МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ

Методичні рекомендації
до виконання лабораторних робіт
з навчальної дисципліни

"ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ"

для студентів напряму підготовки
"Комп'ютерні науки"
всіх форм навчання

Частина 1

Харків. Вид. ХНЕУ, 2008

Затверджено на засіданні кафедри інформаційних систем.
Протокол №7 від 29.02.2008 р.

М54 Методичні рекомендації до виконання лабораторних робіт з навчальної дисципліни "Об'єктно-орієнтоване програмування" для студентів напряму підготовки "Комп'ютерні науки" всіх форм навчання. Ч. 1 / Укл. Ю. Е. Парфьонов, В. М. Федорченко, М. Ю. Лосєв, О. В. Щербаков. – Харків: Вид. ХНЕУ, 2008. – 72 с. (Укр. мов.)

Наведено завдання, що сприяють поглибленню й закріпленню знань з теоретичних питань об'єктно-орієнтованого програмування та придбанню практичних навичок з розробки додатків на мові програмування C#.
Рекомендовано для студентів напряму підготовки "Комп'ютерні науки" всіх форм навчання.

ВСТУП

Відомо, що використання потужних комп’ютеризованих засобів неможливо без програмного забезпечення. Галузь розробки програмного забезпечення набуває все більшого значення, оскільки тенденції розвитку комп’ютерної техніки свідчать про те, що з одного боку складність та функціональні можливості комп'ютерної техніки постійно і швидко зростають, а з другого боку, це потребує більш досконалих програмних засобів для задоволення потреб користувачів.

Істотною рисою таких програмних систем є рівень складності: для одного розробника практично неможливо охопити всі її аспекти. Причому ця складність є неминучою: з нею можливо справитися, але позбавитися від неї неможливо.

У теперішній час найбільш розповсюдженим методом боротьби зі складністю є об'єктно-орієнтований підхід до розробки програмного забезпечення. З використанням цього підходу розробляється більша частина програм у всьому світи. Це потребує від відповідних фахівців чіткого уявлення концепцій об'єктно-орієнтованого програмування, що дає можливість їх практичного використання при розробці додатків будь-якою мовою програмування.

Базовою мовою програмування для виконання лабораторних робіт є C#, оскільки в теперішній час вона інтенсивно використовується компаніями-розробниками програмного забезпечення, а також її досить легко вивчати. C# (вимовляється Сі-шарп) – об'єктно-орієнтована мова програмування, яка розроблена компанією Microsoft (MS) у 2001 році. Вона успадкувала багато корисних можливостей від інших мов програмування і безпосередньо пов'язана з двома найширше вживаними у світі мовами програмування – C++ та Java.

У загальному випадку програми на будь-якій мові програмування призначені для виконання на певній комп‘ютерній платформі. Комп‘ютерна платформа – це сполучення апаратної архітектури, що визначається типом процесору, який використовується, та операційної системи. При написанні програм розробник використовує засоби платформи для доступу до мережі, роботи з графічним інтерфейсом користувача та ін. Різні платформи підтримують різні інтерфейси програмування додатків – API (Application Programming Interface).

Слід зазначити, що у світи є комп'ютери різноманітних типів – Pentium PC, Macintosh, робочі станції Sun та ін. У цих комп'ютерах використовуються різні процесори, головним чином виробництва фірм Intel і AMD. У свою чергу, для процесорів Intel та AMD існує кілька операційних систем, наприклад, Microsoft Windows, Sun Solaris, різні різновиди операційної системи UNIX. Тому технології та мови програмування, які призначені для роботи на різних комп‘ютерних платформах і не залежать від конкретного типу процесору й операційної системи, мають істотні переваги над аналогами.

Програма мовою C# під час її виконання не звертається прямо до API операційної системи. Як посередника для цього вона використовує так звану "віртуальну машину" програмної платформи Microsoft .NET. Таким чином, платформа Microsoft .NET дозволяє створювати додатки, які не потребують перекомпіляції при перенесенні їх на різні комп‘ютерні платформи, що широко використовується для розробки Internet-додатків. Крім того, платформа Microsoft .NET містіть стандартну бібліотеку класів, у якій є компоненти для організації інтерфейсу користувача, звертання до файлів, роботи в мережі тощо. Суттєвою перевагою платформи Microsoft .NET також є підтримка багатомовного програмування. Під багатомовним програмуванням розуміється здатність коду, написаного різними мовами, працювати спільно. Ця можливість дуже важлива при створенні великих програм, а також бажана при програмуванні окремих компонентів, які можна було б використовувати в багатьох комп'ютерних мовах і в різному операційному середовищі.

Курс лабораторних робіт, що пропонується, спрямований на формування твердих практичних навичок щодо розробки додатків із використанням об‘єктно-орієнтованого підходу. Для практичної розробки програм рекомендується використовувати інтегроване середовище розробки Microsoft (MS) Visual Studio.

Загальні рекомендації до виконання лабораторних робіт

Звіт кожної лабораторної роботи повинен містити:

1. Титульний лист на якому повинни бути: назва дисципліни; тема лабораторної роботи; дата виконання роботи; П.І.Б. студента, курс, номер групи; П.І.Б., посада викладача.

2. Лист змісту на якому повинен бути нумерований перелік назв пунктів роботи (див. пункт 3) із зазначенням номерації сторінок.

3. Опис виконаних завдань з: умовою завдання; описом архітектури програми (склад, структура класів, зв‘язки між ними, алгоритми тощо); вихідним кодом програми з документаційними XML-коментарями для класів (призначення класу) та методів (призначення, опис параметрів кожного методу та значення, що він повертає); прикладами результатів роботи програми на тестових вихідних даних (табл. 1):

Таблиця 1

№ з/п	Вихідні дані	Одержаний результат	Очікуваний результат	Позначка про прохождення тесту (+/-)

4. Висновки з лабораторної роботи з урахуванням усіх завдань, що були виконані:

аналіз результатів, які були одержані за кожним пунктом завдання;

аналіз результатів тестування програм;

ступінь відповідності програм, що були розроблені при постановці завдання;

інша інформація.

Вимоги до оформлення звіту

1. Звіт із лабораторної роботи виконується в друкованому вигляді на аркушах формату А4 відповідно до державних стандартів;

2. аркуші звіту з’єднуються скріпками або іншим загальноприйнятим способом.

Лабораторна робота 1.
Основи використання мови C#

Цілі лабораторної роботи

1. Придбання практичних навичок розробки найпростіших консольних додатків мовою з використанням основних елементів мови C#.

2. Удосконалювання навичок роботи з інтегрованим середовищем розробки MS Visual Studio й довідковою системою Microsoft Developer Network (MSDN).

Перед виконанням лабораторної роботи студент повинен знати:

1. Загальні відомості про мову C#;
2. Основи синтаксису мови C#.

Після виконання лабораторної роботи студент повинен вміти:

1. Розробляти прості консольні додатки з використанням базових елементів мови C#.

Основи мови програмування C# та використання системи програмування MS Visual Studio щодо розробки програм цією мовою

Структура програми на C#

Програма на **C#** складається з одного або кількох файлів. У кожному додатку на **C#** повинен бути метод **Main()**, визначений в одному з його класів. Виконання будь-якої програми починається з методу **Main()**.

Програма "Hello, World" мовою C# виглядає таким чином:

```
using System;
class HelloWorld01
{
  public static void Main()
  {
    Console.WriteLine("Hello, World!");
  }
}
```

Уся стандартна бібліотека .Net розбита на окремі групи, які позначуються терміном namespace (простір імен). Ці групи складаються з класів а класи, у свою чергу, містять методи. Наприклад, функції ReadLine() і WriteLine() належать класу Console. Клас Console належить до групи System. Таким чином, повне ім'я функції ReadLine() буде виглядати так: System.Console.ReadLine().

Ключове слово using позбавляє від необхідності щораз при виклику функції набирати назву її групи. Наприклад, завдяки рядку using System; можливо записати Console.ReadLine() замість System.Console.ReadLine().

Типи даних

Система типів C# повністю відображає систему типів .NET C#, підрозділяє типи на два види: вбудовані типи, які визначені в мові, і типи, які визначає **користувач**, тобто програміст.

Важлива особливість цієї системи типів полягає в явному розділенні всіх типів на типи-значення та посилальні типи. Основна різниця між ними – це спосіб, яким їх значення зберігаються в пам'яті. Типи-значення зберігають своє фактичне значення в стеці. Посилальні типи зберігають у стеці лише адреси об'єкта, а сам об'єкт зберігається в динамічної області пам‘яті.

До типів-значень відноситься широкий набір примітивних типів даних, включаючи цілі числа різної розрядності, типи з плаваючою комою різної точності, спеціальний тип decimal із фіксованою точністю, призначений для фінансових обчислень, а також символьний тип char, здатний зберігати символи у форматі Unicode й тому зручний при розробці інтернаціональних застосувань. Усі цілочисельні типи існують у двох варіантах: знаковому і беззнаковому. C# містить також спеціальний тип для булевих значень; змінні булевого типу можуть містити значення true або false.

Окрім примітивних типів, у C# існує можливість організовувати дані в структури, що складаються зі змінних будь-якого типу, або в перерахування, складені з декількох змінних одного й того ж цілочисельного типу. Важливою особливістю перерахувань у C# є необхідність явного приведення до базового типу за бажання проінтерпретувати значення з перерахування як число.

Таблиця 2 показує ключові слова для вбудованих типів C#, які є псевдонімами типів .NET у просторі імен System:

Таблиця 2

Ключові слова вбудованих типів мови C#	Типи .NET Framework
1	2
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal

Закінчення табл. 2

1	2
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

Прості цілочисельні типи

Таблиця 3 показує розміри й діапазони цілочисельних типів, які становлять підмножину простих типів:

Таблиця 3

Тип	Діапазон значень	Розмірність, біт
sbyte	-128 to 127	8
byte	0 to 255	8
char	U+0000 to U+ffff	16
short	-32,768 to 32,767	16
ushort	0 to 65,535	16
int	-2,147,483,648 to 2,147,483,647	32
uint	0 to 4,294,967,295	32
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	64
ulong	0 to 18,446,744,073,709,551,615	64

Прості типи з плаваючою крапкою (табл. 4):

Тип	Діапазон значень	Точність
float	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7 знаків
double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15-16 знаків

Вирази й оператори C# в основному аналогічні операторам мови C++, за виключенням:

1. Оператор вибору варіанта switch

Можливо використовувати рядок як перевірочне значення:

```
string myString;
……….
switch (myString)
{
 case "Hello":
 // робити щось
 break;
 case "Goodbye":
 ……..
}
```

Кожна конструкція case повинна мати явний вихід. Не допускається наскрізне виконання case, якщо тільки початковий case не порожній.

2. Оператор циклу foreach

C# пропонує додатковий оператор управління потоком виконання програми – foreach. Цикл foreach виконує ітерації за елементами масиву або колекції, без необхідності явної специфікації індексів. Цикл foreach, що працює з масивом, повинен виглядати так, як показано нижче. У цьому прикладі MyArray – масив елементів типу double, і потрібно вивести кожне значення в консольне вікно. Щоб зробити це, використовується наступний код:

```
foreach (double someElement in myArray)
{
  Console.WriteLine(someElement);
}
```

У цьому циклі someElement є ім'ям, призначене змінної, яка використовується для ітерації в циклі. Цей цикл буде мати такий самий ефект, що й

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

```
{
  Console.WriteLine(myArray[i]);
}
```

Масиви

Масив у C# – це посилальний об'єкт. Є можливість як використовувати "класичні" масиви, так і працювати з масивами за допомогою стандартного класу System.Array.

Одномірні масиви

Приклад:

Синтаксис	Опис
int [ ] MyBox;	оголошення посилання на масив;
int [ ] MyBox = {5; -4; 75};	оголошення посилання на масив та ініціалізація масіву з трьох елементів;
int [ ] MyBox; MyBox = new int [3];	оголошення посилання на масив і визначення масиву з трьох елементів;
int [ ] MyBox = new int [3];	оголошення посилання на масив та одночасне визначення масиву з трьох елементів.

У C#, як і в C/C++, нумерація елементів масиву починається з нуля. Таким чином, у цьому прикладі початковий елемент масиву – це MyBox[0], а останній – MyBox[2]. Елемента MyBox[3], природно, немає.

Багатовимірні масиви

Синтаксис	Опис
int [,] MyBox = new int [2,3];	оголошення посилання на масив та одночасне визначення двовимірного масиву із шести елементів;
int [, ,] MyBox = new int [10,10,10];	оголошення посилання на масив та одночасне визначення тривимірного масиву з тисячі елементів;
int [, ,] MyBox = {(2,-2),(3,-22), (0,4)};	оголошення посилання на масив та ініціалізація тривимірного масіву з шести елементів

Робота з масивами за допомогою класу System.Array

Поняття масиву в C# є значно ширшим, ніж просте об’єднання множини значень у змінній з одним ім’ям. У C# масиви є об’єктами, похідними від базового класу System.Array. Тому, при створенні масиву

насправді відбувається створення екземпляра класу, успадкованого від System.Array.

Будь-який масив є об'єктом класу, похідним від System.Array. Як і будь-який інший клас, System.Array має свої методи й властивості. Одним з найважливіших методів класу System.Array є метод GetLength(). Він дозволяє одержати розмір певного виміру багатовимірного масиву. Інше важливе достоїнство класу System.Array - властивість Rank, що дозволяє програмним шляхом визначити ранг масиву, тобто кількість його вимірів. Крім того, клас System.Array має властивість Length, що повертає загальну кількість елементів у масиві.

Розробка консольного додатка C# у системі програмування MS Visual Studio

1. Запустіть MS Visual Studio. Після цього на екрані з'явитися головне вікно програми.

2. Для створення консольного додатка виберіть пункт меню File, у ньому – команду New, а потім - команду Project... З'явиться діалогове вікно New Project (новий проект), рис. 1. У цьому діалоговому вікні вкажіть у розділі Project Types (тип проекту) значення Visual C#, у розділі Templates (шаблони) – значення Console Application (консольний додаток), вкажіть у полі Name (ім'я) ім'я проекту, наприклад, HelloWorld і виберіть папку, в яку буде поміщений новий проект. Для цього в полі Location (Розміщення) виберіть диск і папку.

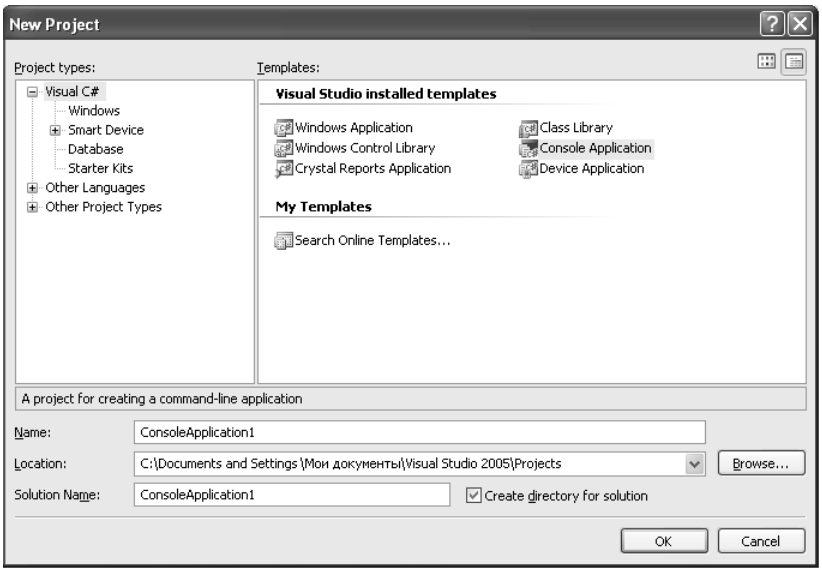

Рис. 1. **Діалогове вікно New Project**

3. Натиснить кнопку ОК. У головному вікні MS Visual Studio з'являться автоматично згенерована "заготівка" вихідного коду програми та вікно провідника рішення (Solution Explorer) із папками й файлами проекту (рис. 2).

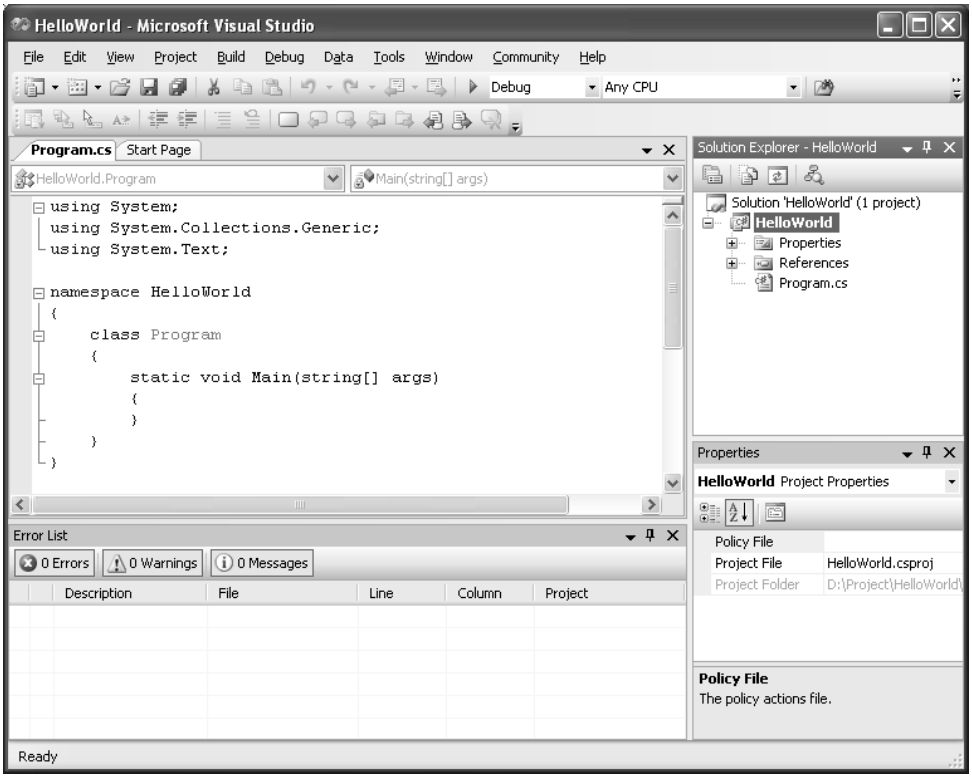

Рис. 2. **Головне вікно MS Visual Studio з кодом програми, та вікном Solution Explorer**

Для одержання довідкової інформації про елементи стандартної бібліотеки .Net за допомогою утиліти Object Browser необхідно обрати меню View – пункт Object Browser.

Документування вихідного коду програми

У C# допускаються однорядкові й багаторядкові коментарі.

Перші починаються з двох символів скісної риски. Весь текст до кінця рядка, що розміщується за цією парою символів, сприймається як коментар, що не впливає на виконання програмного коду. Початком багаторядкового коментарю є пара символів /*, а кінцем - */.

Тут же, в консольному проекті, ми зустрічаємося з ще однією досить важливою новинкою C# – XML-тегами, яки призначені для "самодокументування" коду.

Приклад використання документаційних XML-коментарів:

```
using System;
namespace ConsoleHello
{
  /// <summary>
  /// Перший консольний проект
  /// </summary>
 class Class1
 {
    /// <summary>
    /// Точка входу. Запитує ім'я й видає вітання.
    /// </summary>
    static void Main()
    {
      Console.WriteLine("Введіть Ваше ім'я");
      string name;
      name = Console.ReadLine();
      if (name == "")
        Console.WriteLine ("Добридень, світ!");
      else
        Console.WriteLine("Добридень, " + name + "!");
    }

     /// <summary>
     /// Математичний метод
     /// </summary>
     /// <param name="Number">Дійсне  число</param>
     /// <returns>Квадрат аргументу</returns>
     double MyMethod(double Number)
     {
      return Math.Pow(Number, 2);
     }
 }
}
```

Опису класу Class1 і опису методу Main передує заданий у рядковому коментарі XML-тег <summary>. Цей тег розпізнається спеціальним інструментарієм, що будує XML-звіт проекту. Крім тегу <summary> можливі й інші теги, що включаються у звіти. Наприклад, тег <param> слугує для опису параметра, який передається методу, тег <returns> – для опису значення, що повертає метод. Деякі теги додаються майже автоматично. Якщо в потрібному місці (перед оголошенням класу, методу) набрати підряд три символи косої риси, то автоматично вставиться необхідний тег документування, так що залишиться тільки доповнити його відповідною інформацією.

Порядок виконання лабораторної роботи

Загальні вимоги до розроблювальних програм:

наявність найпростішого текстового інтерфейсу користувача;

забезпечення коректної обробки вихідних даних із метою запобігання появи помилок при виконанні програми.

Розробити консольні додатки, зазначені в завданнях:

1. Додаток, що виводить на консоль прізвище студента, а також поточну дату й час.

Примітка*: для одержання поточної дати й часу використовуйте структуру .Net* System.DateTime

2. Використовуючи елементи класу System.Math (стандартний клас .Net для підтримки математичних операцій), розробити програму, що обчислює й виводить на консоль значення одного з виразів:

Номер варіанта	Вираз
1	$a^{1/2}$
2	|a|
3	exp(a)
4	lg (a)
5	sin(a)
6	cos(a)
7	tg(a)
8	ctg(a)
9	a^b
10	ln(a)
11	перетворення a (радіани→градуси)
12	перетворення a (градуси →радіани)

Вихідні дані повинні вводитися з консолі.

Примітка: *Потрібно врахувати, що стандартні методи класу Math для обчислення тригонометричних функцій очікують параметр у радіанах, а вихідні дані й результат повинні бути представлені в градусах.*

3. Додаток, що приймає вихідні дані з консолі, перераховує температуру за шкалою Фаренгейта в температуру за шкалою Цельсия й виводить на консоль "таблицю" відповідності температур із заданого діапазону. Перерахування температур повинно бути реалізовано у вигляді статичного методу класу.

Вихідні дані : $t_{ф.нач}$ – початкова температура за Фаренгейтом; $t_{ф.кін}$ - кінцева температура за Фаренгейтом; $\Delta t_ф$ – крок зміни температури.

Математичний опис завдання:

$$C = \frac{5}{9} \cdot (F - 32),$$

де F – значення температури за Фаренгейтом;

C – значення температури за Цельсієм.

Контрольні питання

1. Охарактеризуйте структуру програми мовою C#.
2. Перелічіть вбудовані типи даних C#.
3. Призначення методів класу System.Math.
4. Запишіть й охарактеризуйте загальний формат методу, що повертає результат.
5. Порядок створення проекту консольного додатка C# у MS Visual Studio.
6. Порядок відкриття існуючого проекту в MS Visual Studio.
7. Порядок додавання файлу в проект у MS Visual Studio.
8. Порядок компіляції та запуску програми в MS Visual Studio.
9. Порядок використання утиліти Object Browser.
10. Документування вихідного коду програми в MS Visual Studio за допомогою документаційних XML-коментарів.

Лабораторна робота 2.
Розробка додатків із використанням базових елементів об'єктно-орієнтованого програмування

Цілі лабораторної роботи

1. Ознайомлення з основними елементами мови C#.
2. Придбання практичних навичок використанням базових елементів об‘єктно-орієнтованого програмування (ООП) під час розробки консольних додатків мовою C#.
3. Удосконалювання навичок роботи з інтегрованим середовищем розробки MS Visual Studio й довідковою системою MSDN.

Перед виконанням лабораторної роботи студент повинен знати:

1. Загальні синтаксичні конструкції мови C#.
2. Реалізацію базових елементів ООП у мові C#.

Після виконання лабораторної роботи студент повинен вміти:

Розробляти прості консольні додатки мовою C# із використанням базових елементів ООП.

Реалізація базових елементів ООП у мові C#

Парадигма програмування – це набір теорій, методів, стандартів, які використовуються при розробці та реалізації програм на комп'ютері. ООП – одна з парадигм програмування, яка розглядає програму як множину взаємодіючих "об'єктів". Кожний об'єкт здатен отримувати повідомлення, обробляти дані та надсилати повідомлення іншим об'єктам. Кожний об'єкт можна розглядати як незалежний автомат з окремим призначенням або відповідальністю.

ООП основане на трьох основних принципах:

1. Інкапсуляція – об'єднання в єдине ціле (клас) даних і алгоритмів обробки цих даних. В ООП дані називаються полями, а алгоритми – методами.
2. Спадкування – можливість створення ієрархії класів, коли класи-нащадки отримують від класа-попередника поля й методи.
3. Поліморфізм (від гр. poly - багато і morphos – форма, означає багато форм) – це властивість класів однієї ієрархії вирішувати схожі за змістом завдання за допомогою різних алгоритмів.

Класи

Класи – це основний спосіб організації даних у C#. Будь-яка програма, що написана цією мовою, повинна складатися не менш чим із одного класу (отже C# є "справжньою" об'єктно-орієнтованою мовою, на відміну, скажімо, від C++, у якому використання класів можливе, але необов'язково).

Функції в мові C#

Будь-яка функція повинна бути членом класу (методом). Методи можуть приймати або не приймати параметри, повертати або не повертати значення. Параметри можуть передаватися в методи за значенням або за посиланням. Методи можуть бути статичними або методами екземплярів класу (об'єктів).

Модифікатори

Одним з важливих принципів об'єктно-орієнтованого підходу до програмування є інкапсуляція даних: вважається, що внутрішній устрій класу й конкретна реалізація його методів повинні бути невідомі зовнішнім споживачам. Для виразу різних рівнів доступності елементів програми в C# існує обширний набір модифікаторів. Наприклад, поля та методи можуть мати таки основні модифікатори доступу:

public (даний елемент класу доступний усім зовнішнім споживачам);

private (елемент недоступний за межами опису класу; цей модифікатор ставиться за умовчанням);

internal (елемент доступний тільки для класів, визначених у тій же збірці, що і даний клас).

Крім того, існують модифікатори, що змінюють поведінку елементів класу, наприклад:

const (свідчить, що дана змінна не може бути змінена);

readonly (описує змінні, які можуть набути значення тільки при самому описі або в конструкторі класу);

static (вказує, що даний елемент належить типу об'єкта, а не конкретному екземпляру).

Приклад програми на C#, у якій є два класа: Hero, FairyTale наведений нижче:

```
class Hero {
  private int Age;
  private string Name;
  private readonly bool Alive;
  // Конструктор
  public Hero (int Age, string Name) {
```

```
    this.Age=Age; this.Name = Name;
    Alive = true;
  }
  public int GetAge() { return Age; }
  public bool IsAlive() { return Alive; }
  public static void Hobby() { ...}
  public void Talk() {...}
  ……..
}

public class FairyTale {
 public static void Main() {
  // Створення об'єкта класу  Hero
  Hero LittleHero = new Hero(19, "Student");
 Console.WriteLine(LittleHero.GetName() + ", " +
                    LittleHero.GetAge() + ", " + LittleHero.IsAlive());
 }
}
```

Ключове слово this

є посиланням на поточний екземпляр об'єкта;

посилання this є схованим покажчиком, що передається в кожний нестатичний метод класу;

будь-який метод може використовувати this для доступу до інших нестатичних методів і змінних цього об'єкта.

Конструктори класів:

конструктор – метод, який ініціалізує стан об'єкта після його створення;

конструктор має те ж ім'я, що й клас, у якому він використовується;

конструктор не повертає значення (навіть типу void);

конструктор викликається автоматично;

якщо конструктор не заданий у програмі, то він буде автоматично згенерований компілятором для побудови відповідних об'єктів;

конструктор можна перевантажувати.

Масиви об'єктів у C#

Далі наведений фрагмент програми, в якому створюється масив для зберігання двох об‘єктів класу MyClass та два об‘єкта цього класу додаються до масиву:

```
MyClass[ ]  ArrayOfObject= new MyClass[2];
   for (int i = 0; i < ArrayOfObject.Length; i++)
        ArrayOfObject[i] = new MyClass();
```

Порядок виконання лабораторної роботи

Загальні вимоги до розроблювальних програм:

наявність найпростішого текстового інтерфейсу користувача;

забезпечення коректної обробки вихідних даних із метою запобігання появи помилок при виконанні програми;

розробка програми для обробки відомості (див. варіанти завдань).

Для цього необхідно описати клас, поля якого відповідають вихідним полям відомості. Для установки значень полів повинен використовуватися конструктор. Обчислення значень розрахункових полів відомості, одержання значень вихідних полів повинне виконуватися за допомогою відповідних нестатичних методів класу.

Програма повинна забезпечувати створення масиву об'єктів цього класу, уведення вихідних даних з консолі й вивід на консоль вихідних значень і полів, що розраховуються, кожної із записів відомості, а також підсумкової інформації з відомості.

Варіант 1

Відомість нарахування зарплати співробітникам підприємства:

№ з/п	Прізвище	Зарплата, грн	Утримано, грн	Видано, грн
1	F	Z	P	S = Z – P
2				
…				
	Разом	∑	∑	∑

Варіант 2

Відомість витрат палива на автобазах міста:

№ з/п	Автобаза	Витрачено палива, кг	Кількість автомашин, шт.	Середня витрата палива, кг
1	A	T	K	C = T/K
2				
…				
n				
	Разом	∑	∑	∑/n

Варіант 3

Відомість використання машинного часу в обчислювальному центрі:

№ з/п	Кафедра	Використання машинного часу, годин		Відхилення від плану	
		за планом	фактично	у годинах	у %
1	K	P	F	O_1 = P-F	$O_2 = O_1 \times 100/P$
2					
…					
	Разом	∑	∑		

Варіант 4

Відомість споживання електроенергії на заводах міста:

№ з/п	Завод	Споживання електроенергії, кВт/год.		Відхилення від плану	
		за планом	фактично	у кВт/год.	у %
1	Z	P	F	O_1= P-F	$O_2=O_1\times100/P$
2					
…					
	Разом	∑	∑		

Варіант 5

Відомість руху матеріалів на складі підприємства за звітний період:

№ з/п	Склад	Рух матеріалів за період, грн			Залишок на кінець періоду
		залишок на початок періоду	отримано	видано	
1	C	O_c	P	V	$R = O_c + P – V$
2					
…					
	Разом	∑	∑	∑	∑

Варіант 6

Відомість прибутку підприємства за звітний період за видами продукції:

№ з/п	Продукція	Кількість, шт.	Оптова ціна, грн	Собівартість, грн	Прибуток, грн
1	Pr	K	Z	C	P = K(Z – C)
2					
…					
	Разом	∑			∑

Варіант 7

Відомість відвідування занять студентами:

№ з/п	Прізвище	Пропущено, год.		Пропуск	
		усього	виправдано	у годинах	у %
1	F	V	O	$P_1 = V – O$	$P_2 = P_1 \cdot \times 100/V$
2					
…					
	Разом:	∑	∑	∑	

Варіант 8

Відомість обсягу поставок продукції в натуральному та вартісному виразах:

№ з/п	Продукція	Шифр	Обсяг постачань, шт.	Оптова ціна, грн	Обсяг постачань, грн
1	P	H	V	Z	O = V × Z
2					
…					
	Разом		∑	∑	∑

Варіант 9

Відомість розрахунку середньої вартості перевезення авіапасажирів:

№ з/п	Тип літака	Рейс	Витрати на рейс, грн.	Кількість пасажирів	Середня вартість перевезення, грн.
1	T	R	Z	K	S = Z/K
2					
…					
n					
	Разом:		∑	∑	∑/n

Варіант 10

Відомість обліку часу роботи верстатів підприємства:

№ з/п	Тип верстата	Час роботи, год.		Відхилення від плану	
		за планом	фактично	у годинах	у %
1	Z	P	F	$O_1 = P-F$	$O_2 = O_1 \times 100/P$
2					
…					
	Разом	∑	∑		

Варіант 11

Відомість випуску деталей робітниками цеху:

№ з/п	Прізвище	Кількість деталей, шт.		Брак	
		виготовлено	прийнято	шт.	%
1	Z	P	F	$O_1 = P-F$	$O_2=O_1\times100/P$
2					
…					
	Разом	$\sum$	$\sum$	$\sum$	

Варіант 12

Відомість наявності та руху основних фондів підприємства:

№ з/п	Назва фонду	Наявність на початок року, шт.	Поступило, шт.	Вибуло, шт.	Наявність на конець року, шт.
1	F	N_1	P	V	$N_2 = N_1 + P – V$
2					
…					
	Разом	$\sum$	$\sum$	$\sum$	$\sum$

Контрольні питання

1. Принципи об’єктно-орієнтованого програмування
2. Поняття класу й об'єкта, співвідношення між ними..
3. Життєвий цикл об'єкта в програмі.
4. Призначення й варіанти використання посилання this.
5. Що означає перевантаження методу?
6. Призначення конструктора.
7. Основні особливості конструктора.

Лабораторна робота 3
Застосування концепції повторного використання класів при розробці додатків.

Цілі лабораторної роботи:

1. Удосконалювання практичних навичок з розробки класів мовою C#.

2. Придбання практичних навичок застосування агрегації, спадкування й поліморфізму в мові C#.

3. Удосконалювання навичок роботи з інтегрованим середовищем розробки MS Visual Studio та довідковою системою MSDN.

Перед виконанням лабораторної роботи студент повинен знати:

1. Поняття класу й об'єкта, співвідношення між ними.
2. Принципи об’єктно-орієнтованого програмування.
3. Синтаксис опису класу.
4. Життєвий цикл об'єкта в програмі.
5. Порядок використання конструкторів.
6. Синтаксис агрегації й спадкування в C#.
7. Порядок виклику конструкторів при спадкуванні.
8. Принципи перевантаження й перевизначення методів.
9. Принципи реалізації поліморфізму в C#.
10. Порядок використання абстрактних класів і інтерфейсів.

Після виконання лабораторної роботи студент повинен вміти:

1. Самостійно розробляти класи мовою C#.
2. Застосовувати механізми спадкування й поліморфізму при розробці програм мовою C#.

Реалізація агрегації, спадкування й поліморфізму в мові C#

Агрегація описує відношення цілого й частини, яке спричиняє появлення відповідної ієрархії об‘єктів, причому, ідучи від цілого (агрегату), можливо прийти до його частин (атрибутів).

Агрегація може означати фізичне входження одного об‘єкта до іншого, але це не обов‘язково. Наприклад, літак складається з крил, двигунів, шасі та інших частин. З другого боку, відношення акціонера та його акцій – це агрегація, яка не передбачає фізичного включення. Акціонер володіє своїми акціями, але вони в нього не входять фізично. Це, безсумнівно, відношення агрегації, яке є скоріше концептуальним, чим фізичним за своїм походженням.

Об‘єкт, який є атрибутом іншого об‘єкта (агрегату), має зв‘язок зі своїм агрегатом. Через цей зв‘язок агрегат може надсилати йому повідомлення.

Синтаксично агрегація – це використання посилання на об‘єкт одного класу як поля іншого.

Приклад відношення агрегації (автомобіль із радіоприймачем) наведений на рис. 3:

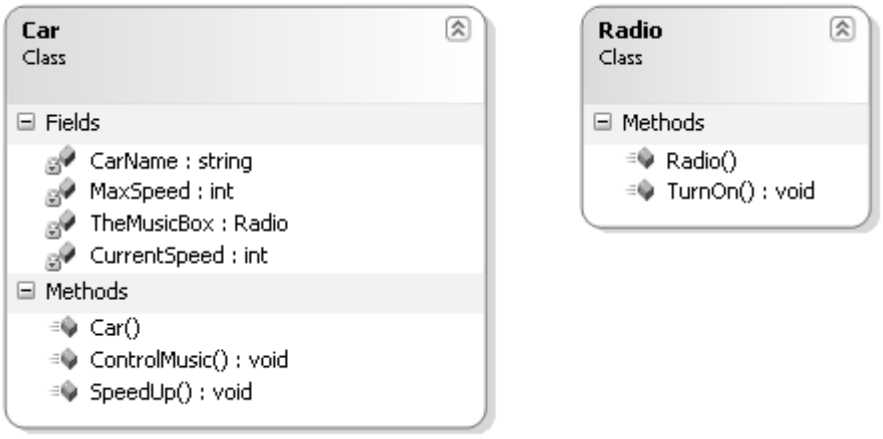

Рис. 3. **Приклад відношення агрегації**

Реалізація відношення агрегації мовою C# показана далі:

```
class Radio
{
      public Radio() { }

      public void TurnOn(bool On)
      {
         if (On)
            Console.WriteLine("Увімкнули радіо...");
         else
            Console.WriteLine("Виключили радіо...");
      }
}

class Car
{
   private int CurrentSpeed;
   private int MaxSpeed;
   private string CarName;
   private Radio TheMusicBox;
```

```
  public Car(string CarName, int MaxSpeed, int CurrentSpeed)
  {
     this.CarName = CarName;
     this.MaxSpeed = MaxSpeed;
     this.CurrentSpeed = CurrentSpeed;
     TheMusicBox = new Radio();
   }

   public void ControlMusic(bool State)
   {
       TheMusicBox.TurnOn(State);
   }

   public void SpeedUp(int Delta)
   {
        CurrentSpeed += Delta;
        if (CurrentSpeed >= MaxSpeed)
        {
           Console.WriteLine("Швидкість автомобіля " +  CarName + "
більше максимальної !");
           Console.WriteLine("Швидкість буде примусово знижена на "
+ (2 * Delta));
            CurrentSpeed -= 2 * Delta;
        }
       else
            Console.WriteLine("\tскорость автомобіля = " +
                                                    CurrentSpeed);

   }
}

class CarApp
{
   public static void Main()
   {
        Car C1;
        C1 = new Car("Копійка", 100, 10);
```

```
            C1.ControlMusic(true);
            for (int i = 0; i < 10; i++)
            {
               C1.SpeedUp(20);
            }
            C1.ControlMusic(false);
        }
}
```

Принцип успадкування:

ієрархічне відношення між класами, при якому клас використовує структуру або поведінку іншого класу;

при встановленні між класами відношення спадкування між ними також встановлюється залежність "часткове – загальне";

класи-спадкоємці доповнюють або перевизначають структуру й поведінку класів-предків, які успадковані.

Приклад відношення спадкування подано на рис. 4.

Усі герої казки мають властивості та методи класу Hero.
Людина (Human) вміє читати книги, тварина (Animal) вміє полювати, жінка (Woman) – пекти пироги, а чоловік (Man) – рубати дерева.

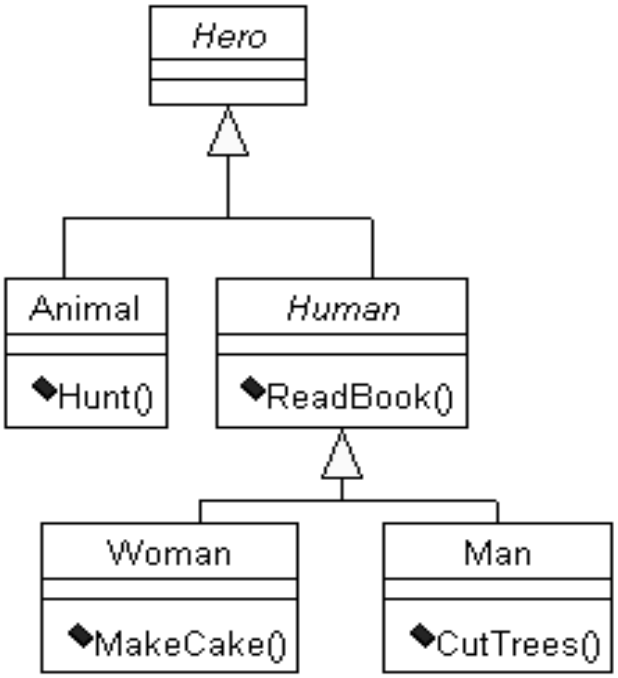

Рис. 4. **Діаграма спадкування**

Далі наведена реалізація діаграми спадкування (рис. 4) мовою C#:

```
class Hero { }
class Animal : Hero {
```

```
  public void Hunt() { }
}
class Human : Hero {
  public void ReadBook() { }
}
class Woman : Human {
  public void MakeCake() { }
}
class Man : Human {
  public void CutTrees() { }
}
```

Ініціалізація об'єктів при спадкуванні:

а) конструктори без параметрів

```
class Art {
 public Art() { Console.WriteLine("Art");  }
}
class Drawing : Art {
 public Drawing() { Console.WriteLine("Drawing"); }
}
class Cartoon : Drawing {
 public Cartoon() { Console.WriteLine("Cartoon"); }
 public static void Main(String[ ] args) {
    // створення об‘єкта класу Cartoon
    Cartoon x = new Cartoon();
 }
}
```

У цьому прикладі після створення об‘єкта класу Cartoon викликається його конструктор. У ньому спочатку автоматично викликається конструктор найближчого базового класу (тобто конструктор класу Drawing). Із конструктора класу Drawing спочатку автоматично викликається конструктор найближчого базового класу (тобто конструктор класу Art). Таким чином послідовність викликів конструкторів така: Cartoon() → Drawing() → Art(). Далі виконується програмний код усіх конструкторів ієрархії: Art() → Drawing() → Cartoon().

Результат роботи програми:

Art
Drawing
Cartoon

б) конструктори з параметрами

```
class Box {
  private double Width;
  private double Height;
  private double Depth;
  Box(double Width, double Height, double Depth) {
    this.Width = Width;
    this.Height = Height;
    this.Depth = Depth;
  }

class WeightBox : Box {
 private double Weight;
 WeightBox(double Width, double Height, double Depth, double
          Weight) : base (Width, Height, Depth)
 {
  // Виклик конструктора найближчого базового класу
  this.Weight = Weight;
 }
}

class BoxManager {
 public static void Main(String[ ] args) {
  WeightBox box = new WeightBox(2.0, 3.0, 5.0, 125.5);
 }
}
```

У цьому випадку концептуально відбувається теж саме, що й у випадку а), але для виклику конструктора найближчого базового класу необхідний явний виклик за допомогою ключового слова base**.**

Основні варіанти використання спадкування:

"В – це А"

```
class A
{       // базовий клас
  private int field;
  public void f() { }
}

class B : A { }
```

"B схожий на A"

```
class A
{       // базовий клас
  private int field1;
  public void f() { }
}

class B : A
{
   private int field2;
   public void g() { }
}
```

перевантаження методу базового класу

```
class A
{       // базовий клас
        private int field1;
        public void f() { }
}

class B : A {
        private int field2;
        public void f(int argument) { }
}
```

перевизначення методу базового класу

```
class A
{       // базовий клас
   private int field1;
```

```
  public virtual void f() { }
}

class B : A
{
  private int field2;
  public override void f() { }
    //Наприклад виклик методу базового класу
    base.f();
  }
}
```

Можлива заборона спадкування від класу за допомогою ключового слова **sealed**:

```
sealed class A
{
  private int field1;
  public void f() { }
}
```

Принцип поліморфізму

Поліморфізм - здатність об'єктів з однієї ієрархії класів відповідати на один запит (рис. 5), наприклад "малювати":

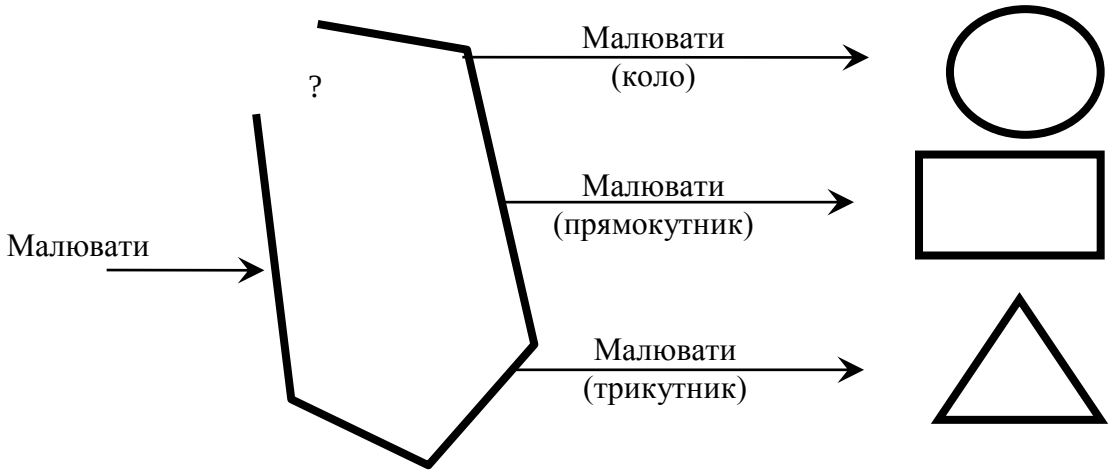

Рис. 5. **Приклад поліморфізму – малювання геометричних фігур**

Використання принципу поліморфізму на прикладі геометричних фігур (рис. 6):

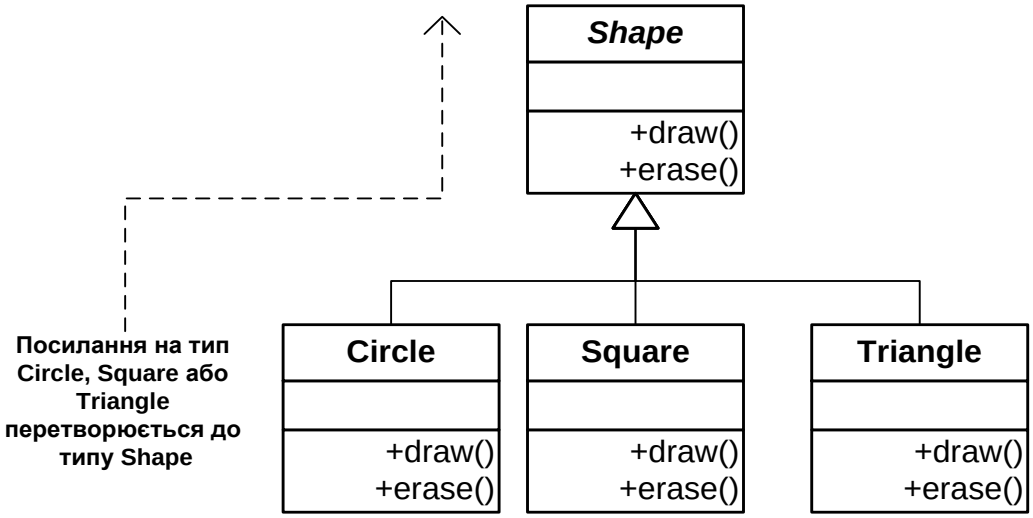

Рис. 6. **Діаграма класів-геометричних фігур**

Одною з необхідних умов реалізації поліморфізму є спадкування. Другою – виклик віртуальних методів, що визначені в базовій абстракції (загальний інтерфейс ієрархії), та перевизначаються в похідних класах, через посилання на базовий тип (див. рис. 6).

Реалізація поліморфної поведінки мовою C# наведена дали:

```
class Shape
{
  //Малювати геометричну фігуру
  public virtual void draw()
  {
    Console.WriteLine("Shape.draw()");
  }
  //Стирати геометричну фігуру
  public virtual void erase()
  {
    Console.WriteLine("Shape.erase()");
  }
}

class Circle : Shape
{
  public override void draw()
```

```
  {
    Console.WriteLine("Circle.draw()");
  }
  public override void erase()
  {
    Console.WriteLine("Circle.erase()");
  }
}

class Square : Shape
{
  public override void draw()
  {
   Console.WriteLine("Square.draw()");
  }
  public override void erase()
  {
    Console.WriteLine("Square.erase()");
  }
}
class Triangle : Shape
{
  public override void draw()
  {
  Console.WriteLine("Triangle. draw()");
  }
  public override void erase()
  {
    Console.WriteLine("Triangle.erase()");
  }
}

public class ShapeManager
{
  public static void Main()
  {
      // "Створення" геометричної фігури
```

```
    Shape s = new Shape();
    s.draw();// викликається draw() із Shape
    s.erase();// викликається erase() з Shape
    s = new Circle();
    s.draw(); // викликається draw() із Circle
    s.erase(); // викликається erase() з Circle
    s = new Square();
    s. draw(); // викликається draw() із Square
    s.erase(); // викликається erase() з Square
    s = new Triangle();
    s. draw(); // викликається draw() із Triangle
    s.erase(); // викликається erase() з Triangle
  }
}
```

Помітьте, що створення об‘єкта класу Shape, тобто якоїсь абстрактної геометричної фігури, не має особливого сенсу, тому що незрозуміло як її малювати та стирати.

Як правило, базовою абстракцією при реалізації поліморфізму є абстрактний клас або інтерфейс.

Абстрактні класи надають поняття предметної області, наділені всіма найбільш загальними властивостями й/або поведінкою, які передбачається перевизначати в нащадках (похідних класах).

Правила використання абстрактних класів:

реалізують вершину в ієрархії спадкування;

не можна створити їхні об'єкти;

можна оголосити посилання на абстрактний клас, що згодом буде використовуватися для роботи з об'єктами похідних класів;

можуть містити абстрактні й звичайні методи;

абстрактні методи не мають потреби в реалізації;

усі абстрактні методи повинні бути перевизначені в похідному класі або похідному класі повинен бути теж оголошений абстрактним.

Реалізація поліморфної поведінки з абстрактним класом:

```
abstract class Shape
{
  //Малювати геометричну фігуру
```

```
  public abstract void draw();
  //Стирати геометричну фігуру
  public abstract void erase();
}

class Circle : Shape
{
  public override void draw()
  {
    Console.WriteLine("Circle.draw()");
  }
  public override void erase()
  {
    Console.WriteLine("Circle.erase()");
  }
}

class Square : Shape
{
  public override void draw()
  {
   Console.WriteLine("Square.draw()");
  }
  public override void erase()
  {
    Console.WriteLine("Square.erase()");
  }
}

class Triangle : Shape
{
  public override void draw()
  {
  Console.WriteLine("Triangle. draw()");
  }
  public override void erase()
  {
```

```
    Console.WriteLine("Triangle.erase()");
  }
}

public class ShapeManager
{
  public static void Main()
  {
     // Shape s = new Shape(); // помилка
    Shape s; // не має помилки
     s.draw();// викликається draw() із Shape
     s.erase();// викликається erase() з Shape
     s = new Circle();
     s.draw(); // викликається draw() із Circle
     s.erase(); // викликається erase() з Circle
     s = new Square();
     s. draw(); // викликається draw() із Square
     s.erase(); // викликається erase() з Square
     s = new Triangle();
     s. draw(); // викликається draw() із Triangle
     s.erase(); // викликається erase() з Triangle
  }
}
```

Інтерфейс (interface) – це іменований набір семантично зв'язаних абстрактних елементів. Інтерфейси використовуються для реалізації моделі поведінки об'єкта.

Правила використання інтерфейсів:

реалізують вершину ієрархії спадкування;

не можна створити їх об'єкти;

можна оголосити посилання на інтерфейс, що згодом буде використовуватися для роботи з об'єктами класів реалізації;

елементи інтерфейсу неявно є public;

можуть містити тільки абстрактні методи;

усі абстрактні методи повинні бути перевизначені в класах реалізації;

кожний член інтерфейсу (властивість або метод) автоматично є абстрактним;

клас може реалізовувати необмежена кількість інтерфейсів.

Реалізація поліморфної поведінки з абстрактним класом:

```
interface IShape
{
  //Малювати геометричну фігуру
  void draw();
  //Стирати геометричну фігуру
  void erase();
}

class Circle : Shape
{
  public void draw()
  {
    Console.WriteLine("Circle.draw()");
  }
  public void erase()
  {
    Console.WriteLine("Circle.erase()");
  }
}

class Square : Shape
{
  public void draw()
  {
   Console.WriteLine("Square.draw()");
  }
  public void erase()
  {
    Console.WriteLine("Square.erase()");
  }
}
```

```
class Triangle : Shape
{
  public void draw()
  {
  Console.WriteLine("Triangle. draw()");
  }
  public void erase()
  {
    Console.WriteLine("Triangle.erase()");
  }
}

public class ShapeManager
{
  public static void Main()
  {
     // IShape s = new IShape(); // помилка
     IShape s; // не має помилки
     s.draw();// викликається draw() із Shape
     s.erase();// викликається erase() з Shape
     s = new Circle();
     s.draw(); // викликається draw() із Circle
     s.erase(); // викликається erase() з Circle
     s = new Square();
     s. draw(); // викликається draw() із Square
     s.erase(); // викликається erase() з Square
     s = new Triangle();
     s. draw(); // викликається draw() із Triangle
     s.erase(); // викликається erase() з Triangle
  }
}
```

Порядок виконання лабораторної роботи

Завдання 1

Перетворити програму обробки відомості (див. лабораторну роботу 2) так, щоб у ній використовувалося відношення агрегації. При цьому

"цілим" повинен бути клас, що подає відомість із безліччю записів, а "частиною" - клас, що подає один запис відомості.

Завдання 2

Розробити програму, що використовує принцип поліморфізму на базі абстрактного класу відповідно до одного з варіантів завдань на лабораторну роботу (див. нижче).

Завдання 3

Перетворити програму із завдання 2 так, щоб замість абстрактного класу використовувався інтерфейс.

Вариант 1

Є відношення спадкування: птиця (Bird) – базова абстракція, орел (Eagle), та качка (Duck) – похідні класи.

Bird має методи Eat() (їсти) і Move() (рухатися). Кожний із похідних класів перевизначає методи Eat(), Move() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму для виведення текстових повідомлень щодо руху та прийому їжі орлом і качкою.

Вариант 2

Є відношення спадкування: геометрична фігура (Shape) – базова абстракція, квадрат (Square) та круг (Circle) – похідні класи.

Shape має метод GetArea(), який обчислює та повертає значення площі геометричної фігури. Клас Square має поле A – довжина сторони квадрата. Клас Circle має поле R – радіус окружності. Кожний з похідних класів перевизначає метод GetArea() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму при обчисленні площ квадрата ($S=A^2$) та круга ($S=\pi \cdot R^2$). Значення довжини сторони квадрата, радіуса кола ввести з консолі. Вивести значення площ квадрата та круга на консоль.

Вариант 3

Є відношення спадкування: геометрична фігура (Shape) – базова абстракція, паралелограм (Parallelogram) та рівнобедрена трапеція (Trapezium) – похідні класи.

Shape має метод GetArea(), який обчислює та повертає значення площі геометричної фігури. Клас Parallelogram має поля A – основа паралелограма та H – його висота. Клас Trapezium має поля C, D – основи трапеці, та H – висота трапеції. Кожний із похідних класів перевизначає метод GetArea() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму при обчисленні площ паралелограма (S=A·H) та трапеції (S=0,5·(C+D)·H). Значення основи та висоти паралелограма, основ та висоти трапеції ввести з консолі. Вивести значення площ паралелограма та трапеції на консоль.

Вариант 4

Є відношення спадкування: геометричне тіло (Solid) – базова абстракція, прямокутний паралелепіпед (RectSolid) та куб (Cube) – похідні класи.

Solid має метод GetVolume(), який обчислює та повертає значення об‘єму геометричного тіла. Клас Cube має поле A – довжина ребра куба. Клас RectSolid має поля C, D – довжина та ширина основи прямокутного паралелепіпеда, та H – його висота. Кожний із похідних класів перевизначає метод GetVolume() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму при обчисленні об‘ємів куба ($V=A^3$) та прямокутного паралелепіпеда (V=C·D·H). Значення довжини ребра куба, довжини й ширини основи прямокутного паралелепіпеда, його висоти ввести з консолі. Вивести значення об‘ємів куба та прямокутного паралелепіпеда на консоль.

Вариант 5

Є відношення спадкування: персона (Person) – базова абстракція, італієць (Italian) та українець (Ukranian) – похідні класи.

Person має методи PrintCountryName() (надрукувати назву держави) та Speak() (говорити деякою мовою). Кожний із похідних класів перевизначає методи PrintCountryName(), Speak() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму для виведення текстових повідомлень щодо назви держави та рідної мови італійця й українця.

Вариант 6

Є відношення спадкування: судно (Vessel) – базова абстракція, парусник (SailingVessel) та підводний човен (Submarine) – похідні класи.

Vessel має методи PrepareToMovement() (підготуватися до руху) та Move() (рухатися). Кожний із похідних класів перевизначає методи PrepareToMoving(), Move() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму для виведення текстових повідомлень щодо підготовки до руху та руху парусника й підводного човна.

Вариант 7

Є відношення спадкування: геометричне тіло (Solid) – базова абстракція, піраміда з квадратною основою (Pyramid) та куля (Sphere) – похідні класи.

Solid має метод GetVolume(), який обчислює та повертає значення об‘єму геометричного тіла. Клас Pyramid має поля A – довжина сторони основи піраміди та H – висота піраміди. Клас Sphere має поле R – радіус кулі. Кожний із похідних класів перевизначає метод GetVolume() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму при обчисленні об‘ємів піраміди ($V=(1/3)\cdot A^2\cdot H$) та кулі ($V=(4/3)\cdot\pi\cdot R^3$). Значення довжини сторони основи піраміди і її висоти та радіуса кулі ввести з консолі. Вивести значення об‘ємів піраміди та кулі на консоль.

Вариант 8

Є відношення спадкування: геометричне тіло (Solid) – базова абстракція, піраміда з квадратною основою (Pyramid) та конус (Cone) – похідні класи.

Solid має метод GetVolume(), який обчислює та повертає значення об‘єму геометричного тіла. Клас Pyramid має поля A, H – довжина сторони основи піраміди, та H – висота піраміди. Клас Cone має поле R – радіус основи конуса, та H – висота конуса. Кожний із похідних класів перевизначає метод GetVolume() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму при обчисленні об‘ємів піраміди ($V=(1/3)\cdot A^2\cdot H$) та конуса ($V=1/3\cdot\pi\cdot R^2\cdot H$). Значення довжини сторони основи піраміди та висоти

піраміди, радіуса основи конуса та його висоти ввести з консолі. Вивести значення об‘ємів піраміди й конуса на консоль.

Вариант 9

Є відношення спадкування: планета (Planet) – базова абстракція, Земля (Earh) та Місяць (Moon) – похідні класи.

Planet має методи ReportAboutMovement() (повідомити навколо якого небесного тіла рухається планета) та ReportAboutLife() (повідомити про наявність життя на планеті). Кожний із похідних класів перевизначає методи ReportAboutMovement(), ReportAboutLife() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму для виведення текстових повідомлень щодо до руху й наявності життя на Землі та Місяці.

Вариант 10

Є відношення спадкування: геометрична фігура (Shape) – базова абстракція, паралелограм (Parallelogram) та рівнобедрена трапеція (Trapezium) – похідні класи.

Shape має метод GetPerimeter(), який обчислює та повертає значення периметра геометричної фігури. Клас Parallelogram має поля A – основа паралелограма, H – його висота, та Alfa – кут між основою й боковою стороною паралелограма. Клас Trapezium має поля C, D – основи трапеції, H – висота трапеції. Кожний із похідних класів перевизначає метод GetPerimeter() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму при обчисленні периметрів паралелограма ($P=2\cdot X+2\cdot A$; $X=H/\sin(Alfa)$) та трапеції ($P=2\cdot X+D+C$; $X=[(0,5\cdot(C-D))^2 + H^2]^{0,5}$). Значення основи та висоти паралелограму, основ і висоти трапеції ввести з консолі. Вивести значення периметрів паралелограма та трапеції на консоль.

Вариант 11

Є відношення спадкування: геометричне тіло (Solid) – базова абстракція, прямокутний паралелепіпед (RectSolid) та куб (Cube) – похідні класи.

Solid має метод GetSurfaceArea(), який обчислює та повертає значення площі поверхні геометричного тіла. Клас Cube має поле A –

довжина ребра куба. Клас RectSolid має поля C, D – довжина та ширина основи прямокутного паралелепіпеду, та H – його висота. Кожний із похідних класів перевизначає метод GetSurfaceArea() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму при обчисленні площ поверхонь куба ($S_п=6\cdot A^2$) та прямокутного паралелепіпеда ($S_п=2\cdot C\cdot D+2\cdot D\cdot H+2\cdot C\cdot H$). Значення довжини ребра куба, довжини та ширини основи прямокутного паралелепіпеда, його висоти ввести з консолі. Вивести значення площ поверхонь куба та прямокутного паралелепіпеда на консоль.

Вариант 12

Є відношення спадкування: геометрична фігура (Shape) – базова абстракція, прямокутник (Rectangle) та прямокутний трикутник (Triangle) – похідні класи.

Shape має метод GetArea(), який обчислює та повертає значення площі геометричної фігури. Клас Rectangle має поле A – довжина прямокутника, та B – ширина прямокутника. Клас Triangle має поля C, D – катети прямокутного трикутника. Кожний із похідних класів перевизначає метод GetArea() базової абстракції.

Розробити консольну програму, яка використовує принцип поліморфізму при обчисленні площ прямокутника ($S=A\cdot B$) та прямокутного трикутника ($S=0,5\cdot C\cdot D$). Значення довжини та ширини прямокутника, катетів прямокутного трикутника ввести з консолі. Вивести значення площ прямокутника та прямокутного трикутника на консоль.

Контрольні питання

1. Принципи об’єктно-орієнтованого підходу.
2. Синтаксис опису класу.
3. Особливості статичних елементів класу.
4. Специфікатори доступу в C#.
5. Порядок ініціалізації об'єкта класу.
6. Агрегація й спадкування.
7. Синтаксис агрегації та спадкування в C#.
8. Порядок виклику конструкторів при спадкуванні.
9. Перевантаження "базового" методу.
10. Перевизначення "базового" методу.

11. Поліморфізм в C#.
12. Переваги концепції поліморфізму.
13. Абстрактні класи.
14. Абстрактні класи – правила.
15. Інтерфейси.
16. Інтерфейси – правила.

Лабораторна робота 4.
Перевантаження операцій у c#

Цілі лабораторної роботи

1. Придбання практичних навичок використання перевантаження операцій у мові C#.

2. Удосконалювання навичок роботи з інтегрованим середовищем розробки MS Visual Studio й довідковою системою MSDN.

Перед виконанням лабораторної роботи студент повинен знати:

1. Принципи перевантаження операцій.

Після виконання лабораторної роботи студент повинен вміти:

1. Самостійно використовувати механізм перевантаження операцій при розробці програм мовою C#.

2. Використовувати основні можливості MS Visual Studio при розробці консольних програм.

Перевантаження операцій у C#

Мова C# дозволяє визначити значення стандартних операцій відносно об‘єктів класу. Цей процес має назву перевантаження операцій.

Для перевантаження операторів використовується ключове слово operator, яке дозволяє створити метод, який визначає дію оператора, пов’язану з його класом.

Перевантаження операцій стало невід'ємною частиною C#. Наприклад, більшість класів у C# за умовчанням перенавантажують операцію порівняння (операція == практично завжди означає виклик методу Equals(), успадкованого від класу System.Object).

Перевантаження операцій зазвичай використовується для того, щоб скоротити й привести до звичного вигляду запис операцій над об'єктами, визначеними програмістом. Скажімо, з об'єктами, що подають математичні або фізичні величини, зазвичай асоціюються арифметичні операції. Візьмемо як приклад квадратні матриці: для них природно ввести операції складання і множення. Без перевантаження операторів ці дії довелося б записувати таким чином: A.Add(B) або Matrices.Add (A,B). І та, і інша форми запису декілька незвичні, оскільки традиційною формою є запис вигляду A+B.

Правила при перевантаженні операцій:

операція повинна бути визначена як відкритий статичний метод класу (специфікатори public static);

параметри в операцію повинні передаватися за значенням (тобто, при передачі параметрів не можна використовувати ключові слова ref або out);

сигнатури всіх операцій класу повинні бути різними;

типи, які використовуються в операції, повинні мати не менші права доступу, ніж сама операція (тобто повинні бути доступними при використанні операції).

Існує дві форми перевантаження операцій: одна використовується для унарних операцій, а друга – для бінарних. Метод operator для перевантаження унарних операцій повинен мати один параметр, а для перевантаження унарних операцій – два параметрі.

Приклад перевантаження унарних операцій "інкремент" та "декремент".

```
namespace Sample01
{
   class Point
   {
      int x, y;
      Point()
      {
         x = 0;
         y = 0;
      }
```

```
        Point(Point key)
        {
            x = key.x;
            y = key.y;
        }
        // Перевантаження операції інкремента
        public static Point operator ++(Point par)
        {
            par.x++;
            par.y++;
            return par;
        }
        // Перевантаження операції декременту
        public static Point operator --(Point par)
        {
            par.x--;
            par.y--;
            return par;
        }

        class Program
        {
            static void Main(string[ ] args)
            {
                Point p = new Point();
                p++;
                Console.WriteLine("x={0}   y={1}", p.x, p.y);
                ++p;
                Console.WriteLine("x={0}   y={1}", p.x, p.y);
                p--;
                Console.WriteLine("x={0}   y={1}", p.x, p.y);
                --p;
                Console.WriteLine("x={0}   y={1}", p.x, p.y);
            }
        }
    }
}
```

Приклад перевантаження бінарної операції "додавання":

```
public static Point operator +(Point par1, Point par2)
{
  return new Point(par1.x+par2.x,par1.y+par2.y);
}
```

Не всі операції можна перевантажувати, а деякі з них можуть перевантажуватися з певними обмеженнями (табл. 4):

Таблиця 4

Операція	Можливість перевантаження
+, -, !, ~, ++, --, true, false	Унарні операції, які дозволяють перевантаження
+, -, *, /, %, &, |, ^, <<, >>	Бінарні операції, які дозволяють перевантаження
==, !=, <, >, <=, >=	Операції порівняння перевантажуються парами
[ ]	Операції доступу до елементів масивів перевантажуються неявно за рахунок індексаторів
()	Операції перетворення типів перевантажуються з використанням ключових слів implicit і explicit
+=, -=, *=, /=, %=, &=, |=, ^=,<<=, >>=	Операції не перевантажуються, бо неможна перевантажувати операцію присвоювання
=, ., ?:, ->, new, is, sizeof, typeof	Операції, які не можна перевантажувати

Порядок виконання лабораторної роботи

Робота складається з двох етапів. У звіті повинні бути тексти програм із кожного з них.

Етап 1

Розробити клас – "вектор дійсних чисел" відповідно до наступних мінімальних вимог:

1. Наявність поля – масиву для збереження елементів "вектору".
2. Клас повинен мати поле – "ємність" вектора, що визначає кількість елементів.
3. Клас повинен містити конструктор із параметрами, метод для

додавання елемента в кінець вектора, метод для виведення елементів вектора на консоль, методи, що повертають значення полів.

4. Клас повинен використовувати перевантажену операцію для виконання обробки елементів вектора відповідно до індивідуального варіанта завдання.

Програма повинна створювати два об'єкти-вектори, уводити елементи векторів із консолі, демонструвати використання всіх методів вектора, виводити дані на консоль.

Варіанти виконання завдання

Перевантажена операція для виконання обробки елементів вектора, повинна визначати:

Варіант 1

Суму від’ємних елементів двох "векторів".

Варіант 2

Суму додатних елементів двох "векторів".

Варіант 3

Добуток елементів двох "векторів" із парними номерами.

Варіант 4

Суму елементів двох "векторів" із непарними номерами.

Варіант 5

Кількість від’ємних елементів двох "векторів".

Варіант 6

Кількість додатних елементів двох "векторів".

Варіант 7

Добуток від’ємних елементів двох "векторів".

Варіант 8

Добуток додатних елементів двох "векторів".

Варіант 9

Кількість елементів двох "векторів", що перебувають в інтервалі від A до B.

Варіант 10

Кількість елементів двох "векторів", рівних 0.

Варіант 11

Кількість елементів двох "векторів", більших C.

Варіант 12

Кількість елементів двох "векторів", менших за C.

Етап 2

Замінити в програмі (етап 1) методи, що повертають значення полів, на відповідні "властивості".

Контрольні питання

1. Коли доцільно використовувати перевантаження операцій?
2. Чим відрізняється перевантаження унарних і бінарних операцій?
3. Які операції не можуть бути перевантажені?
4. Чи змінюється сенс перевантаженої операції щодо стандартних типів, для яких вона визначена?
5. Як реалізувати перевантаження операції інкремента (декременту) в префіксному та постфіксному варіантах?

Лабораторна робота 5
Використання основних бібліотек платформи MICROSOFT .NET при розробці додатків.

Цілі лабораторної роботи:

1. Удосконалювання практичних навичок із розробки класів мовою C#.
2. Придбання практичних навичок використання класів виключень, уведення-виведення, контейнерів платформи Microsoft .Net.

3. Удосконалювання навичок роботи з інтегрованим середовищем розробки MS Visual Studio і довідковою системою MSDN.

Перед виконанням лабораторної роботи студент повинен знати:

1. Організацію системи уведення-виведення в Microsoft .Net.
2. Принципи обробки виключень у Microsoft .Net.
3. Структуру бібліотеки контейнерів платформи Microsoft .Net і принципи їхнього використання.

Після виконання лабораторної роботи студент повинен вміти:

1. Самостійно розробляти програми з використанням основних бібліотек платформи Microsoft .Net.

Загальні відомості про основні елементи бібліотеки класів платформи Microsoft .Net

Цей опис не містіть інформації щодо полів, властивостей і методів абстракцій бібліотеки класів платформи Microsoft .Net. ЇЇ можна отримати за допомогою утілити Object Browser системи програмування MS Visual Studio, довідкової системи MSDN або рекомендованої літератури.

Контейнери (колекції)

Під *колекцією розуміють* групу об’єктів. Простір імен System. Collections містить безліч інтерфейсів і класів, які визначають та реалізують колекції різних типів. Колекції спрощують програмування, пропонуючи вже готові рішення для побудови структур даних, розробка яких "з нуля" відрізняється великою трудомісткістю. Мова йде про вбудовані колекції, які підтримують, наприклад, функціонування стеків, черг і хеш-таблиць.

Основна перевага колекцій полягає в тому, що вони стандартизують спосіб обробки груп об'єктів у прикладних програмах. Усі колекції розроблені на основі набору чітко визначених інтерфейсів. Ряд вбудованих реалізацій таких інтерфейсів, як ArrayList, Hashtable, Stack і Queue, можна використовувати "як є". У кожного програміста також є можливість реалізувати власну колекцію, але в більшості випадків досить убудованих.

Платформа Microsoft .NET підтримує три основних типи колекцій: загального призначення, спеціалізовані й орієнтовані на побітову організацію даних. Колекції загального призначення реалізують ряд основних структур даних, включаючи динамічний масив, стек і чергу.

Сюди також відносяться словники, призначені для зберігання пар ключ-значення. Колекції загального призначення працюють із даними типу object, тому їх можна використовувати для зберігання даних будь-якого типу.

Колекції спеціального призначення орієнтовані на обробку даних конкретного типу або на обробку унікальним способом. Наприклад, існують спеціалізовані колекції, призначені тільки для обробки рядків або односпрямованого списку.

Класи колекцій, орієнтованих на побітову організацію даних, служать для зберігання груп бітів. Колекції цієї категорії підтримують такий набір операцій, що не характерний для колекцій інших типів. Наприклад, у біт-орієнтованої колекції BitArray визначені такі побітові операції, як "і" та "або".

Основним для всіх колекцій є реалізація *перечисельника (нумератора),* що підтримується інтерфейсами IEnumerator і IEnumerable.

Перечисельник забезпечує стандартизований спосіб поелементного доступу до вмісту колекції. Оскільки кожна колекція повинна реалізувати інтерфейс IEnumerable, до елементів будь-якого класу колекції можна одержати доступ за допомогою методів, визначених в інтерфейсі IEnumerator. Отже, після внесення невеликих змін, код, що дозволяє циклічно опитувати колекцію одного типу, можна успішно використовувати для циклічного опитування колекції іншого типу.

Простір імен System.Collections містить класи й інтерфейси, які визначають різні колекції об'єктів (табл. 5):

Таблиця 5

Інтерфейси	
Інтерфейс	Опис
1	2
ICollection	Визначає розмір, перечисельники й методи синхронізації для всіх колекцій
IComparer	Надає іншим додаткам метод для порівняння двох об'єктів
IDictionary	Визначає загальну поведінку колекцій пар " ключ-значення"
IDictionaryEnumerator	Здійснює нумерацію елементів словника
IEnumerable	Надає перечисельник, що підтримує просте переміщення по колекції
IEnumerator	Підтримує просте переміщення по колекції

Закінчення табл. 5

1	2
IList	Визначає загальну поведінку колекцій, до об‘єктів яких можна одержати доступ за індексом
Класи	
Клас	Опис
ArrayList	Реалізує структуру даних "динамічний масив"
Queue	Реалізує структуру даних "черга", що обслуговується за принципом "першим прийшов — першим вийшов"
Stack	Реалізує структуру даних "черга", що обслуговується за принципом "останнім прийшов — першим вийшов"
SortedList	Реалізує структуру даних "асоціативний масив". Надає колекцію пар " ключ-значення", які впорядковані по ключах. Доступ до пар можна одержати за ключем та за індексом
Hashtable	Реалізує структуру даних "асоціативний масив". Надає колекцію пар " ключ-значення", адресація яких визначається хеш-кодом ключа
BitArray	Управляє компактним масивом двоїстих значень, поданих логічними величинами, де значення true відповідає 1, а значення false відповідає 0

Клас ArrayList

Клас ArrayList призначений для підтримки динамічних масивів, які при необхідності можуть збільшуватися або скорочуватися. У C# стандартні масиви мають фіксовану довжину, що не може змінитися під час виконання програми. Це означає, що програміст повинен знати заздалегідь, скільки елементів буде зберігатися в масиві. Але іноді до виконання програми не можна точно сказати, масив якого розміру знадобиться. У таких випадках і використовується клас ArrayList. Об'єкт класу ArrayList є масивом змінної довжини, елементами якого є об'єктні посилання. Будь-який об'єкт класу ArrayList створюється з деяким початковим розміром. При перевищенні цього розміру колекція автоматично його збільшує. У випадку видалення об'єктів цей розмір автоматично скорочується.

Приклад використання класу ArrayList:

```
using System;
using System.Collections;
```

```
public class SamplesArrayList
   {
      public static void Main()
      {
         // Створюється й ініціалізується об'єкт класу ArrayList
         ArrayList myAL = new ArrayList();
         myAL.Add("Україна, ");
         myAL.Add("уперед");
         myAL.Add("!");
         // Відображення значень деяких властивостей ArrayList
         Console.WriteLine("myAL");
         Console.WriteLine("Count: {0}", myAL.Count);
         Console.WriteLine("Capacity: {0}", myAL.Capacity);
         Console.Write("Values: ");
         PrintValues(myAL);
         // PrintValues1(myAL);
      }

     // Відображення елементів ArrayList з використанням
     // перечисельника
      public static void PrintValues(IEnumerable myList)
      {
         // Для ефективної роботи з об'єктом ArrayList
         // створюється перечисельник myEnumerator, який
         // забезпечує перебір елементів
         IEnumerator myEnumerator = myList.GetEnumerator();
         while (myEnumerator.MoveNext())
            Console.Write("{0}", myEnumerator.Current);
      }

// Спрощений спосіб відображення елементів ArrayList
public static void PrintValues1(ArrayList myList)
{
  foreach (object element in myList)
    Console.Write("{0}", element);
}
```

```
   }
```
Результат роботи програми:

myAL
Count: 3
Capacity: 16
Values: Україна, уперед!

Клас Queue

Ще однією розповсюдженою структурою даних є черга. Додавання елементів у чергу й видалення їх з неї здійснюється за принципом "першим прийшов - першим обслужений" (first-in, first-out – FIFO). Інакше кажучи, перший елемент, поміщений у чергу, першим же з неї і вилучається. Наприклад, кожному доводилося стояти в черзі до квиткової каси в кінотеатрі або до каси в супермаркеті, щоб оплатити покупку. У програмуванні черги використовуються для організації таких механізмів, як виконання декількох процесів у системі й підтримка списку незакінчених транзакцій (у системах ведення баз даних) або пакетів даних, отриманих із Інтернету. Черги також часто використовуються в галузі імітаційного моделювання.

Клас колекції, призначений для підтримки черги, називається Queue. Черга – це динамічна колекція, що при необхідності збільшується, щоб прийняти для зберігання нові елементи, причому щораз коли така необхідність виникає, поточний розмір черги множиться на коефіцієнт зростання, що за замовчуванням дорівнює значенню 2,0.

Приклад використання класу Queue:

```
using System;
using System.Collections;

public class SamplesQueue
{
   public static void Main()
   {
      // Створюється й ініціалізується об'єкт класу Queue
      Queue my = new Queue();
      my.Enqueue("Україна, ");
      my.Enqueue("уперед");
      my.Enqueue("!");
```

```
      // Відображення значень деяких властивостей Queue
      Console.WriteLine( "my" );
      Console.WriteLine( "Count: {0}", my.Count );
      Console.Write( "Values: " );
      PrintValues(my);
      // PrintValues1(my);
   }

// Відображення елементів Queue з використанням
// перечисельника
public static void PrintValues(IEnumerable myList)
{
  IEnumerator myEnumerator = myList.GetEnumerator();
     while (myEnumerator.MoveNext())
       Console.Write("{0}", myEnumerator.Current);
}

// Спрощений спосіб відображення елементів Queue
public static void PrintValues1(Queue myList)
{
   for (int i = 0; i < myList.Count; i++)
     Console.Write("{0}", myList.Dequeue());
}
    }
```

Результат роботи програми:

```
my
Count: 3
Values: Україна, вперед!
```

Клас Stack

Стек – це список, додавання й видалення елементів до якого здійснюється за принципом "останнім прийшов – першим обслужений" (last-in, first-out – LIFO). Щоб зрозуміти, як працює стек, уявіть собі купу тарілок на столі. Тарілку, поставлену на стіл першою, можна буде взяти лише останньою, тобто коли будуть зняті все поставленні зверху тарілки. Стек – найбільш затребувана структура даних у програмуванні. Її часто

використовують у системному програмному забезпеченні, компіляторах і програмах з області створення штучного інтелекту.

Клас колекції, призначений для підтримки стека, називається Stack. Стек – це динамічна колекція, що при необхідності збільшується, щоб прийняти для зберігання нові елементи, причому щораз коли стік повинен розширитися, його ємність подвоюється. Клас Stack допускає в якості дійсного значення "порожнє посилання", а також допускає наявність повторюваних елементів.

Нижче наведений приклад створення й ініціалізації класу Stack і виведення його значень:

```
public class SamplesStack
{
  public static void Main()
  {
    // Створюється й ініціалізується об'єкт класу Stack
    Stack myStack = new Stack();
    myStack.Push("Україна,");
    myStack.Push("вперед ");
    myStack.Push("!");
    // Відображення значень деяких властивостей Stack
    Console.WriteLine("myStack");
    Console.WriteLine("Count: {0}", myStack.Count);
    Console.Write("Values:");
    // PrintValues(myStack);
    PrintValues1(myStack);
  }

  // Відображення елементів Stack із використанням
  // перечисельника
  public static void PrintValues(IEnumerable myCollection)
  {
    IEnumerator myEnumerator = myCollection.GetEnumerator();
   while (myEnumerator.MoveNext())
      Console.Write("{0}", myEnumerator.Current);
  }
```

```
  // Спрощений спосіб відображення елементів Stack
  public static void PrintValues1(Stack my)
  {
    int Counter = my.Count;
    for (int i = 0; i < Counter; i++)
      Console.Write("{0}", my.Pop());
  }
}
```

Результат роботи програми з використанням стека відрізняється від результатів роботи аналогічних програм з використанням класів ArrayList, Queue:

myStack
Count: 3
Values: !вперед Україна,

Це пов‘язано з тим що дисципліна обслуговування стека – LIFO, тобто метод Pop() забирає дані з "вершини" стека.

У бібліотеки класів платформи Microsoft .Net також є простір імен System.Collections.Generic, який містіть так звані "типізовані" контейнери. Вони можуть зберігати посилання не тільки на об‘єкти класу System.Object, а і на об‘єкти будь-якого іншого класу. Основні класи простору імен System.Collections.Generic та їх аналоги в просторі імен System.Collections наведені в табл. 6, де T – ім‘я класу:

Таблиця 6

Класи простору імен System.Collections.Generic	Аналог у просторі імен System.Collections
List <T>	ArrayList
Queue <T>	Queue
Stack <T>	Stack
SortedDictionary <K, V>	SortedList
Dictionary <K, V>	Hashtable
LinkedList <T>	-

Приклад використання "типізованого" контейнеру:

```
using System;
using System.Collections.Generic;

namespace TempConsoleApplication
{
   class Person
   {
      private string Name;

      public Person(string Name)
      { this.Name = Name; }

      public override string ToString()
      { return Name; }
   }

   public class MainClass
   {
      public static void Main()
      {
         // "Типізований" список посилань на об‘єкти класу Person
         List<Person> list = new List<Person>();
         list.Add(new Person("Василь"));
         list.Add(new Person("Галина"));
         list.Add(new Person("Петро"));
         PrintValues(list);
      }

      static void PrintValues(List<Person> myCollection)
      {
         foreach (Person element in myCollection)
            Console.WriteLine("Це - " + element);
      }
   }
}
```

Результат роботи програми:

Це - Василь

Це - Галина

Це – Петро

З іншими колекціями можна познайомитися за допомогою довідкової системи MSDN або рекомендованої літератури.

Обробка виключень

Проблеми "традиційного підходу" до обробки помилок під час виконання програми:

1. "Помилки можуть траплятися з іншими, але не в моєму коді".
2. "Перевірочний" код повинен перебувати всередині "корисного" коду.
3. Код із численними перевірками помилок легко стає нечитабельним.

Переваги обробки виключень у порівнянні з "традиційним підходом" до обробки помилок:

1. Відділення "корисного" коду від "перевірочного".
2. Групування типів помилок.
3. Можливість передачі помилки в інший контекст.

Послідовність обробки виключення подана на рис. 7:

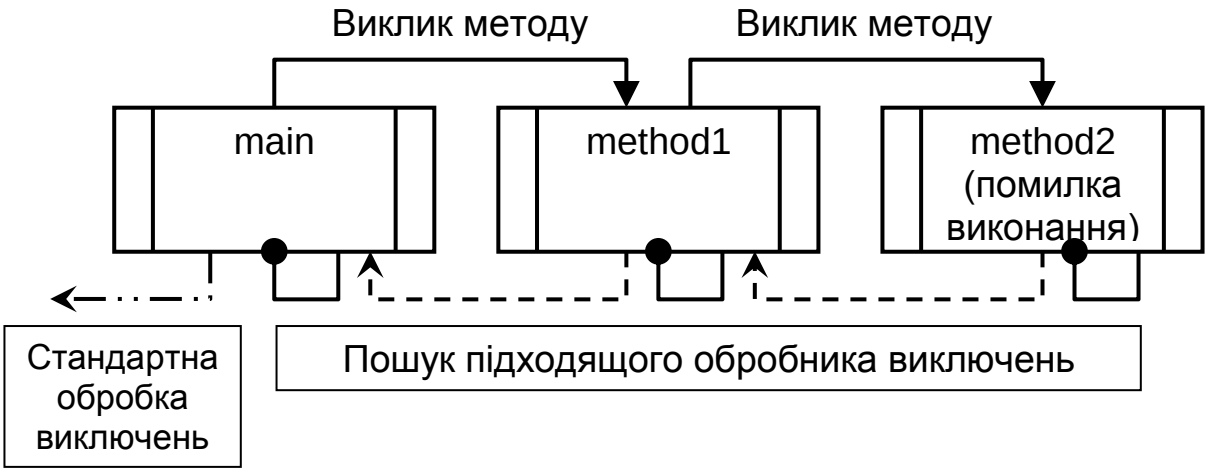

Рис. 7. **Механізм обробки виключень**

Синтаксис обробки виключень:

```
try {
    // блок, у якому може виникнути помилка
```

```
}
catch(ExceptionType1 ex) {
    //блок обробки виключення типу ExceptionType1
}
……..
catch(ExceptionTypeN ex) {
    // блок обробки виключення типу ExceptionTypeN
}
// необов'язковий блок
finally {
// оператори, які виконуються в будь-якому разі
}
```

Ієрархія класів системи обробки виключень бібліотеки класів платформи Microsoft .Net наведена на рис. 8.

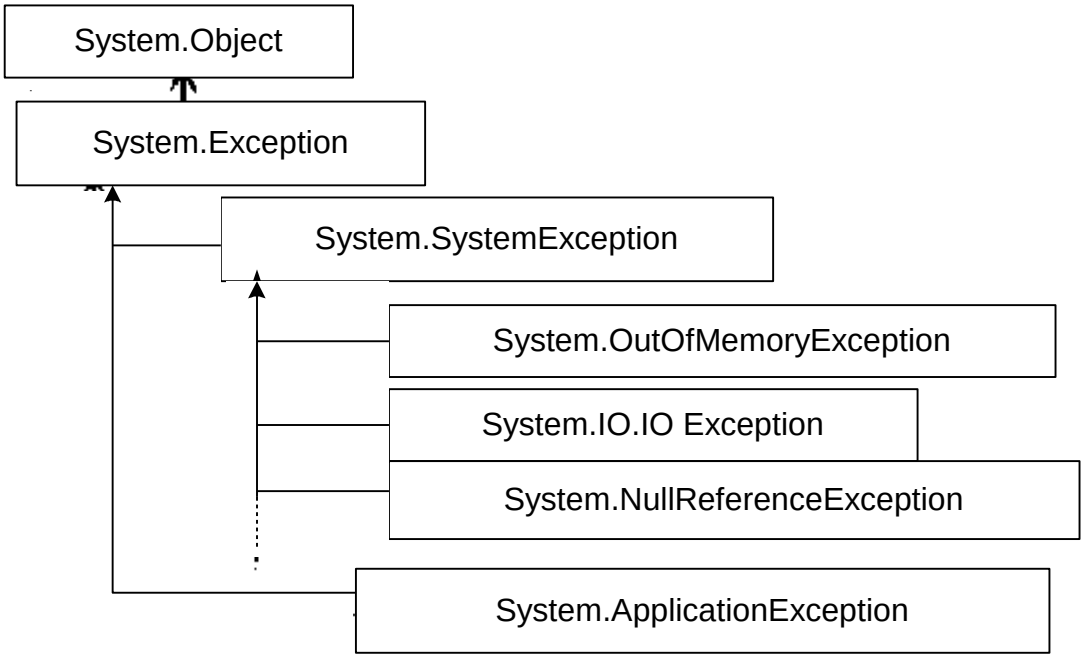

Рис. 8. **Класи системи обробки виключень Microsoft .Net**

Введення-виведення даних

Класи системи введення-виведення Microsoft .Net (простір імен System.IO) наведені на рис. 9:

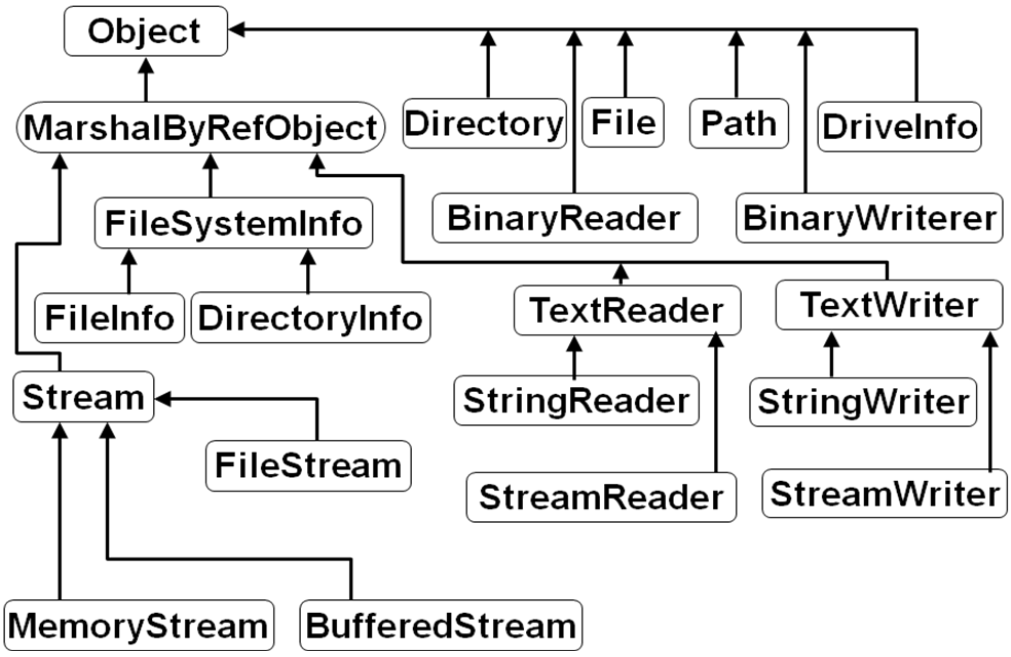

Рис. 9. **Основні класи системи введення-виведення Microsoft .Net**

Введення даних з файла за допомогою методу ReadAllLines() класу File:

```
try
{
  String[ ] Lines = File.ReadAllLines("d:\myfile.txt");
}
catch(FileNotFoundException e) { }
```

Виведення даних до файла за допомогою методу WriteAllLines() класу File:

```
try
{
  File.WriteAllLines("d:\myfile.txt");
}
catch(FileNotFoundException e) { }
```

Введення даних із файлу за допомогою символьного потоку введення (класу StreamReader):

```
StreamReader sw = new StreamReader(ChosenFile);
```

```
string line = null;
while ((line = sw.ReadLine()) != null)
{
   string[ ] tokens = line.Split(delimeter);
   CarList.Add(new Car(tokens[0], tokens[1], tokens[2]));
}
sw.Close();
```

Виведення даних до файла за допомогою символьного потоку виведення (класу StreamWriter):

```
StreamWriter sw = new StreamWriter(chosenFile, false);
foreach (string line in textBoxContents.Lines)
        sw.WriteLine(line);
sw.Close();
```

Порядок виконання лабораторної роботи

Розробити програму для обробки відомості, дані якої зберігаються в текстовому файлі.

При розробці даної програми за основу необхідно взяти відповідну програму з лабораторної роботи № 2.

Програма повинна:

1) містити клас, поля якого відповідають вихідним та розрахунковім полям відомості (для установки значень полів використовувати конструктор з параметрами, для обчислення значень розрахункових полів відомості використовувати нестатичні методи, для повернення значень полів використовувати властивості типу get);

2) містити клас, статичні методи якого призначені для виведення даних відомості до текстового файла і введення їх у програму з цього файла.

3) у головному класі (який містить метод Main()) забезпечувати:

а) уведення вихідних дані відомості й кількості її записів у поточному сеансі роботи з консолі;

б) запис вихідних і розрахункових даних відомості в "контейнер" об'єктів класу (див. пункт 1);

в) виведення даних відомості з "контейнера" у текстовий файл, ім'я якого вводиться з консолі;

г) уведення даних відомості з текстового файла в "контейнер" об'єктів класу (див. пункт 1);

д) виведення полів, що розраховуються, кожної із записів відомості, а також підсумкової інформації з відомості з "контейнера" на консоль.

4) Обробляти стандартні виключення введення-виведення, які можуть виникнути при виконанні програми.

Таблиця 7

Варіанти завдань

№ з/п	Символ-роздільник полів записів файла	Клас контейнера	Тип контейнера
1	;	Dictionary <K, V>	Типізований
2	:	List <T>	Типізований
3	|	Queue <T>	Типізований
4	/	Hashtable	Нетипізований
5	\	ArrayList	Нетипізований
6	$	Stack	Нетипізований
7	&	Queue	Нетипізований
8	!	SortedList	Нетипізований
9	#	Stack <T>	Типізований
10	%	SortedDictionary <K, V>	Типізований
11	+	LinkedList <T>	Типізований
12	пробіл	Dictionary <K, V>	Типізований

Контрольні питання

1. Проблеми "традиційного підходу" до обробки помилок під час виконання програми.
2. Переваги обробки виключень у порівнянні з "традиційним підходом" до обробки помилок.
3. Механізм обробки виключень.
4. Особливості байтових і символьних потоків вводу – виводу.

5. Базові класи байтових потоків вводу-виводу .Net та їх основні методи.

6. Базові класи символьних потоків і їх основні методи.

7. Призначення основних класів символьних потоків вводу-виводу .Net.

8. Призначення основних класів байтових потоків вводу-виводу .Net.

9. У чому різниця між буферизованими й небуферизованими потоками вводу-виводу .Net?

10. Склад бібліотеки контейнерів .Net.

11. Основні інтерфейси бібліотеки контейнерів .Net.

12. Коротка характеристика структури даних "динамічний масив" і її реалізації в.Net.

13. Коротка характеристика структури даних "двозв’язний список" та її реалізації в.Net.

14. Поняття про хеш-таблицю й хеш-функцію.

15. Коротка характеристика структури даних "асоціативний масив" і її реалізації в.Net.

16. Ітератори і їх використання.

Лабораторна робота 6.
Використання регулярних виразів при розробці додатків мовою c#

Цілі лабораторної роботи

1. Придбання практичних навичок використання регулярних виразів для обробки текстової інформації у програмі мовою C#.

2. Удосконалювання навичок роботи в середовищі MS Visual Studio.

Перед виконанням лабораторної роботи студент повинен знати:

1. Принципи створення й використання регулярних виразів.

2. Властивості й методи класу Regex і інших класів, пов'язаних із регулярними виразами.

Після виконання лабораторної роботи студент повинен вміти:

1. Самостійно використовувати механізм регулярних виразів при розробці програм розбору текстів і пошуку за зразком мовою C#.

Теоретичний матеріал

Регулярні вирази

Один із найменш відомих, але корисних класів у всій бібліотеці класів платформи Microsoft .NET - Regex, який знаходиться у просторі імен System.Text.RegularExpressions. Regex подає регулярні вирази, які є мовою для розбору й перетворення тексту. Regex підтримує три основних типи операцій:

розбивка рядку на підрядки з використанням регулярних виразів, що задають роздільники;

пошук підрядків за шаблонами, заданими за допомогою регулярних виразів;

виконання операцій пошуку й заміни із застосуванням регулярних виразів для вказівки тексту, що повинен бути замінений.

Одним із варіантів практичного застосування регулярних виразів є перевірка користувальницького введення.

При створенні об'єкта Regex його конструкторові передається регулярний вираз:

```
Regex regex = new Regex ("[ a-z]");
```

Мовою регулярних виразів "[a-z]" означає будь-яку малу літеру алфавіту. Можна також указати другий параметр, що задає опції Regex. Так, оператор

```
Regex regex = new Regex ("[ a-z]",  RegexOptions.IgnoreCase)
```

створює об'єкт для пошуку будь-якої букви алфавіту без урахування регістру. Якщо конструктору Regex передається неприпустимий регулярний вираз, він генерує виключення ArgumentException.

Після ініціалізації об'єкта Regex можно використовувати його методи для застосування відповідного регулярного виразу до рядків тексту. Нижче ми довідаємося, як використовується Regex у керованих додатках, і розглянемо приклади його використання.

Розбивка рядків

Метод Regex.Split розбиває рядок на складові частини, використовуючи регулярний вираз для пошуку роздільників. Наступний приклад розділяє повне ім'я (шлях до файла) на букву пристрою й імена каталогів:

```
Regex regex = new Regex (@"\\");
string[ ]  parts = regex.Split (@"c:\inetpub\wwwroot\wintellect");
foreach (string  part  in  parts)  Console.WriteLine (part);
```

У результаті одержимо:

```
c:
inetpub
wwwroot
wintellect
```

Зверніть увагу на подвійну зворотну схисну лінію, передану конструкторові Regex. Взагалі символ @ перед рядковим літералом, що має спеціальне призначення, дозволяє не використовувати подвоєння символу для вказівки компіляторові, що він буде застосовуватися як звичайний символ (див. параметр методу Split()). Але у даному випадку її однаково потрібно подвоювати, тому що вона є спеціальним символом також і в регулярних виразах.

Пошук у рядках

Імовірно, найбільше часто класс Regex використовується для пошуку підрядків, що відповідають заданому шаблону. Він надає три методи пошуку рядків і визначення наявності збігів: Match(), Matches() і IsMatch().

Найпростіший із них – IsMatch(). Він просто дає відповідь "так" або "ні" на питання, чи містить рядок тексту, що задовольняє регулярному виразу.

Іншим застосуванням IsMatch() є перевірка користувальницького введення. Наприклад, метод IsValid() повертає true, якщо переданий рядок містить 16 цифр, згрупованих у четвірки, розділені тире, і false – у противному випадку:

```
bool IsValid  (string  input)
{
  string Pattern = "^[0-9]{4}-[0-9]{4}-[0-9]{4}-[0-9]{4}$";
  Regex regex = new Regex(Pattern);
  return  regex.IsMatch (input);
}
```

Рядки, подібні "5678-8765-4321", проходять перевірку успішно, а рядки "*123456788765432" та "ABCD-8765-4321" – ні. Символи ^ і $ позначають початок і кінець рядку, відповідно. Подібні регулярні вирази часто застосовуються в програмах для попередньої перевірки введення номерів кредитних карт. При бажанні "[0-9]" у регулярному виразі можна замінити на "\d". Таким чином, вираз:

"^\d{4}-\d{4}-\d{4}-\d{4}$"

еквівалентний попередньому.

IsMatch() дозволяє визначити, чи містить рядок текст, що задовольняє регулярному виразу, але нічого не говорить про те, в якому місці перебуває цей текст і скільки збігів знайдено. Для цієї мети служить метод Match().

Трохи більше елегантний метод перебору всіх збігів із шаблоном пошуку пропонує метод Matches(). Він повертає набір об'єктів Match, вміст якого можна перебрати за допомогою циклу foreach. Кожний об'єкт Match, відповідає одному збігу. Об'єкт Match має властивість Groups, що дозволяє виділити підрядки всередині збігу.

Заміна рядків

Наділити програму можливістю пошуку й заміни дозволяє метод Regex.Replace(), що заміщає текст, що відповідає регулярному виразу, текстом, переданим як параметр. Наступний приклад заміняє в рядку input усі входження "Hello" на "Goodbye":

```
Regex regex = new Regex ("Hello");
string  output =  regex.Replace (input,   "Goodbye");
```

Інший приклад видаляє з рядка весь текст, укладений у кутові дужки, шляхом заміни виразів у кутових дужках на порожній рядок:

```
Regex regex = new Regex ("<[^>]*>");
string output =  regex. Replace (input,   "");
```

Базові знання про регулярні вирази можуть бути дуже корисні при розборі й перетворенні текстових рядків у додатках мовою C#.

Загальні відомості про синтаксичні аналізатори

Синтаксичним аналізатором зовнішнього подання поняття, заданого за допомогою синтаксичних правил, називається така

програма, що дозволяє ввести довільний текст і або друкує повідомлення про те, що цей текст побудований відповідно до правил зовнішнього подання цього поняття (коли це дійсно так), або повідомляє одну чи кілька причин, через які цей текст не є зовнішнім поданням відповідного поняття.

При описі понять будемо використовувати металінгвістичні формули (розширені форми Бекуса – Наура).

Метасимволи:

::= (означає "це є");

{ } (фігурні дужки використовуються для об'єднання декількох елементів);

{ }* (указує, що вкладений у фігурні дужки елемент формули може бути опущений або повторений довільне число раз);

{ }$^{+}$ (указує, що вкладений у фігурні дужки елемент формули може бути повторений один або довільне число раз);

| - означає "або".

Порядок виконання лабораторної роботи

Загальна постановка завдання

Розробити програму для синтаксичного аналізу вхідної послідовності даних із використанням пакета регулярних виразів платформи Microsoft .Net.

Варіанти індивідуальних завдань

1. Побудувати синтаксичний аналізатор для поняття "просте_логічне":

просте_логічне ::= TRUE | FALSE | простий_ідентифікатор | NOT просте_логічне

простий_ідентифікатор ::= буква

знак_операції ::= AND | OR

буква ::= *a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z*

2. Побудувати синтаксичний аналізатор для поняття "відношення":

відношення ::= { операнд знак_відношення операнд }

знак_відношення ::= $<|\leq|>|\geq|<|>|=$

операнд ::= доданок

доданок ::= константа | змінна
константа ::= цифра {цифра}*
змінна ::= буква {буква | цифра}*
цифра ::= 0|1|2|3|4|5|6|7|8|9
буква ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

3. Побудувати синтаксичний аналізатор для поняття "список_списків":

список_списків ::= список {; список}*
список ::= елемент {, елемент }*
елемент ::= буква
буква ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
4. Побудувати синтаксичний аналізатор для поняття "послідовність":

послідовність ::= {елемент роздільник}*
елемент ::= ціле_число
роздільник ::= пробіл+
ціле_число ::= { + | - } ціле_без_знака
ціле_без_знака ::= цифра+
цифра ::= 0|1|2|3|4|5|6|7|8|9

5. Побудувати синтаксичний аналізатор для поняття "присвоювання":

присвоювання ::= { ліва_частина=права_частина }
ліва_частина ::= змінна
права_частина ::= змінна | { змінна знак_операції змінна }
змінна ::= буква { буква | цифра }*
знак_операції ::= + | – | * | /
цифра ::= 0|1|2|3|4|5|6|7|8|9
буква ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

6. Побудувати синтаксичний аналізатор для поняття "список_параметрів":

список_параметров ::= параметр {, параметр }*
параметр ::= { ім'я = цифра+ }| { ім'я = (список_параметрів) }
ім'я ::= буква+

цифра ::= 0|1|2|3|4|5|6|7|8|9
буква ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

7. Побудувати синтаксичний аналізатор для поняття "скалярний_тип":

скалярний_тип ::= { ім'я_константи {, ім'я_константи}*} | { константа . . константа } | ім'я_скаляр_типу
ім'я_константи ::= буква {цифра | буква}*
ім'я_скаляр_типу ::= буква {цифра | буква}*
константа ::= цифра {цифра}* | ім'я_константи
цифра ::= 0|1|2|3|4|5|6|7|8|9
буква ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

8. Побудувати синтаксичний аналізатор для поняття "дійсне_число":

дійсне_число ::= { ціле_число. ціле_без_знака } | { ціле_число. ціле_без_знака E ціле_число } | { ціле_число E ціле_число }
ціле_без_знака ::= цифра {цифра}*
ціле_число ::= ціле_без_знака | { {+ | - } ціле_без_знака }
цифра ::= 0|1|2|3|4|5|6|7|8|9

9. Побудувати синтаксичний аналізатор для поняття "опис_міток":

опис_міток ::= LABEL мітка { , мітка }*;
мітка ::= ідентифікатор | послідовність_цифр
послідовність_цифр ::= 0 . . 9999
ідентифікатор ::= буква {буква | цифра}*
цифра ::= 0|1|2|3|4|5|6|7|8|9
буква ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

10. Побудувати синтаксичний аналізатор для поняття "добуток":

добуток ::= ціле {знак_операції ціле }*
ціле ::= цифра {цифра}*
знак_операції ::= * | /
цифра ::= 0|1|2|3|4|5|6|7|8|9

11. Побудувати синтаксичний аналізатор для поняття "текст":

текст ::= { пробел }$^{+}$ | { елемент текст }
елемент ::= буква | цифра | знак | (текст)
цифра ::= 0|1|2|3|4|5|6|7|8|9
буква ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
знак ::= + | – | ∗

Наприклад, $A+(45-\text{F(x)}*(B-C))$

12. Побудувати синтаксичний аналізатор для поняття "формула":

формула ::= цифра | (формула знак формула)
знак ::= + | – | ∗
цифра ::= 0|1|2|3|4|5|6|7|8|9

Контрольні питання

1. Коли доцільно використовувати регулярні вирази?
2. Який клас подає регулярні вирази?
3. Які основні операції підтримуються класом регулярних виразів?
4. Які основні параметри конструктора класу регулярних виразів?
5. Наведіть приклад використання методу Split().
6. Наведіть приклад використання методу Matches().
7. Наведіть приклад використання методу Match().
8. Наведіть приклад використання методу IsMatch().
9. Чим відрізняються методи Match(), IsMatch() і Matches()?
10. Наведіть приклад використання методу Replace().

Рекомендована література

1.Троелсен Э. C# и платформа.Net / Пер. с англ. – СПб.: Питер, 2007. – 796 с.

2.Троелсен Э. Язык программирования C# 2005 и платформа .Net 2.0 / Пер. с англ. – М.: Изд. дом "Вильямс", 2007. – 1168 с.

3.Шилдт Г. Полный справочник по C#. / Пер. с англ. – М.: Изд. дом "Вильямс", 2004. – 752 с.

4.Нейгел К. C# 2005 для профессионалов / К. Нейген, Б. Ивьен, Дж. Глинн; / [Пер. с англ. – М.: Изд. дом "Вильямс", 2006. – 1376 с.

Ресурси мережі internet

1. www.microsoft.com – сайт компанії Microsoft.
2. www.intuit.ru – Internet-інститут інформаційних технологій.
3. www.c-sharpcorner.com – матеріали з C#.
4. www.functionx.com – матеріали з C# та інших мов програмування.
5. www.rsdn.ru – матеріали з C# й інших мов програмування.

НАВЧАЛЬНЕ ВИДАННЯ

Методичні рекомендації
до виконання лабораторних робіт
з навчальної дисципліни

"ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ"

для студентів напряму підготовки
"Комп'ютерні науки"
всіх форм навчання

Частина 1

Укладачі: **Парфьонов Юрій Едуардович**
Федорченко Володимир Миколайович
Лосєв Михайло Юрійович
Щербаков Олександр Всеволодович

Відповідальний за випуск **Пономаренко В. С.**

Редактор **Гнатченко Г. О.**

Коректор **Мартовицька-Максимова В. А.**

План 2008 р. Поз. №213.

Підп. до друку Формат 60 × 90 1/16. Папір MultiCopy. Друк Riso.

Ум.-друк. арк. 4,5. Обл.-вид. арк. 5,63. Тираж прим. Зам. №

Видавець і виготівник — видавництво ХНЕУ, 61001, м. Харків, пр. Леніна, 9а

Свідоцтво про внесення до Державного реєстру суб'єктів видавничої справи ***Дк №481 від 13.06.2001 р.***

Методичні рекомендації
до виконання лабораторних робіт
з навчальної дисципліни

"ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ"

для студентів напряму підготовки "Комп'ютерні науки"
всіх форм навчання

Частина 1