Міністерство освіти і науки України
Національний технічний університет
«Харківський політехнічний інститут»

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ C#

Методичні вказівки до лабораторних робіт
з курсу «Об'єктно-орієнтоване програмування»

для студентів спеціальності
122 – Комп'ютерні науки
126 – Інформаційні системи та технології

Затверджено
редакційно-видавничою
радою університету,
протокол № 2 от 29.06.2021

Харків
НТУ «ХПІ»
2022

Об’єктно-орієнтоване програмування мовою C#. Методичні вказівки до лабораторних занять з курсу «Об’єктно-орієнтоване програмування» для студентів спеціальностей 122 – Комп’ютерні науки, 126 – Інформаційні системи та технології / уклад. Нікуліна О. М., Іванов Л. В., Коцюба Н. В. – Х. : НТУ «ХПІ», 2022. – 68 с.

Укладачі: О. М. Нікуліна
Л. В. Іванов
Н. В. Коцюба
Рецензент В. В. Москоленко

Кафедра інформаційні системи та технології

ВСТУП

Вказівки призначені для студентів, які вивчають мову C# на аудиторних заняттях або самостійно. Класи, узагальнення, успадкування, винятки, розглядаються на прикладах, супроводжуваних необхідними теоретичними відомостями.

Предметом навчальної дисципліни «Об'єктно-орієнтоване програмування» є методи алгоритмізації та програмування на мовах C++, C# та Java. Завдання дисципліни – освоєння методів та засобів об'єктно-орієнтованного програмування у візуальному середовищі Visual Studio та Eclipse.

Мета цих методичних вказівок – допомогти студентам засвоїти основи об'єктно-орієнтованого програмування мовою C#.

Лабораторні роботи передбачають вивчення розділів: побудова та об'ява класів, вкладені класи та композиція, перевантаження, успадкування, інтерфейси та абстрактні класи, винятки, використання контейнерів, узагальнення, написання програм з використанням графічного інтерфейсу користувача. Для виконання завдань студент повинен в достатньому ступені володіти процедурним програмуванням на мові C#. До кожної лабораторної роботи в методичних вказівках наведені: мета роботи, теоретичні основи та варіанти контрольних задач за розділами курсів, що вивчаються. Варіанти робіт відповідають вимогам навчального плану та навчальної програми і можуть бути застосовані для здійснення контролю знань студентів зі спеціальностями 122 – «Комп'ютерні науки» та 126 – «Інформаційні системи та технології».

ЛАБОРАТОРНА РОБОТА 1

КЛАСИ ТА ОБ'ЄКТИ

Мета лабораторної роботи – опанувати написання програм з використанням об'єктів.

1.1. Теоретичні основи

Як і в Java, весь код, що виконується, у C# знаходиться всередині методів класів або структур. Нижче наводиться приклад опису класу:

```
class Rectangle
{
  public double Width;
  public double Height;
  public double Area()
  {
    return Width * Height;
  }
}
```

Після останньої фігурної дужки не слід ставити крапку з комою. На відміну від C++, неабстрактні та нечасткові методи завжди реалізуються всередині визначення класу або структури.

Під час створення об'єкта класу поля ініціалізуються значеннями за умовчанням. C# допускає ініціалізацію полів початковими значенням:

```
class Rectangle
{
  public double Width = 10;
  public double Height = 20;
  public double Area()
  {
    return Width * Height;
  }
}
```

Сам клас може бути визначений у просторі імен чи в іншому класі. На відміну від Java, не існує спеціальних блоків ініціалізації усередині тіла класу. Необхідний код міститься в конструкторах.

Поля і методи можуть бути оголошені з ключовим словом **static**. Звертання до таких полів і методів може здійснюватися без створення

екземпляра класу. C# дозволяє звертання до таких елементів тільки через ім'я класу. Звертання до статичних елементів через ім'я об'єкта приводить до помилки компіляції. Як і в Java, статичні елементи даних не потрібно окремо визначати в глобальній області видимості. Статичні поля можуть бути проініціалізовані під час опису:

```
class NewClass
{
  public static double x = 10;
  public static int    i = 20;
}
```

Як елементи класу можуть виступати константи. Їх описують за допомогою модифікатора **const**. Такі константи створюються компілятором і не можуть бути ніде змінені. За замовчуванням такі константи є еквівалентними **static final** у Java.

Мова C# підтримує закритий (**private**), захищений (**protected**), відкритий (**public**), внутрішній (**internal**) і захищений внутрішній (**protected internal**) рівні доступу. Внутрішні рівні доступу визначають видимість у межах складання (assembly). Сам клас може бути оголошений як **public**. На відміну від C++, C# вимагає окремої специфікації доступу для кожного елемента (або групи елементів даних, описаних з одним специфікатором типу), при чому за умовчанням передбачається **private**:

```
public class TestClass
{
  private int i;
  double x; // private за умовчанням
  public void SetI(int value)
  {
    i = value;
  }
  public int GetI()
  {
    return i;
  }
  public void SetX(double value)
  {
    x = value;
  }
```

```
  public double GetX()
  {
    return x;
  }
}
```

Елементи **internal** доступні в тому ж складанні, що і їхній клас. Екземпляр класу створюється шляхом застосування операції **new** до конструктора:

```
TestClass tc = new TestClass();
```

У результаті об'єкт створюється в динамічній пам'яті. У даному прикладі `tc` – ім'я посилання на об'єкт. Змінні типу посилання на об'єкт класу можуть не посилатися ні на який екземпляр класу. У цьому випадку їм доцільно присвоїти значення порожнього посилання – **null.**

Екземпляр класу створюється шляхом застосування операції **new** до конструктора. Конструктор являє собою функцію, яка здійснює ініціалізацію об'єкта. Ім'я конструктора збігається з ім'ям класу. Не можна вказувати тип результату конструктора. У класі може бути визначено кілька конструкторів. Якщо жоден конструктор явно не визначений, автоматично створюється конструктор за замовчуванням (без параметрів). Такий конструктор ініціалізує усі поля початковими значеннями за умовчанням.

Ключове слово **this** використовується як посилання на об'єкт, для якого викликаний метод. Використання **this** багато в чому аналогічно Java. Статичні методи не можуть використовувати посилання **this.** Один конструктор можна викликати з іншого з використанням слова **this** і конструкції, аналогічної списку ініціалізації в C++.

Замість блоку статичної ініціалізації, який застосовано в Java, у C# використовуються *статичні конструктори*. Статичний конструктор повинен бути описаний за допомогою ключового слова **static**, для нього не можна визначати специфікатор доступу. Крім того, з тіла статичного конструктора не можна звертатися до нестатичних елементів класу. Статичний конструктор завжди викликається неявно – до створення екземпляра чи класу використання будь-якого статичного еле-

мента класу. Наведемо приклад статичного конструктора:

```
class Counter
{
  static int count;
  static Counter()
  {
    count = 0;
  }
}
```

У класі можуть бути одночасно статичний і нестатичний конструктори з однаковим набором параметрів.

Існує варіант нестатичних констант – поля з модифікатором **readonly**. Значення таких полів можуть задаватися тільки в конструкторі – це так звані константи екземпляра, що можуть бути різними для різних екземплярів того самого класу:

```
class NewClass
{
  readonly double x = 10;
  public NewClass(double initX)
  {
    x = initX;
  }
}
```

Елементи **readonly** можуть бути **static.** Їхні початкові значення задаються в статичних конструкторах. У C# є деструктори. Синтаксис їхнього опису цілком аналогічний C++.

```
class NewClass
{
  ~NewClass() // деструктор
  {

  }
}
```

У C# деструктор викликається збирачем сміття перед ліквідацією об'єкта. У деяких випадках об'єкт може бути не вилучений збирачем сміття ніколи, отже деструктор може бути ніколи не викликаний. Для деяких об'єктів можна створити функцію `Dispose()`, яка містить код завершення роботи з об'єктом, зокрема звільнення ресурсів. Для того, щоб цей метод був гарантовано викликаний, об'єкт можна створити у

конструкції **using**:

```
using (X x = new X())
{

} // об’єкт x знищено
```

Примітка: клас `X` повинен реалізовувати інтерфейс `IDisposable`. Інтерфейси будуть розглянуті у наступній темі.

За допомогою **using** в C# можна створювати синоніми для імен класів:

```
using AliasToClass = NewSpace.SomeClass;
  AliasToClass atc = new AliasToClass(); // всередині функції якогось класу
```

1.2. Задачі

У всіх завданнях, крім зазначених операцій, повинні бути реалізовані наступні методи: метод ініціалізації *Init*, введення з клавіатури *Read*, виведення на екран *Display*, перетворення в рядок *toString*. Всі поля закриті.

Задача 1.1

Створити клас *Vector3d*, що задається трійкою координат. Реалізувати методи: додавання і віднімання векторів, скалярний добуток векторів, множення на скаляр, порівняння векторів, обчислення довжини вектора, порівняння довжини вектора.

Задача 1.2

Створити клас *Money* для роботи з грошовими сумами. Число представлено двома полями: гривні і копійки. Реалізувати методи: додавання і віднімання, ділення сум, розподіл суми на дробове число, множення суми на дробове число, операції порівняння.

Задача 1.3

Створити клас *Triangle* для подання трикутника. Поля: сторони і кути. Реалізувати операції: отримання і зміни полів даних, обчислення площі, обчислення периметра, обчислення висот, визначення виду трикутника.

Задача 1.4

Створити клас *Angle* для роботи з кутами на площині. Поля: гра-

дуси і хвилини. Реалізувати методи: переклад в радіани, приведення до діапазону 0–360, збільшення і зменшення кута на задану величину, отримання синуса, порівняння кутів.

Задача 1.5

Створити клас *Data* для роботи з датами. Поля: рік, місяць, день. Реалізувати операції: віднімання, порівняння дат, визначення високосного року, переклад в кількість днів.

Задача 1.6

Створити клас *Time* для роботи з часом. Поля: години, хвилини, секунда. Реалізувати операції: віднімання, порівняння часу, переклад в хвилини і секунди.

Задача 1.7

Створити клас *Account*, який представляє собою банківський рахунок. Поля: прізвище власника, номер рахунку, відсоток нарахування, сума в гривнях. Реалізувати операції: зміна власника рахунку, зняття деякої суми з рахунку, покласти гроші на рахунок, нарахувати відсоток.

Задача 1.8

Створити клас *Fraction*, для роботи з дробовими числами. Поля: ціла частина і дрібна. Реалізувати операції: додавання, віднімання, множення, операції порівняння.

Задача 1.9

Створити клас *Goods* (Товар). Поля: найменування товару, дата оформлення, ціна товару, кількість одиниць товару, номер накладної. Реалізувати методи: зміни ціни товару, зміни кількості товару, обчислення вартості товару.

Задача 1.10

Створити клас *Payment* (Зарплата). Поля: ПІБ, оклад, рік надходження на роботу, прибутковий податок, відсоток надбавки, кількість відпрацьованих днів на місяць, нарахована і утримана суми. Реалізувати методи: обчислення нарахованої суми, обчислення утриманої суми, сума, що видається, обчислення стажу.

ЛАБОРАТОРНА РОБОТА 2

ПЕРЕВАНТАЖЕННЯ ОПЕРАЦІЙ

Мета лабораторної роботи – навчитися перевантажувати операції у класі.

2.1. Теоретичні основи

Під час проектування класу можна визначити набір операцій, які можна виконувати над об'єктами. До операцій, що перевантажуються, належать унарні операції +, –, !, ~, ++, --, **true**, **false** і бінарні +, –, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=. Якщо для об'єкта перевантажені операції **true** і **false**, посилання на нього може бути використане у якості аргументів умовної операції, **if** і циклів. Операції **true** і **false**, == і !=, > і <, >= і <= є парними: якщо в класі перевантажується одна з них, обов'язково повинна бути перевантажена й інша.

Операції, що перевантажується, відповідає відкритий статичний метод, назва якого складається з ключового слова **operator** і позначення операції.

Функції **explicit operator** Ім'я_типу() і **implicit operator** Ім'я_типу() використовуються для явних і неявних перетворень типів відповідно. Неявні перетворення в C# відповідають спеціальним операторам перетворення (випадок перетворення з класу в інший тип) і конструкторам з одним параметром (випадок перетворення з іншого типу в даний клас).

Відмінністю перевантаження операцій є те, що якщо перевантажена арифметична операція, то автоматично вважається перевантаженою відповідна операція складеного присвоювання.

2.2. Задачі

Виконати завдання лабораторної роботи 1 (свій варіант) з конструктором і перевантаженням операцій. Реалізувати завдання двома способами: всі операції визначити як зовнішні дружні функції і як методи класу.

Задача 2.1

У класі *Vector3d* перевантажити операції: додавання і віднімання векторів, порівняння векторів, порівняння довжини вектора.

Задача 2.2

У класі *Money* перевантажити операції: додавання і віднімання, операції порівняння.

Задача 2.3

У класі *Triangle* перевантажити операції: обчислення площі, обчислення периметра, обчислення висот.

Задача 2.4

У класі *Angle* перевантажити операції: переклад в радіани, збільшення і зменшення кута на задану величину, порівняння кутів.

Задача 2.5

У класі *Data* перевантажити операції: додавання, віднімання, порівняння дат.

Задача 2.6

У класі *Time* перевантажити операції: додавання, віднімання, порівняння часу.

Задача 2.7

У класі *Account* перевантажити операції: зняття деякої суми з рахунку, покласти гроші на рахунок.

Задача 2.8

У класі *Fraction* перевантажити операції: додавання, віднімання, множення, операції порівняння.

Задача 2.9

У класі *Goods* перевантажити операції: зміни ціни товару, обчислення вартості товару.

Задача 2.10

У класі *Payment* перевантажити операції: сума, що видається, обчислення стажу.

ЛАБОРАТОРНА РОБОТА 3

ВКЛАДЕНІ КЛАСИ. КОМПОЗИЦІЯ

Мета лабораторної роботи – опанувати написання програм з вкладеними класами.

3.1. Теоретичні основи

У мові C # будь-який клас у своїй реалізації може містити оголошення іншого класу. Клас, який оголошується в межах фігурних дужок іншого класу, називається вкладеним класом.

У найпростішому випадку, загальна форма оголошення вкладеного класу в класі має вигляд:

```
class Outer
{
  // поля та методи класу Outer
  // ...

  class Inner
  {
    // поля та методи класу Inner
    // ...
  }

  // поля та методи класу Outer
  // ...
}
```

Класи `Outer` і `Inner` можуть містити різні специфікатори, які визначають доступ (**`private`**, **`protected`**, **`public`**) або інші властивості класу (**`static`**, **`sealed`** і т.п.)

Окрім композиції класів, є також композиція об’єктів.

Відносини, при якому існує взаємозв’язок між двома об’єктами. Є два типи таких відносин.

Відносини типу has-a (клас містить інший клас, has-a relationship). В цьому випадку в класі оголошується один або кілька об’єктів іншого класу. Тут також існує поділ: агрегація, композиція. Якщо вкладені об’єкти можуть існувати незалежно від класу (не є складовою частиною класу), то це є агрегація (aggregation). Якщо вкладені об’єкти

(об’єкт) доповнюють клас таким чином, що існування класу неможливо без цих об’єктів, то це є композиція (composition) або об’єднання;

Відносини типу uses (клас «використовує інший клас). Це узагальнене ставлення при якому можливі різні форми використання одного класу іншим. Якщо в програмі оголошені два класи, то необов’язково один клас повинен містити примірник іншого класу. Клас може використовувати тільки певний метод іншого класу, клас може звертатися до імені іншого класу (використовувати ім’я), клас може використовувати поле даних іншого класу і т.п.

При відносинах типу has-a клас містить один або кілька об’єктів (екземплярів) іншого класу. Існує два різновиди відносини типу has-a:

– агрегація. Це випадок, коли один або кілька вкладених об’єктів не є частиною класу, тобто клас може існувати без цих об’єктів. Клас може містити будь-яку кількість таких об’єктів (навіть 0). Дивіться наведені нижче приклади;

– композиція. У цьому випадку один або кілька вкладених об’єктів є частиною класу, тобто без цих об’єктів неможливо логічне існування самого класу.

3.2. Задачі

У всіх завданнях необхідно реалізувати по два класи. Реалізувати завдання двома способами:

1) композиція класів – допоміжний клас повинен бути визначений як незалежний, об’єкти цього класу використовуються в якості полів основного;

2) вкладені класи.

Задача 3.1

Реалізувати клас *Vector3d* (задача 1.1), використовуючи клас Fraction (задача 1.8).

Задача 3.2

Реалізувати клас *Money* (задача 1.2), використовуючи клас Fraction (задача 1.8).

Задача 3.3

Реалізувати клас *Triangle* (задача 1.3), спираючись на клас *Angle* (задача 1.4) для подання кутів.

Задача 3.4

Реалізувати клас *Quadrangle*, спираючись на класи *Triangle* (задача 1.3) і Angle (задача 1.4) для подання кутів.

Задача 3.5

Реалізувати клас *Goods* (задача 1.9), додати поле-дату надходження товару на склад використовуючи клас Data (задача 1.5).

Задача 3.6

Реалізувати клас *Data* (задача 1.5), використовуючи клас *Time* (задача 1.6).

Задача 3.7

Реалізувати клас *Account* (задача 1.7), використовуючи суми клас Money (задача 1.2).

Задача 3.8

Реалізувати клас *Calculator* з повним набором арифметичних операцій, використовуючи клас (задача 1.8).

Задача 3.9

Реалізувати клас Goods (задача 1.9), використовуючи для ціни клас Money (задача 1.2).

Задача 3.10

Реалізувати клас Payment (задача 1.10), використовуючи для полів з датою клас Data (задача 1.5).

ЛАБОРАТОРНА РОБОТА 4
УСПАДКУВАННЯ

Мета лабораторної роботи – опанувати написання програм з об’єктами, які успадковуються.

4.1. Теоретичні основи

Успадкування – це процес створення похідних класів від базових. Об’єкти похідних класів неявно містять усі поля базового класу, вклю-

чаючи закриті, не зважаючи на те, що методи похідного класу не мають доступу до закритих елементів базового. Крім того, успадковуються всі відкриті властивості та методи. Елементи базового класу з модифікатором **protected** (захищені) доступні з похідних класів.

Як і Java, C# дозволяє тільки одиничне успадкування класів – у кожного класу може бути тільки один базовий клас. Успадкування завжди відкрите (похідний клас не може обмежити доступ до відкритих елементів базового класу). Успадкування має наступний синтаксис:

```
class DerivedClass : BaseClass
{
  // тіло класу
}
```

Усі класи безпосередньо чи опосередковано походять від класу `System.Object`.

Конструктори не успадковуються – для кожного класу ієрархії повинен бути реалізований свій власний конструктор. Перед виконанням конструктора похідного класу викликається конструктор базового класу і конструктори об'єктів, які створюються під час опису класу.

Ключове слово **base** використовують для доступу до елементів базового класу з похідного класу, зокрема: для виклику перекритого методу базового классу та для передачі параметрів конструктора базового класу. Наприклад:

```
class BaseClass
{
  int i, j;
  public BaseClass()
  {
    i = j = 0;
  }
  public BaseClass(int i, int j)
  {
    this.i = i;
    this.j = j;
  }
}
class DerivedClass : BaseClass
{
  int k;
```

```
  public DerivedClass() : base(0, 0)
  {
    k = 0;
  }
  public DerivedClass(int i, int j, int k) : base(i, j)
  {
    this.k = k;
  }
}
```

Класи можуть бути визначені з модифікатором **sealed.** Такі класи не можуть використовуватися в якості базових. Методи з модифікатором **sealed** не можуть бути перекриті.

Як і в інших мовах об'єктно-орієнтованого програмування, існує неявне приведення типу посилання на похідний клас до посилання на базовий, але не навпаки.

```
BaseClass b = new DerivedClass();
DerivedClass d = new DerivedClass();
b = d; // OK
d = b; // Помилка компіляції!
```

Для того щоб привести посилання на базовий клас до посилання на похідний клас, може бути використане явне перетворення. У випадку, якщо таке перетворення неможливе, генерується виняткова ситуація `System.InvalidCastException`. Працювати з винятковими ситуаціями не завжди зручно. Існує спеціальна форма приведення типів з перевіркою можливості такого приведення – операція **as**:

```
BaseClass b1 = new DerivedClass();
BaseClass b2 = new BaseClass();
DerivedClass d1 = b1 as DerivedClass;   // OK
DerivedClass d2 = b2 as DerivedClass; // null
```

Операція **is** повертає **true**, якщо перетворення можливе, і **false** у протилежному випадку:

```
BaseClass b1 = new DerivedClass();
BaseClass b2 = new BaseClass();
if (b1 is DerivedClass)
{
  DerivedClass d1 = (DerivedClass) b1; // OK
}
if (b2 is DerivedClass)
{
```

```
  DerivedClass d2 = (DerivedClass) b2; // не виконується
}
```

Конструкції **as** та **is** запозичені з Delphi Pascal.

4.2. Задачі

Реалізувати класи завдань лабораторної роботи 1 (свій варіант) з конструкторами, але без перевантаження операцій, в якості базових класів. Реалізувати класи-спадкоємці, в яких методи базових класів використовуються для реалізації перевантажених операцій.

Задача 4.1

У класі *Vector3d* перевантажити операції у класі-спадкоємці: додавання і віднімання векторів, порівняння векторів, порівняння довжини вектора.

Задача 4.2

У класі *Money* перевантажити операції у класі-спадкоємці: додавання і віднімання, операції порівняння.

Задача 4.3

У класі *Triangle* перевантажити операції у класі-спадкоємці: обчислення площі, обчислення периметра, обчислення висот.

Задача 4.4

У класі *Angle* перевантажити операції у класі-спадкоємці: переклад в радіани, збільшення і зменшення кута на задану величину, порівняння кутів.

Задача 4.5

У класі *Data* перевантажити операції у класі-спадкоємці: додавання, віднімання, порівняння дат.

Задача 4.6

У класі *Time* перевантажити операції у класі-спадкоємці: додавання, віднімання, порівняння часу.

Задача 4.7

У класі *Account* перевантажити операції у класі-спадкоємці: зняття деякої суми з рахунку, покласти гроші на рахунок.

Задача 4.8

У класі *Fraction* перевантажити операції у класі-спадкоємці: дода-

вання, віднімання, множення, операції порівняння.

Задача 4.9

У класі *Goods* перевантажити операції у класі-спадкоємці: зміни ціни товару, обчислення вартості товару.

Задача 4.10

У класі *Payment* перевантажити операції у класі-спадкоємці: сума, що видається, обчислення стажу.

ЛАБОРАТОРНА РОБОТА 5
ВІРТУАЛЬНІ ТА АБСТРАКТНІ КЛАСИ

Мета лабораторної роботи – навчитися будувати інтерфейси та абстрактні класи.

5.1. Теоретичні основи

Поліморфізм – це властивість класів, згідно з якою поведінка об'єктів класу може визначатися не на етапі компіляції, а на етапі виконання. Поняття поліморфізму та поліморфних класів загалом спільні в усіх мовах об'єктно-орієнтованого програмування, зокрема, в Java та C#.

Усі класи C# є поліморфними, оскільки вони походять від поліморфного класу `System.Object`. Як і в інших мовах об'єктно-орієнтованого програмування, поліморфізм у C# реалізований через механізм *віртуальних функцій*. Однак, на відміну від Java, необхідно явно вказувати модифікатор **`virtual`** у базовому класі і модифікатори **`override`** у похідних класах. Якщо ми хочемо перекрити віртуальну функцію, обірвавши ланцюжок поліморфізму, необхідно використовувати ключове слово **`new`**:

```
class Shape
{
  public virtual void Draw()
  {
     . . .
  }
}
```

```
class Circle : Shape
{
  public override void Draw()
  {
     . . .
  }
}

class Rectangle : Shape
{
  public new void Draw()
  {
     . . .
  }
}
```

Найчастіше виникає необхідність у перекритті віртуальних функцій класу `System.Object`. Наприклад, для того, щоб отримати представлення об'єкта у вигляді рядка, необхідно для класу визначити функцію `ToString()`, яка повертає необхідний рядок. Таке представлення можна вживати з будь-якою метою, наприклад, виводити усі дані про об'єкт за допомогою функції `Console.WriteLine()`:

```
class MyClass
{
  int k;
  double x;
  public MyClass(int k, double x)
  {
    this.k = k;
    this.x = x;
  }
  public override string ToString()
  {
    return k + " " + x;
  }
  static void Main(string[] args)
  {
    MyClass mc = new MyClass(1, 2.5);
    Console.WriteLine(mc);
  }
}
```

Можна також перекрити метод `Equals()`, який дозволяє та порів-

нювати об’єкти між собою.

Іноді класи створюються для представлення абстрактних концепцій, а не для створення екземплярів. Такі концепції можуть бути представлені *абстрактними класами*. У C# для цього використовується ключове слово **abstract** перед визначенням класу.

```
abstract class SomeConcept
{
  . . .
}
```

Абстрактні методи аналогічні чисто віртуальним функціям у C++. Абстрактний клас може містити абстрактні методи, такі, для яких не приводиться реалізація. Такі методи не мають тіла функції. Їхнє оголошення аналогічне оголошенню функцій-елементів у C++, але оголошенню повинне передувати ключове слово **abstract**. У такому випадку мається на увазі, що метод – віртуальний, але вказувати слово **virtual** не можна. Методи, що перевантажують абстрактний метод, у похідних класах повинні мати модифікатор **override**.

Від абстрактного класу не вимагають обов’язкову наявність абстрактних методів. Але кожен клас, у якому є хоч один абстрактний метод, чи хоча б один абстрактний метод базового класу не був визначений, повинен бути оголошений як абстрактний (з використанням ключового слова **abstract**).

Абстрактні класи можуть містити абстрактні властивості, для таких властивостей не задається код доступу, а тільки позначається необхідність завдання такого коду в нащадках.

У C# використовується поняття *інтерфейсів*. Інтерфейс може розглядатися як чисто абстрактний клас, що містить тільки абстрактні методи:

```
interface Int1 {
  void F();
  int G(int x);
}
```

Кожен клас може бути створений тільки від одного базового класу, але при цьому реалізовувати один чи кілька інтерфейсів. Клас, що

реалізує інтерфейс, повинен забезпечити реалізацію всіх методів, оголошених в інтерфейсі. У іншому разі такий клас буде абстрактним і має бути оголошений зі специфікатором **abstract**.

Інтерфейси, що реалізуються класом, перелічуються через кому там же, де вказується базовий клас. Базовий клас повинен завжди бути першим у цьому списку. Для того, щоб розрізняти базові класи від інтерфейсів, імена інтерфейсів рекомендується починати з великої літери I.

Методи, визначені в інтерфейсі, є за умовчанням абстрактними і відкритими. У класі, реалізує інтерфейс, такі методи повинні бути оголошені як **public**:

```
interface ISomeFuncs {
  void F();
  int G(int x);
}
class SomeClass : ISomeFuncs {
  public void F() {
  }
  public int G(int x)
  {
    return x;
  }
}
```

Клас може реалізувати кілька інтерфейсів:

```
interface IFirst
{
  void F();
  int G(int x);
}
interface ISecond
{
  void H(int z);
}
interface IDerived : IFirst, ISecond
{
```

```
}
class AnotherClass : IFirst, ISecond
{
  public void F()
  {

  }
  public int G(int x)
  {
    return x;
  }
  public void H(int z)
  {

  }
}
```

Елементами інтерфейсів можуть бути також властивості.

На відміну від Java, в C# існує додаткова можливість визначення явної реалізації інтерфейсів, що запобігає конфліктам імен у класах, які реалізують декілька інтерфейсів. У класі, який реалізує інтерфейс, ім'я функції, визначеної в інтерфейсі, складається з двох частин.

```
тип Ім'я_інтерфейсу.ім'я_методу()
{
  ...// реалізація
}
```

Для таких методів не вказується рівень доступу. Методи класу, які явно реалізують інтерфейс, можуть викликатися тільки через ім'я посилання на інтерфейс, а не на клас, який його реалізує.

5.2. Задачі

У наступних завданнях потрібно реалізувати абстрактний базовий клас. Функції визначаються в похідних класах. В базових класах повинні бути оголошені віртуальні функції введення / виводу, які реалізуються в похідних класах. Замінити абстрактний клас на інтерфейс.

Задача 5.1

Створити абстрактний базовий клас *Figure* з віртуальними мето-

дами обчислення площі та периметра. Створити похідні класи: *Rectangle* (прямокутник), *Trapezium* (трапеція) зі своїми функціями площі і периметра. Самостійно визначити, які поля необхідні, які з них можна задати в базовому класі, а які – в похідних.

Задача 5.2

Створити абстрактний базовий клас *Number* з віртуальними методами – арифметичними операціями. Створити похідні класи *Integer* (ціле) і *Real* (дійсне).

Задача 5.3

Створити абстрактний базовий клас *Body* (тіло) з віртуальними функціями обчислення площі поверхні і об'єму. Створити похідні класи: *Parallelepiped* (паралелепіпед) і *Ball* (куля) зі своїми функціями площі поверхні і об'єму.

Задача 5.4

Створити абстрактний клас *Currency* (валюта) для роботи з грошовими сумами. Визначити віртуальні функції перекладу в рублі і виведення на екран. Реалізувати похідні класи *Dollar* (долар) і *Euro* (євро) зі своїми функціями перекладу і виведення на екран.

Задача 5.5

Створити абстрактний базовий клас *Triangle* для подання трикутника з віртуальними функціями обчислення площі та периметра. Поля даних повинні включати дві сторони і кут між ними. Визначити класи-спадкоємці: прямокутний трикутник, рівнобедрений трикутник, рівносторонній трикутник зі своїми функціями обчислення площі та периметра.

Задача 5.6

Створити абстрактний базовий клас *Root* (корінь) з віртуальними методами обчислення коренів і виведення результату на екран. Визначити похідні класи *Linear* (лінійне рівняння) і *Square* (квадратне рівняння) з власними методами обчислення коренів і виведення на екран.

Задача 5.7

Створити абстрактний базовий клас *Function* (функція) з віртуальними методами обчислення значення функції в заданій точці і виве-

дення результату на екран. Визначити похідні класи *Ellipse* (еліпс), *Hyperbola* (гіпербола) з власними функціями обчислення у залежно від вхідного параметра.

Задача 5.8

Створити абстрактний базовий клас *Integer* (ціле) з віртуальними арифметичними операціями і функцією виведення на екран. Визначити похідні класи *Decimal* (десяткове) і *Binary* (двійкове), що реалізують власні арифметичні операції і функцію виведення на екран. Число представляється масивом, кожен елемент якого – цифра.

Задача 5.9

Створити абстрактний базовий клас *Series* (прогресія) з віртуальними функціями обчислення і-го елемента прогресії і суми прогресії. Визначити похідні класи: *Linear* (арифметична) і *Exponential* (геометрична).

Задача 5.10

Створити абстрактний базовий клас *Container* з віртуальними методами *sort*() і поелементної обробки контейнера foreach. Розробити похідні класи *Bubble* (бульбашка) і *Choice* (вибір). У першому класі сортування реалізується методом бульбашки, а поелементна обробка полягає в добуванні квадратного кореню. В другом класі сортування реалізується методом вибору, а поелементна обробка обчисленні логарифму.

ЛАБОРАТОРНА РОБОТА 6
ВИНЯТКИ

Мета лабораторної роботи – опанувати написання програм з використанням блоків з винятками.

6.1. Теоретичні основи

Використання механізму обробки виняткових ситуацій є дуже важливою складовою частиною практики програмування на всіх сучасних об’єктно-орієнтованих мовах. Об’єкти-винятки дозволяють

програмісту відокремити точки виникнення помилок часу виконання від коду, де ці помилки повинні оброблятися. Виняткова ситуація являє собою подію, що виникає під час виконання програми та порушує нормальне виконання інструкцій коду.

Для генерації виняткової ситуації використовується оператор **throw**. Після ключового слова **throw** повинен бути розташований об'єкт класу System.Exception чи класів, похідних від нього. Такі похідні класи відбивають специфіку конкретної програми.

```
class SpecificException : Exception
{
}
```

Клас System.Exception містить ряд властивостей, за допомогою яких можна одержати доступ до інформації про виняткову ситуацію, зокрема:

Message – текстовий опис помилки, що задається як параметр конструктора під час створення об'єкта-винятку;

Source – ім'я об'єкта чи застосунку, що згенерувало помилку;

StackTrace – послідовність викликів, що привели до виникнення помилки (аналогічно printStackTrace() у Java).

У більшості випадків об'єкт-виняток створюється в місці генерації виняткової ситуації за допомогою оператора **new**, однак іноді об'єкт-виняток створюється заздалегідь. Типове твердження **throw** може виглядати так:

```
void F()
{
  . . .
  if (/* помилка */)
    throw new SpecificException();
  . . .
}
```

У заголовку функції, на відміну від Java, не специфікуються типи винятків, які генеруються цією функцією. У наступному прикладі функція Reciprocal() генерує виняткову ситуацію у випадку ділення на нуль.

```
class DivisionByZero : Exception
{
}

class Test
{
  public double Reciprocal(double x)
  {
    if (x == 0)
      throw new DivisionByZero();
    return 1 / x;
  }
}
```

На відміну від C++, C# не допускає створення винятків примітивних типів. Дозволені тільки об'єкти класів, похідних від `Exception`.

Як і в Java, у блоці **`try`** розміщують код, що може генерувати виняткову ситуацію:

```
double x, y;
. . .
try
{
  y = Reciprocal(x);
}
```

Після блоку **`try`** повинен випливати один чи кілька оброблювачів (блоків **`catch`**). Кожен такий оброблювач відповідає визначеному типу винятку. Блок **`catch`** без дужок обробляє всі інші виняткові ситуації:

```
catch (DivisionByZero d)
{
  // обробка виняткової ситуації
}
catch (SpecificException)
{
  // обробка виняткової ситуації
}
catch
{
  // обробка виняткової ситуації
}
```

Як видно з приклада, на відміну від Java, у заголовку блоку **`catch`** можна опускати ідентифікатор об'єкта-винятку, якщо важливий тільки тип.

Класи винятків утворять ієрархію. При зіставленні типів винятків, оброблювач базового типу сприймає також винятки всіх створених від нього типів. Звідси випливає, що оброблювачі похідних типів варто розміщувати до оброблювачів базових типів.

У деяких випадках оброблювач виняткових ситуацій не може цілком обробити виняток і повинен передати його зовнішньому оброблювачу. Це можна зробити за допомогою оператора **`throw`**:

```
catch (SomeEx ex)
{
  // локальна обробка виняткової ситуації
  throw (ex);  // повторна генерація
}
```

Якщо в заголовку оброблювача не визначений ідентифікатор, то можна використовувати **`throw`** без виразу:

```
catch (Exception)
{
  // локальна обробка виняткової ситуації
  throw;
}
```

Після останнього блоку **`catch`** можна розмістити блок **`finally`**. Цей код завжди виконується незалежно від того, виникла виняткова ситуація чи ні.

```
try
{
  openFile();
  // інші дії
}
catch (FileError f)
{
  // обробка виняткової ситуації
}
catch (Exception ex)
```

```
{
  // обробка виняткової ситуації
}
finally {
  closeFile();
}
```

В .NET Framework визначені стандартні особливі ситуації – класи, що теж є нащадками `Exception`. Один із найчастіше виникаючих стандартних винятків – `System.NullReferenceException`, який генерується при спробі звертатися до елементів класу або структури через посилання, яке дорівнює **`null`**. Виняток `System.IndexOutOfRangeException` генерується, коли відбувається вихід за межі масиву.

Внутрішні виняткові ситуації .NET сигналізують про серйозну проблему під час виконання програми і можуть виникнути при виконанні будь-якого оператора. До них відносяться `ExecutionEngineException` (внутрішня помилка CLR), `StackOverflowException` (переповнення стеку), `OutOfMemoryException` (брак оперативної пам'яті). Зазвичай такі винятки не перехоплюються.

6.2. Задачі

Реалізувати обробку винятків в трьох варіантах:

– використовувати відповідні стандартні винятки;

– використовувати – винятки-спадкоємці від стандартних винятків;

– визначити власні – винятки.

Виконати завдання лабораторної роботи 1 (свій варіант).

Задача 6.1

Клас *Vector3d* з конструкторами, перевантаженням операцій і обробкою винятків.

Задача 6.2

Клас *Money* з конструкторами, перевантаженням операцій і обробкою винятків.

Задача 6.3

Клас *Triangle* з конструкторами, перевантаженням операцій і об-

робкою винятків.

Задача 6.4

Клас *Angle* з конструкторами, перевантаженням операцій і обробкою винятків.

Задача 6.5

Клас *Data* з конструкторами, перевантаженням операцій і обробкою винятків.

Задача 6.6

Клас *Time* з конструкторами, перевантаженням операцій і обробкою винятків.

Задача 6.7

Клас *Account* з конструкторами, перевантаженням операцій і обробкою винятків.

Задача 6.8

Клас *Fraction* з конструкторами, перевантаженням операцій і обробкою винятків.

Задача 6.9

Клас *Goods* з конструкторами, перевантаженням операцій і обробкою винятків.

Задача 6.10

Клас *Payment* з конструкторами, перевантаженням операцій і обробкою винятків.

ЛАБОРАТОРНА РОБОТА 7
УЗАГАЛЬНЕННЯ

Мета лабораторної роботи – навчитися створювати та використовувати узагальнення функцій та класів.

7.1. Теоретичні основи

Узагальнене програмування (generic programming) це парадигма програмування, яка полягає у відокремленні даних, структур даних та алгоритмів обробки даних. *Узагальнені структури даних* дозволяють

зберігати колекції даних різних типів, при чому механізми зберігання не залежать від типів даних об'єктів. *Узагальнені алгоритми*, в свою чергу, повинні бути побудовані так, щоб вони не залежали ні від типів даних, ні від конкретних типів структур, які ці дані зберігають. Узагальнене програмування фокусується на пошуку спільності серед подібних реалізацій контейнерних класів та алгоритмів та створенні типів та функцій більш високого рівня абстракції.

Мови, які не підтримують узагальнень, розв'язують цю проблему шляхом використання вказівників на будь-яку змінну (у C), або посилань на спільний базовий клас (як у Delphi Pascal, а також у перших версіях Java та C#). Такий підхід не є зручним та безпечним з точки зору типів. Наприклад, якщо у першій версії C# необхідно було створити клас для зберігання пари об'єктів одного типу, треба було б створити два посилання на клас **object**:

```
public class Pair
{
  object First, Second;
  public Pair(object first, object second)
  {
    First = first;
    Second = second;
  }

}
```

Оскільки клас **object** є базовим для усіх типів, новий клас можна, наприклад, застосувати для зберігання пари рядків:

```
Pair p = new Pair("Прізвище", "Ім\'я");
```

Такий підхід має певні недоліки:

– Для читання об'єктів необхідно застосувати явне перетворення типів:

```
string s = (string) p.First; // Замість string s = p.first;
```

– Немає впевненості, що у парі зберігаються об'єкти саме того типу, який нас цікавить:

```
int i = (int) p.Second; // Помилка часу виконання
```

– Не можна гарантувати, що обидва поля будуть одного типу:

```
Pair p1 = new Pair("Прізвище", 2); // Жодного повідомлення про
помилку
```

Крім того, зберігання типів-значень у такому підході пов'язана з автоматичним упакуванням, що не є досить ефективним. Можна, звичайно, створити декілька класів для зберігання пари даних, наприклад `IntegerPair`, `FloatPair`, `StringPair` тощо. Такий підхід – громіздкий, та незручний. Він також ускладнює пошук та виправлення помилок.

Інша проблема – створення функцій для роботи з масивами різних типів. Багато типових алгоритмів (наприклад, обміняти два певних елемента місцями, змінити порядок елементів на протилежний) не залежать від типів елементів масиву. Як і у випадку створення класу, можна використовувати object як тип елементів. Наприклад:

```
public static void SwapElements(object[] arr, int i, int j)
{
  object e = arr[i];
  arr[i] = arr[j];
  arr[j] = e;
}
```

Знову ті ж самі проблеми: відсутність можливості контролю типів під час компіляції, "безглузде" перетворення типів, тощо. Створення окремих функцій для різних типів – громіздке, пов'язане з помилками, для всіх типів – взагалі неможливе.

Починаючи з версії 2, мова C# дозволяє створювати та використовувати *узагальнення* – синтаксичну конструкцію, що включає в себе параметри класу, структури або функції, які містять додаткову інформацію про типи елементів та інших даних. Ці параметри беруть у кутові дужки та розділяють комами. В найчастішому випадку список містить один параметр.

Узагальнення надають можливість створення та використання типів, безпечних з точки зору типів. Типи, опис яких містить такі параметри, мають назву узагальнених. Під час створення об'єкта узагальненого типу у кутових дужках вказують імена реальних типів. На відміну від Java, де в якості реальних типів можна використовувати тільки типи-посилання, C# дозволяє використовувати будь-які типи.

Узагальнені класи не можуть містити функцію `Main()`.

Якщо ми намагаємось додати до пари дані різних типів, компілятор згенерує помилку. Помилковою є також спроба явно перетворити

тип:

```
Pair<string> p = new Pair<string>("1", "2");
int i = (int) p.Second; // Помилка компіляції
```

Тип даних з параметром у кутових дужках (наприклад, `Pair<string>`) має назву *параметризованого типу*.

Окрім узагальнених класів і структур, можна створювати узагальнені функції всередині як узагальнених, так і звичайних (неузагальнених) класів (структур):

```
public class ArrayPrinter
{
  public static void PrintArray<T>(T[] a)
  {
    foreach (T x in a)
      Console.Write("{0, 10}", x);
    Console.WriteLine();
  }

  static void Main(string[] args)
  {
    string[] sa = {"First", "Second", "Third"};
    PrintArray(sa);
    int[] ia = {1, 2, 4, 8};
    PrintArray(ia);
  }
}
```

Рекомендованими іменами формальних параметрів є імена, які починаються з великої літери `T`.

Узагальнення може мати два і більше параметрів. У наступному прикладі пара може містити об’єкти різних типів:

За замовчуванням узагальнений тип вважають безпосередньо або опосереднено похідним від класу **`object`**. Іноді такого припущення замало для створення корисних класів та функцій. Наприклад, ми не можемо викликати жодного методу, окрім методів, описаних у класі **`object`**, навіть не можемо створити об’єкт за допомогою операції **`new`**, оскільки не впевнені, що тип надає конструктор без параметрів. Для того, щоб розширити функціональність узагальнених класів та методів, для опису узагальнень використовують так звані обмеження. Після узагальнення розміщують ключове слово **`where`**, а далі можна вказува-

ти про наявність конструктора без параметрів, базовий клас, інтерфейси, які реалізує тип. Наприклад:

```
class A<T> where T : new() // тип T повинен мати конструктор
без параметрів
  {
    public T Data = new T();
  }

  interface Int1
  {
    void doSomething();
  }

  class B<T> where T : Int1 // тип T повинен реалізовувати
вказаний інтерфейс
  {
    T data;
    public void f()
    {
      data.doSomething();
    }
  }

  class C<T> where T : class // тип T повинен бути типом-
посиланням
  {
  }

  class D<T> where T : struct // тип T повинен бути типом-
значенням
  {
  }
```

Від узагальнених класів шляхом спадкування можна створювати як узагальнені, так і неузагальнені класи. Якщо створюється неузагальнений клас, необхідно вказати конкретний тип для параметра-узагальнення. Наприклад:

```
class X<T>
  {
  }

  class Y<T> : X<T>
  {
```

```
}

class Z : X<int>
{
}
```

Можна також визначати узагальнені перевантажені операції.

Також можна серіалізувати об'єкти узагальнених класів. У такому випадку для імен тегів (атрибутів) серіалізатор автоматично обирає `Ім'яВластивостіOfІм'яТипу`. Наприклад, для властивості з ім'ям `SomeProperty` класу `SomeClass<T>`, якщо у створеному об'єкті тип `T` спеціалізовано як **`string`**, отримаємо таке ім'я тегу (атрибуту): `SomePropertyOfString`.

7.2. Задачі

Виконати завдання лабораторної роботи 7 (свій варіант) за наступними умовами.

Задача 7.1

Реалізувати стек у вигляді узагальнень однозв'язного списку.

Задача 7.2

Реалізувати чергу у вигляді узагальнень двозв'язного списку.

Задача 7.3

Реалізувати дек у вигляді узагальнень двозв'язного списку.

Задача 7.4

Реалізувати чергу у вигляді узагальнень однозв'язного списку.

Задача 7.5

Реалізувати стек у вигляді узагальнень однозв'язного списку.

Задача 7.6

Реалізувати дек у вигляді узагальнень двозв'язного списку.

Задача 7.7

Реалізувати чергу у вигляді узагальнень однозв'язного списку.

Задача 7.8

Реалізувати чергу у вигляді узагальнень двозв'язного списку.

Задача 7.9

Реалізувати дек у вигляді узагальнень двозв'язного списку.

Задача 7.10

Реалізувати чергу у вигляді узагальнень двозв'язного списку.

ЛАБОРАТОРНА РОБОТА 8
КОНТЕЙНЕРИ

Мета лабораторної роботи – опанувати написання програм з використанням контейнерів.

8.1. Теоретичні основи

Контейнерний клас – це клас, об'єкти якого надають можливість зберігання інших об'єктів або посилань. Найпростіший контейнер – це звичайний масив. Функціональність масивів не є достатньою для багатьох задач, тому сучасні мови і платформи програмування надають різноманітні можливості для створення контейнерів.

C# надає варіанти створення контейнерів як із застосуванням узагальнень (простір імен `System.Collections.Generic`), так і без них (простір імен `System.Collections`). Варіант без узагальнень вважається застарілим і небажаним. Далі розглядатимуться саме узагальнені контейнери. Простір імен `System.Collections.Generic` автоматично підключається до всіх автоматично згенерованих програм.

Усі стандартні контейнери реалізують інтерфейс `IEnumerable`. У наведеній нижче таблиці наведені деякі узагальнені інтерфейси та стандартні узагальнені класи, які реалізують ці інтерфейси:

Усі класи надають велику кількість статичних і нестатичних функцій для роботи з послідовностями елементів. Контейнери дозволяють зберігати як посилання, так і безпосередньо елементи типу **`int`**, **`double`**, тощо.

Клас `Stack` дозволяє створювати структуру даних, організовану за принципом "останній зайшов – перший вийшов" ("last in first out", LIFO). Клас `Queue` надає структуру даних, організовану за принципом "перший зайшов – перший вийшов" ("first in first out", FIFO).

Інтерфейс `IList` описує упорядковану колекцію (послідовність).

Узагальнений клас `List`, який реалізує інтерфейс `IList`, представляє список, створений за допомогою масиву (аналогічний `ArrayList` у Java). Як і в масивах, доступ до елементів може здійснюватися за індексом (через операцію `[]`). На відміну від масивів, розмір списків може динамічно змінюватися. Властивість `Count` повертає число елементів, які містяться в списку. Як і елементи масивів, елементи списків пронумеровані з нуля.

Створити порожній список об'єктів деякого типу (`SomeType`) можна за допомогою конструктора за замовчуванням:

```
IList<SomeType> list1 = new List<SomeType>();
```

Можна також одразу описати посилання на `List`:

```
List<SomeType> list1 = new List<SomeType>();
```

Другий варіант є менш бажаним, оскільки в такому випадку знижується гнучкість програми. Перший варіант дозволить легко замінити реалізацію списку `List` на будь-яку іншу реалізацію інтерфейсу `IList`, яка більше відповідає вимогам конкретної задачі. У другому випадку є спокуса викликати методи, специфічні для `List`, тому перехід на іншу реалізацію буде ускладнено. Для локальних змінних можна вживати опис за допомогою ключового слова **`var`**. Наприклад, останній опис міг би бути таким:

```
var list1 = new List<SomeType>();
```

За допомогою спеціального конструктору можна створити список з існуючого масиву:

```
int[] a = { 1, 2, 3 };
IList<int> list2 = new List<int>(a);
```

Можна створити новий список із використанням існуючого. Новий список містить копії елементів. Наприклад:

```
var list3 = new List<int>(list2);
```

Списки, як і масиви, можна створювати з використанням ініціалізаторів:

```
IList<int> list4 = new List<int> { 11, 12, 13 };
```

Створивши список, у нього можна додавати елементи за допомогою функції `Add()`. Метод `Add()` з одним аргументом додає елемент у кінець списку:

```
IList<string> list5 = new List<string>();
```

```
list5.Add("abc");
list5.Add("def");
list5.Add("xyz");
```

До списку можна додати всі елементи іншого списку (або іншої колекції) за допомогою функції `Concat()`. Результатом буде нова колекція (об'єкт типу `IEnumerable<>`):

```
var result = list3.Concat(list4); // 1 2 3 11 12 13
```

На відміну від Java, у списках зберігати у списках елементи типу **`int, double`** тощо. Як і масиви, списки можна обходити за допомогою конструкції **`foreach`**:

```
foreach (int i in list3)
  Console.WriteLine(i);
```

У тих випадках, коли частіше, ніж вибір довільного елемента, застосовують операції додавання і видалення елементів у довільних місцях, доцільно використовувати клас `LinkedList<>`, що зберігає об'єкти за допомогою зв'язаного списку.

Перед тим, як створювати свої власні узагальнені класи та методи, слід поцікавитися, чи є відповідні стандартні засоби. Для роботи з масивами існує стандартний клас `System.Array`, який надає велику кількість статичних узагальнених функцій для роботи з масивами, наприклад, таких, як `Resize()` (зміна розміру зі зберіганням існуючих елементів), `Reverse()` (зміна порядку елементів на протилежний), `Copy()` (копіювання одного масиву в інший, або в той самий з іншої позиції), `Sort()` (сортування) та багато інших. Роботу цих функцій можна продемонструвати на наступному прикладі:

```
using System;
namespace Arrays
{

  class Program
  {
    public static void Print<TElem>(IList<TElem> arr)
    {
      foreach (TElem elem in arr)
        System.Console.Write("{0, 6}", elem);
```

```
      System.Console.WriteLine();
    }
    static void Main(string[] args)
    {
      int[] a = { 1, 2, 3, 4 };
      Array.Resize(ref a, 5);
      Print(a);                // 1     2     3     4     0
      Array.Reverse(a);
      Print(a);                // 0     4     3     2     1
      Console.WriteLine(Array.IndexOf(a, 4)); // 1
      Array.Sort(a);           // 0     1     2     3     4
      Print(a);
      int[] b = new int[2];
      Array.Copy(a, b, b.Length);
      Print(b);                // 0     1
      Array.Copy(a, 2, b, 0, b.Length);
      Print(b);                // 2     3
    }
  }
}
```

З наведеного приклада видно, що клас `System.Array` реалізує інтерфейс `IList`. Тому його екземпляр може бути використаний як фактичний параметр функції з параметром типу `IList`.

У протилежність масивам, відповідні методи для списків – нестатичні:

```
static void Main(string[] args)
    {
      List<int> a = new List<int> { 1, 2, 3, 4 };
      a.Add(0);
      Print(a);         // 1     2     3     4     0
      a.Reverse();
      Print(a);         // 0     4     3     2     1
      Console.WriteLine(a.IndexOf(4)); // 1
      a.Sort();
      Print(a);           // 0     1     2     3     4
      int[] b = new int[5];
      a.CopyTo(b, 0); // копіювання у масив
      Print(b);       // 0     1     2     3     4
```

```
}
```

Для масивів і списків числових значень сортування здійснюється за збільшенням. Для класів і структур упорядкування визначається функцією `CompareTo()` інтерфейсу `IComparable`. Цей метод повинен повернути від'ємне значення (наприклад, -1), якщо об'єкт, для якого викликаний метод, менше об'єкта о, нульове значення, якщо об'єкти рівні, і додатне значення в протилежному випадку. Для того, щоб масив або список можна було відсортувати за замовчуванням, елементи масиву (списку) повинні бути об'єктами, для яких реалізовано інтерфейс `IComparable`. Для типів, які не реалізують інтерфейс `IComparable`, спроба здійснити сортування за замовчуванням призводить до генерації винятку `InvalidOperationException`.

Якщо ми не хочемо (чи не можемо) визначити функцію `CompareTo()`, можна створити окремий клас, що реалізує інтерфейс `IComparer`. Посилання на об'єкт такого класу передаються в якості в другого параметру функції `Sort()` для масивів (або першого параметру для списків). Сортування списків здійснюється аналогічно.

Асоціативні масиви можуть зберігати пари об'єктів (посилань на об'єкти). Асоціативні масиви теж є узагальненими типами. Асоціативні масиви у C# представлені узагальненим інтерфейсом `IDictionary`, який реалізовано, класами `Dictionary` і `SortedDictionary`. Останній клас вимагає впорядкованого за ключем зберігання пар. Ключі, на відміну від значень, не можуть повторюватися.

Кожне значення (об'єкт), яке зберігається в асоціативному масиві, пов'язується з певним значенням іншого об'єкту (ключа). Метод `Add(key, value)` додає значення (`value`) та асоціює з ним ключ (`key`). Якщо асоціативний масив раніше містив пари з зазначеним ключем, нове значення заміняє старе. Для перевірки знаходження ключа та значення застосовуються методи `ContainsKey()` та `ContainsValue()`. Обійти асоціативний масив можна за допомогою ітерації по елементам типу `KeyValuePair` з властивостями `Key` та `Value`. Крім того, звертатися як до існуючих, так і до відсутніх елементів можна через індексатор. Видалити елемент за ключем можна за допомогою метода `Remove()`.

Ключ і значення можуть бути як різних, так і однакових типів. Властивості `Keys` та `Values` відповідно повертають колекції ключів та значень.

8.2. Задачі

Наступні завдання виконати двома способами:

– з використанням масивом, з виділенням пам'яті, розмір заздалегідь не відомий. Елементи можуть вставлятися в будь-яке місце масиву; може бути видалений будь-який елемент або група елементів. Реалізувати операцію індексування.

– з використанням двозв'язного списку. Замість операції індексування реалізувати ітератор як набір методів списку.

Для роботи з контейнерами реалізувати методи: додавання та видалення елементу, вводу та виводу всього контейнеру.

Задача 8.1

Створити клас «Відділ кадрів». Поля: прізвище співробітника, ім'я, по батькові, посаду, стаж роботи, оклад.

Задача 8.2

Створити клас «Червона книга». Поля: вид тварини, рід, сімейство, місце проживання, чисельність популяції.

Задача 8.3

Створити клас «Бібліотека». Поля: автор книги, назва, рік видання, код УДК, ціна, кількість в бібліотеці.

Задача 8.4

Створити клас «Супутники планет». Поля: назву, назву планети-господаря, рік відкриття, діаметр, період обертання.

Задача 8.5

Створити клас «Радіодеталі». Поля: позначення, тип, номінал, кількість на схемі, позначення можливого замінника.

Задача 8.6

Створити клас «Побут студентів». Поля: прізвище студента, ім'я, по батькові, факультет, розмір стипендії, число членів сім'ї.

Задача 8.7

Створити клас «Міський транспорт». Поля: вид транспорту, номер маршруту, початкова зупинка, кінцева зупинка, час у дорозі.

Задача 8.8

Створити клас «Спортивні змагання». Поля: прізвище спортсмена, ім'я, команда, вид спорту, заліковий результат, штрафні очки.

Задача 8.9

Створити клас «Оптова база». Поля: назву товару, кількість на складі, вартість одиниці, назва постачальника, термін поставки.

Задача 8.10

Створити клас «Зоопарк». Поля: вид тварини, кличка, вік, категорія рідкості, вага, добовий раціон м'яса, овочів, молока.

ЛАБОРАТОРНА РОБОТА 9
ГРАФІЧНИЙ ІНТЕРФЕЙС

Мета лабораторної роботи – освоїти написання підпрограм.

9.1. Теоретичні основи

Інтерфейс користувача – це набір технічних та програмних засобів, за допомогою яких людина взаємодіє з комп'ютером. Далі йтиметься про програмні засоби інтерфейсу комп'ютеру.

Графічний інтерфейс користувача (Graphical user interface, GUI) дає можливість користувачеві взаємодіяти з комп'ютером за допомогою графічних елементів управляння (вікон, піктограм, меню, кнопок, списків тощо) та технічних пристроїв позиціонування, таких як маніпулятор "миша" Програми, які реалізують цей тип інтерфейсу, мають назву *застосунків графічного інтерфейсу користувача*.

Невід'ємною частиною практики сучасного програмування стало використання зворотних викликів (callback) і повідомлень (notifications). Для реалізації зворотних викликів і повідомлень у мовах C і C++ використовують покажчики на функції. Такий підхід несе в собі деякі недоліки, тому що не надає типової захищеності, що може привести до появи помилок, які не виявляються на стадії компіляції.

У C# є нова синтаксична конструкція – *делегати* (delegates). Їхнє призначення аналогічне покажчикам на функції в C++, але делегати є керованими об'єктами і прив'язані до типів. Середовище виконання гарантує, що делегат указує на припустимий об'єкт. У C# є дві основні області застосування делегатів: методи зворотного виклику (замість покажчиків на функції) і оброблювачі подій. Делегати дозволяють викликати методи, вибір яких відбувається під час виконання програми. Делегати являють собою типи-посилання і є нащадками стандартного типу `System.MulticastDelegate`. Делегат – це клас, що містить дані про сигнатуру методу. Екземпляр делегата (delegate instance) – об'єкт, що дозволяє прив'язатися до конкретного методу, що відповідає визначеній сигнатурі. У сигнатуру методу входить тип значення, що повертається, і список аргументів.

Нижче приведений приклад оголошення делегата:

```
delegate string MyDelegate(int x);
```

Делегат може бути оголошений як у просторі імен, так і в класі. Для виклику використовується звичайний синтаксис виклику методу, тільки як ім'я методу використовується ім'я екземпляра делегата. Для ініціалізації делегата методом потрібно передати ім'я цього методу в конструктор делегата при його створенні:

```
тип-делегата ім'я = new тип-делегата (метод);
```

Це може бути метод будь-якого класу, за умови, що параметри методу і параметри делегата збігаються за типами.

Важливою властивістю делегатів є те, що для них допускається композиція. Це означає, що кілька делегатів, що посилаються на різні методи, можна комбінувати в один, у результаті чого при спробі виклику методу через такий складений делегат замість одного методу викликається кілька методів у тій послідовності, у якій їхні делегати додавалися в результуючий делегат. Така технологія дозволяє при виконанні програми визначати послідовність методів (наприклад, з того самого коду при виконанні однієї умови можуть викликатися методи `A()` і `B()`, іншого – `A()` і `C()`, третьої умови – взагалі `A()`, `B()` і `C()`, при цьому код, що їх викликає, як і раніше, може знаходитися в бібліотеці, і змінювати його не буде потрібно).

Реалізація застосунків графічного інтерфейсу користувача базується на механізмі отримання та обробки подій. Уся програма складається з ініціалізації (реєстрації візуальних елементів управління) та основного циклу отримання та обробки подій. Події – це переміщення або натискання кнопок миші, клавіатурне введення, тощо. Кожний зареєстрований візуальний елемент управління може отримувати події, які до нього стосуються, та виконувати функції обробки цих подій. Робота з подіями в C# відповідає моделі "видавець – передплатник", де клас публікує подію, яку він може ініціювати, і будь-які класи можуть підписатися на цю подію. При ініціації події середовище стежить за тим, щоб повідомити всіх передплатників про виникнення події.

Метод, який викликається при виникненні події, визначається делегатом. Потрібно, щоб такий делегат приймав два аргументи. Ці аргументи повинні представляти два об'єкти: той, що ініціював подію (видавець), і інформаційний об'єкт події, що повинний бути похідним від класу `EventArgs` .NET Framework.

Спеціальне ключове слово **`event`** використовують для опису елементів класу типу відповідного делегату.

Бібліотека Windows.Forms надає прості й у той же час потужні механізми для керування графічним інтерфейсом користувача.

Форма являє собою екранний об'єкт, який можна застосовувати для надання інформації користувачу і для обробки введення інформації від користувача. Найпростіший спосіб задати інтерфейс користувача для форми – розмістити елементи керування на її поверхні. З іншого боку, форма – це об'єкт, що задається властивостями, що визначають їхній зовнішній вигляд, методами, що визначають їхню поведінку, і подіями, що визначають їхню взаємодію з користувачем. Форми, як і всі візуальні об'єкти в .NET, є екземплярами класів. Зокрема, форми успадковані від `System.Windows.Forms.Form`.

При проектуванні додатків графічного інтерфейсу користувача програміст взаємодіє зі спеціальним засобом – дизайнером форм. Дизайнер форм призначений для візуальної розробки користувацького інтерфейсу програми. До основних елементів дизайнера форм можна

прилічити:
`Properties Window` (пункт меню `View | Properties Window`);
`Layout Toolbar` (пункт меню `View | Toolbars | Layout`);
`Toolbox` (пункт меню `View | Toolbox`).

Проектування графічного інтерфейсу здійснюється аналогічно відповідним технологіям Visual Basic, Delphi, тощо.

Більшість візуальних компонентів надає властивість `Text` типу **`string`**. Для форми безпосередньо – це заголовок, для мітки (`Label`), для області введення тексту (`TextBox`) – безпосередньо текст тощо.

Особливу роль у проектуванні програмного забезпечення інформаційних систем відіграють так звані компоненти, обізнані про дані (data-aware components). Робота з даними, які ці компоненти відображають або дозволяють редагувати, може здійснюватися як вручну, так і через так зване джерело даних. *Джерело даних* (Data Source) – це об'єкт в пам'яті, який, як правило, являє собою контейнер для колекції інших об'єктів, або ж це може бути єдиний екземпляр об'єкта. Використання джерела даних дозволяє уникнути дублювання інформації, оскільки візуальні компоненти працюють безпосередньо з наборами даних.

Для відображення табличних даних використовують компонент `DataGridView`. Робота цього компоненту буде проілюстрована нижче на відповідних прикладах.

Від початку існування графічної оболонки MS Windows за відображення графічних об'єктів відповідає підсистема Graphics Device Interface (GDI). Ця підсистема дозволяє відтворювати графіку на різних пристроях відображення – дисплеях, принтерах тощо. GDI відповідає за растеризацію ліній і кривих, відображення шрифтів і обробку палітри. Після виходу Windows XP базовою підсистемою стала GDI+. GDI+ є поліпшеним середовищем для 2D графіки, в якому, зокрема реалізоване згладжування ліній, градієнтна заливка, підтримка сучасних графічних форматів (JPEG і PNG) тощо. Окрім цілих, можна застосовувати координати з плаваючою точкою. В .NET можна використовувати функції GDI+ через простір імен `System.Drawing`. Зокрема, це про-

стір імен містить найважливіший тип GDI – `Graphics`, що інкапсулює в собі основні можливості двовимірної графіки.

Найбільш природний спосіб отримання графічного зображення на формі – реалізація для певного компоненту (наприклад, панелі) події Paint. Відповідний метод отримує, зокрема, параметр типу `System.Windows.Forms.PaintEventArgs`. Одне з властивостей

Об'єкт, який реалізує можливості класу `Graphics`, містить інформацію про стан системи відображення – систему координат, колір, шрифт і режими малювання. За замовчуванням графічні зображення викреслюються в стандартній для комп'ютерної графіки системі координат, в якій координати можуть приймати цілі значення, а осі направлені зліва направо і зверху вниз. Для малювання контурів фігур використовується об'єкт типу `Pen`, який зазвичай передається функціям як параметр. Для визначення способу заповнення використовуються об'єкти класів, вироблених від `Brush`, наприклад `SolidBrush`. Параметри конструкторів цих об'єктів вимагають, зокрема, завдання кольору. Для цього використовується структура типу `Color`. Її властивості визначають конкретні кольору.

Для завдання шрифтів тексту використовують об'єкт типу `Font`.

Об'єкти зазначених типів можна створювати як заздалегідь, так і під час виклику функцій відображення.

Кожну з фігур можна намалювати заповненою, або вималювати тільки її границі. Для цієї мети використовуються методи, імена яких починаються відповідно на `Fill...` і `Draw...`. Наприклад, метод `FillRectangle()` зображує заповнений прямокутник, `DrawRectangle()` – контурний. Кожен з методів `DrawRectangle()`, DrawEllipse(), `FillRectangle() i FillEllipse()` викликається з п'ятьма параметрами: перо (пензлик для заповнених фігур), `x, y, width i height`. Координати x і в задають положення верхнього лівого кута фігури, параметри `width` і `height` визначають її межі. Метод `DrawString(str, font, brush, x, y)` дозволяє вивести рядок `str` починаючи з координат `x, y` заданим шрифтом і пензлем. Усі перелічені методи мають різні варіанти виклику, що використовують,

зокрема, допоміжні об’єкти.

Існує велика кількість функцій, що реалізують інші графічні можливості, зокрема, роботу з растровими зображеннями.

9.2. Задачі

Задача 9.1

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма зверху вниз на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від’ємними.

Задача 9.2

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма зліва направо на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від’ємними.

Задача 9.3

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма справа наліво на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від’ємними.

Задача 9.4

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми» і 3–4 кнопки установки стилю зафарбовування. Будується стовпчикова діаграма справа наліво на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від’ємними.

Задача 9.5

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова

діаграми», «Очищення форми». Будується стовпчикова діаграма з прямокутників, відцентрувати щодо середини форми вибраним кольором. Числа можуть бути тільки додатними.

Задача 9.6

Дано текстовий файл з десятьма числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма з прямокутників, необхідно відцентрувати щодо середини форми за типом дитячої пірамідки, тобто масив повинен бути відсортований. Числа можуть бути тільки додатними.

Задача 9.7

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма знизу вгору на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від'ємними.

Задача 9.8

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма зліва направо на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути тільки додатними.

Задача 9.9

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма справа наліво на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути тільки додатними.

Задача 9.10

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми» і 3–4 кнопки установки стилю зафарбовування. Будується стовпчикова діаграма справа наліво на весь розмір

форми вибраним кольором. Передбачити, що числа можуть бути тільки додатними.

ПРИКЛАДИ ДО ЛАБОРАТОРНИХ РОБІТ

Лабораторна робота 1

Приклад 1.1.

```
class Rectangle
{
  double width;
  double height;
  public Rectangle(double width, double height)
  {
    this.width = width;
    this.height = height;
  }
  public Rectangle() : this(10, 20)  {
  }
}
```

Лабораторна робота 2

Приклад 2.1. У наступному класі перевантажується операція + і операція неявного приведення до типу **string**:

```
using System;
namespace PointTest
{

  class Point
  {
    private double x, y;
    public Point(double x, double y)
    {
      this.x = x;
      this.y = y;
    }
    public static Point operator+(Point a, Point b)
    {
      return new Point(a.x + b.x, a.y + b.y);
    }

    public static implicit operator string(Point p)
    {
      return p.x + " " + p.y;
    }
```

```
  }

  class Test
  {
    static void Main(string[] args)
    {
      Point p1 = new Point(1, 2);
      Point p2 = new Point(3, 4);
      Point p3 = p1 + p2;
      Console.WriteLine(p3); // Приведення до string
    }
  }
}
```

Лабораторна робота 3

Приклад 3.1. У прикладі демонструється оголошення і доступ до вкладених класів, які містять різні модифікатори доступу.

Лістинг класу `Outer` наступний:

```
public class Outer
{
  public int d;
  static public int sd;
  public Inner1 GetInner1()
  {
    Inner1 i1 = new Inner1();
    i1.d1 = 25;
    Inner1.sd1 = 30;
    return i1;
  }
  private class Inner1
  {
    public int d1;
    public static int sd1;
  }
  public class Inner2
  {
    public int d2;
    public static int sd2;
  }
  internal class Inner3
  {
    public int d3;
```

```
    public static int sd3;
  }
  protected class Inner4
  {
    public int d4;
    public static int sd4;
  }
  protected internal class Inner5
  {
    public int d5;
    public static int sd5;
  }
}
```

Використання вкладених класів в деякому методі може бути наступним:

```
Outer o = new Outer();
o.d = 300;
Outer.sd = 230;
Outer.Inner2 i2 = new Outer.Inner2();
i2.d2 = 440;
Outer.Inner2.sd2 = 500;
Outer.Inner3 i3 = new Outer.Inner3();
i3.d3 = 100;
Outer.Inner3.sd3 = 400;
Outer.Inner5 i5 = new Outer.Inner5();
i5.d5 = 200;
Outer.Inner5.sd5 = -100;
```

Як видно з вищенаведеного коду, створювати об'єкти можна тільки для класів `Inner2`, `Inner3`, `Inner5`. Щоб створити об'єкт protected-класу `Inner4`, потрібно оголосити інший клас, який успадковує клас `Outer`, наприклад:

```
class Outer2 : Outer
{
  void SomeMethod()
  {
    Inner4 i4 = new Inner4();
    i4.d4 = 23;
    Inner4.sd4 = 330;
  }
}
```

Даний приклад також показує, що вкладений private-клас прихо-

вує доступ до оголошеної в ньому статичної змінної з методів інших класів. Так само, приховує доступ і вкладений protected-клас.

Приклад 3.2. Приклад агрегації для типу відносини has-a. Клас Figures містить масив класів *Point* (точка) і масив класів *Line* (лінія). Кількість елементів в масивах може бути довільним, навіть рівним 0.

```
using System;
using static System.Console;
namespace ConsoleApp1
{

  class Point
  {
    public double x;
    public double y;
  }

  class Line
  {
    public Point pt1 = null;
    public Point pt2 = null;
  }

  class Figures
  {
    public Point[] points;
    public Line[] lines;
    public Figures()
    {
      points = null;
      lines = null;
    }
    public void Print()
    {
      WriteLine("Array points:");
      for (int i = 0; i < points.Length; i++)
      {
        WriteLine("x = {0}, y = {1}", points[i].x,
        points[i].y);
      }
      WriteLine("Array lines:");
      for (int i=0;i<lines.Length; i++)
      {
```

```
        WriteLine("pt1.x = {0}, pt1.y = {1}", lines[i].pt1.x,
        lines[i].pt1.y);
        WriteLine("pt2.x = {0}, pt2.y = {1}", lines[i].pt2.x,
        lines[i].pt2.y);
      }
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Figures fg = new Figures();
      fg.points = new Point[5];
      for (int i = 0; i < fg.points.Length; i++)
      {
        fg.points[i] = new Point();
        fg.points[i].x = i * i;
        fg.points[i].y = i * i * i;
      }
      fg.lines = new Line[3];
      for (int i = 0; i < fg.lines.Length; i++)
      {
        fg.lines[i] = new Line();
        fg.lines[i].pt1 = new Point();
        fg.lines[i].pt2 = new Point();
        fg.lines[i].pt1.x = i;
        fg.lines[i].pt1.y = i * 2;
        fg.lines[i].pt2.x = i * 3;
        fg.lines[i].pt2.y = i * i;
      }
      fg.Print();
    }
  }
}
```

Приклад 3.3. Приклад композиції для типу відносини has-a. Розглянемо клас *Line*, який описує лінію по двох точках. Точки описуються класом *Point*. Клас *Line* містить 2 примірника класів *Point*. Без цих примірників (об'єктів) клас *Line* існувати не може, оскільки обидва примірники становлять важливу частину лінії (крайні точки лінії). Таким чином, обидва екземпляра класу *Point* є частиною класу *Line*. Фрагмент прикладу наступний.

```
class Point
{
  public double x;
  public double y;
}
class Line
{
  public Point pt1 = null;
  public Point pt2 = null;
}
```

Лабораторна робота 4

Приклад 4.1. Припустимо, необхідно розробити ієрархію класів "Регіон" – "Населений район" – "Країна". Окремі класи цієї ієрархії можуть стати базовими для інших класів (наприклад "Незаселений острів", "Національний парк", "Адміністративний район", "Автономна республіка" і т.д.). Ієрархію класів можна доповнити класами "Місто" і "Острів". Доцільно в кожен клас додати конструктор, який ініціалізує усі поля. Можна також створити масив посилань на різні об'єкти ієрархії і. для кожного об'єкта вивести на екран рядок даних про нього. Для того, щоб одержати рядкове представлення об'єкта, необхідно перекрити функцію `ToString()`. Можна запропонувати наступну ієрархію класів:

```
namespace HierarchyTest
{
  // ієрархія класів

  class Region {
    public string Name { get; set; }
    public double Area { get; set; }
    public Region(string name, double area)
    {
      Name = name;
      Area = area;
    }
    public override string ToString()
    {
      return Name + ".\nТериторія " + Area + " кв.км.\n";
    }
  }
```

```
  class PopulatedRegion : Region {
    public int Population { get; set; }
    public PopulatedRegion(string name, double area, int popu-
lation)
      : base(name, area)
    {
      Population = population;
    }
    public int Density() {
      return (int) (Population / Area);
    }
    public override string ToString()
    {
      return base.ToString() +
        "Населення " + Population + " чол.\n" +
        "Щільність населення " + Density() + " чол/кв.км.\n";
    }
  }

  class Country : PopulatedRegion
  {
    public string Capital { get; set; }
    public Country(string name, double area, int population,
string capital) :
      base(name, area, population)
    {
      Capital = capital;
    }
    public override string ToString()
    {
      return "Країна " + base.ToString() + "Столиця " +
               Capital + "\n";
    }
  }

  class City : PopulatedRegion
  {
    public int Boroughs { get; set; } // Кількість районів
    public City(string name, double area, int population,
           int boroughs) :
      base(name, area, population)
    {
      Boroughs = boroughs;
```

```
    }
    public override string ToString()
    {
     return "Місто " + base.ToString() + "Районів - " +
     Boroughs + "\n";
    }

  }

  class Island : PopulatedRegion
  {
    public string Sea { get; set; }
    public Island(string name, double area, int population,
     string sea) :
      base(name, area, population)
    {
      Sea = sea;
    }
   public override string ToString()
    {
      return "Острів " + base.ToString() + "Море -    " + Sea
+ "\n";
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Region[] a = { new City("Київ", 839, 2679000, 10),
             new Country("Україна", 603700, 46294000, "Київ"),
             new City("Харків", 310, 1461000, 9),
             new Island("Зміїний", 0.2, 30, "Чорне") };
      foreach (Region region in a)
        System.Console.WriteLine(region);
    }
  }
}
```

Лабораторна робота 5

Приклад 5.1. Абстрактний клас Shape (геометрична фігура) реалізує поля і методи, що можуть бути використані різними похідними

класами. До таких полів можна, наприклад, віднести поточну позицію і метод переміщення екраном `MoveTo()`. У класі `Shape` також оголошені абстрактні методи, такі як `Draw()`, що повинні бути реалізовані у всіх похідних класах, але по-різному. Реалізація за умовчанням не має сенсу.

```
abstract class Shape
{
  int x, y;
    . . .
  public void MoveTo(int newX, int newY)
  {
    . . .
    Draw();
  }
  public abstract void Draw();
}
```

Конкретні класи, створені від `Shape`, такі як `Circle` чи `Rectangle`, визначають реалізацію методу `Draw()`.

```
class Circle : Shape
{
  public override void Draw()
  {
    . . .
  }
}
class Rectangle : Shape
{
  public override void Draw()
  {
    . . .
  }
}
```

Приклад 5.2. Абстрактні класи можуть містити абстрактні властивості, для таких властивостей не задається код доступу, а тільки позначається необхідність завдання такого коду в нащадках.

```
abstract class Shape
{
  public abstract double Area
  {
    get;
```

```
  }
}

class Rectangle : Shape
{
  double width, height;
  public Rectangle(double width, double height)
  {
    this.width = width;
    this.height = height;
  }
  public override double Area
  {
    get
    {
      return width * height;
    }
  }
}
```

Лабораторна робота 6

Приклад 6.1.

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 5;
            int y = x / 0;
            Console.WriteLine($"Результат: {y}");
        }
        catch
        {
            Console.WriteLine("Виник виняток!");
        }
        finally
        {
            Console.WriteLine("Блок finally");
        }
        Console.WriteLine("Кінець програми");
        Console.Read();
    }
```

```
}
```

Лабораторна робота 7

Приклад 7.1. Наведемо приклад використання узагальнень.

```
public class Pair<T>
{
  public T First  { get; set; }
  public T Second { get; set; }
  public Pair(T first, T second)
  {
    First = first;
    Second = second;
  }
}

class Program
{
 static void Main(string[] args)
 {
   Pair<string> p = new Pair<string>("Прізвище", "Ім\'я");
   string s = p.First; // Отримуємо рядок без приведення типів
// Можна використовувати цілі константи
   Pair<int> p1 = new Pair<int>(1, 2);
// Отримуємо ціле значення без приведення типів
   int i = p1.Second;
 }
}
```

Приклад 8.2.

```
public class PairOfDifferentObjects<TFirst, TSecond>
{
  public TFirst First  { get; set; }
  public TSecond Second { get; set; }
  public PairOfDifferentObjects(TFirst first, TSecond second)
  {
    First = first;
    Second = second;
  }
}

class Program
{
 static void Main(string[] args)
```

```
 {
  PairOfDifferentObjects<int, String> p =
    new PairOfDifferentObjects<int, String>(1000, "thousand");
    PairOfDifferentObjects<int, int> p1 =
      new PairOfDifferentObjects<int, int>(1, 2);
  }
}
```

Лабораторна робота 8

Приклад 8.1. Можна самостійно створити клас, що реалізує інтерфейс `IComparable`. Наприклад, масив прямокутників сортується за площею:

```
class Rectangle : IComparable<Rectangle>
  {
    double width, height;
    public Rectangle(double width, double height)
    {
      this.width = width;
      this.height = height;
    }
    public double Area()
    {
      return width * height;
    }
    public int CompareTo(Rectangle rect)
    {
      return Area().CompareTo(rect.Area());
    }
    public override String ToString()
    {
      return "[" + width + ", " + height + ", area = " +
     Area() + "]";
    }

  }

class Program
{
  static void Main(string[] args)
  {
   Rectangle[] a = {new Rectangle(2, 7), new Rectangle(5, 3),
   new Rectangle(3, 4)};
```

```
   Array.Sort(a);
   foreach (Rectangle rect in a)
      Console.WriteLine(rect);
    }
}
```

Приклад 8.2. Інтерфейс `IComparer` містить опис методу `Compare()` з двома параметрами. Функція повинна повернути від'ємне число, якщо перший об'єкт під час сортування необхідно вважати меншим, чим інший, значення 0, якщо об'єкти еквівалентні, і додатне число в протилежному випадку.

```
class Rectangle
{
   double width, height;
   public Rectangle(double width, double height)
   {
     this.width = width;
     this.height = height;
   }
   public double Area()
   {
     return width * height;
   }
   public override String ToString() {
     return "[" + width + ", " + height + ", area = " + Area()
            + "]";
    }
 }

class CompareByArea : IComparer<Rectangle>
 {
  public int Compare(Rectangle r1, Rectangle r2) {
     return r1.Area().CompareTo(r2.Area());
   }
}

class Program
{
  static void Main(string[] args)
  {
   Rectangle[] a = { new Rectangle(2, 7), new Rectangle(5, 3),
                     new Rectangle(3, 4) };
   Array.Sort(a, new CompareByArea());
```

```
   foreach (Rectangle rect in a)
   Console.WriteLine(rect);
    }
  }
```

Приклад 8.3.

```
using System;
using System.Collections.Generic;
namespace DictionaryTest
{

  class Program
  {
    static void Main(string[] args)
    {
    SortedDictionary<string, string> dictionary =
      new SortedDictionary<string, string>();
    dictionary.Add("sky", "небо");
    dictionary.Add("house", "дім");
    dictionary.Add("white", "білий");
    dictionary.Add("dog", "собака");
    dictionary["dog"] = "собака"; // аналогічно dictionary.Add
                                  // ("dog", "собака");
    foreach (var pair in dictionary) // pair типу KeyValuePair
// виведення за абеткою
     Console.WriteLine("{0}\t {1}", pair.Key, pair.Value);
// True
     Console.WriteLine(dictionary.ContainsValue("небо"));
// False
     Console.WriteLine(dictionary.ContainsKey("city"));
     dictionary.Remove("dog");
     dictionary["house"] = "будинок";
     foreach (var pair in dictionary)
        Console.WriteLine("{0}\t {1}", pair.Key, pair.Value);
    }
  }
}
```

Лабораторна робота 9

Приклад 9.1. Застосування делегатів для зворотного виклику можна продемонструвати на прикладі створення класу для розв’язання методом дихотомії будь-якого рівняння. Делегати дозволяють здійснити узагальнений опис функції, яка визначає ліву частину рівняння:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DelegatesTest
{
 public class Solver
 {
  public delegate double LeftSide(double x);
  public static double Solve(double a, double b, double eps,
                             LeftSide f)
    {
      double x = (a + b) / 2;
      while (Math.Abs(b - a) > eps)
      {
        if (f(a) * f(x) > 0)
          a = x;
        else
          b = x;
        x = (a + b) / 2;
      }
      return x;
    }
  }
}
```

Приклад 9.2. У наступному класі здійснюється розв'язання певного рівняння.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DelegatesTest
{

  class Program
  {
    public static double F(double x)
    {
      return x * x - 2;
    }
    static void Main(string[] args)
    {
      Solver.LeftSide ls = new Solver.LeftSide(F);
      Console.WriteLine(Solver.Solve(0, 2, 0.000001, ls));
    }
```

```
  }
}
```

Приклад 9.3. Можна обійтися без окремого створення екземпляра делегату. Такий екземпляр буде створено автоматично:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DelegatesTest
{

  class Program
  {
    public static double F(double x)
    {
      return x * x - 2;
    }
    static void Main(string[] args)
    {
      Console.WriteLine(Solver.Solve(0, 2, 0.000001, F));
    }
  }
}
```

Приклад 9.4. У версії 2.0 C# уводиться поняття безіменних методів. Такі методи створюють безпосередньо в місці, де потрібен екземпляр делегату. Для створення безіменного методу використовують таку конструкцію:

```
delegate(відповідний_список_параметрів) { тіло_функції }
```

Попередній приклад програми, яка використовує делегат, можна буде ще скоротити:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DelegatesTest
{

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(Solver.Solve(0, 2, 0.000001,
      delegate(double x) { return x * x - 2; } ));
```

```
    }
  }
}
```

Список літератури

1. Голуб Б. М. C#. Концепція та синтаксис. Навч. посібник / Б. М. Голуб – Львів: Видавничий центр ЛНУ імені Івана Франка, 2006. – 136 с.

2. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4 5-е издание / Э. Троелсен ; пер. с англ. – Москва: ООО "И. Д. Вильямс", 2011. – 1392 с.

3. Шилдт Г. C# 3.0. Полное руководство / Г. Шилдт ; пер. с англ. – Москва: Диалектика-Вильямс, 2009. – 992 с.

4. Нейгел К. C# 4.0 и платформа .NET 4 для профессионалов / К. Нейгел, Б. Ивьен, Д. Глинн ; пер. с англ. – Киев: Диалектика, 2011. – 1440 с.

5. Уотсон К. Microsoft Visual C# 2008. Базовый курс / К. Уотсон, К. Нейгел, Я. Х. Педерсен, Дж. Д. Рид, М. Скиннер, Э. Уайт ; пер. с англ. – Москва: Диалектика–Вильямс, 2009. – 1216 с.

6. Рихтер Дж. Программирование на платформе Microsoft .NET Framework 2.0 на языке C# / Дж. Рихтер ; пер. с англ. – Санкт-Петербург: Питер, М:Русская Редакция, 2007. – 656 с.

7. Рихтер Дж. Программирование на C# для профессионалов / Дж. Рихтер, М. Ван де Боспурт ; пер. с англ. – Москва: Вильямс, 2014. – 368 с.

8. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес ; пер. с англ.. – Санкт-Петербург: Питер, 2001.

9. Об’єктно-орієнтоване програмування (частина 2). Розробник курсу Л.В. Іванов. http://www.iwanoff.inf.ua/oop_ua/index.html

ЗМІСТ

Навчальне видання

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ C#

Методичні вказівки до лабораторних робіт
з курсу «Об'єктно-орієнтоване програмування»

для студентів спеціальності
122 – Комп'ютерні науки
126 – Інформаційні системи та технології

Укладачі **НІКУЛІНА** Олена Миколаївна
ІВАНОВ Лев Вадимович
КОЦЮБА Ніна Вікторівна

Відповідальний за випуск В. В. Москаленко

Роботу до видання рекомендував

В авторській редакції

План 2021 р., поз.207

Підп. до друку Формат 60x84 1/16. Папір друк.№2.
Гарнітура Times New Roman. Ум. друк. арк.
Тираж 50 прим. Зам. № . Ціна договірна.

Віддруковано в ТОВ «ДРУКАРНЯ МАДРИД»
61024, м. Харків, вул. Максиміліанівська, 11
Тел.: (057) 756-53-25
Свідоцтво суб'єкта видавничої справи
Серія ДК, № 4399 от 27.08.2012 р.
www.madrid.in.ua e-mail: info@madrid.in.ua 61002