Міністерство освіти і науки України
Державний вищий навчальний заклад
“Ужгородський національний університет”
Математичний факультет
Кафедра системного аналізу і теорії оптимізації

ОСНОВИ ОБ’ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ У C#

Методичні вказівки до лабораторних робіт для студентів І-го курсу математичного факультету спеціальності "Прикладна математика"

Ужгород – 2014

Брила А.Ю., Антосяк П.П., Глебена М.І., Чупов С.В., Семйон І.В. Основи об’єктно-орієнтованого програмування у C#. Методичні вказівки до лабораторних робіт для студентів І-го курсу математичного факультету спеціальності "Прикладна математика". – Ужгород, 2014. – 73 с.

Розглядаються основні поняття і принципи об’єктно-орієнтованого програмування реалізовані у мові C#. Наведено необхідний теоретичний матеріал та приклади розв’язання деяких задач. Запитання для самоконтролю та завдання для самостійного виконання дозволяють закріпити розглядуваний матеріал.

Рецензенти: канд. фіз.-мат. наук, доц. Мич І.А.,
канд. фіз.-мат. наук, доц. Погоріляк О.О.

Рекомендовано до друку Вченою радою математичного факультету ДВНЗ “Ужгородський національний університет” 20 листопада 2014 року, протокол № 4.

1. КЛАСИ ТА ОБ'ЄКТИ

1.1. Поняття класу та об'єкта

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

Приклад. Прямокутник характеризується наступними даними:

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

Також для прямокутника можна визначити наступні дії:

1) знаходження периметру;
2) знаходження площі;
3) визначення рівності двох прямокутників, які задаються за допомогою своїх сторін;
4) масштабування (збільшення або зменшення сторін прямокутника у певну кількість разів).

Отже, прямокутник можна описати наступною логічною структурою

<u>Прямокутник</u>

Сторона *a*
Сторона *b*
} *Дані*

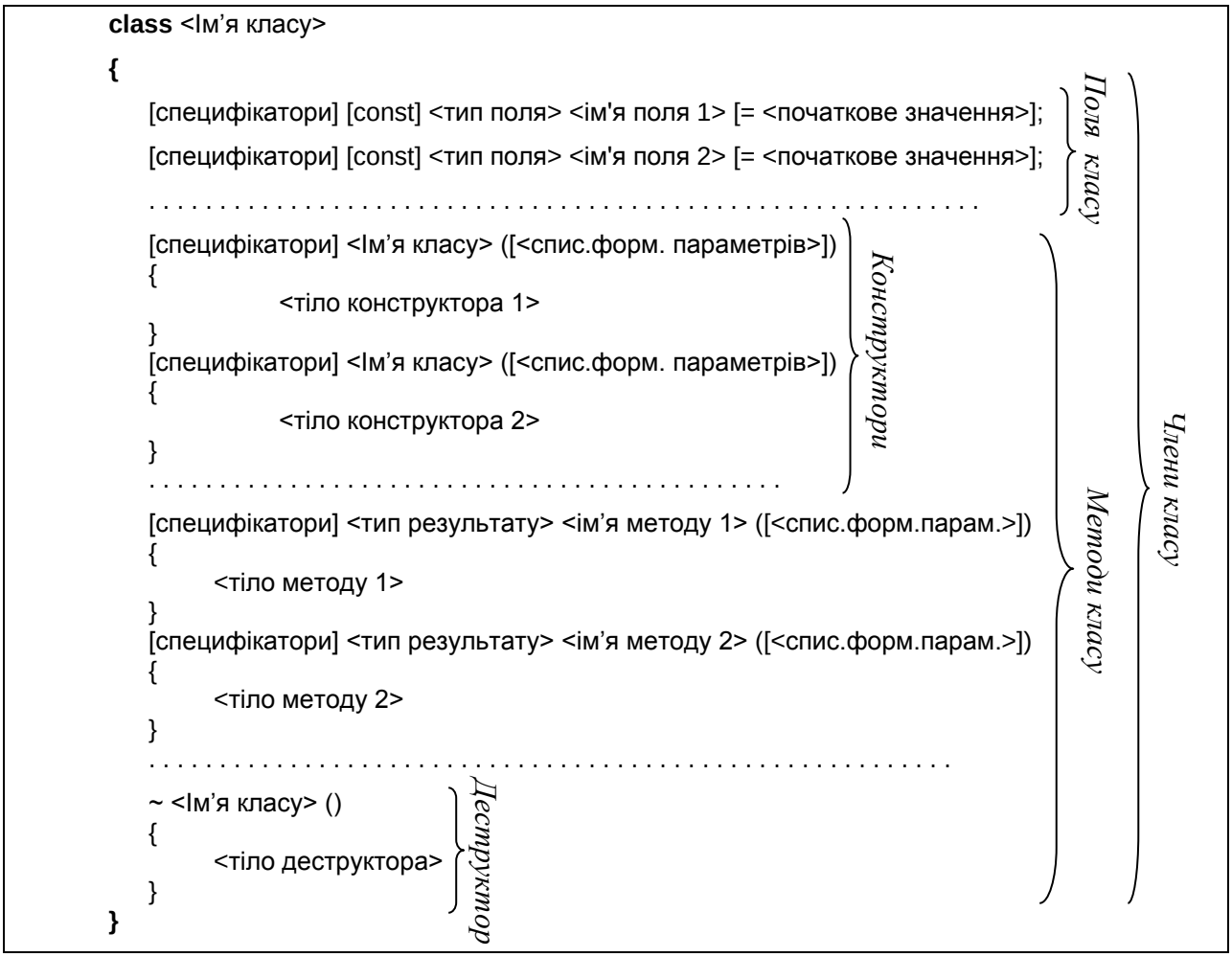
Дія *знаходження периметру*
Дія *знаходження площі*
Дія *порівняння* *(з іншим прямокутником)*
Дія *масштабування*
} *Функціональні можливості*

В мовах програмування для опису об'єктів реальної дійсності використовується тип даних, який має назву *клас* і який дозволяє описувати як властивості об'єкту так і його поведінку або ж функціональні можливості

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

1.2. Опис класів

У найпростішому випадку загальна схема опису класу в C# наступна:



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

При описі членів класу можна вказувати *специфікатори доступу*, які визначають в яких місцях опису даного класу чи іншого класу вказані члени є доступними (звідки до них можна звернутись):

- **`private`** – закриті члени класу. Доступні тільки у методах цього ж класу;
- **`public`** – відкриті члени класу. Доступ до них може бути здійснений з поза меж класу (з іншого класу);
- **`protected`** – захищені члени класу. Доступні у методах цього ж класу та методах його нащадків(про нащадків дивись пункт 4).

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

Приклад. Опишемо розглядувану логічну структуру "Прямокутник" за допомогою класу у C#.

Об'єкт предметної області	**Програмна структура** `class`

Прямокутник
 	
Сторона *a*
 	
Сторона *b*
 	
Дія *створення трикутника* (конструктор)
 	
Дія *знаходження периметру*
 	
Дія *знаходження площі*
	`class Rectangle`
	`{`
	
	`public double a;`
	
	`public double b;`
	
	`public Rectangle (double a, double b)`
	`{`
	`this.a = a;`
	`this.b = b;`
	`}`
	
	`public double Perimeter ()`
	`{`
	`return 2 * (a + b);`
	`}`
	
	`public double Square ()`
	`{`
	`return a * b;`
	`}`
	
	`public bool IsEqual (double a, double b)`

Об’єкт предметної області	Програмна структура `class`
Дія *порівняння* *(з іншим прямокутником)*	`{ return (this.a == a && this.b == b) || (this.a == b && this.b == a); }`
Дія *масштабування*	`public void Scale (double k) { a = k * a; b = k * b; }`
Дія *знищення трикутника* (деструктор)	`~Rectangle () { }`
	`}`

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

1.3. Конструктори. Створення об’єктів

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

1.3.1. Конструктор без параметрів. Якщо в класі явно не визначено жодного конструктора (і тільки в цьому випадку!), то він надається за замовчуванням (*конструктор за замовчуванням*). За визначенням такий конструктор не має формальних параметрів. Такий конструктор тільки виділить пам’ять для полів об’єкта, але поля не буде проініціалізовано. Тому бажано розробляти власний конструктор без параметрів.

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

```
class Rectangle
{
    . . . . . .
    public Rectangle()
    {
        a = b = 1;
    }
    . . . . . .
}
```

1.3.2. Конструктори з параметрами. Як було зазначено раніше, опис конструктора може містити параметри, які, як правило, використовують для ініціалізації полів об’єктів.

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

```
class Rectangle
{
    . . . . . .
    public Rectangle(double a)
    {
        this.a = a;
        b = 1;
    }
    . . . . . .
}
```

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

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

```
class Rectangle
{
    . . . . . .
    public Rectangle(double a)
    {
        this.a = a;
        b = 1;
    }
    public Rectangle (double a, double b):this(a)
    {
        this.b = b;
    }
    . . . . . .
}
```

1.3.3. Конструктор копіювання. У деяких випадках зручно мати у класі конструктор, який би давав можливість ініціалізувати поля об’єкта значеннями полів іншого об’єкта. Тоді у класі визначають конструктор з одним параметром, тип якого співпадає із типом класу. Такий конструктор називають *конструктором копіювання*.

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

```
class Rectangle
{
    . . . . . .
    public Rectangle(Rectangle r)
    {
        this.a = r.a;
        this.b = r.b;
    }
    . . . . . .
}
```

1.3.4. Створення об’єктів. Наявність конструкторів у класі дозволяє створювати та ініціалізувати об’єкти у програмі. Об’єкт створюється за допомогою оператора `new`.

Загальне правило використання оператора `new`:

```
<Ім’я класу> <ім’я об’єкта> = new <Ім’я_класу>(список_параметрів);
```

Приклади:

```
Rectangle r1 = new Rectangle();    // a=1, b=1
Rectangle r2 = new Rectangle(3);   // a=3, b=1
Rectangle r3 = new Rectangle(4,5); // a=4, b=5
Rectangle r4 = new Rectangle(r2);  // a=3, b=1
```

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

```
Rectangle r1 = new Rectangle();
Rectangle r2 = r1;
```

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

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

Приклад:

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

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

Приклади:

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

1.4. Доступ до членів класу

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

Загальне правило	Приклад

<Ім’я об’єкта>.<Ім’я поля>	`Rectangle r = new Rectangle();`
	`r.a=4;`
	`r.b=5;`

<Ім’я об’єкта>.<Ім’я методу>	`Console.WriteLine("S={0}", r.Square());` `Console.WriteLine("P={0}", r.Perimeter());`

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

Приклад. Розглянемо детально реалізацію конструктора:

```
public Rectangle (double a, double b)
{
    this.a = a;
    this.b = b;
}
```

Формальні параметри

Поля, описані у класі

1.5. Статичні поля та методи

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

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

```
<Ім’я класу>.<Ім’я статичного поля>
```

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

Приклад опису статичного поля	Приклад звертання до статичного поля

`class Rectangle`
`{`
`  public static int Count;`
`  . . . . . .`
`  public Rectangle()`
`  {`
`  Count++;`
`  a = b = 1;`
`  }`
`  . . . . . .`
`}`
	`Rectangle r1 = new Rectangle();`
	`Rectangle r2 = new Rectangle();`
	`Console.WriteLine(Rectangle.Count); // 2`

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

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

```
class Rectangle
{
    . . . . . .
    static Rectangle()
    {
        Count = 0;
    }
    . . . . . .
}
```

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

```
<Ім'я класу>.<Ім'я статичного метода>
```

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

Приклад опису статичного методу	Приклад звертання до статичного методу

`class Rectangle`
`{`
`. . . . . .`
`public static double Square(Rectangle r)`
`{`
`return r.a * r.b;`
`}`
`. . . . . .`
`}`
	`Rectangle r1 = new Rectangle(3,4);`
	`Console.WriteLine(Rectangle.Square(r1));`

1.6. Деструктор

Як було зазначено раніше, при створенні об'єктів деякого класу явно чи неявно (автоматично) викликається конструктор. У конструкторі, як правило, здійснюють ініціалізацію полів об'єкта, динамічно виділять необхідний об'єм пам'яті, підключають деякі додаткові ресурси та інші дії. Аналогічно,

при знищенні об’єктів автоматично викликається метод, який називають *деструктором*. Деструктор викликається безпосередньо перед знищенням об’єкта, тому усередині деструктора виконують звільнення усіх виділених для цього об’єкта ресурсів: звільнення динамічно виділеної пам’яті, закритті файлів та ін. Якщо додаткових ресурсів при створенні об’єкта не виділяється, то власний деструктор описувати не обов’язково. Зазначимо, що деструктори викликають у порядку, зворотному до порядку виклику конструкторів.

Загальний вигляд деструктора:

```
~<Ім’я класу>()
{
      <тіло деструктора>
}
```

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

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

```
class Rectangle
{
    . . . . . .
    ~Rectangle ()
    {
        Count--;
    }
    . . . . . .
}
```

Питання для самоконтролю

1. *Що таке клас?*
2. *Для чого використовують класи?*
3. *Наведіть загальну схему опису класу та його реалізації.*
4. *Що таке поле?*
5. *Що таке метод?*
6. *Як здійснюється доступ до членів класу?*
7. *Що таке об'єкт?*
8. *Що таке конструктор?*
9. *Що таке конструктор за замовчуванням?*
10. *Що таке деструктор?*
11. *Що таке статичне поле?*
12. *Для чого призначені статичні конструктори?*
13. *Яка різниця між статичним та нестатичним методами?*
14. *Що таке "збирач сміття"?*
15. *Як здійснюється доступ до членів класу?*
16. *Що таке специфікатор доступу?*

Завдання для самостійної роботи

1. Об'єкт "Текст"	
поля	▪ для зберігання послідовності символів; ▪ для зберігання тематики вказаного тексту;
методи	▪ визначення кількості букв у тексті; ▪ визначення кількості пробілів (між словами один пробіл); ▪ заміни кожного входження однієї букви на іншу; ▪ видалення слова з вказаним порядковим номером.
2. Об'єкт "Вектор"	
поля	▪ для зберігання координат вектора; ▪ для зберігання розмірності вектора;
методи	▪ введення елементів вектора; ▪ виведення елементів вектора у рядку; ▪ визначення довжини вектора; ▪ нормування вектора.
3. Об'єкт "Матриця"	
поля	▪ для зберігання елементів матриці; ▪ для зберігання розмірності матриці;
методи	▪ введення елементів матриці; ▪ виведення елементів матриці;

	▪ знаходження найбільшого елемента; ▪ знаходження найменшого елемента.
4. Об’єкт “Число”	
поля	▪ для зберігання натурального числа;
методи	▪ введення числа; ▪ виведення числа; ▪ знаходження кількості цифр; ▪ знаходження суми цифр.
5. Об’єкт “Число – масив цифр”	
поля	▪ для зберігання кількості цифр; ▪ для зберігання натурального числа, як масиву цифр;
методи	▪ введення числа; ▪ виведення числа; ▪ знаходження кількості входження деякої цифри; ▪ знаходження суми цифр; ▪ порівняння з іншим числом-масивом.
6. Об’єкт “Арифметична прогресія ”	
поля	▪ для зберігання першого члена; ▪ для зберігання різниці;
методи	▪ введення та виведення першого члена; ▪ виведення та виведення різниці; ▪ знаходження n-го члена прогресії; ▪ знаходження суми n перших членів прогресії.
7. Об’єкт “Геометрична прогресія ”	
поля	▪ для зберігання першого члена; ▪ для зберігання знаменника;
методи	▪ введення та виведення першого члена; ▪ виведення та виведення знаменника; ▪ знаходження n-го члена прогресії; ▪ знаходження суми n перших членів прогресії.
8. Об’єкт “Пряма на площині ”	
поля	▪ для зберігання коефіцієнтів канонічного рівняння прямої;
методи	▪ введення та виведення коефіцієнтів; ▪ знаходження точки перетину з іншою прямою; ▪ встановлення паралельності з іншою прямою; ▪ встановлення належності деякої точки прямій.
9. Об’єкт “Площина ”	
поля	▪ для зберігання рівняння площини;
методи	▪ введення та виведення коефіцієнтів рівняння площини; ▪ перевірка належності точки площині; ▪ знаходження проекції точи на площину; ▪ знаходження точки перетину прямої з площиною; ▪ встановлення паралельності з іншою площиною.
10. Об’єкт “Множина цифр”	
поля	▪ для зберігання множини цифр;
методи	▪ додавання нової цифри; ▪ виведення цифр, які входять у множину на екран; ▪ знаходження найбільшої цифри; ▪ знаходження суми цифр.

11.	**Об’єкт “**Стек**” (**реалізація стеку за допомогою одновимірного масиву цілих чисел)	
	поля	▪ для зберігання вершини стеку (номера останнього доданого елемента); ▪ масив елементів;
	методи	▪ виведення на екран; ▪ додавання нового елементу; ▪ видалення елементу; ▪ знаходження суми елементів.
12.	**Об’єкт “**Черга**” (**реалізація стеку за допомогою одновимірного масиву цілих чисел)	
	поля	▪ для зберігання номерів першого та останнього елементів черги; ▪ масив елементів;
	методи	▪ виведення на екран; ▪ додавання нового елементу; ▪ видалення елементу; ▪ знаходження суми елементів.
13.	**Об’єкт “**Впорядкований масив**” (**елементами масиву є цілі числа, які завжди впорядковані за зростанням)	
	поля	▪ для зберігання кількості елементів масиву; ▪ масив елементів;
	методи	▪ виведення на екран; ▪ додавання нового елемента; ▪ видалення вказаного елемента; ▪ знаходження елемента, з використанням бінарного пошуку.
14.	**Об’єкт “Многокутник” (**зберігаються координати вершин)	
	поля	▪ для зберігання кількості вершин; ▪ масив вершин;
	методи	▪ введення/виведення координат вершин; ▪ знаходження периметру; ▪ знаходження площі; ▪ належності точки ребру многокутника.

2. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

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

2.1. Унарні оператори

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

```
+   -   !   ~   ++   --   true   false
```

Загальне правило перевантаження унарного оператора:

```
public static <тип результату> operator <унарний оператор>(<Ім’я класу> <аргумент>)
{
    <тіло оператора>
}
```

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

Унарний оператор повинен повертати:

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

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

Унарні оператори `true`, `false` повинні перевантажуватись разом.

Приклад перевантаження	Приклад використання

`class Complex`
`{`
`  public double a;`
 	
`  public double b;`	

Приклад перевантаження	Приклад використання

public Complex(double a, double b)
{
this.a = a;
this.b = b;
}
 	
public static Complex operator -(Complex c)
{
return new Complex(-c.a, -c.b);
}
 	
public static Complex operator ~(Complex c)
{
return new Complex(c.a, -c.b);
}
 	
public static double operator !(Complex c)
{
return Math.Sqrt(c.a*c.a + c.b*c.b);
}
 	
public static Complex operator ++(Complex c)
{
return new Complex(++c.a, ++c.b);
}
 	
public static bool operator true(Complex c)
{
return c.b == 0;
}
public static bool operator false(Complex c)
{
return c.b != 0;
}
}
	Complex c1 = new Complex(1,2); // 1+2i
	
	// протилежне комплексне число
	Complex c2 = -c1; // -1-2i
	
	// спряжене комплексне число
	Complex c3 = ~c1; // 1-2i
	
	// модуль комплексного числа
	double a = !c1; // $\sqrt{5}$
	
	// префіксний та постфіксний інкримент
	Complex c4 = ++c1; // 2+3i
	Complex c5 = c1++; // 2+3i
	
	if(c1)
	Console.WriteLine("Уявна частина = 0.");
	else
	Console.WriteLine("Уявна частина != 0.");

2.2. Бінарні оператори

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

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

Загальне правило перевантаження бінарного оператора:

```
public static <тип результату> operator <бін. опер.>(<Тип1> <арг.1>, <Тип2> <арг.2>)
{
    <тіло оператора>
}
```

Принаймні один аргумент у бінарному операторі повинен мати тип класу, для якого цей оператор перевантажується. Бінарний оператор може повертати величину будь-якого типу. Операції `==` та `!=`, `>` та `<`, `>=` та `<=` повинні перевантажуватись парами та повертати логічне значення. Бінарний оператор, який повертає величину типу класу, для якого цей оператор

перевантажується, повинен створювати новий об’єкт цього класу, виконати з ним необхідні дії і повернути його в якості результату. Складні операції присвоювання (+=, -= і т.д.) перевантажувати не можна. Але якщо у класі перевантажена відповідна бінарна операція, то при виконанні складної операції присвоювання викликаються спочатку відповідна бінарна операція, а потім операція присвоювання.

Приклад перевантаження	Приклад використання

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

2.3. Операції перетворення типів

У мові C# є можливість перевантажувати операції явного та неявного перетворення типів.

Загальне правило перевантаження операції *неявного* перетворення типів:

```
public static implicit operator <цільовий тип>(<Тип> <аргумент>)
{
    <тіло оператора>
}
```

Загальне правило перевантаження операції *явного* перетворення типів:

```
public static explicit operator <цільовий тип>(<Тип> <аргумент>)
{
     <тіло оператора>
}
```

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

Приклад перевантаження	Приклад використання

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

Питання для самоконтролю

1. *Що таке перевантаження операторів?*
2. *Яка загальна форма опису функцій перевантаження унарних операторів?*
3. *Які унарні оператори можна перевантажувати?*
4. *Що слід пам'ятати при перевантаженні унарних операторів?*
5. *Яка загальна форма опису функцій перевантаження бінарних операторів?*
6. *Які бінарні оператори можна перевантажувати?*
7. *Що слід пам'ятати при перевантаженні бінарних операторів?*
8. *Яка загальна форма опису функцій перевантаження явного та неявного перетворення типів?*

Завдання для самостійної роботи

1. Клас "Вектор у R^2" – TVector2D	
поля	▪ для зберігання координат вектора;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення елементів вектора; ▪ визначення довжини вектора; ▪ нормування вектора; ▪ порівняння з іншим вектором; ▪ перевантаження операторів + (додавання векторів), – (віднімання векторів), * (знаходження скалярного добутку).
2. Клас "Прямокутник " – TRectangle	
поля	▪ для зберігання довжин сторін;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі; ▪ визначення периметру; ▪ порівняння з іншим прямокутником; ▪ перевантаження операторів + (додавання відповідних сторін), – (віднімання довжин відповідних сторін), * (множення сторін на деяке число).
3. Клас "Квадрат " – TSquare	
поля	▪ для зберігання довжини сторін;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі; ▪ визначення периметру; ▪ порівняння з іншим квадратом; ▪ перевантаження операторів + (додавання довжин сторін), – (віднімання довжин сторін), * (множення сторін на деяке число).
4. Клас "Трикутник " – TTriangle	
поля	▪ для зберігання довжин сторін;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі; ▪ визначення периметру; ▪ порівняння з іншим трикутником; ▪ перевантаження операторів + (додавання довжин сторін), – (віднімання довжин відповідних сторін), * (множення сторін на деяке число).
5. Клас "прямокутний трикутник " – TPTriangle	
поля	▪ для зберігання довжин катетів;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі; ▪ визначення периметру; ▪ порівняння з іншим трикутником;

	▪ перевантаження операторів + (додавання довжин катетів), – (віднімання довжин відповідних катетів), * (множення сторін на деяке число).
6. Клас “коло” – TCircle	
поля	▪ для зберігання радіуса;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі круга (обмеженого колом); ▪ визначення довжини кола; ▪ порівняння з іншим колом; ▪ перевантаження операторів + (додавання радіусів), – (віднімання радіусів), * (множення радіуса на число).
7. Клас “круг” – TCircle	
поля	▪ для зберігання радіуса; ▪ для зберігання центра кола
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі круга; ▪ перевірка належності точки кругу; ▪ перевантаження операторів + (додавання радіусів), – (віднімання радіусів), * (множення радіуса на число).
8. Клас “одновимірний масив” – TArray	
поля	▪ для зберігання елементів масиву; ▪ для зберігання кількості елементів.
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження найбільшого/найменшого елемента; ▪ сортування масиву; ▪ знаходження суми елементів; ▪ перевантаження операторів + (додавання елементів), – (віднімання елементів), * (множення масиву на число).
9. Клас “матриця ” – TMatrix	
поля	▪ для зберігання елементів матриці; ▪ для зберігання розмірності матриці.
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження найбільшого/найменшого елемента; ▪ знаходження суми елементів.
10. Клас “квадратна матриця ” – TSMatrix	
поля	▪ для зберігання елементів матриці; ▪ для зберігання розмірності матриці.
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження найбільшого/найменшого елемента; ▪ знаходження суми елементів. ▪ перевантаження операторів + (додавання елементів), – (віднімання елементів).

11. Клас “звичайний дріб” – TFraction	
поля	▪ для зберігання чисельника і знаменника;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ скорочення дробів (якщо чисельник і знаменник містять спільні множники); ▪ перевантаження операторів +, –, *, / .
12. Клас “пряма на площині” – TLine2D	
поля	▪ для зберігання коефіцієнтів канонічного рівняння прямої;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження точки перетину з іншою прямою; ▪ визначення належності точки прямій; ▪ перевантаження операторів + (додавання коефіцієнтів прямих), – (віднімання коефіцієнтів прямих).
13. Клас “відрізок на площині” – TInterval2D	
поля	▪ для зберігання координат початку і кінця відрізка;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження точки перетину з іншим відрізком; ▪ визначення довжини відрізка; ▪ визначення середини відрізка; ▪ перевантаження операторів + (утворюється відрізок початок якого є початком першого, а кінець – кінцем другого), * (збільшення довжини відрізка у вказану кількість разів зберігаючи початок відрізка незмінним).
14. Клас “точка на площині” – TPoint2D	
поля	▪ для зберігання довжин сторін;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення відстані до іншої точки; ▪ визначення квадранту, до якого належить точка; ▪ порівняння з іншою точкою; ▪ перевантаження операторів + (додавання відповідних координат), – (віднімання відповідних координат).

3. ІНКАПСУЛЯЦІЯ. ВЛАСТИВОСТІ ТА ІНДЕКСАТОРИ

3.1. Інкапсуляція

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

```
Rectangle r1 = new Rectangle();
r1.a=-2;
r1.b=-9;
```

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

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

```
class Rectangle
{
   // Опис закритих членів класу
   private double a_;
   private double b_;
   public double Get_a() { return a_; } // Зчитування a_
   public void Set_a(double a)          // Встановлення a_
   {
       if (a>0) a_= a;               // Перевірка коректності значення a
   }
```

```
    public double Get_b() { return b_; }// Зчитування b_
    public void Set_b(double b)         // Встановлення b_
    {
        if (b>0) b_= b;               // Перевірка коректності значення b
    }
    public Rectangle (double a, double b)
    {
        // Виклик функцій для встановлення значень закритих полів
        Set_a(a);
        Set_b(b);
    }
}
```

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

```
Rectangle r = new Rectangle(3,8);
r.Set_a(2);   r.Set_b(9);          // Встановлення значень
double s= r.Get_a()*r.Get_b();       // Зчитування значень
```

3.2. Властивості

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

Загальний синтаксис опису властивості:

```
[специфікатор доступу] <тип властивості> <Ім'я властивості>
{
        get
        {
            // тіло функції зчитування значення
            return <результат>;
        }
        set
        {
            // тіло функції встановлення значення
        }
}
```

Як правило, властивість оголошують відкритою (зі специфікатором доступу `public`). Якщо властивість використовується для доступу до деякого закритого поля, то тип властивості повинен співпадати із типом цього поля. Тип властивості не може бути `void`. При описі властивості може бути відсутньою або частина `get` або частина `set`, але не обидві одночасно. Якщо у описі властивості присутня тільки частина `get`, то така властивість

називається *властивістю тільки для читання*. Якщо у описі властивості присутня тільки частина `set`, то така властивість називається *властивістю тільки для запису*. Метод `get` повинен містити оператор `return`. В методі `set` для доступу до значення, яке встановлюється, використовується неявний параметр `value`.

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

```
class  Rectangle
{
    //Опис закритих членів класу
    private  double a_;
    private  double b_;
    public double a
    {
        get { return a_; } //Зчитування a_
        set                //Встановлення a_
        {
           if (value > 0) a_ = value; //Перевірка коректності значення a
        }
    }
    public double b
    {
        get { return b_; } //Зчитування b_
        set                //Встановлення b_
        {
           if (value > 0) b_ = value; //Перевірка коректності значення b
        }
    }
    public Rectangle (double a, double b)
    {
      //Звертання до властивостей для встановлення значень закритих полів
        this.a = a;
        this.b = b;
    }
 }
```

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

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

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

3.3. Індексатори

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

Загальний синтаксис опису індексатора:

```
[специфікатор доступу] <тип індексатора> this[<список індексів>]
{
        get
        {
            //тіло функції одержання значень за індексами
            return <результат>;
        }
        set
        {

            //тіло функції встановлення значень за індексами
        }
}
```

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

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

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

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

Приклад опису одновимірного індексатора	**Приклад звертання до індексатора**

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

Питання для самоконтролю

1. *Що таке інкапсуляція?*
2. *Що таке властивість?*
3. *Який загальний синтаксис опису властивості?*
4. *Чим відрізняються властивості тільки для читання і властивості тільки для запису?*
5. *Що таке індексатор?*
6. *Який загальний синтаксис опису індексатора?*
7. *Чим відрізняються індексатори тільки для читання і властивості тільки для запису?*
8. *Скільки індексів може бути описано в індексаторі?*
9. *Чи є якесь обмеження на тип індексів при описі у індексаторі?*

Завдання для самостійної роботи

Завдання 1

У кожному із варіантів передбачити конструктор класу та метод ToString(), який дозволяє одержати рядкове представлення даних об’єкта. Доступ до даних реалізувати за допомогою властивостей.

1. Створити клас TMoney для роботи з грошовими сумами. Сума повинна зберігатися у вигляді доларового еквіваленту. Реалізувати методи додавання/вилучення грошової маси, вказуючи необхідну суму у гривнях, та визначення курсу долара, при якому сума у гривнях збільшиться на 100. Курс долара зберігати в окремому полі.
2. Створити клас TAngle для роботи з кутами. Кут зберігається як ціле число у градусній мірі. Реалізувати збільшення та зменшення кута на величину, яка задається у радіанах.
3. Створити клас TRational для роботи з раціональними дробами. Число представляється парою цілих чисел. Реалізувати збільшення, зменшення, множення і ділення числа на число, яке задається як дійсне число. Введення та виведення числа реалізувати за допомогою методу ToString (при цьому чисельник від знаменника відділяється символом “/”).
4. Створити клас TDate для роботи із датами у форматі “день.місяць.рік”. Дата представляється структурою із трьома полями. Реалізувати методи збільшення/зменшення дати на певну кількість днів, місяців чи років. Введення та виведення дати реалізувати за допомогою методу ToString.
5. Створити клас TFraction для роботи із десятковими дробами. Число представляється за допомогою двох полів, у яких зберігається ціла та дробова частини числа. Реалізувати методи збільшення та зменшення числа.

6. Створити клас TBankomat, який моделює роботу банкомата. Клас повинен містити поля для зберігання кількості купюр кожного із номіналів від 5 до 200 гривень. Реалізувати методи знаходження максимальної та мінімальної сум, які може видати банкомат, та метод зняття деякої суми.
7. Створити клас TTime для роботи із часом у форматі “години:хвилини”. Час представляється структурою із двома полями. Реалізувати методи збільшення/зменшення часу на певну кількість годин чи хвилин.
8. Створити клас TMan для зберігання інформації про людину з полями: ім’я, вік, стать. Реалізувати методи визначення знаку гороскопа та встановлення того, чи є людина дитиною, юнаком або дорослою людиною.
9. Створити клас TGoods, який характеризує деякий товар на складі. Клас повинен містити поля: назва товару, одиниці виміру, кількість, ціна однієї одиниці. Реалізувати методи визначення сумарної вартості товару та методи збільшення/зменшення кількості товару.
10. Створити клас TBus, який характеризує автобус і містить наступні поля: марка автобуса, вага, кількість місць, кількість пасажирів. Передбачити методи збільшення/зменшення кількості пасажирів та визначення наближеної ваги автобуса з пасажирами без багажу.
11. Створити клас TQuadrangle, який характеризує чотирикутник і містить наступні поля: довжини сторін, кути. Передбачити методи для визначення типу чотирикутника, знаходження периметру та площі.
12. Створити клас TFirm, який характеризує приватне підприємство і містить наступні поля: назва, вид діяльності, кількість працівників, кількість продукції, що може бути виготовлена за один день. Передбачити методи для визначення кількості продукції, яка може бути виготовлена за вказаний період часу та часу, який необхідно для виготовлення вказаної кількості продукції.

Завдання 2

У кожному із варіантів самостійно визначити властивості, необхідні для реалізації вказаного класу та реалізувати відповідні індексатори.

1. Реалізувати клас, що представляє одновимірний масив і містить опис індексатора для доступу до елементів. Передбачити методи введення/виведення, знаходження максимального та мінімального елементів.
2. Реалізувати клас, що представляє матрицю і містить опис індексатора для доступу до елементів масиву. Передбачити методи введення/виведення, знаходження максимального та мінімального елементів.
3. Реалізувати клас, що представляє прямокутник (прямокутник задається довжинами сторін) і містить опис індексатора для доступу до сторін прямокутника (a – перша сторона, b – друга сторона). Передбачити методи введення/виведення, знаходження периметру та площі.
4. Реалізувати клас, що представляє трикутник (трикутник задається довжинами сторін) і містить опис індексатора для доступу до сторін трикутника(a – перша сторона, b – друга сторона, c – третя сторона). Передбачити методи введення/виведення, знаходження периметру та площі.
5. Реалізувати клас, що представляє прямокутник (прямокутник задається координатами вершин) і містить опис індексатора для доступу до сторін прямокутника (a – перша сторона, b – друга сторона). Передбачити методи введення/виведення, знаходження периметру та площі.

6. Реалізувати клас, що представляє трикутник (трикутник задається координатами вершин) і містить опис індексатора для доступу до сторін трикутника(a – перша сторона, b – друга сторона, c – третя сторона). Передбачити методи введення/виведення, знаходження периметру та площі.
7. Реалізувати клас, що представляє багатокутник (багатокутник задається довжинами сторін) і містить опис індексатора для доступу до сторін багатокутника. Передбачити методи введення/виведення, знаходження периметру та площі.
8. Реалізувати клас, що представляє багатокутник(багатокутник задається координатами вершин) і містить опис індексатора для доступу до сторін багатокутника. Передбачити методи введення/виведення, знаходження периметру та площі.
9. Реалізувати клас, що представляє квадратне рівняння і містить опис індексатора, який дозволяє одержувати перший та другий розв'язки (якщо вони існують). Передбачити методи введення/виведення, перевірки існування розв'язків.
10. Реалізувати клас, що представляє рівняння прямої на площині ($Ax + By + C = 0$) і містить опис індексатора для доступу до коефіцієнтів цього рівняння. Передбачити методи введеня/виведення, знаходження точки перетину з іншою прямою та метод перевірки належності точки прямій.
11. Реалізувати клас, що представляє рівняння площини ($Ax + By + Cz + D = 0$) і містить опис індексатора для доступу до коефіцієнтів цього рівняння. Передбачити методи введеня/виведення, знаходження точки перетину з осями координат та метод перевірки належності точки площині.
12. Реалізувати клас, що представляє систему двох рівнянь з двома невідомими

    i. $\begin{cases} a_1 x + b_1 y = c_1, \\ a_1 x + b_1 y = c_1. \end{cases}$

    Передбачити індексатор для доступу до коефіцієнтів першого та другого рівняння (індексатор з двома індексами: перший – номер рівняння, другий – номер коефіцієнта). Передбачити також методи введення/виведення, знаходження розв'язку, перевірки точки на те, чи є вона розв'язком системи рівнянь.
13. Реалізувати клас, що представляє натуральне число. Передбачити індексатор для доступу до цифр цього числа, методи обернення числа, та підрахунку кількості нулів.
14. Реалізувати клас, що представляє арифметичну прогресію. Передбачити індексатор для одержання i-го члена прогресії, методи введення/виведення, знаходження суми n членів цієї прогресії.
15. Реалізувати клас, що представляє геометричну прогресію. Передбачити індексатор для одержання i-го члена прогресії, методи введення/виведення, знаходження суми n членів цієї прогресії.

4. УСПАДКУВАННЯ

4.1. Поняття успадкування

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

Загальний синтаксис опису класу-нащадка:

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

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

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

```
public class Rectangle
{
    private double a_;
    private double b_;
    public double a
    {
        get { return a_; }
        set
        {
            if (value > 0) a_ = value;
        }
    }
    public double b
    {
        get { return b_; }
        set
        {
            if (value > 0) b_ = value;
        }
    }
    public double Perimeter()
    {
        return 2 * (a + b);
    }
}
```

```
public class RParallelepiped : Rectangle
{
    private double c_;
    public double c
    {
        get { return c_; }
        set
        {
            if (value > 0) c_ = value;
        }
    }
    public double Volume()
    {
        return a * b * c;
    }
}
```

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

4.2. Успадкування та конструктори

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

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

```
[специфікатори] <Ім'я класу> ([<спис. форм. парам.>]) : base ([<спис. факт. парам.>])
{
    <тіло конструктора класу-нащадка>
}
```

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

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

```
public class Rectangle
{
  . . . . . .
   public Rectangle(double a, double b)
   {
       this.a = a;
       this.b = b;
   }
}


public class RParallelepiped : Rectangle
{
   . . . . . .
    public RParallelepiped(double a, double b, double c) : base(a, b)
    {
        this.c = c;
    }
}
```

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

4.3. Перевизначення методів

Часто при визначенні класу-нащадка доводиться описувати однойменні дії, які також виконуються у класі предка, але мають інший зміст. Наприклад, розглянемо методи знаходження площі прямокутника (клас-предок `Rectangle`) і площі повної поверхні прямокутного паралелепіпеда (клас-нащадок `RParallelepiped`). Як у класі предка, так і у класі нащадка їх доцільно назвати `Square`. Опис у класі нащадка методу, ім'я якого використане у класі предка, називається *перевизначенням* (*простий*

поліморфізм). При цьому застосовується *механізм приховування*, згідно якого метод, описаний у класі нащадка, “приховує” однойменний метод класу-предка. Та це не означає, що приховані методи стають недоступними. Завжди є можливість звернутися до відкритого чи захищеного прихованого методу класу-предка за допомогою ключового слова `base`:

```
base.<Ім'я прихованого методу класу-предка>([<спис. факт. параметрів>]);
```

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

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

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

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

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

```
public class Rectangle
{
   . . . . . .
    public double Square()
    {
        return a * b;
    }
}

public class RParallelepiped : Rectangle
{
   . . . . . .
    public new double Square()
    {
        return Perimeter() * c + 2 * base.Square();
    }
}
```

4.4. Заборона успадкування

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

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

```
public sealed class RParallelepiped : Rectangle
{
        ...
}
```

Питання для самоконтролю

1. *У чому полягає суть успадкування?*
2. *Що таке клас-предок?*
3. *Що таке клас-нащадок?*
4. *Скільки класів-предків може мати клас?*
5. *Як описати клас-нащадок?*
6. *Що таке перевизначення методів?*
7. *Як перевизначити метод класу-предка?*
8. *Як здійснити доступ до перевизначених методів класу-предка?*
9. *Чи має об'єкт-нащадок доступ до перевизначених методів класу-предка?*
10. *Які члени класу-предка є доступними для класу-нащадка?*
11. *Як описати члени класу-предка, щоб вони були доступними для класу-нащадка і недоступними у програмі-клієнті?*
12. *Як описати члени класу-предка, щоб вони не були доступними як для класу-нащадка, так і для програми-клієнта?*
13. *Як заборонити успадкування від класу?*

Завдання для самостійної роботи

Варіант 1.

1. Описати клас, який містять вказані поля і методи.

Клас "Вектор у R^2" – TVector2D	
поля	▪ для зберігання координат вектора;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення елементів вектора; ▪ визначення довжини вектора; ▪ нормування вектора; ▪ порівняння з іншим вектором; ▪ перевантаження операторів + (додавання векторів), – (віднімання векторів), * (знаходження скалярного добутку).

2. Створити клас-нащадок TVector3D (вектор у R^3) на основі класу TVector2D. Додати третю координату вектора та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 2.

1. Описати клас, який містять вказані поля і методи.

Клас "Квадрат " – TSquare	
поля	▪ для зберігання довжини сторін;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі; ▪ визначення периметру; ▪ порівняння з іншим квадратом; ▪ перевантаження операторів + (додавання довжин сторін), – (віднімання довжин сторін), * (множення сторін на деяке число).

2. Створити клас-нащадок TCube (куб) на основі класу TSquare. Додати метод знаходження об'єму куба та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 3.

1. Описати клас, який містять вказані поля і методи.

Клас "Трикутник " – TTriangle	
поля	▪ для зберігання довжин сторін;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі; ▪ визначення периметру; ▪ порівняння з іншим трикутником; ▪ перевантаження операторів + (додавання довжин сторін), – (віднімання довжин відповідних сторін), * (множення сторін на деяке число).

2. Створити клас-нащадок TTrianglePrizm (пряма призма, в основі якої трикутник) на основі класу TTriangle. Додати метод знаходження об'єму призми та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 4.

1. Описати клас, який містять вказані поля і методи.

Клас “Рівносторонній трикутник ” – TRTriangle	
поля	▪ для зберігання довжини сторін;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі; ▪ визначення периметру; ▪ порівняння з іншим трикутником; ▪ перевантаження операторів + (додавання довжин сторін), – (віднімання довжин відповідних сторін), * (множення сторін на деяке число).

2. Створити клас-нащадок TPiramid (правильна трикутна піраміда) на основі класу TRTriangle. Додати метод знаходження об’єму піраміди та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 5.

1. Описати клас, який містять вказані поля і методи.

Клас “прямокутний трикутник ” – TPTriangle	
поля	▪ для зберігання довжин катетів;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі; ▪ визначення периметру; ▪ порівняння з іншим трикутником; ▪ перевантаження операторів + (додавання довжин катетів), – (віднімання довжин відповідних катетів), * (множення сторін на деяке число).

2. Створити клас-нащадок TPPiramid (прямокутна трикутна піраміда, у якій бічне ребро перпендикулярне до катетів) на основі класу TPTriangle. Додати метод знаходження об’єму піраміди та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 6.

1. Описати клас, який містять вказані поля і методи.

Клас “коло” – TCircle	
поля	▪ для зберігання радіуса;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі круга (обмеженого колом); ▪ визначення довжини кола; ▪ порівняння з іншим колом; ▪ перевантаження операторів + (додавання радіусів), – (віднімання радіусів), * (множення радіуса на число).

2. Створити клас-нащадок TCylinder (циліндр) на основі класу TCircle. Додати метод знаходження об’єму циліндра та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 7.

1. Описати клас, який містять вказані поля і методи.

Клас "коло" – TCircle	
поля	▪ для зберігання радіуса;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі круга (обмеженого колом), площі сектора; ▪ визначення довжини кола; ▪ порівняння з іншим колом; ▪ перевантаження операторів + (додавання радіусів), – (віднімання радіусів), * (множення радіуса на число).

2. Створити клас-нащадок TCone (конус) на основі класу TCircle. Додати поле для збереження висоти конуса, метод знаходження об'єму конуса та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 8.

1. Описати клас, який містять вказані поля і методи.

Клас "коло" – TCircle	
поля	▪ для зберігання радіуса;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі круга (обмеженого колом), площі сектора; ▪ визначення довжини кола; ▪ порівняння з іншим колом; ▪ перевантаження операторів + (додавання радіусів), – (віднімання радіусів), * (множення радіуса на число).

2. Створити клас-нащадок TSphere (сфера) на основі класу TCircle. Додати метод знаходження площі поверхні сфери та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 9.

1. Описати клас, який містять вказані поля і методи.

Клас "круг" – TCircle	
поля	▪ для зберігання радіуса; ▪ для зберігання центра кола
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ визначення площі круга; ▪ перевірка належності точки кругу; ▪ перевантаження операторів + (додавання радіусів), – (віднімання радіусів), * (множення радіуса на число).

2. Створити клас-нащадок TBall (куля) на основі класу TCircle. Додати метод знаходження об'єму кулі та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 10.

1. Описати клас, який містять вказані поля і методи.

Клас “одновимірний масив” – TArray	
поля	▪ для зберігання елементів масиву; ▪ для зберігання кількості елементів.
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження найбільшого/найменшого елемента; ▪ сортування масиву; ▪ знаходження суми елементів; ▪ перевантаження операторів + (додавання елементів), – (віднімання елементів), * (множення масиву на число).

2. Створити клас-нащадок TOderedArray (упорядкований масив) на основі класу TArray. Додати методи додавання та вилучення елементів (перевизначивши оператори додавання та віднімання числа).

3. Створити програму-клієнт для тестування.

Варіант 11.

1. Описати клас, який містять вказані поля і методи.

Клас “матриця ” – TMatrix	
поля	▪ для зберігання елементів матриці; ▪ для зберігання розмірності матриці.
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження найбільшого/найменшого елемента; ▪ знаходження суми елементів.

2. Створити клас-нащадок TOpMatrix (матриця, для якої перевантажено оператори +,–,*) на основі класу TMatrix..

3. Створити програму-клієнт для тестування.

Варіант 12.

1. Описати клас, який містять вказані поля і методи.

Клас “квадратна матриця ” – TSMatrix	
поля	▪ для зберігання елементів матриці; ▪ для зберігання розмірності матриці.
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження найбільшого/найменшого елемента; ▪ знаходження суми елементів. ▪ перевантаження операторів + (додавання елементів), – (віднімання елементів).

2. Створити клас-нащадок TDeterminant2 (визначник квадратної матриці порядку 2) на основі класу TSMatrix. Розробити метод для знаходження визначника.

3. Створити програму-клієнт для тестування.

Варіант 13.

1. Описати клас, який містять вказані поля і методи.

Клас “квадратна матриця ” – TSMatrix	
поля	▪ для зберігання елементів матриці; ▪ для зберігання розмірності матриці.
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження найбільшого/найменшого елемента; ▪ знаходження суми елементів. ▪ перевантаження операторів + (додавання матриць), – (віднімання матриць).

2. Створити клас-нащадок TMSMatrix (клас доповнюється новими методами: транспонування, перевантаження оператора * (множення матриці на матрицю) та перевантаження оператора * (множення матриці на число)) на основі класу TSMatrix. Розробити метод для знаходження визначника.

3. Створити програму-клієнт для тестування.

Варіант 14.

1. Описати клас, який містять вказані поля і методи.

Клас “звичайний дріб” – TFraction	
поля	▪ для зберігання чисельника і знаменника;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ скорочення дробів (якщо чисельник і знаменник містять спільні множники); ▪ перевантаження операторів +, –, *, / .

2. Створити клас-нащадок TMixFraction (мішані дроби – ціла частина, чисельник і знаменник) на основі класу TFraction. Додати поле для збереження цілої частини та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

Варіант 15.

1. Описати клас, який містять вказані поля і методи.

Клас “пряма на площині” – TLine2D	
поля	▪ для зберігання коефіцієнтів канонічного рівняння прямої;
методи	▪ конструктор без параметрів, конструктор з параметрами, конструктор копіювання; ▪ введення/виведення даних; ▪ знаходження точки перетину з іншою прямою; ▪ визначення належності точки прямій; ▪ перевантаження операторів + (додавання коефіцієнтів прямих), – (віднімання коефіцієнтів прямих).

2. Створити клас-нащадок TLine3D (пряма у просторі) на основі класу TLine2D. Додати ще одне поле для збереження коефіцієнта при невідомій z та перевизначити відповідні методи.

3. Створити програму-клієнт для тестування.

5. ПОЛІМОРФІЗМ

5.1. Поняття поліморфізму

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

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

5.2. Складний поліморфізм (віртуальні методи)

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

```
RParallelepiped rp = new RParallelepiped(2, 3, 4);
Rectangle r = rp; // тут діє правило неявного висхідного перетворення типів
```

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

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

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

При перевизначенні віртуального методу у класі-нащадку використовують специфікатор `override`. Використання специфікатора

override не є обов'язковим. За його відсутності метод буде перевизначено як статичний поліморфний.

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

```
public class Rectangle
{
   . . . . . .
    public virtual double Square()
    {
        return a * b;
    }
}

public sealed class RParallelepiped : Rectangle
{
   . . . . . .
    public override double Square()
    {
        return Perimeter() * c + 2 * base.Square();
    }
}
```

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

```
Rectangle r = new Rectangle (3, 5);
RParallelepiped rp = new RParallelepiped(2, 3, 4);
```

Тоді:

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

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

Надамо змінній r значення rp.

```
r = rp;
```

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

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

5.3. Перетворення типів при успадкуванні

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

```
Rectangle r = new Rectangle(3, 5);
RParallelepiped rp1 = new RParallelepiped(2, 3, 4);
r = rp1;
// тут діє правило явного низхідного перетворення типів
RParallelepiped rp2 = (RParallelepiped)r;
```

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

```
Rectangle r = new Rectangle(3, 5);
object o = r;
// тут виникне помилка часу виконання
RParallelepiped rp = (RParallelepiped)o;
```

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

```
<об’єкт> as <ім’я_класу>
<об’єкт> is <ім’я_класу>
```

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

```
Rectangle r = new Rectangle(3, 5);
object o = r;
// тут вже не виникне помилка часу виконання
if (o is RParallelepiped) // або if (o as RParallelepiped != null)
{
    RParallelepiped rp = (RParallelepiped)o;
}
```

5.4. Абстрактні методи

Властивості успадкування та поліморфізму дають можливість визначати клас, який є загальним описанням деякої групи подібних об’єктів реальної

дійсності. Але дуже часто, описуючи такий клас, недоцільно чи навіть неможливо вказати реалізацію деяких спільних для всієї групи методів. У цьому випадку методи описують як *абстрактні* і реалізації для них не вказують.

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

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

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

```
public abstract class Figure // клас абстрактний
{
    private double Data_;
    public double Data
    {
        get{ return Data_; }
        set
        {
            if(value > 0) Data_ = value;
            else Data_ = 1;
        }
    }
    public Figure(double data)
    {
        Data = data;
    }
    public void Show_Data()
    {
        Console.WriteLine("Data={0}", Data);
    }
```

```
        public abstract double Square(); // метод абстрактний (реалізація не наводиться)
    }
    public class Rect : Figure
    {
        public Rect(double a) : base(a)
        {
        }
        public override double Square() // абстрактний метод перевизначається
        {
            return Data * Data;
        }
    }

    public class Circle : Figure
    {
        public Circle(double r) : base(r)
        {
        }
        public override double Square() // абстрактний метод перевизначається
        {
            return Math.PI * Data * Data;
        }
    }
```

Питання для самоконтролю

1. *Що таке поліморфізм?*
2. *Для чого у класах описують віртуальні методи?*
3. *Що таке абстрактні методи?*
4. *Які класи називають абстрактними?*
5. *Чи може клас містити крім абстрактних інші методи?*
6. *Чи можна створити об'єкт абстрактного класу?*
7. *У яких випадках використовують абстрактні класи?*
8. *Чи можна визначити тип об'єкта-нащадка, якщо змінна типу клас-предок містить його адресу?*

Завдання для самостійної роботи

Завдання

У кожному варіанті завдання при описі класів самостійно визначити необхідні поля, властивості та методи вводу/виводу. Деякі методи класу-предка можуть бути віртуальними і абстрактними. У програмі-клієнті для збереження сукупності об'єктів використати масив.

1. Створити клас TPrism, який представляє правильну призму і містить методи для знаходження площі поверхні та об'єму. На основі цього класу створити класи-нащадки TPrism3 та TPrism4, які представляють правильну трикутну та чотирикутну призми. З клавіатури вводиться дані для створення правильної трикутної та чотирикутної призми. На їх основі поступово створити m правильних призм (трикутних та чотирикутних), об'єм кожної з яких на 5 більше попередньої. Для трикутних призм знайти сумарний об'єм, а для чотирикутних – суму площ поверхні.
2. Створити клас TNumber з віртуальними методами для знаходження суми цифр та знаходження першої/останньої цифри. На основі цього класу створити класи-нащадки TIntNumber та TRealNumber, у яких реалізовано перевизначені віртуальні методи. Створити m об'єктів цілих чисел та n об'єктів дійсних чисел (дані згенерувати випадковим чином). Знайти суму перших цифр цілих чисел та суму останніх цифр дійсних чисел.
3. Створити клас TBody, який представляє просторову геометричну фігуру з методами обчислення площі поверхні та об'єму. На основі цього класу створити класи нащадки TParallelepiped та TBall. Випадковим чином створити певну кількість паралелепіпедів та куль, щоб їх сумарна кількість дорівнювала n. Знайти сумарну площу поверхонь усіх геометричних тіл.
4. Створити клас TTriangle з віртуальними методами для обчислення площі та периметру. На основі цього класу створити класи, які представляють рівносторонні, прямокутні та рівнобедрені трикутники. Випадковим чином створити певну кількість трикутників кожного виду, щоб їх сумарна кількість дорівнювала n. Для рівносторонніх та прямокутних обчислити суму площ, а для рівнобедрених – суму всіх периметрів.
5. Створити клас TQuadrangle, який представляє чотирикутник і містить віртуальні методи для обчислення площі та периметру. На основі цього класу створити класи, які представляють прямокутник, квадрат, паралелограм (квадрат створити на основі прямокутника). Випадковим чином створити певну кількість чотирикутників кожного виду, щоб їх сумарна кількість дорівнювала n. Обчислити суму площ прямокутників та квадратів і суму периметрів паралелограмів.
6. Створити клас TMatrix, який представляє матрицю і містить методи для обчислення детермінанта та суми елементів матриці. На основі цього класу створити класи, які представляють квадратні матриці порядку 2, та порядку 3. За допомогою цих класів обчислити вираз

$$S = \left(\sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} \right) + |A| + |B|,$$

де $A = \|a_{ij}\|_1^3$ – матриця порядку 3, а $B = \|b_{ij}\|_1^2$ – матриця порядку 2.

7. Створити клас TVector, який представляє вектор і місить методи для обчислення довжини вектора та скалярного добутку векторів. На основі цього класу створити класи, які представляють вектори з просторів R^2 та R^3. За допомогою цих класів, обчислити значення виразу

$$S = \langle a,b \rangle + \langle c,d \rangle + |a|,$$

де $a,b \in R^3$, а $c,d \in R^2$.

8. Створити клас TVector, який представляє вектор і місить методи для визначення того, чи є інший вектор паралельним/перпендикулярним до нього та метод знаходження довжини вектора. На основі цього класу створити класи, які

представляють вектори з просторів R^2 та R^3. У масиві зберегти 3 двовимірні та 4 тривимірні вектори. Знайти суму довжин паралельних до першого по порядку двовимірних векторів та суму перпендикулярних до першого по порядку тривимірних векторів.

9. Створити клас TEquation, який представляє рівняння і містить віртуальні методи для знаходження коренів рівняння та перевірки чи є деяке значення коренем рівняння. На основі цього класу створити класи нащадки, які представляють лінійні рівняння та квадратні рівняння. Випадковим чином згенерувати дані для n лінійних рівнянь та m квадратних рівнянь. Знайти суму коренів для кожного із видів рівнянь (за умови, що вони існують).
10. Створити клас TSystemLinearEquation, який представляє систему лінійних алгебраїчних рівнянь і містить методи для введення/виведення коефіцієнтів, знаходження коренів та перевірки того, чи є деякий набір чисел розв'язком системи. На основі цього класу створити класи-нащадки, які представляють системи двох та трьох лінійних рівнянь відповідно з двома та трьома невідомими. Випадковим чином згенерувавши дані, знайти розв'язок систем лінійних алгебраїчних рівнянь обох видів.
11. Створити клас TVSeries, який представляє прогресію і містить методи для знаходження n-го члена прогресії і знаходження суми перших n членів цієї прогресі. На основі цього класу створити класи нащадки, які представляють арифметичні та геометричні прогресії. Випадковим чином згенерувати дані для n прогресій (геометрична, арифметична, геометрична, арифметична, і т.д.). Знайти суму перших m членів прогресії, n-товий член якої є найбільшим.
12. Створити клас TTriad, який представляє трійку цілих чисел і містить методи для їх збільшення/зменшення на 1. Реалізувати класи нащадки TDate ("число.місяць.рік") та TTime ("години.хвилини.секунди"). Випадковим чином згенерувати n дат та m об'єктів-часу. Визначити, які із дат мають значення, що є допустимими, якщо їх трактувати як час. Всі інші дати зменшити на 1.
13. Створити клас TPair, який представляє пару чисел і містить методи для їх збільшення/зменшення на 1. Реалізувати класи нащадки TTime ("години.хвилини") та TMoney ("гривні.копійки"). Згенерувати поступово випадковим чином n пар (час, гроші), де час – тривалість виконання роботи, а гроші – вартість однієї хвилини роботи працівників. Обчислити витрати на виконання кожної із робіт.
14. Створити клас TArray, який представляє одновимірний масив і містить методи введення/виведення, збільшення/зменшення всіх елементів на 1 та знаходження середнього арифметичного. Реалізувати класи-нащадки, що представляють одновимірні масиви з елементами цілого та дійсного типів. Випадковим чином створивши m масивів кожного виду, знайти масив, середнє арифметичне елементів якого є найбільшим.
15. Створити клас TLine, що представляє пряму і містить методи для визначення того, чи є інша пряма паралельною/перпендикулярною до неї, чи належить вказана точка прямій. Реалізувати класи-нащадки, що представляє пряму на площині і в просторі. Випадковим чином згенерувати дані для створення n прямих у просторі, та m прямих на площині. Визначити, чи є серед заданих прямих у просторі така, що є перпендикулярною до всіх інших прямих у просторі.

6. ІНТЕРФЕЙСИ

6.1. Поняття інтерфейсу. Опис інтерфейсів

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

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

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

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

```
[специфікатор доступу] interface <Ім'я інтерфейсу> [:<Список інтерфейсів-предків>]
    {
        <тип результату> <ім'я методу 1> ([<спис.форм.парам.>]);
        <тип результату> <ім'я методу 2> ([<спис.форм.парам.>]);
        . . . . . .
    }
```

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

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

```
public interface IMeasurable
{
    double Perimeter();
    double Area();
}

public interface ICircumcircleIncircle
{
     double R { get; } // радіус описаного кола
    double r { get; } // радіус вписаного кола
}
```

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

6.2. Реалізація інтерфейсів

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

```
[специфікатор доступу] class <Ім’я класу>:<Ім’я класу-предка>,
<Ім’я інтерфейсу-предка1>,<Ім’я інтерфейсу-предка2>, ..., <Ім’я інтерфейсу-предкаN>
{
        <реалізація членів інтерфейсу-предка1>
        <реалізація членів інтерфейсу-предка2>
        . . . . . .
        <реалізація членів інтерфейсу-предкаN>
        <опис інших членів класу>
}
```

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

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

```
public <тип результату> <Ім’я методу> ([<спис.форм.парам.>])
{
        <реалізація методу>
}
```

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

```
<тип результату> <Ім'я інтерфейсу>.<Ім’я методу> ([<спис.форм.парам.>])
{
        <реалізація методу>
}
```

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

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

```
public class Square: IMeasurable, ICircumcircleIncircle
{
    private double a_;
    public double a
    {
        get { return a_; }
        set
        {
            if (value > 0) a_ = value;
            else a_ = 1;
        }
    }
    public Square (double a)
    {
        this.a = a;
    }
    // неявна реалізація методу Perimeter інтерфейсу IMeasurable
    public double Perimeter()
    {
        return 4 * a;
    }
    // явна реалізація методу Area інтерфейсу IMeasurable
    double IMeasurable.Area()
    {
        return a * a;
    }
    // неявна реалізація властивості R інтерфейсу ICircumcircleIncircle
    public double R
    {
        get
        {
            return (Math.Sqrt(2) / 2) * a;
        }
    }
    // явна реалізація властивості r інтерфейсу ICircumcircleIncircle
    double ICircumcircleIncircle.r
    {
        get
        {
            return a / 2;
        }
     }
}
```

6.3. Доступ до функціональних можливостей інтерфейсу

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

```
Square s1 = new Square (2); // квадрат зі стороною 2
IMeasurable i1 = (IMeasurable)s1;
Console.WriteLine("Периметр квадрата:{0}", i1.Perimeter());
Console.WriteLine("Площа квадрата:{0}", i1.Square());
```

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

```
Square s1 = new Square (2); // квадрат зі стороною 2
Console.WriteLine("Радіус описаного кола:{0}", s1.R);
```

6.4. Стандартні інтерфейси .NET

Окрім можливості визначення власних інтерфейсів, у бібліотеках середовища .NET пропонується величезний набір стандартних (які поставляються із платформою) інтерфейсів.

6.4.1. Порівняння об’єктів

Інтерфейс `System.IComparable` забезпечує поведінку можливості порівняння об’єктів класу та, як правило, реалізовується у класі з метою визначення порядку на множині об’єктів класу. Його визначення виглядає так:

```
public interface IComparable
{
    int CompareTo(object o);
}
```

Отже, підтримка даного інтерфейсу класом вимагає реалізації у цьому класі методу `CompareTo`. Цей метод повертає результат порівняння двох об’єктів − поточного з об’єктом, який передається в метод в якості параметру. Метод повинен повертати ціле число:

- від'ємне число, якщо поточний об'єкт менший за параметр;
- нуль, якщо поточний об'єкт та параметр одинакові;
- додатне число, якщо поточний об'єкт більший за параметр.

Підтримка інтерфейсу `System.IComparable` може виявитися дуже зручним засобом для забезпечення можливості сортування масивів об'єктів класу. Справа в тому, що базовим класом для всіх масивів у мові C# є клас `System.Array` а, отже, методи цього класу можна застосовувати до будь-якого похідного від нього класу, зокрема і до масиву визначеного користувачем. Серед загальнодоступних методів класу `System.Array` є статичний метод `Sort()`, одне із перевантажень якого `Array.Sort(Array array)` сортує одновимірний масив array за зростанням. Цей метод при порівняння елементів масиву автоматично викликає реалізацію (як явну так і неявну) інтерфейсу `System.IComparable`.

Приклад. Реалізуємо підтримку інтерфейсу `System.IComparable` у класі `Square`. Слід мати на увазі, що при реалізації методу `CompareTo` програміст самостійно визначає *критерій порівняння об'єктів*. У нашому випадку будемо порівнювати квадрати за величиною периметру.

```
public class Square: IMeasurable, ICircumcircleIncircle, IComparable
{
    . . . . . .

    //явна реалізація інтерфейсу System.IComparable
    int IComparable.CompareTo(object o)
    {
        Square s = o as Square;
        if (s != null)
        {
            if (this.Perimeter() < s.Perimeter()) return -1;
            if (this.Perimeter() > s.Perimeter()) return 1;
            return 0;
        }
        throw new ArgumentException("Аргумент не сумісний із класом
Square.");
    }
}
```

Тепер відсортувати масив об’єктів класу `Square` за зростанням можна так:

```
Square [] a = { new Square (3), new Square (4), new Square (1) };
Array.Sort(a);
```

Відсортувати ж масив за спаданням можна обернувши відсортований за зростанням масив за допомогою загальнодоступного статичного методу `Array.Reverse()`:

```
    Array.Reverse(a);
```

6.4.2. Сортування за різними критеріями

У багатьох задачах виникає потреба у можливості сортування набору об’єктів за різними критеріями. У C# для цього використовується інтерфейс `System.Collections.IComparer`.

Інтерфейс `System.Collections.IComparer` містить один метод `Compare`, який повертає результат порівняння двох об’єктів, які передаються йому в якості параметрів:

```
    public interface IComparer
    {
        int Compare(object o1, object o2);
    }
```

Метод повертає ціле число, яке є результатом порівняння двох об’єктів. Це число повинно бути:

- від’ємне, якщо перший параметр менший за другий;
- нуль, якщо параметри одинакові;
- додатне число, якщо перший параметр більший за другий.

Принцип застосування цього інтерфейсу полягає в тому, що для кожного критерію сортування набору об’єктів описується допоміжний клас, який реалізує цей інтерфейс. Об’єкт цього допоміжного класу передається у стандартний метод сортування масивів `Array.Sort()` в якості другого параметру.

Приклад. Наведемо приклад сортування масиву об’єктів класу `Square` за величиною периметра (допоміжний клас `SortByPerimeter`) та за

величиною сторони (допоміжний клас SortByA). Класи критеріїв сортування реалізовані як вкладені, оскільки вони потрібні тільки об'єктам класу Square.

```
public class Square : IMeasurable, ICircumcircleIncircle, IComparable
{
    . . . . . .

    // клас критерію сортування за величиною периметра
    public class SortByPerimeter : IComparer
    {
        //явна реалізація інтерфейсу System.Collections.IComparer
        int IComparer.Compare(object o1, object o2)
        {
            Square s1 = o1 as Square;
            Square s2 = o2 as Square;
            if (s1 == null || s2 == null)
                throw new ArgumentException("Один або більше
аргументів не сумісні із класом Square.");
            return ((IComparable)s1).CompareTo(s2);
        }
    }
    // клас критерію сортування за величиною сторони
    public class SortByA : IComparer
    {
        //явна реалізація інтерфейсу System.Collections.IComparer
        int IComparer.Compare(object o1, object o2)
        {
            Square s1 = o1 as Square;
            Square s2 = o2 as Square;
            if (s1 == null || s2 == null)
                throw new ArgumentException("Один або більше
аргументів не сумісні із класом Square.");
            if (s1.a > s2.a) return 1;
            if (s1.a < s2.a) return -1;
            return 0;
        }
    }
}
```

У цьому випадку відсортувати масив об'єктів класу Square за зростанням за різними критеріями можна так:

```
    Array.Sort(a, new Square.SortByPerimeter());
    Array.Sort(a, new Square.SortByA());
```

6.4.3. Перебір об'єктів та ітератори

Для перебору елементів у спеціальним чином організованій групі даних застосовується оператор foreach. Прикладом такої групи є масив. Зручність

цього різновиду циклу полягає в тому, що не потрібно визначати кількість елементів у групі та виконувати їх перебір за індексом: просто вказується необхідність на перебір всіх елементів групи. Загальний синтаксис цього оператора наступний:

```
foreach(<тип> <ім'я змінної> in <ім'я групи даних) <тіло циклу>
```

Ім'я змінної задає локальну по відношенню до циклу змінну, яка буде по черзі приймати всі значення із заданої іменем групи даних. Тип змінної повинен відповідати типу елементів групи даних. Наприклад, нехай задано масив:

```
int[] a = {24, 50, -18, 3, -7}
```

Вивід цього масиву на консоль з допомогою оператора `foreach` виглядає наступним чином:

```
foreach(int x in a) Console.WriteLine(x);
```

У C# є можливість застосування оператора `foreach` до класу, визначеного користувачем. Для цього потрібно у класі реалізувати інтерфейс `IEnumerable`. Вся справа в тому, що об'єкт, який задається іменем групи даних, має належати до класу сумісного із типом `System.Collections.IEnumerable`.

В інтерфейсі `System.Collections.IEnumerable` (переліковний) визначено один метод – `GetEnumerator()`, який повертає об'єкт типу `System.Collections.IEnumerator` (перелічувач) та який можна використовувати для перегляду елементів об'єкта як групи даних. Інтерфейс `System.Collections.IEnumerator` визначає три методи:

- властивість `Current`, яка повертає поточний елемент об'єкта як групи даних;
- метод `MoveNext()`, що переводить перелічувач на наступний елемент об'єкта як групи даних;
- метод `Reset()`, який встановлює перелічувач на початок перегляду групи даних.

Отже, якщо потрібно, щоб для перебору елементів класу міг застосовуватися цикл `foreach`, необхідно реалізувати всі ці методи. Однак, у

другій версії платформи .NET були введені засоби, які спрощують виконання перебору у об'єкті − *ітератори*.

Ітератор представляє собою блок коду, який визначає правило перебору елементів об'єкта як групи даних. На кожній ітерації циклу `foreach` виконується один крок ітератора, який закінчується видачею наступного значення. Видача значення виконується за допомогою ключового слова `yield`.

Приклад. Опишемо клас `SquaresList` (група (список) квадратів) із підтримкою перебору квадратів у наборі.

```
public class SquaresList : IEnumerable
{
   private Square[] list = { new Rect(3), new Rect(1), new Rect(4), new Rect(1) };

    public IEnumerator GetEnumerator()
    {
        // реалізація ітератора
        for (int i = 0; i < list.Length; i++) yield return list[i];
    }
}
```

Вивід інформації про сторони квадратів, які входять до списку:

```
    SquaresList List = new SquaresList();

    foreach (Square square in List)
         Console.WriteLine(square.a);
```

Все, що потрібно зробити у другій версії платформи .NET для підтримки перебору, − вказати, що клас реалізує інтерфейс `IEnumerable`, та реалізувати ітератор. Блок ітератора синтаксично є звичним блоком коду та може бути описаний в тілі методу, операції або блоці `get` властивості, якщо відповідне значення, що повертається, має тип `IEnumerable` або `IEnumerator`. В тілі блоку ітератора можуть зустрічатися дві конструкції:

- `yield return` − формує значення, яке видається на черговій ітерації;
- `yield break` − сигналізує про завершення ітерації.

Код блока ітератора виконується не так, як звичайні блоки. Компілятор формує службовий об'єкт-перелічувач, при виклику метода `MoveNext()` якого виконується код блока ітератора, який видає чергове значення за

допомогою ключового слова `yield`. Наступний виклик метода `MoveNext()` об'єкта-перелічувача поновлює виконання блока ітератора з того моменту, на якому він був призупинений в попередній раз.

Ще однією перевагою використання ітераторів є те, що для одного і того ж класу можна задати різний порядок перебору елементів. Наприклад:

```
public class SquaresList : IEnumerable
{
   . . . . . .

    public IEnumerable MoreThan3()
    {
        for (int i = 0; i < list.Length; i++)
            if (list[i].a > 3)
                yield return list[i];
    }
}
```

Тепер перебір тільки тих квадратів у списку, у яких сторона більша за 3, може бути реалізований, наприклад, так:

```
    SquaresList List = new SquaresList();

    foreach (Rect square in List.MoreThan3())
        Console.WriteLine(square.a);
```

Питання для самоконтролю

1. *Що таке інтерфейс?*
2. *Що спільного між інтерфейсами та абстрактними класами?*
3. *Чим відрізняються інтерфейси від абстрактних класів?*
4. *Яке загальне правило опису інтерфейсів?*
5. *Чи передбачено для інтерфейсів успадкування?*
6. *Чи можна вказувати для інтерфейсу-нащадка одночасно декілька інтерфейсів-предків?*
7. *Що таке неявна реалізація члена інтерфейсу у класі?*
8. *Що таке явна реалізація члена інтерфейсу у класі?*
9. *Чи обов'язково клас, який реалізує інтерфейс має містити реалізації всіх його методів?*

10. *Як можна здійснювати доступ до функціональних можливостей інтерфейсу?*
11. *У яких випадках у класі варто реалізовувати підтримку інтерфейсу* `System.IComparable`*?*
12. *У яких випадках у класі варто реалізовувати підтримку інтерфейсу* `System.Collections.IComparer`*?*
13. *У яких випадках у класі варто реалізовувати підтримку інтерфейсу* `System.Collections.IEnumerable`*?*
14. *Для чого призначений оператор* `foreach`*?*
15. *Що таке ітератор?*

Завдання для самостійної роботи

Завдання

1. Створити клас, що містить методи знаходження площі, периметра та типу опуклого чотирикутника, який може задаватися довжинами сторін або координатами вершин. Створити відповідні інтерфейси для випадку, коли чотирикутник задається довжинами сторін та координатами вершин.
2. Створити клас, що містить методи знаходження площі, периметра, радіуса вписаного і описаного кола для рівностороннього, рівнобедреного, різностороннього та прямокутного трикутників. Для різного типу трикутників передбачити відповідні інтерфейси.
3. Створити клас, що містить методи визначення паралельності та перпендикулярності прямих на площині, що задані різними способами. Для різних способів задання прямих передбачити відповідні інтерфейси.
4. Створити клас, що містить методи знаходження розв'язків системи двох лінійних рівнянь з двома невідомими та системи трьох лінійних рівнянь з трьома невідомими (коефіцієнти можуть бути задані як десяткові дробів або як раціональні дроби). Для системи двох лінійних рівнянь з двома невідомими та системи трьох лінійних рівнянь з трьома невідомими передбачити відповідні інтерфейси.
5. Створити клас, що містить методи розв'язання квадратного рівняння (знаходження розв'язків розглядати як у R, так і просторі комплексних чисел). Для різних випадків передбачити відповідні інтерфейси.
6. Створити клас, що містить методи додавання, віднімання, множення та ділення раціональних дробів та такі ж методи для роботи з комплексними числами. Для випадку раціональних дробів та випадку комплексних чисел передбачити відповідні інтерфейси.
7. Створити клас, що містить методи для знаходження відсотку від числа, збільшення, зменшення числа на певну кількість відсотків (число може бути десятковим або раціональним дробом). При створенні класу використати відповідні інтерфейси.
8. Створити клас, що містить методи для знаходження суми і різниці чисел, які можуть бути або раціональними дробами, або числами, записаними за допомогою римських цифр. При створенні класу використати відповідні інтерфейси.

9. Створити інтерфейс, що містить опис методів для роботи з векторами (знаходження довжини вектора, суми векторів, множення вектора на число, скалярного добутку) та класи, які містять реалізації методів інтерфейсу у випадку двовимірних та тривимірних векторів.
10. Створити інтерфейс, що містить опис методів для роботи з двовимірними векторами (знаходження довжини вектора, суми векторів, множення вектора на число, скалярного добутку) та класи, які містять реалізації методів інтерфейсу у випадку, коли вектор задано його координатами та координатами початку і кінця.
11. Створити інтерфейс, що містить опис методів для знаходження n-го члена та суми перших n членів прогресії. На основі цього інтерфейсу створити класи, що містять реалізації методів інтерфейсу у випадку арифметичної та геометричної прогресій.
12. Створити інтерфейс, що містить опис методів для знаходження суми цифр числа, а визначення кількості нулів у записі числа. На основі цього інтерфейсу створити класи, що містять реалізації методів інтерфейсу у випадку цілого та дійсного чисел
13. Створити інтерфейс, що містить опис методів роботи з матрицями (знаходження детермінанта, сліду, рангу матриці), та класи, що містять реалізації методів інтерфейсу у випадку квадратних матриць порядку 2 і 3.
14. Створити інтерфейс, що містить опис методів знаходження суми, найбільшого, найменшого елементів. На основі цього інтерфейсу створити класи, що містять реалізації методів інтерфейсу, зберігаючи елементи у одновимірному та двовимірному масивах.
15. Створити інтерфейс, що містить опис методів визначення відстані від точки до площини в R^3 та метод перевірки належності точки деякій площині. На основі цього інтерфейсу створити класи, що містять реалізації методів інтерфейсу у випадку задання площини канонічним рівнянням та за допомогою трьох точок.
16. Створити інтерфейс, що містить опис методів знаходження площі поверхні, площі основи та об’єму правильної піраміди. На основі цього інтерфейсу створити класи, що містять реалізації методів інтерфейсу у випадку трикутної та чотирикутної пірамід.

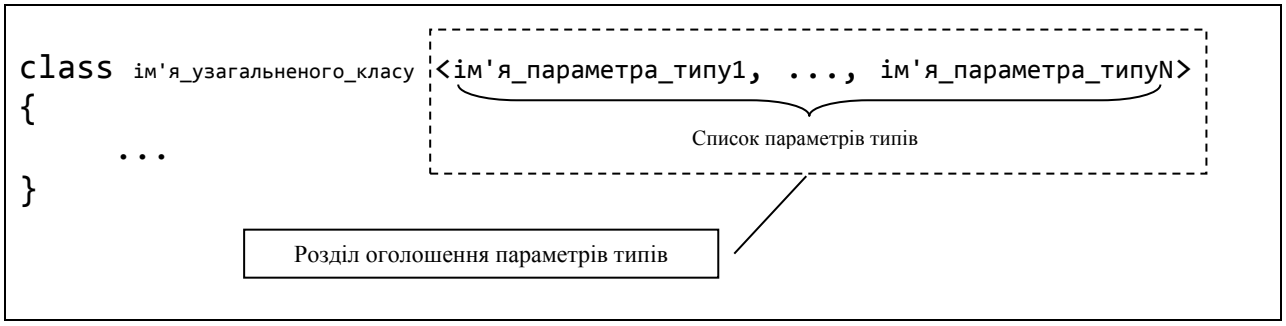
7. УЗАГАЛЬНЕННЯ

7.1. Поняття узагальненого класу

Термін узагальнення, по суті, означає параметризований тип. Особлива роль параметризованих типів полягає в тому, що вони дозволяють створювати класи, структури, інтерфейси, методи та деякі інші члени класу, у яких тип оброблюваних даних вказується у вигляді параметра. За допомогою узагальнень можна, наприклад, створити єдиний клас, який автоматично стає придатним для обробки різнотипних даних. Клас, структура, інтерфейс або метод, що оперує параметризованим типом даних, називається *узагальненим*, як, наприклад, узагальнений клас або узагальнений метод.

Слід відмітити, що у C# є можливість створювати узагальнений код для роботи із різнотипними даними, оперуючи посиланнями типу `object`. Недоліком такого підходу є недотримання безпеки типів. Для перетворення типу `object` у конкретний тип даних необхідне явне приведення типів, яке може бути джерелом помилок часу виконання. Застосування узагальнень, навпаки, забезпечує безпеку типів і тим самим не вимагає явного приведення типів оброблюваних даних. Таким чином, узагальнення розширюють можливості повторного використання коду та дозволяють робити це надійно та просто.

Загальне правило опису узагальненого класу:



Розділ оголошення параметрів типів розташовується після імені узагальненого класу та являє собою кутові дужки (< >), всередині яких міститься список параметрів типів. Ім’я параметру типу − це ідентифікатор, який може бути використаний при оголошенні членів узагальненого класу, визначаючи цим самим місце для підстановки конкретного типу.

Приклад. Розглянемо можливості та переваги використання узагальнень на прикладі класу `Array` (описує одновимірний масив об’єктів довільного типу, який задається параметром типу з іменем T).

```
class Array<T>
{
    // поле для збереження об'єктів довільного типу
    T[] elements;

    // конструктор узагальненого класу
    public Array(params T[] args)
    {
        elements = new T[args.Length];
        args.CopyTo(elements, 0);
    }

    // індексатор для доступу до елементів узагальненого масиву
    public T this[int index]
    {
        set
        {
            if (index < 0 || index >= elements.Length)
              throw new Exception("Індекс елемента виходить за межі масиву.");
            elements[index] = value;
        }
        get
        {
            if (index < 0 || index >= elements.Length)
              throw new Exception("Індекс елемента виходить за межі масиву.");
            return elements[index];
        }
    }
}
```

На основі узагальненого класу можна утворювати (конструювати) різні конкретні класи. Клас, утворений внаслідок підстановки замість параметрів типів конкретних типів, називається *сконструйованим класом (типом)*, а відповідний процес називається *конструюванням класу* на основі узагальненого.

Оскільки сконструйований клас вже є конкретним класом, то можна створювати об’єкти сконструйованих класів. Загальне правило оголошення об’єктів сконструйованих класів наступне:

```
                                  сконструйований клас
ім’я_узагальненого_класу <конкретний_тип1, ..., конкретний_типN> ім’я_об’єкта =
new ім’я_узагальненого_класу <конкретний_тип1, ..., конкретний_типN> ([спис.факт.пар.]);
```

Приклади:

```
// конструювання класу та створення об’єкта для роботи із масивом цілих чисел
Array<int> a1 = new Array<int>(1,2,3,4,5);

// конструювання класу та створення об’єкта для роботи із масивом дійсних чисел
Array<double> a2 = new Array<double>(1.0, 2.0, 3.0, 4.0, 5.0);

// конструювання класу та створення об’єкта для роботи із масивом квадратів
Array<Square> a3 = new Array<Square>(new Square(1), new Square(2));
```

При конструюванні класів на основі узагальнень підставляти можна як конкретний тип (наприклад `int`, `double`, `Square` або `string`), так і інші сконструйовані типи.

Відмітимо, що при різних конкретних типах отримують різні сконструйовані типи. Так, наприклад, типи int та double − це два різних типи, а, отже `Array<int>` та `Array<double>` також два різних типи. Звідси випливає, що неприйнятним є наступне присвоєння:

```
Array<double> s = new Array<int>(1,2,3,4,5);      // Помилка
```

7.2. Обмеження для параметрів типів

При описі узагальненого класу передбачається, що параметр типу може бути довільним. Тому за замовчуванням будь-який параметр типу розцінюється компілятором як тип `object`. Таким чином, при роботі в узагальненому класі з об’єктами параметру типу програміст може використовувати тільки методи, успадковані від класу `object` (`Equals()`, `GetType()`, `ToString()` та ін.). Ця особливість сильно обмежує можливість реалізації деяких методів в узагальнених класах. Нехай, наприклад, в узагальненому класі `Array` виникла необхідність у методі для підрахунку кількості елементів у масиві, які більші за заданий. Реалізація відповідного методу (здавалося б) повинна виглядати наступним чином:

```
public int countGreaterThan(T elem)
{
    int count = 0;
    foreach (T e in elements)
        if (e > elem)  // помилка компіляції
            count++;
    return count;
}
```

Помилка компіляції у цьому методі буде викликана тим, що параметр типу `T` розцінюється компілятором як тип `object`. А для об'єктів типу `object` не визначена операція відношення «<».

Для виходу із подібних ситуацій у C# передбачено обмеження для параметрів типів: вказуючи параметр типу, можна накласти певне обмеження на цей параметр. Це робиться за допомогою оператора `where` при описанні параметрів типів:

```
class ім'я_узагальненого_класу <ім'я_параметра_типу1, ..., ім'я_параметра_типуN>
where ім'я_параметра_типу1: список_обмежень_для_параметра_типу1
. . .
where ім'я_параметра_типуN: список_обмежень_для_параметра_типуN
{
    ...
}
```

Опис обмежень для параметрів типів

Список обмежень для параметра типу складається із вказаних через кому ключових слів, кожне із яких задає певний тип обмеження на відповідний параметр типу. У мові C# передбачено наступні типи обмежень для параметрів типу.

Тип обмеження на параметр типу	**Ключове слово**	**Опис**
Обмеження на клас предка	`Ім'я_класу-предка`	Параметр типу повинен бути класом нащадком від класу `Ім'я_базового_класу`.
Обмеження на реалізацію інтерфейсу	`Ім'я_інтерфейсу`	Параметр типу повинен реалізовувати інтерфейс `Ім'я_інтерфейсу`. Можна задавати декілька інтерфейсів, розділяючи їх комами. Якщо обмеження накладається одночасно на клас предка і на реалізацію інтерфейсу, то першим у списку повинен бути зазначений клас-предок.
Обмеження на належність до типів значень	`struct`	Параметр типу повинен бути структурою. У цьому випадку, всяка спроба використовувати посилальний тип, наприклад `string`, замість параметру типу призведе до помилки під час компіляції.

Тип обмеження на параметр типу	Ключове слово	Опис
Обмеження на належність до типів посилань	`class`	Параметр типу повинен належати до типів посилань. У цьому випадку, всяка спроба використовувати тип-значення, наприклад `int` або `bool`, замість параметру типу призведе до помилки під час компіляції. Слід пам'ятати, що, якщо є декілька обмежень на узагальнення, то обмеження `class` або `struct` повинно бути накладеним першим.
Обмеження на наявність конструктора за замовчуванням	`new()`	Параметр типу повинен мати конструктор за замовчуванням. Використовується у випадку, якщо є необхідність в узагальненому класі створювати екземпляри параметра типу. Слід пам'ятати, що в узагальненні із декількома обмеженнями це обмеження повинно бути вказане останнім.

Приклад. Накладемо у нашому узагальненому класі `Array<T>` обмеження для параметру типу `T` на реалізацію інтерфейсу `IComparable`. Це дасть змогу при реалізації методів цього узагальнення порівнювати об'єкти параметру типу `T`, використовуючи метод `CompareTo()` інтерфейсу `IComparable`. Так, наприклад, реалізація методу для підрахунку кількості елементів у масиві, які більші за заданий буде наступною:

```
class Array<T> where T:IComparable
{
    . . .
    public int countGreaterThan(T elem)
    {
        int count = 0;
        foreach (T e in elements)
            if (e.CompareTo(elem) > 0)
                count++;
        return count;
    }
}
```

7.3. Узагальнені методи

Неузагальнений клас може містити окремі узагальнені методи. Узагальнені методи визначаються в такий спосіб:

```
тип_результату ім’я_функції<список параметрів типу>([список форм.парам.])
[опис обмежень для параметрів типів]
{
    . . .
}
```

У якості типу результату може бути вказаний як конкретний тип, так і один із типів параметрів зі списку параметрів типів. Так само у списку формальних параметрів узагальненої функції можуть бути описані параметри як із зазначенням конкретного типу даних, так і одного із типів зі списку параметрів типів. Також при описі узагальненої функції можна накладати обмеження на параметри типів. Як правило, узагальнені методи реалізовують у неузагальнених класах як статичні члени цього класу.

Приклад. Опишемо звичайний (неузагальнений) клас `Sorter`, що містить метод сортування одновимірних масивів простим вибором. Для того щоб можна було використати цей метод для сортування масивів із різними типами елементів, його варто визначити як узагальнений.

```
class Sorter
{
    // сортування простим вибором
    public static void SelectionSort<T>(T[] array) where T:IComparable
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            int index = i;
            for (int j = i + 1; j < array.Length; j++)
                if (array[j].CompareTo(array[index]) < 0)
                    index = j;

            T t = array[index];
            array[index] = array[i];
            array[i] = t;
        }
    }
}
```

Виклик узагальненого методу неузагальненого класу здійснюється із зазначенням конкретного типу елементів масиву. Наприклад:

```
        int[] a = { 2, 5, 6, 1, 3, 4 };
        Sorter.SelectionSort<int>(a);
```

7.4. Недолік обмеження операцій

При розробці узагальнень слід пам'ятати, що обмеження операцій в існуючих версіях мови C# не підтримуються. Це означає, що наступне обмеження є помилкою.

```
// Недопустиме узагальнення!
public class AddOperation<T> where T : operator +
{
    public T Add(T arg1, T arg2)
    {
      return arg1 + arg2;
    }
}
```

Необхідність такого роду обмежень (коли в узагальненні виникає потреба у використанні операцій додавання, множення, віднімання, ділення, тощо для об'єктів-змінних параметру типу) на практиці виникає досить часто. Наприклад, така необхідність буде виникати при реалізації багатьох методів нашого узагальнення `Array<T>`. Зокрема, операцію додавання над об'єктами типу `T` вимагає метод обчислення суми елементів масиву.

Та все ж таки бажаного результату у цьому випадку можна досягти. Наприклад, для цього ми можемо описати узагальнений інтерфейс, який визначає зазначену операцію (або при потребі декілька операцій), і потім вказати для відповідного параметру типу обмеження на підтримку цього інтерфейсу. Проілюструємо цей підхід на прикладі реалізації методу `Sum()` знаходження суми елементів масиву у нашому узагальненні `Array<T>`.

Спочатку опишемо узагальнений інтерфейс, який задає можливість визначати суму двох аргументів деякого загального типу `T`.

```
interface IAddOperation<T>
{
    T Add(T a, T b);
}
```

Тепер в узагальненні `Array<T>` накладемо обмеження для параметру типу `T` на підтримку інтерфейсу `IAddOperation<T>`. Після цього ми можемо реалізувати метод `Sum()` із використанням методу `Add()`:

```
class Array<T> where T: IComparable, IAddOperation<T>
{
    . . . . . .
    public T Sum()
    {
        T s = default(T);
        foreach (T e in elements)
            s = s.Add(s, e);
        return s;
    }
}
```

Ключове слово `default` в узагальненнях використовується для встановлення значень за замовчуванням для параметра типу. Замовчування для параметрів типу наступні:

- значення за замовчуванням числових величин дорівнює нулю;
- типи посилань мають значення за замовчуванням `null`;
- поля структур встановлюються у 0 (для типів значень) або в `null` (для типів посилань).

Із опису класу `Array<T>` випливає, що замість параметру типу `T` можуть бути використані тільки ті класи, які реалізують інтерфейси `IComparable` та `IAddOperation<T>`. Зокрема з цієї причини ми наразі не можемо сконструювати на основі узагальнення `Array<T>` тип для роботи із масивом цілих чисел. Але ми можемо реалізувати для типу `int` клас-оболонку, який би підтримував відповідні інтерфейси.

```
struct Integer: IComparable, IAddOperation<Integer>
{
    public int value;
    public Integer(int v)
    {
        value = v;
    }
    public int CompareTo(object o)
    {
        Integer x = (Integer)o;
        if (value > x.value) return 1;
        if (value < x.value) return -1;
        return 0;
    }
    public Integer Add(Integer a, Integer b)
    {
        return new Integer(a.value + b.value);
    }
}
```

Тепер розроблене нами узагальнення `Array<T>` може бути використане для роботи із даними типу `Integer`, зокрема для знаходження суми елементів:

```
Array<Integer> a = new Array<Integer>(new Integer(1) ,new Integer(2), new Integer(3));

Console.Write(a.Sum().value);
```

Питання для самоконтролю

1. *Що таке узагальнений клас?*
2. *Яке загальне правило опису узагальненого класу?*
3. *Що таке сконструйований тип?*
4. *У яких випадках використовують обмеження для параметрів типів?*
5. *Яке загальне правило опису обмежень для параметрів типів?*
6. *Які типи обмежень для параметрів типів передбачено у мові C#?*
7. *Що таке узагальнений метод?*
8. *Яке загальне правило опису узагальненого методу?*
9. *Як обійти недолік обмеження операцій?*
10. *Яка роль ключового слова default?*

Завдання для самостійної роботи

Реалізувати узагальнення класів у завданнях для самостійної роботи розділів 4 та 5.

ЛІТЕРАТУРА

1. *Павловская Т.А.* C#. Программирование на языке высокого уровня. Учебник для вузов / Т.А. Павловская – СПб.: Питер, 2007. – 432 с.
2. *Голуб Б.М.* C#. Концепція та синтаксис. Навч. посібник / Б.М. Голуб, – Львів: Видавничий центр ЛНУ імені Івана Франка, 2006. – 136 с.
3. *Троелсен Э.* Язык программирования C# 2010 и платформа .NET 4.0, 5-е изд. / Э Троелсен. – М.: ООО “И.Д. Вильямс”, 2011. – 1392 с.
4. *Шилдт Г.* C# 4.0: полное руководство / Г. Шилдт. – М.: ООО “И.Д. Вильямс”, 2011. – 1056 с.
5. *Иванова Г.С.* Объектно-ориентированное программирование: Учеб. для вузов / Г.С Иванова, Т.Н. Ничушкина, Е.К. Пугачев / Под ред. Г.С. Ивановой. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2001. – 320 с.
6. C++. Объектно-ориентированное программирование. Задачи и упражнения. / В.В. Лаптев, А.В. Морозов, А.В. Бокова. – СПб.: Питер, 2007. – 288 с.

ЗМІСТ

Відповідальний за випуск: завідувач кафедрою системного аналізу і теорії оптимізації к.ф.-м.н., доц. Кузка О.І.

Автори: к. ф.-м. н., доц. Брила А.Ю.,
к. ф.-м. н., Антосяк П.П.,
к. ф.-м. н., Глебена М.І.,
к. ф.-м. н., доц. Чупов С.В.,
к. т. н., доц. Семйон І.В.

Рецензенти: к.ф.-м.н., доц. Мич І.А.,
к. ф.-м.н., доц. Погоріляк О.О.

ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ У C#

Методичні вказівки до лабораторних робіт для студентів І-го курсу математичного факультету спеціальності "Прикладна математика"