Лабораторна робота № 3

Тема: Реалізація класів в Java

Мета роботи: Вивчити принципи створення та реалізації класів

Будь-яка програма на Java має клас, що утворює додаток. Такий клас виконує роль певного контейнеру, в якому повинен бути щонайменше один метод, за допомогою якого запуститься виконання програми. Але фундаментальною одиницею програмування в Java слугують класи, що вміщують поля, методи для виконання операцій з цими полями, а також можуть дозволяти утворювати екземпляри класів або іншими словами - об’єкти. Такі класи є створюваними типами даних, структура яких визначається полями класу, а методи класу реалізують певні функції або операції над даними такого типу. У найпростішому випадку, якщо рішення певної задачі може бути спрощено завдяки використанню певного поєднання декількох змінних, то такі змінні можна об’єднати саме у класі в якості його полів, а також можна додати декілька методів, що реалізують або певну модель обробки цих полів, або розширюють їх використання. Наприклад, у такій задачі, як запис студента на курс за вибором, було б складно окремо утримувати і зв’язувати між собою імена, прізвища та по-батькові студентів, а алгоритми обробки таких записів були б мало прозорими. Якщо утворити клас із відповідними полями ім’я, прізвище та по-батькові, то на його основі вже можна створювати окремі екземпляри або певні масиви цих екземплярів, наприклад, тих, хто записався на курс. Обробка таких даних стає прозорою, бо виконується над одиницею сукупності даних, що пов’язані з одним реальним об’єктом — студентом. Безумовно прикладів зручності об’єктного підходу щодо зберігання та обробки інформації можна навести безліч, але в Java об’єктний підхід розширено і на принцип створення програмного забезпечення.

Мова програмування Java, як об’єктно-орієнтована мова програмування, має засоби для побудови класів та об’єктів. Об’єкти в Java оголошуються за допомогою імен класів подібно оголошенням змінних за певним типом. Тому об’єкти при деяких обставинах можна вважати змінними, що мають тип, визначений іменем класу. Але, на відмінну від змінних базових типів, при оголошені об’єкти не утворюються. У програмі резервується ім’я об’єкту і очікується зв’язування його з адресою локалізації об’єкту - посиланням. Зв’язування відбувається тільки при створені об’єкту або при застосовуванні оператору привласнення “=”, що зв’яже ім’я об’єкту із вже існуючим об’єктом того ж самого класу або його похідних. Утворені об’єкти знаходяться в області системної пам’яті, що зветься *кучею (heap)*. Якщо посилання не відноситься ні до одного об’єкту, то воно має значення null.

Відмінність між самим об’єктом та посиланням на нього є технічною стороною Java, тому у ряді питань акцент на цьому може не робитись. Наприклад, передавання об’єкту до методу, або повернення об’єкту методом може не деталізуватись через механізм зміни посилань. Але удосконалення алгоритмів обробки масивів, колекцій, тощо безумовно будуть потребувати розуміння технічної реалізації на Java взаємодії з об’єктами.

Кожний клас складається із полів та методів. Це не означає, що клас обов’язково повинен вміщувати і поля, і методи. В Java навіть можна створити клас, в якому у явному вигляді не буде не полів, не методів:

```
public class NewClass {

}
```

І такий клас також можна використовувати, хоча зрозуміло, що і його використання, і сенс від такого використання є занадто обмеженими:

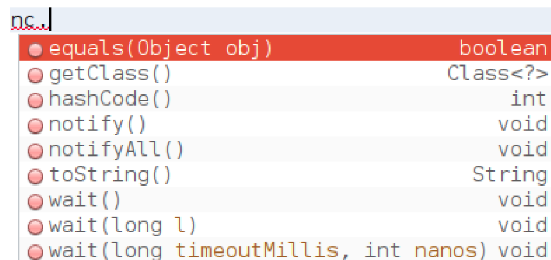
```
NewClass nc = new NewClass();
System.out.println(nc.getClass());
```

Із наведеного видно, що новий клас створюється за допомогою зарезервованого слова *class* та обов'язково повинен мати ім'я та блок із дужок {}. Звичайне оголошення та утворення екземпляру класу — об'єкту, виконується у одному рядку, подібно до наведеного:

```
NewClass nc = new NewClass();
```

де `NewClass` — ім'я класу; `nc` — ім'я об'єкту; `new` — оператор, за допомогою якого утворюється об'єкт; `NewClass()` — спеціальний метод класу — конструктор, який застосовується при утворені об'єкту.

Чому створюється об'єкт класу, якщо ні полів, ні методів не було задано?В Java навіть "пустий" клас реально не є пустим, бо він автоматично наслідує клас Object з пакету java.lang. Створений у прикладі об'єкт *nc* наслідує від класу Object наступні методи:



Саме метод `getClass()` , отриманий від класу Object було використано у наведеному вище прикладі.

Конструктор класу може утворюватись автоматично при компіляції коду класу, якщо в ньому в явному вигляді не задано ні одного конструктору. Конструктор завжди має те саме ім'я, що і клас — це обов'язкова вимога. Конструктор має обмежене використання — під час утворення об'єкту. Якщо в класі не було задано конструктор, то буде створено конструктор за замовчуванням. Такий конструктор не має аргументів для ініціалізації полів і їх ініціалізація відбувається за загальною логікою компілятора: числові поля отримують значення 0, поля типу boolean — false, а інші поля — значення "null". Наприклад, для достатньо простого класу, який призначено для представлення точки на плоскості:

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

компілятором буде створено конструктор за замовчуванням, який при створені нових об'єктів цього класу буде ініціалізувати поля x та y значенням 0. У класі Point є тільки поля x та y, які будуть мати значення відповідних декартових координат точки на плоскості, але відсутні методи. У такому класі важливу роль відіграє модифікатор доступу до полів: public. Він дозволяє прямий доступ на читання та зміну значень цих полів у об'єктів класу Point. З математичної точки зору значення декартових координат не мають явних обмежень. Як за координатою x, так і за координатою y значення можуть змінюватись від -∞ до +∞. Але для багатьох практичних задач навіть координати точки на плоскості можуть мати обмеження, які слід контролювати. Саме тому, для класів, на основі яких утворюються об'єкти, застосовується одна із основних парадигм об'єктно-

орієнтованого програмування — інкапсуляція. Відповідно до цієї парадигми клас вміщує не тільки поля, а і методи для їх обробки. Суттєву роль у реалізації цієї парадигми грають модифікатори доступу, оскільки регулюють доступ як до полів, так і до методів. Найпоширеніший підхід — усім полям класу призначається модифікатор доступу private, що забороняє прямий доступ до них іззовні, а доступ до полів реалізується через get- та set- методи. Інколи їх називають гетери та сетери. Якщо потрібно реалізувати отримання значення певного поля, то створюється get-метод, ім'я якого комбінується з префікса get та імені поля з великої літери. Якщо потрібно реалізувати зміну значення певного поля, то створюється set-метод, ім'я якого комбінується з префікса set та імені поля з великої літери. Зазвичай get- та set- методи отримують модифікатор доступу public.

Наприклад, наведений клас Point для більшої відповідності парадигмі інкапсуляції може бути представлений наступним чином:

```
public class Point {
    private double x,y;
    public void setX(double x) {
        this.x = x;
    }

    public void setY(double y) {
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }
}
```

Задача get-методів доволі проста — їм необхідно повернути значення відповідних полів. Тому тип цих методів такий самий, як і тип полів, а в тілі методу тільки один оператор return. Зазвичай такі методи не мають аргументів. Set-методи мають один аргумент, тип і ім'я якого співпадають з відповідним полем. Конфлікт іменування не виникає, тому що поле прив'язано до об'єкту, для позначення якого використовується слово this. Так позначається об'єкт, у якого викликається відповідний set-метод. Методи get- та set- знаходяться всередині класу, тому вони мають прямий доступ до будь-яких полів цього класу.

Зазвичай поля класу використовуються для надання об'єктам своїх певних характеристик. Наприклад, у наведеному класі Point поля x та y слугують для утримання значень декартових координат точки і різні об'єкти цього класу можуть мати різні значення цих полів. Щоб підкреслити належність таких змінних об'єктам їх можуть називати змінні екземпляру або змінні об'єкту. Але інколи виникає необхідність спільного використання певного поля усіма об'єктами класу. Наприклад, масив об'єктів класу Point може використовуватись для відображення якоїсь фігури і необхідно забезпечити перенесення цієї фігури між різними вікнами або фрагментами користувацького інтерфейсу. Дані про таке вікно або фрагмент можуть зберігатись як контекст для об'єктів. Якщо такий контекст зберігається разом з об'єктом у деякому полі, то його зміна буде потребувати обробки усіх цих об'єктів і виконання однієї і тієї самої операції за їх кількістю. Тобто ця операція не буде достатньо швидкою. Одне із рішень — це додати до класу спеціальне спільне поле, яке не буде переноситись та копіюватись до об'єкту, а буде одним для об'єктів цього класу. Таке поле зветься *змінною класу*.

Для опису змінної класу відповідне поле оголошується з ключовим словом static, тому такі поля також звуться *статичними*. Нижче наведено приклад класу Point зі статичним полем content:

```
public class Point {
    static public String content;
    private double x,y;
    public void setX(double x) {
        this.x = x;
    }
    public void setY(double y) {
        this.y = y;
    }
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }
```

Оскільки поле має модифікатор доступу public, то звернення до нього може бути безпосереднім. На відміну від значень полів об’єкту змінити значення змінної класу можна або через звернення до цього поля через будь-який екземпляр цього класу, або через звернення до цього поля з використанням імені самого класу:

```
public static void main(String[] args){
            Point[] A = new Point[10];
            for(int i=0; i<10; i++)
               A[i] = new Point();
            for(int i=0; i<10; i++){
                A[i].setX(Math.random()*100);
                A[i].setY(Math.random()*100);
            }
            for(Point x: A){
                System.out.println(x.getX()+"  "+x.getY());
            }
            System.out.println("content Point = "+Point.content);
            A[0].content = "first";
            System.out.println("content Point = "+Point.content);
            System.out.println("content a[0] = "+A[0].content);
            System.out.println("\n");
            Point[] B = new Point[10];
            for(int i=0; i<10; i++)
               B[i] = new Point();
            for(int i=0; i<10; i++){
                B[i].setX(Math.random()*1000);
                B[i].setY(Math.random()*1000);
            }
            for(Point x: B){
                System.out.println(x.getX()+"  "+x.getY());
            }
            System.out.println("content Point = "+Point.content);
            System.out.println("content b[0] = "+B[0].content);
        }
```

Фрагмент результату виконання цього коду:

```
92.8237589118521  15.882511265528965
19.44987428351207  11.257481919926137
content Point = null
content Point = first
content a[0] = first


648.4460847369868  686.8081753455601
741.7179884630832  390.39697336240954
. . . . . . . . . . . . . . . . . . .
523.9158270976895  8.92318355340782
content Point = first
content b[0] = first
```

У наведеному прикладі створюється масив A об'єктів класу Point, який ініціалізується випадковими значеннями з використанням методу Math.random(). Слід зауважити на особливість утворення масиву об'єктів певного класу — спочатку утворюється масив за типом цього класу з необхідними параметрами, а потім, з використанням необхідного конструктору, утворюється об'єкти цього класу, які зв'язуються з елементами масиву. У дійсності масив створюється з елементами, що будуть мати посилання на об'єкти. Тобто масив не вміщує об'єкти, а лише посилання на них. Саме тому і є необхідним другий етап — утворення самих об'єктів класу і ініціалізація елементів масиву посиланнями на ці об'єкти.

У прикладі перше звернення до статичного поля content зроблено через клас: `Point.content`. Це поле не було ініціалізовано — тому повертає значення null. У наступному рядку коду поле отримує значення через екземпляр класу A[0]: `A[0].content = "first"`. Тепер змінна класу `content` має значення і звернення до неї у будь-який спосіб буде повертати одне і те саме значення. Більш того, наступне створення нового масиву екземплярів цього класу не буде впливати на її значення — для елементів цього масиву це поле залишається тим самим і тому має значення `"first"`.

Статичні поля класу утримують загальне значення тільки для об'єктів класу однієї програми. Програмний код класу є частиною байт-коду програми після компіляції, тому створення екземплярів цього класу у будь-якій її частині залишає спільне статичне поле класу. Це залишається справедливим навіть у випадку, коли клас із статичним полем використовується в інших класах із яких потім утворюються об'єкти:

```
public class Example {
public static void main(String[] args){
            NewClassA A1 = new NewClassA(2);
            System.out.print("\n");
            NewClassB B1 = new NewClassB(3);
        }
}

public class NewClassA {
    Point[] A;
    NewClassA(int n){
        A = new Point[n];
        for(int i=0; i<n; i++)
            A[i] = new Point();
        for(int i=0; i<n; i++){
            A[i].setX(Math.random()*100);
            A[i].setY(Math.random()*100);
        }
        for(Point x: A){
            System.out.println(x.getX()+"  "+x.getY());
```

```
        }
        A[0].setContent("first");
        System.out.println("content a[0] = "+A[0].getContent());
    }
}

public class NewClassB {
    Point[] B;
    NewClassB(int n){
        B = new Point[n];
        for(int i=0; i<n; i++)
            B[i] = new Point();
        for(int i=0; i<n; i++){
            B[i].setX(Math.random()*1000);
            B[i].setY(Math.random()*1000);
        }
        for(Point x: B){
            System.out.println(x.getX()+"  "+x.getY());
        }
        System.out.println("content b[0] = "+B[0].getContent());
    }
}
```

Приклад результату роботи цієї програми показує, що значення статичної змінної класу Point, яке задається у конструкторі класу NewClassA через `A[0].setContent("first")`, залишається тим самим і для екземплярів цього класу, що утворюються в конструкторі іншого класу NewClassB:

```
89.19438989861716  51.488208373490906
64.06095882456681  38.613636502690255
content a[0] = first

37.5012476798019  235.509427523239
98.81548255880779  673.7726785695329
496.9017607518552  517.7609791723806
content b[0] = first
```

Таким чином немає значення де саме у коді однієї програми утворюються екземпляри класу, в якому є статичне поле — змінна класу, її значення буде одним і тим самим для усіх екземплярів. Важливим є тільки час, коли змінюється значення такого поля і коли воно використовується. Обробка статичних полів JVM відрізняється від обробки полів екземплярів класу. При утворені екземпляру класу JVM виділяє необхідну пам’ять для усіх його полів і звільняє цю пам’ять за допомогою так званого збирача сміття, якщо в програмі зникають усі посилання на цей екземпляр класу навіть, якщо програма продовжує працювати. Для статичних полів класу пам’ять виділяється при запуску програми, в якому цей клас використовується, і звільняється тільки після зупинки програми.

Методи та конструктори класів

Кількість аргументів у методі може бути фіксованою або змінною. Також аргументи можуть бути відсутніми, як це було у деяких наведених раніше прикладах. Якщо в якості аргументу задається масив, то метод отримує не набір елементів, а посилання на них, що для деяких задач спрощує реалізацію обробки даних. При передаванні даних через аргументи може бути виконано автоматичне приведення типів. Наприклад, якщо аргумент у методі має тип double, то при його виклику допускається значення як типу double, так і інших числових типів.

Кількість даних, що передається методу, може бути наперед невідомою або може залежати від умов задачі. Для передавання таких даних методу можна використати масив або так званий аргумент змінної довжини:

```
public class NewClassC {
    public static void outprint(Point ...p){
        for(Point x: p){
            System.out.println(" x="+x.getX()+"  y="+x.getY());
        }
    }
    public static void outprint2(Point[] A){
        for(Point x: A){
            System.out.println(" x="+x.getX()+"  y="+x.getY());
        }
    }
}
```

У класі `NewClassC` є два методи, що виконують одне і те саме, але з різними типами аргументів: `outprint(Point ...p)` використовує аргумент змінної довжини, який позначається як `“...p”` , а `outprint2(Point[] A)` використовує масив. Обидва методи дозволяють вивести значення елементів масиву:

```
NewClassC.outprint(A);
NewClassC.outprint2(A);
```

де A — масив об’єктів класу Point.

Результат в обох випадках буде одним і тим самим, тому що реалізація методів одна і та сама. Різниця в методах полягає в тому, що метод `outprint(Point ...p)` можна використовувати для виведення окремих екземплярів класу Point, наприклад:

```
Point q1 = new Point();
Point q2 = new Point();
Point q3 = new Point();
q1.setX(1);
q1.setY(1);
q2.setX(2);
q2.setY(2);
q3.setX(3);
q3.setY(3);
NewClassC.outprint(q1,q2,q3);
```

Сукупність та тип параметрів, а також ім’я методу утворюють сигнатуру методу. В одному класі може бути декілька однойменних методів, що відрізняються сигнатурою та мають певну специфіку своєї реалізації. Це зветься перевантаженням методу. З точки зору використання це різні методи, але тому, що усі вони мають однакове ім’я, то про них говорять як про один метод. Відносно функціональності різних варіантів перевантаженого методу, то при відсутності формальних обмежень загальноприйнятним є реалізація одного засобу. Наприклад, перевантажений метод призначено для відображення віконних елементів, а відмінність у сукупності аргументів його варіантів дозволяє підбирати найбільш відповідний з них у певному випадку — можливість задавати геометричні розміри елементів, колір, положення, робити ієрархічне зв’язування об’єктів. З технічної точки зору варіанти перевантаженого методу можуть мати не тільки різні реалізації чи алгоритми, але взагалі різну функціональність. Механізм обирання методу за його сигнатурою є достатньо потужним, але зміна функціонального призначення у межах одного перевантаженого методу є небажаною. Зрештою, це може уповільнити

застосування такого методу розробниками, тому що слід пам'ятати, яке поєднання аргументів відповідає тій чи іншій його функціональності.

Особливе місце серед методів класу займає конструктор. Конструктор, подібно іншим методам класу, може мати аргументи, або бути без них. Конструктори також можуть бути перевантаженими. Кількість конструкторів, їх варіанти, як правило, обмежується призначенням класу. Конструктор, який створюється за замовчуванням, не має аргументів, тобто його сигнатура найпростіша. Якщо у класі є опис принаймні одного конструктору (з аргументами або без них), то конструктор за замовчуванням не створюється. Якщо для створення об’єктів будуть необхідними як конструктори з аргументами, так і без них, то конструктор без аргументів доведеться явно описати.

Розглянемо приклад перевантаження конструктору для класу, що дозволяє представити правильний дріб. Створимо клас Fraction:

```
public class Fraction {
    int a;
    int b;
    double c;
    Fraction(int a, int b){
        this.a = a;
        this.b = b;
        this.c = (double) this.a/this.b;
    }
}
```

Екземпляри цього класу утримують чисельник у змінній *a* та знаменник у змінній *b* цілочисельного типу int. Також екземпляри мають поле *c*, що відповідає десятковому значенню дробу. Це поле типу double. Клас має один конструктор `Fraction(int a, int b)`, за допомогою якого будь-який дріб можна представити через визначення чисельника та знаменника, що достатньо відповідає математичним визначенням. Але цілі числа також є правильними дробами із знаменником 1. За допомогою конструктору `Fraction(int a, int b)` екземпляри класу `Fraction` , що представляють цілі числа, будуть утворюватись з чисельником, що має значення самого числа та знаменником 1. Для зручності використання класу і утворення його екземплярів такого типу до класу можна додати наступний конструктор:

```
    Fraction(int a){
        this.a = a;
        this.b = 1;
        this.c = (double) this.a;
    }
```

або задля зменшення повторюваності коду у вигляді

```
    Fraction(int a){
        this(a,1);
    }
```

Екземпляр класу може надаватись методу в якості аргументу, а також екземпляр класу може повертатись методом як результат. Якщо методу необхідно отримати в якості аргументу екземпляр певного класу, то це робиться подібно до аргументів вбудованих типів, але замість типу вказується ім’я цього класу. Якщо необхідно, щоб метод в якості результату повертав екземпляр певного класу, то перед іменем методу замість типу вказується ім’я цього класу, а також використовується оператор return з екземпляром цього класу. У дійсності, і при передаванні аргументу до методу, і при повернені

результату методу будь-які операції виконуються не з самими екземплярами класу, а з відповідними посиланнями на них.

Передавання аргументів методам може здійснюватись або за значенням, або за посиланням. При передаванні аргументу за значенням метод отримує копію аргументу, тому всі операції в методі виконуються саме з копією і зовні аргумент зберігає своє первинне значення. Якщо аргумент передається за посиланням, то всі операції в методі виконуються з аргументом. У Java змінні базових типів передаються за значенням, а екземпляри класів — за посиланням, що пов’язано із способом створення екземплярів класів і реальним зв’язком екземпляру класу та об’єктної змінної. Передавання методу екземпляру класу відбувається за його ім’ям, з яким пов’язано посилання на об’єкт. Саме це значення і передається методу. У базових типів з ім’ям змінної пов’язано значення самої змінної, тому це значення і передається методу. Тобто реальний механізм передавання методу аргументу залишається тим самим — ім’я зв’язано з певним значенням і це значення передається методу, тільки у випадку зі змінною базового типу це значення самої змінної, а у випадку з екземпляром класу — посилання на нього.

Крім статичних полів у класі також можна оголосити статичні методи. Такі методи є методами класу, а не об’єктів на його основі. Статичний метод може оперувати або зі статичними змінними класу, або із змінними, що оголошуються і утворюються всередині такого методу. Поля об’єктів не будуть доступними статичному методу, тому що ні імена, ні кількість об’єктів не може бути відомою статичному методу. На відміну від цього методи об’єктів мають доступ до статичного поля, навіть коли статичне поле має модифікатор доступу private.

Статичні методи зазвичай призначаються для виконання операцій, які є специфічними для цього класу і часто використовуються в класах, з яких не створюються об’єкти, а тільки реалізують певні функції або алгоритмічні методи. Прикладом такого підходу є клас Math та його методи. Усі методи цього класу оголошено як статичні і вони реалізують базові математичні функції для різних типів даних. Слід зауважити, що сам клас Math має модифікатор final, який робить цей клас закритим як для розширення, так і до перевизначення методів, а дозволяється тільки використання його полів та методів.

Статичні методи можна використовувати як із статичних, так і з нестатичних методів, але статичним методам пряме використання методів, що не мають модифікатору доступу static, заборонено. Наприклад, наступні рядки помилки не мають і можуть використовуватись:

```
public static int func1(int a, int b){
    return func2(a,b);
}

public static int func2(int a, int b){
    return a+b;
}
```

Але, якщо для func2() прибрати модифікатор static, то це буде помилка і її використання в тілі func1() стає неможливим.

Використання статичних та нестатичних полів та методів розглянемо на прикладі простої задачі про волонтерів, якім необхідно зібрати певні кошти, наприклад, на медичне обладнання. Безумовно, існує безліч варіантів представлення такої задачі і без використання статичних полів та методів. Клас volunteersAccount має три поля: цільова сума cost, загальні кошти на усіх рахунках allAccounts та кошти на поточному рахунку funds. Змінні cost та allAccounts мають модифікатор static, тому що для усіх екземплярів цього класу вони будуть одними і тими самими.

```
public class volunteersAccount {
    private static double cost;
    private static double allAccounts = 0;
```

```
    private double funds = 0;

    public static void setCost(double cost){
        volunteersAccount.cost = cost;
    }

    public static double getAllFunds(){
        return volunteersAccount.allAccounts;
    }

    public void addFunds(double funds){
        this.funds += funds;
        volunteersAccount.allAccounts += funds;
    }

    public double getFunds(){
        return this.funds;
    }
}
```

Методи, що обслуговують статичні поля також мають модифікатор static, тому що у даному випадку немає ніякого сенсу тиражувати ці методи разом із створенням нових екземплярів класу. З точки зору Java у цих методів модифікатор static може бути відсутнім — компіляція відбудеться і програма буде нормально працювати, але це не буде ефективним використанням пам’яті.

Екземпляр наведеного класу буде мати власне поле `funds` , значення якого буде відповідати наявним коштам на відповідному рахунку, а також два методі: `addFunds` та `getFunds` , які дозволять виконати операцію поповнення (зміни коштів) рахунку та повертає поточне значення поля `funds`. Статичні поля cost та allAccounts в екземплярах класу не існують, як і статичні методи. Статичні поля будуть існувати тільки у класі, тобто єдині для всіх його екземплярів, а доступ до них буде здійснюватись через його статичні методи. Для задачі, що вирішується, це більш менш оптимально — кількість рахунків буде відповідати різноманітним філіям або підрозділам волонтерської організації, кошти на них будуть накопичуватись і контролюватися окремо, а загальна мета та її досягнення будуть єдиними для усіх рахунків.

Необхідність створення статичного методу в класі залежить від задачі, що вирішується. Основні правилами, при цьому, такі:

1. Якщо необхідно створити метод, який буде використовувати статичні змінні та/або викликає статичні методи, то такий метод також може мати модифікатор static. Але якщо окрім доступу до членів класу, метод отримує доступ до членів екземпляра класу, то такий метод має бути методом екземпляра, а не статичним методом.
2. Метод main є початковою точкою для всіх програм і, як такий, він виконується до створення будь-яких об’єктів. Саме для забезпечення цієї функціональності метод main є статичним. Якщо у створеній програмі метод main виявився досить довгим і містить чітко визначені підзавдання, то краще спробувати реалізувати ці підзавдання за допомогою власних методів, які також будуть статичними. Реалізація підзавдань main в якості статичних методів обумовлює більш прозоре їх використання — відпадає необхідність створення екземплярів класу з наступним викликом їх методів та звернення до змінних, все залишається у межах самого класу, а не його екземплярів.
3. Якщо створюється метод загального призначення, то він є кандидатом стати статичним методом. Такі методи називаються методами корисності і вони не прив’язуються до об’єктів. Прикладом корисних методів є методи класу Math, як-от Math.round і Math.sqrt та інші.

Завдання до лабораторної роботи

1. Розробіть клас Vehicle (транспортний засіб), в якому передбачте поля, що ідентифікують власника транспорту, основні характеристики транспорту (не менше 3), а також його поточний стан: положення, що відповідає координатам на мапі, напрямок та швидкість руху, кількість палива у баку. Додайте до класу конструктори. Додайте get- та set-методи, а також методи, що контролюють та керують поточним станом транспорту. Напишіть програму, що демонструє використання класу на декількох тестових прикладах.
2. У наукових та інженерних задачах часто використовується вираз поліномінального типу:

$$P(x)=\sum_{k=0}^{n} a_k x^k$$

Нижче наведено клас Polynom, що реалізує основні властивості таких виразів. Клас має такі методи: додавання, віднімання, множення поліномів, множення та поділ поліному на число, обчислення похідної поліному.

```
public class Polynom {
	private int n;
	private double[] a;

	void set(double[] a){
		this.n=a.length;
		this.a=new double[n];
		int i;
		for(i=0;i<n;i++)
			this.a[i]=a[i];
	}

	void set(int n, double z){
		this.n=n;
		this.a=new double[n];
		int i;
		for(i=0;i<n;i++)
			this.a[i]=z;
	}

	void set(int n){
		set(n, 0);}

	double value(double x){
		double z=0,q=1;
		for(int i=0;i<n;i++){
			z+=a[i]*q;
			q*=x;}
		return z;
	}

	void show(){int i;
		System.out.print("Степінь x:\t");
		for(i=0;i<n-1;i++){ System.out.print(" "+i+"\t");}
		System.out.println(" "+(n-1));
		System.out.print("Коефіцієнт:\t");
		for(i=0;i<n-1;i++){ System.out.print(a[i]+"\t");}
		System.out.println(a[n-1]);
	}

	void show(double x){
```

```
		System.out.println("Значення аргументу x="+x);
		System.out.println("Значення поліному P(x)="+value(x));
	}

	Polynom diff(){
		Polynom t=new Polynom(n-1);
		for(int i=0;i<n-1;i++)
			t.a[i]=a[i+1]*(i+1);
		return t;
	}

	Polynom diff(int k){
		if(k>=n) return new Polynom(1);
		if(k>0) return diff().diff(k-1);
		else return new Polynom(a);
	}

	Polynom plus(Polynom Q){
		Polynom t;
		int i;
		if(n>=Q.n){
			t=new Polynom(a);
			for(i=0;i<Q.n;i++)
				t.a[i]+=Q.a[i];
		}
		else{
			t=new Polynom(Q.a);
			for(i=0;i<n;i++)
				t.a[i]+=a[i];
		}
		return t;
	}

	Polynom minus(Polynom Q){
		return plus(Q.prod(-1));
	}

	Polynom div(double z){
		return prod(1/z);
	}

	Polynom prod(double z){
		Polynom t=new Polynom(a);
		for(int i=0; i<n; i++)
			a[i]*=z;
		return t;
	}

	Polynom prod(Polynom Q){
		int N=n+Q.n-1;
		Polynom t=new Polynom(N);
		for(int i=0;i<n;i++){
			for(int j=0;j<Q.n;j++){
				t.a[i+j]+=a[i]*Q.a[j];
			}
		}
		return t;
	}

	Polynom(double[] a){
		set(a);
	}
	Polynom(int n,double z){
		set(n,z);
```

```
    }
    Polynom(int n){
          set(n);}
    }
```

У класі Polynom приватне цілочисельне поле *n* визначає порядок поліному, який буде дорівнювати *n*-1, а також визначає набір із *n* коефіцієнтів, значення яких буде зберігати масив *a* типу double. Ініціалізація елементів масиву *a* виконується за допомогою перевантаженого методу *set*(). У першій реалізації методу *set* передається посилання на масив типу double, який вміщує необхідні значення коефіцієнтів. У другій його реалізації методу передається 2 аргументи: один визначає розмір масиву коефіцієнтів, другий — одне значення для усіх коефіцієнтів, наприклад, 1. Остання реалізація методу отримує тільки один параметр — розмір масиву коефіцієнтів, а значення усіх коефіцієнтів будуть 0. У останній реалізації використовується другий варіант методу *set*(), де перший аргумент задає розмір масиву коефіцієнтів, а другий аргумент має значення 0.

Метод *value*() повертає значення поліному у точці, значення якої задається його аргументом *x* типу *double*.

Перевантажений метод *show*() має дві реалізації. Перша його реалізація не має аргументів і забезпечує виведення значень коефіцієнтів поліному. Друга його реалізація повертає значення поліному за певним значенням аргументу, що передається методу *show*().

Метод *diff*() в класі Polynom також є перевантаженим, має дві реалізації і повертає поліном, що є похідною поліному, для якого метод викликається. Реалізація методу *diff*() без аргументів повертає похідну поліному першого порядку, а друга реалізація методу повертає похідну поліному порядку k, який задається через його аргумент. Остання реалізація використовує рекурсивне звернення до методу *diff*().*diff*(*k*-1), що дозволяє отримувати поліном похідної і для нього викликати метод із зменшеним на 1 порядком.

Операцію додавання двох поліномів реалізовано у методі *plus*(). Результатом операції є об'єкт класу Polynom. Для одного із поліномів викликається метод *plus*() , і в якості аргументу йому передається інший поліном. Зазвичай поліноми мають різний порядок, тому основою для результату операції обирається поліном вищого порядку. Для обчислення суми поліномів виконується цикл, в якому додаються коефіцієнти, що відповідають однаковим показникам ступені змінної поліному.

Для обчислення різниці поліномів у класі створено метод *minus()*. Реалізацію методу засновано на використані методу *prod*() , за допомогою якого виконується множення усіх коефіцієнтів на -1 та наступне виконання операції додавання.

Метод *prod*() є перевантаженим методом і дозволяє виконати операцію множення поліному на число у першому варіанті реалізації та множення поліному на поліном — у другому. Множення поліному на число — це множення кожного коефіцієнту цього поліному на задане число. Результатом множення одного поліному на інший є сума, в якій кожний доданок є добутком усіх доданків одного поліному на кожний із доданків іншого. Тому, при обчислені добутку двох поліномів *n* та *m* порядку відповідно, результатом буде поліном порядку *n*+*m*. Якщо поліноми представити як:

$$P(x)=\sum_{i=0}^{n} a_i x^i \quad \text{та} \quad Q(x)=\sum_{j=0}^{m} b_j x^j$$

то результатом добутку буде

$$R(x) \equiv P(x)Q(x) = \sum_{i=0}^{n} a_i x^i \sum_{j=0}^{m} b_j x^j = \sum_{\substack{0 \leqslant i \leqslant n, \\ 0 \leqslant j \leqslant m}} a_i b_j x^{i+j}$$

При розрахунку коефіцієнту для x^k необхідно знайти суму $\sum_{i+j=k} a_i b_j$ добутку коефіцієнтів початкових поліномів, сума індексів яких дорівнює k. Наприклад, якщо k = 4, то відповідний коефіцієнт буде отримано з розрахунку суми $a_0 b_4 + a_1 b_3 + a_2 b_2 + a_3 b_1 + a_4 b_0$.

Конструктори класу Polynom реалізовано за допомогою різних версій методу set().

Напишіть програму, що використовує наведений класу та демонструє використання усіх його методів.

3. Які методи класу Polynom, що наведено вище, можуть буди модифіковані у статичні? Модифікуйте ці методи у статичні та перевірте їх роботу.

4. Якими статичними методами можна доповнити клас Polynom? Реалізуйте ці методи та перевірте їх роботу.

5. У розрахункових задачах інколи виникає необхідність реалізації математичних функцій або операцій. Нижче наведено приклад класу, в якому реалізуються декілька математичних функцій:

```
public class MyMath {
static double L=Math.PI;

static double Exp(double x,int N){
      int i;
      double s=0,q=1;
      for(i=0;i<N;i++){
            s+=q;
            q*=x/(i+1);
            }
    return s+q;
}

static double Sin(double x,int N){
      int i;
      double s=0,q=x;
      for(i=0;i<N;i++){
            s+=q;
            q*=(-1)*x*x/(2*i+2)/(2*i+3);
            }
      return s+q;
}

static double Cos(double x,int N){
      int i;
      double s=0,q=1;
      for(i=0;i<N;i++){
            s+=q;
            q*=(-1)*x*x/(2*i+1)/(2*i+2);
            }
      return s+q;
      }

static double BesselJ(double x,int N){
      int i;
      double s=0,q=1;
      for(i=0;i<N;i++){
            s+=q;
```

```
                q*=(-1)*x*x/4/(i+1)/(i+1);
                }
        return s+q;
        }

static double FourSin(double x,double[] a){
        int i,N=a.length;
        double s=0;
        for(i=0;i<N;i++){
                s+=a[i]*Math.sin(Math.PI*x*(i+1)/L);
                }
        return s;}

static double FourCos(double x,double[] a){
        int i,N=a.length;
        double s=0;
        for(i=0;i<N;i++){
                s+=a[i]*Math.cos(Math.PI*x*i/L);
                }
        return s;
        }
}
```

У наведеному класі алгоритми обчислення функцій exp(x), sin(x), cos(x) та Бесселя реалізовано через їх розклад у ряд Тейлора.
Для експоненти використано ряд:

$$\exp(x)\approx\sum_{k=0}^{N}\frac{x^k}{k!}$$

Синус та косинус обчислюються за формулами:

$$\sin(x)\approx\sum_{k=0}^{N}\frac{(-1)^k x^{2k+1}}{(2k+1)!}$$

$$\cos(x)\approx\sum_{k=0}^{N}\frac{(-1)^k x^{2k}}{(2k)!}$$

Для функції Бесселя нульового індексу використовується ряд:

$$J_0(x)\approx\sum_{k=0}^{N}\frac{(-1)^k (x/2)^{2k}}{(k!)^2}$$

У класі реалізація кожної функції має два аргументи. Перший аргумент має тип double та безпосередньо визначає аргумент математичної функції, а другий цілочисельний аргумент визначає кількість доданків у ряді Тейлора, що використовуються при обчислені функції.

Для обчислення функції Бесселя в якості першого аргументу вказано значення нуля функції Бесселя $\mu_1 \approx 2.404825558$. Нулями μ_n (n = 1, 2, ...) функції Бесселя нульового індексу $J_0(x)$ зветься невід'ємні рішення рівняння $J_0(\mu_n)=0$. Тому для вказаного аргументу значення функції повинно бути нульовим у межах точності обчислень.

Також в класі MyMath визначено функції для обчислення рядів Фур'є за базовими функціями $\sin\left(\frac{\pi n x}{L}\right)$ (функція FourSin()) та $\cos\left(\frac{\pi n x}{L}\right)$ (функція FourCos()).

Реалізації функцій мають два аргументи: перший відповідає змінній *x* типу double, а другий є масивом *a* з елементами типа double. Зокрема, у функції FourSin(), в якості результату повертається значення суми, що має вигляд:

$$\sum_{n=1}^{N} a_{n-1} \sin\left(\frac{\pi n x}{L}\right) ,$$

де a_n - елементи масиву *a*; N - верхня границя суми, яка визначається кількістю елементів цього масиву (N = a.length).

Зазначена сума є розкладом у ряд, з обмеженням за кількістю доданків, на інтервалі від 0 до *L* певної функції з коефіцієнтами розкладу, які записано у масив *a*. Параметр *L* оголошується в класі MyMath як статичне поле із значенням *Math.PI*.

Напишіть програму, що використовує наведений клас для обчислення функцій, що реалізовано в ньому, для заданих значень аргументів та параметром N.

6. Перевірте, що результат обчислення функції Бесселя при значенні першого аргументу 2.404825558 є достатньо близьким до 0.

7. Модифікуйте наведений клас MyMath таким чином, щоб всі методи в ньому були без модифікатору static. Як зміниться використання методів цього класу? Реалізуйте приклади його використання.

Запитання для самоперевірки

1. Чим визначається поведінка об’єкту?
2. Чим визначається стан об’єкту?
3. У яких відносинах знаходяться клас і об’єкт?
4. Яку роль в класі грає конструктор?
5. Що означають терміни змінна екземпляра та метод екземпляра?
6. Як багато можна створити екземплярів одного класу?
7. Що означає “перевантаження методів”?
8. Що відрізняє перевантажені методи?
9. Що повертає конструктор після використання оператору new?
10. Як отримати доступ до члена екземпляру класу зі статичного методу?