МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
КАФЕДРА ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ

МЕТОДИЧНІ ВКАЗІВКИ

ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ
З ДИСЦИПЛІНИ «ВСТУП ДО ПРОГРАМУВАННЯ .NET та JAVA»

для студентів освітнього ступеню «Бакалавр» спеціальностей
121 Інженерія програмного забезпечення,
125 Кібербезпека

Луцьк – 2023

УДК 004.42(072)
М 54
Методичні вказівки до виконання лабораторних робіт з дисципліни «Вступ до програмування .NET та JAVA» для студентів освітнього ступеню «бакалавр» спеціальностей 121 Інженерія програмного забезпечення, 125 Кібербезпека / уклад. А. О. Нікітенко. – Луцьк : ДонНТУ, 2023. – 126 с.

Наведено завдання до виконання робіт з дисципліни «Вступ до програмування .NET та JAVA», у стиснутій формі надано теоретичні відомості. Сформульовано завдання й описано порядок виконання робіт. Надано вимоги до оформлення звітів, базові приклади, перелік літературних джерел. Завдання орієнтовані на закріплення знань, отриманих бакалаврами при вивченні лекційного матеріалу та придбання навичок створення й супроводження програмного забезпечення написаного з використанням платформ .NET і JAVA.

Укладач: А.О. Нікітенко, асистент кафедри ПМІ

Рецензент: С.О. Ковальов, к.т.н, доцент кафедри електронної техніки

Відповідальний за випуск:
Н.О. Маслова, завідувач кафедри прикладної математики та інформатики

Затверджено на засіданні навчально-методичного відділу ДонНТУ,
протокол № 3 від «28» листопада 2023 р.

Ухвалено на засіданні кафедри прикладної математики та інформатики,
протокол № 14 від «11» листопада 2023 р.

ЗМІСТ

ВСТУП

Java та .NET є найпотужнішими платформами для розробки сучасних програмних продуктів, що швидко розвиваються, особливо у сфері веб-технологій та корпоративних додатків. Мови Java та C# посідають високі місця в рейтингу популярності мов програмування та в рейтингу зарплат розробників.

Багато в чому це пов'язано з тим, що найбільшим попитом на ринку програмного забезпечення користуються корпоративні додатки, які автоматизують різні аспекти діяльності компаній, організацій і підприємств. Вимоги до таких додатків постійно зростають, а концепції розробки, як наслідок, стають складнішими. Це, в свою чергу, ускладнює концепцію розробки. Платформа Java інтегрує в собі всі рівні технології розробки програмного забезпечення, необхідні для вирішення таких завдань. Не всі ці рівні можна охопити в рамках одного курсу. Цей курс закладає основу професійних навичок, пов'язаних з основними компонентами технології Java. Студенти, які закінчать цей курс, зможуть створювати десктопні додатки на Java, але для тих, хто вирішить в майбутньому зробити кар'єру Java-розробника, цей курс є лише першим кроком.

В цьому курсі вивчення починається з огляду платформ .NET та Java. Далі розглядаються основні концепції та конструкції платформи Java, об’єктно-орієнтоване програмування (ООП) при проектуванні складних архітектур у програмному забезпеченні. Паралельно розглядаються прийоми роботиз інструментальними засобами Java-розробки – набором розробника Java Development Kit (JDK) і інтегрованим середовищем розробки IntelliJ IDEA. Розглядаються як загальні принципи розробки сучасних програмних продуктів, так і окремі технології: для роботи зі структурами даних, роботи з файловою системою, створення розвиненого GUI, організації багатопотокових рішень та інші.

Знання та навички, отримані на цьому курсі, є основою для подальшого більш грунтовного опанування спеціалізації Java розробника, а саме з технологіями баз даних (JDBC, JPA, Hibernate), створення клієнт-серверних

додатків, додатків для Android, J2EE Web J2EE технології, а також зі сучасними фреймворками (наприклад, Spring, Maven, JUnit).

ЛАБОРАТОРНА РОБОТА. ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ JAVA В СЕРЕДОВИЩІ INTELLIJ IDEA

Мета роботи: Ознайомитися з мовою програмування Java. Ознайомитись із середовищем програмування Intellij IDEA.

Основні теоретичні відомості

Ознайомлення з мовою програмування Java

Java – це об'єктно-орієнтована мова. Вона підтримує поліморфізм, успадкування та статичну типізацію. Об'єктно-орієнтований підхід вирішує проблему побудови великих, але в той же час гнучких, масштабованих і розширюваних додатків.

Головна особливість мови Java полягає в тому, що код спочатку транслюється в спеціальний незалежний від платформи байт-код. Цей байт-код потім виконується JVM (віртуальною машиною Java). У цьому відношенні Java відрізняється від стандартних інтерпретованих мов, таких як PHP і Perl. Java також не є повністю скомпільованою мовою, як C або C++.

Така архітектура забезпечує крос-платформну сумісність і апаратну переносимість програм на Java, дозволяючи запускати схожі програми на різних платформах, таких як Windows, Linux і Mac OS без перекомпіляції. Кожна платформа може мати власну реалізацію JVM, але кожна може виконувати той самий код.

Віртуальна машина Java (JVM) – це рушій, який забезпечує середовище виконання для запуску коду Java або додатків. Він перетворює байт-код Java у машинну мову. JVM є частиною середовища виконання Java (JRE). В інших мовах програмування компілятор створює машинний код для конкретної системи. Однак компілятор Java створює код для віртуальної машини, відомої як Java Virtual Machine.

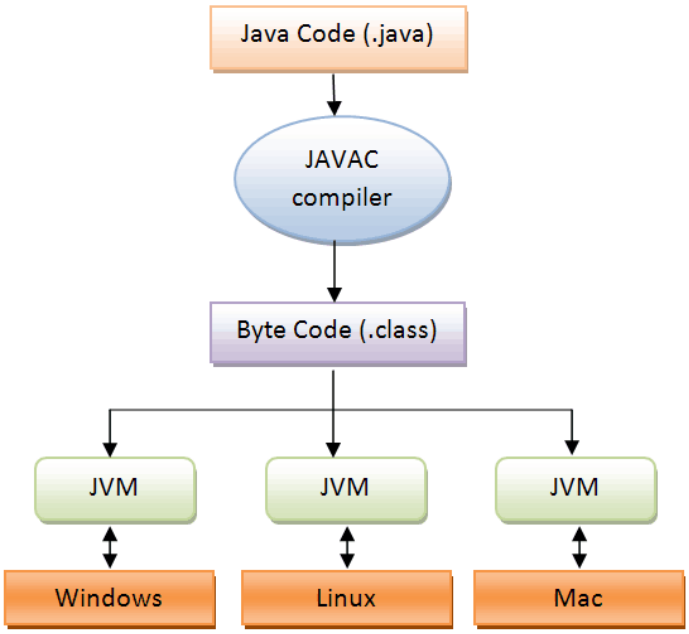


Рисунок 1.1 – Робота Java Virtual Machine (JVM)

Ще однією важливою особливістю Java є підтримка автоматичного збору сміття. Це означає, що немає необхідності вручну звільняти пам'ять від раніше використаних об'єктів, як у C++.

Oracle JDK і OpenJDK

Для розробки мовою програмування Java нам знадобиться спеціальний комплект інструментів, який називається JDK або Java Development Kit. Однак варто зазначити, що існують різні реалізації JDK, хоча всі вони використовують одну й ту саму мову - Java. Дві найпопулярніші реалізації - Oracle JDK і OpenJDK. У чому їхня різниця? Oracle JDK цілком розвивається компанією Oracle. OpenJDK же розвивається як компанією Oracle, так і ще низкою компаній спільно. Найбільші відмінності з точки зору ліцензування. Згідно з ліцензією Oracle JDK можна використовувати безкоштовно для персональних потреб, а також для розробки, тестування та демонстрації додатків. В інших випадках (наприклад, для отримання підтримки) необхідна комерційна ліцензія у вигляді підписки. А OpenJDK повністю безкоштовна. У плані функціоналу, набору можливостей Oracle JDK і OpenJDK практично не повинні відрізнятися. А ось у плані продуктивності відзначається, що Oracle JDK працює дещо

швидше, ніж OpenJDK. Крім того, деякі розробники відзначають, що OpenJDK трохи більш глючна, а Oracle JDK більш стабільна. У лабораторних роботах ми використовуватимемо Oracle JDK, однак якщо ви використовуєте OpenJDK, жодних проблем не повинно виникнути. Посилання на офіційний сайт для завантаження OracleJDK: https://www.oracle.com/java/technologies/downloads/.

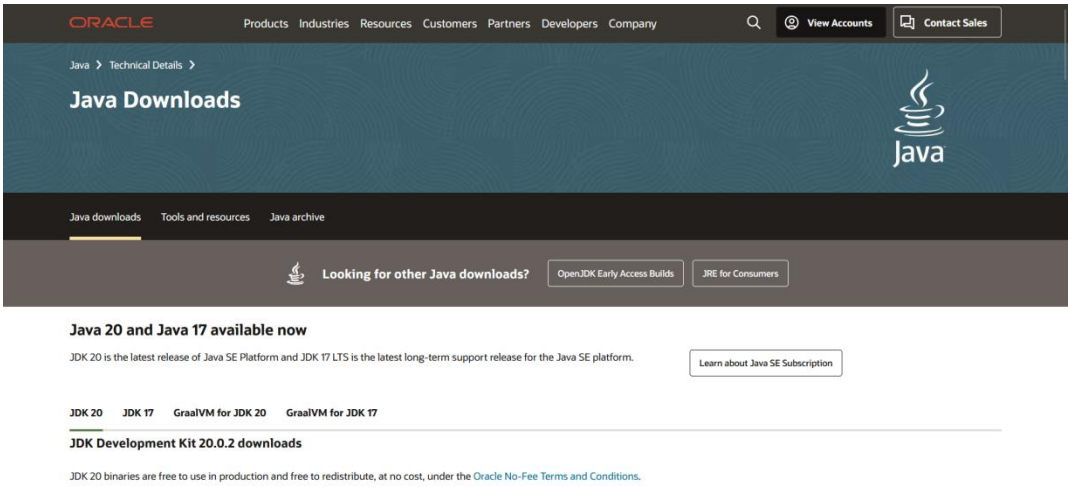


Рисунок 1.2 – Сторінка завантаження OracleJDK

Intellij IDEA

IntelliJ IDEA – це спеціальне середовище програмування або інтегроване середовище розробки (IDE), в основному призначене для Java. Це середовище використовується спеціально для розробки програм. Що відрізняє IntelliJ IDEA від її аналогів - це простота використання, гнучкість та надійний дизайн.

IntelliJ IDEA – найпотужніша та найпопулярніша IDE для Java розробників. Розроблена та підтримується компанією JetBrains. Має ліцензію Apache 2.0. Intellij доступний у двох версіях:

- Community Edition;
- Ultimate Edition.

Для початку роботи можна обрати Community Edition, але на момент написання методичних вказівок компанія JetBrains дає можливість студентам зареєструвати студентську ліцензію для платної Ultimate Edition на 1 рік. Посилання : https://www.jetbrains.com/community/education/#students.

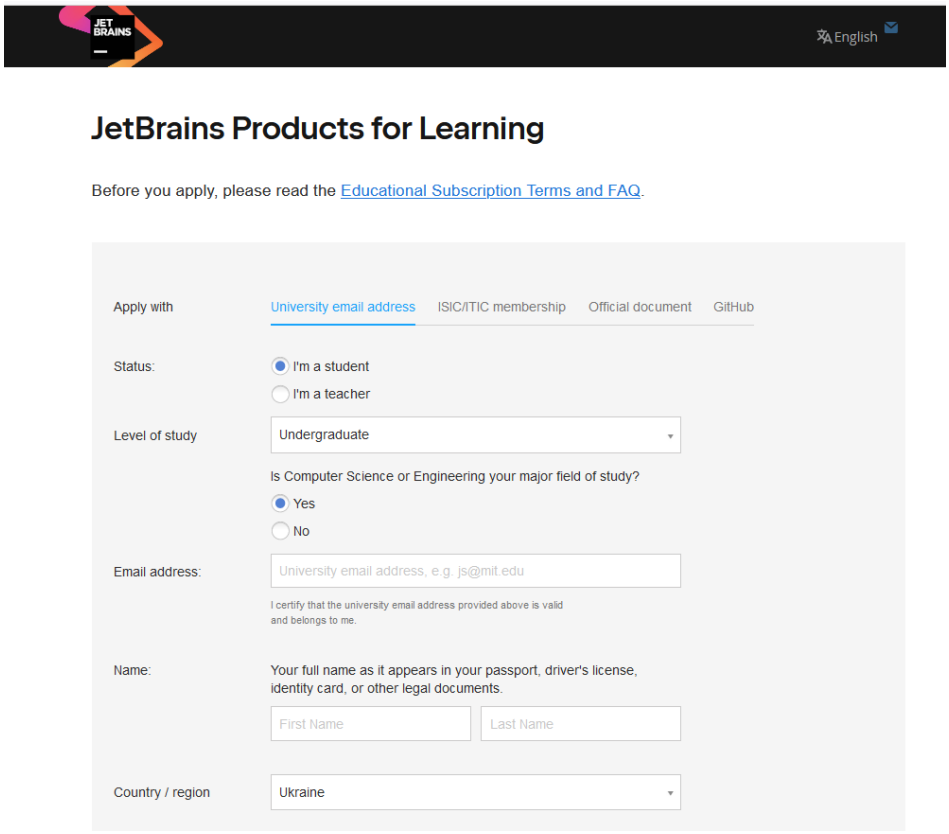


Рисунок 1.3 – Вікно отримання студентської ліцензії на офіційному сайті JetBrains

Після цього завантажуємо Intellij IDEA з офіційного сайту https://www.jetbrains.com/idea/download/?section=windows.

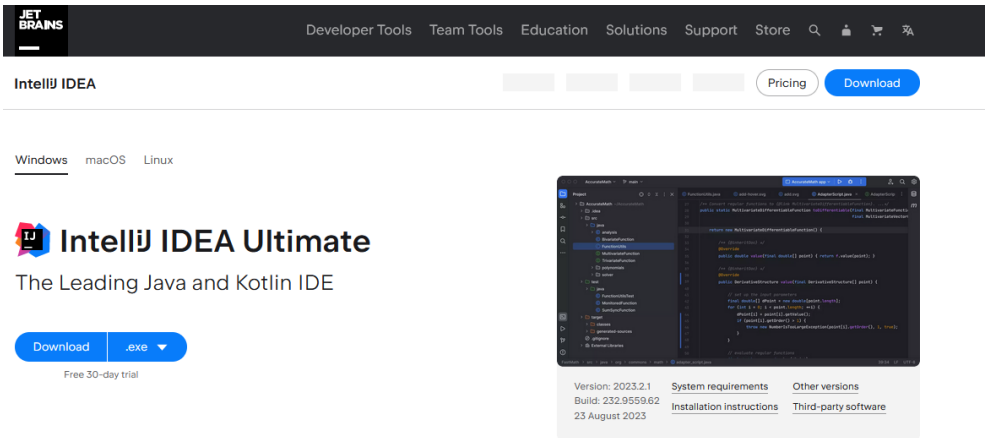


Рисунок 1.4 – Вікно завантаження Intellij IDEA

При першому запуску IntellijIDEA ви повинні зайти до свого облікового запису JetBrains, де вже зареєстровано вашу студентську ліцензію. Тепер

створимо перший проект. В ньому ви повинні обрати версію Java (на момент написання останьою є 20 версія). Дати назву проекту, директорію для розташування та яким чином ви бажаєте зібрати проект, рекомендується використовувати maven. Також рекомендується прибрати «add sample code» для прибрання непотрібної генерації початкового коду.

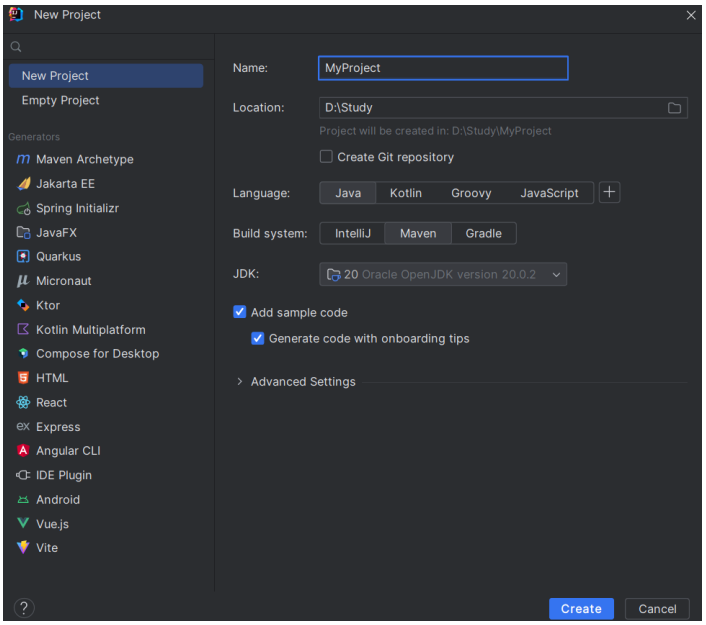


Рисунок 1.5 – Створення першого проекту

Ваш проект має мати такий вигляд після створення:

Рисунок 1.6 – Вигляд першого проєкту

Типи даних

Однією з основних особливостей Java є те, що ця мова є строго типізованою. А це означає, що кожна змінна і константа представляє певний

тип і цей тип суворо визначений. Тип даних визначає діапазон значень, які може зберігати змінна або константа.

Отже, розглянемо систему вбудованих базових типів даних, яка використовується для створення змінних у Java. А вона представлена такими типами.

boolean: зберігає значення true або false;

byte: зберігає ціле число від -128 до 127 і займає 1 байт;

short: зберігає ціле число від -32768 до 32767 і займає 2 байти;

int: зберігає ціле число від -2147483648 до 2147483647 і займає 4 байти;

long: зберігає ціле число від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 і займає 8 байт;

double: зберігає число з плаваючою крапкою від ±4.9*10-324 до ±1.7976931348623157*10308 і займає 8 байт. Як роздільник цілої та дробової частини в дробових літералах використовується крапка;

float: зберігає число з плаваючою крапкою від -3.4*1038 до 3.4*1038 і займає 4 байти;

char: зберігає одиночний символ у кодуванні UTF-16 і займає 2 байти, тому діапазон збережених значень від 0 до 65535.

При цьому змінна може приймати тільки ті значення, які відповідають її типу. Якщо змінна представляє цілочисельний тип, то вона не може зберігати дробові числа.

Цілі числа

Усі цілочисельні літерали, наприклад, числа 10, 4, -5, сприймаються як значення типу int, однак ми можемо присвоювати цілочисельні літерали іншим цілочисельним типам: byte, long, short. У цьому випадку Java автоматично здійснює відповідні перетворення:

```
byte a = 1;
short b = 2;
long c = 2121;
```

Однак якщо ми захочемо присвоїти змінній типу long дуже велике число, яке виходить за межі допустимих значень для типу int, то ми зіткнемося з помилкою під час компіляції:

```
long num = 2147483649;
```

Тут число 2147483649 є допустимим для типу long, але виходить за граничні значення для типу int. І оскільки всі цілочисельні значення за замовчуванням розцінюються як значення типу int, то компілятор вкаже нам на помилку. Щоб розв'язати проблему, треба додати до числа суфікс l або L, який вказує, що число представляє тип long:

```
long num = 2147483649L;
```

Числа з плаваючою крапкою

Під час присвоєння змінній типу float дробового літерала з плаваючою крапкою, наприклад, 3.1, 4.5 тощо, Java автоматично розглядає цей літерал як значення типу double. І щоб вказати, що дане значення має розглядатися як float, нам треба використовувати суфікс f:

```
float fl = 30.6f;
double db = 30.6;
```

І хоча в цьому випадку обидві змінні мають практично одне значення, але ці значення будуть по-різному розглядатися і займатимуть різне місце в пам'яті.

Символи та рядки

Як значення змінна символьного типу отримує одиночний символ, укладений в одинарні лапки: char ch='e';. Крім того, змінній символьного типу також можна присвоїти цілочисельне значення від 0 до 65535. У цьому випадку змінна знову ж таки зберігатиме символ, а цілочисельне значення вказуватиме на номер символу в таблиці символів Unicode (UTF-16). Наприклад:

```
char ch=102; // символ 'f'
System.out.println(ch);
```

Ще однією формою задавання символьних змінних є шістнадцяткова форма: змінна отримує значення в шістнадцятковій формі, яке слідує після символів "\u". Наприклад, char ch='\u0066'; знову ж таки зберігатиме символ 'f'.

Символьні змінні не варто плутати з рядковими, 'a' не ідентично "a". Рядкові змінні представляють об'єкт String, який на відміну від char або int не є примітивним типом у Java:

```
String hello = "Hello...";
System.out.println(hello);
```

Крім власне символів, які представляють літери, цифри, розділові знаки, інші символи, є спеціальні набори символів, які називають керуючими послідовностями. Наприклад, найпопулярніша послідовність - "\n". Вона виконує перенесення на наступний рядок. Наприклад:

```
String text = "Hello \nworld";
System.out.println(text);
```

У цьому випадку послідовність \n буде сигналом, що необхідно зробити переклад на наступний рядок.

Починаючи з версії 15 Java підтримує тестові блоки (text blocks) - багаторядковий текст, одягнений у потрійні лапки. Розглянемо, у чому їхня практична користь. Наприклад, виведемо великий багаторядковий текст:

```
String text = "Ось думка, якій весь я відданий,\n "+
                "Підсумок усього, що розум зібрав.\n "+
                "Лише той, ким бій за життя звіданий,\n "+
                "Життя і свободу заслужив.";
System.out.println(text);
```

За допомогою операції + ми можемо приєднати до одного тексту інший, причому продовження тексту може розташовуватися на наступному рядку. Щоб під час виведення тексту відбувалося перенесення на наступний рядок, застосовується послідовність \n.

```
Результат виконання цього коду:
Ось думка, якій весь я відданий,
Підсумок усього, що розум зібрав.
Лише той, ким бій за життя звіданий,
Життя і свободу заслужив.
```

Текстові блоки, які з'явилися в JDK15, дають змогу спростити написання багаторядкового тексту:

```
String text = "Ось думка, якій весь я відданий,\n "+
                "Підсумок усього, що розум зібрав.\n "+
                "Лише той, ким бій за життя звіданий,\n "+
                "Життя і свободу заслужив.";
```

Увесь текстовий блок обертається в потрійні лапки, при цьому не треба використовувати з'єднання рядків або послідовність \n для їх перенесення. Результат виконання програми буде тим самим, що й у прикладі вище.

Виведення на консоль

Найпростіший спосіб взаємодії з користувачем являє консоль: ми можемо виводити на консоль деяку інформацію або, навпаки, зчитувати з консолі деякі

дані. Для взаємодії з консоллю в Java застосовується клас System, а його функціональність власне забезпечує консольне введення і виведення.

Для створення потоку виведення в клас System визначено об'єкт out. У цьому об'єкті визначено метод println, який дає змогу вивести на консоль деяке значення з подальшим переведенням курсора консолі на наступний рядок. Наприклад:

```
public class Program {
    public static void main(String[] args) {
        System.out.println("Hello world!");
        System.out.println("Bye world...");
    }
}
```

У метод println передається будь-яке значення, як правило, рядок, який треба вивести на консоль. І в цьому випадку ми отримаємо такий висновок:

```
Hello world!
Bye world...
```

За необхідності можна і не переводити курсор на наступний рядок. У цьому випадку можна використовувати метод System.out.print(), який аналогічний println за тим винятком, що не здійснює переведення на наступний рядок.

```
public class Program {
    public static void main(String[] args) {
        System.out.print("Hello world!");
        System.out.print("Bye world...");
    }
}
```

Консольний вивід цієї програми:

```
Hello world!Bye world...
```

Введення з консолі

Для отримання введення з консолі в класі System визначено об'єкт in. Однак безпосередньо через об'єкт System.in не дуже зручно працювати, тому, як правило, використовують клас Scanner, який, у свою чергу, використовує System.in. Наприклад, напишемо маленьку програму, яка здійснює введення чисел:

```
import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Input a number: ");
        int num = in.nextInt();
        System.out.printf("Your number: %d \n", num);
```

```
            in.close();
        }
    }
```

Оскільки клас Scanner знаходиться в пакеті java.util, то ми спочатку його імпортуємо за допомогою інструкції import java.util.Scanner.

Для створення самого об'єкта Scanner у його конструктор передається об'єкт System.in. Після цього ми можемо отримувати значення, що вводяться. Наприклад, у цьому випадку спочатку виводимо запрошення до введення і потім отримуємо число, що вводиться, у змінну num.

Клас Scanner має низку методів, які дають змогу отримати введені користувачем значення:

next(): зчитує введений рядок до першого пробілу;

nextLine(): зчитує весь введений рядок;

nextInt(): зчитує введене число int;

nextDouble(): зчитує введене число double;

nextBoolean(): зчитує значення boolean;

nextByte(): зчитує введене число byte;

nextFloat(): зчитує введене число float;

nextShort(): зчитує введене число short.

Тобто для введення значень кожного примітивного типу в класі Scanner визначено свій метод.

Більшість операцій у Java аналогічні тим, які застосовуються в інших сі-подібних мовах. Є унарні операції (виконуються над одним операндом), бінарні - над двома операндами, а також тернарні - виконуються над трьома операндами. Операндом є змінна або значення (наприклад, число), що бере участь в операції. Розглянемо всі види операцій.

В арифметичних операціях беруть участь числа. У Java є бінарні арифметичні операції (проводяться над двома операндами) і унарні (виконуються над одним операндом). До бінарних операцій відносять такі:

+операція додавання двох чисел;

-операція віднімання двох чисел;

*операція множення двох чисел;

/операція ділення двох чисел.

Під час ділення варто враховувати, що якщо в операції беруть участь два цілих числа, то результат ділення буде округлятися до цілого числа, навіть якщо результат присвоюється змінній float або double:

```
System.out.println(k);
```

Також є дві унарні арифметичні операції, які виконуються над одним числом: ++ (інкремент) і -- (декремент). Кожна з операцій має два різновиди: префіксний і постфіксний;

++ (префіксний інкремент) - передбачає збільшення змінної на одиницю, наприклад, z=++y (спочатку значення змінної y збільшується на 1, а потім її значення присвоюється змінній z);

-- (префіксний декремент) - зменшення змінної на одиницю, наприклад, z=--y (спочатку значення змінної y зменшується на 1, а потім її значення присвоюється змінній z);

-- (постфіксний декремент) - z=y-- (спочатку значення змінної y присвоюється змінній z, а потім значення змінної y зменшується на 1).

Пріоритет арифметичних операцій

Одні операції мають більший пріоритет, ніж інші, і тому виконуються спочатку. Операції в порядку зменшення пріоритету:

++ (постфіксний інкремент), -- (постфіксний декремент);

++ (префіксний інкремент), -- (префіксний декремент);

* (множення), / (ділення), % (залишок від ділення);

+ (додавання), - (віднімання).

Операції з числами з плаваючою крапкою

Слід зазначити, що числа з плаваючою крапкою не підходять для фінансових та інших обчислень, де помилки при округленні можуть бути критичними. Наприклад:

```
double d = 2.0 - 1.1;
System.out.println(d);
```

У цьому випадку змінна d дорівнюватиме не 0.9, як можна було б спочатку припустити, а 0.899999999999999999999. Подібні помилки точності виникають через те, що на низькому рівні для подання чисел із плаваючою

крапкою застосовується двійкова система, однак для числа 0.1 не існує двійкового подання, так само як і для інших дробових значень. Тому в таких випадках зазвичай застосовується клас BigDecimal, який дозволяє обійти подібні ситуації.

Завдання до виконання

Встановити мову програмування Java, інтегроване середовище розробки Intellij IDEA (або інше за власним бажанням). Та виконати завдання:

- Створити 2 змінні name, surname типу рядок, з власним ім’ям та прізвищем;
- Поєднати рядки через прогалину та зберегти результат у змінній full_name ;
- Створіть змінну X, Y. В змінну X помістіть останню літеру вашого ім’я та в змінну Y помістіть останню літеру вашого прізвища зі змінної full_name;
- Підключити додатковий клас Math та виконати розрахунок формули:

$$(2 * (X \bmod 2)) * function(X),$$

Де:

X – номер студента у журналі;

function – визначається з згідно з варіантом.

Таблиця 1.1 – Варіанти завдань

№	Метод
1.	abs
2.	acos
3.	asin
4.	log10
5.	log
6.	tan
7.	tanh
8.	exp
9.	sin
10.	sinh

- Всі проміжні дані надрукувати

Приклад виконання

```
public static void main(String[] args){
    int x = 11;
    String name = "Andrii";
    String surname = "Nikitenko";
    String full_name = name + ' ' + surname;
    System.out.println(full_name);
    double equation = (2 * (x % 2)) * Math.sin(x);
    System.out.println(equation);
}
```

Рисунок 1.7 – Приклад виконання завдання

Контрольні питання

1. Надайте визначення мові програмування Java;
2. Надайте визначення поняттю «IDE»;
3. Поясніть, що означає термін "JVM" (Java Virtual Machine);
4. Що розуміється під терміном "тип даних", і які існують типи даних у Java?
5. Що означають префіксний та постфіксний інкремент у контексті програмування?
6. Роз'ясніть префіксний та постфіксний декремент у програмуванні.
7. Який пріорітет у арифметичних операцій у мові програмування Java?
8. З якою метою використовується клас Scanner у мові програмування Java?

ЛАБОРАТОРНА РОБОТА. ОЗНАЙОМЛЕННЯ З БАЗОВИМИ ЕЛЕМЕНТАМИ ТА КЕРУЮЧИМИ КОНСТРУКЦІЯМИ В JAVA

Мета роботи: ознайомлення з елементами та керуючими конструкціями в мові програмування Java.

Основні теоретичні відомості

Умовні вирази

Умовні вирази являють собою деяку умову і повертають значення типу boolean, тобто значення true (якщо умова істинна), або значення false (якщо умова хибна). До умовних виразів належать операції порівняння та логічні операції.

Операції порівняння

В операціях порівняння порівнюються два операнди, і повертається значення типу boolean - true, якщо вираз є правильним, і false, якщо вираз є неправильним.

1) == порівнює два операнди на рівність і повертає true (якщо операнди рівні) і false (якщо операнди не рівні)

```
int a = 10;
int b = 4;
boolean c = a == b;          // false
boolean d = a == 10;        // true
```

2) != порівнює два операнди і повертає true, якщо операнди НЕ рівні, і false, якщо операнди рівні

```
int a = 10;
int b = 4;
boolean c = a != b;          // true
boolean d = a != 10;        // false
```

3) < (менше ніж) - повертає true, якщо перший операнд менший за другий, інакше повертає false

```
int a = 10;
int b = 4;
boolean c = a < b;    // false
```

4) > (більше ніж) - повертає true, якщо перший операнд більший за другий, інакше повертає false

```
int a = 10;
int b = 4;
boolean c = a > b;   // true
```

5) >= (більше або дорівнює) – повертає true, якщо перший операнд більший за другий або дорівнює другому, інакше повертає false

```
boolean c = 10 >= 10;    // true
```

6) <= (менше або дорівнює) - повертає true, якщо перший операнд менший за другий або дорівнює другому, інакше повертає false

```
boolean c = 10 <= 10;    // true
```

Логічні операції

Також у Java є логічні операції, які також представляють умову та повертають true або false і зазвичай об'єднують кілька операцій порівняння. До логічних операцій відносять такі:

- | - c=a|b; (c дорівнює true, якщо або a, або b (або і a, і b) дорівнюють true, інакше c дорівнюватиме false)
- & - c=a&b; (c дорівнює true, якщо і a, і b дорівнюють true, інакше c дорівнюватиме false)
- ! - c=!b; (c дорівнює true, якщо b дорівнює false, інакше c дорівнюватиме false)
- ^ - c=a^b; (c дорівнює true, якщо або a, або b (але не одночасно) дорівнюють true, інакше c дорівнюватиме false)
- || - c=a|||b; (c дорівнює true, якщо або a, або b (або і a, і b) дорівнюють true, інакше c дорівнюватиме false)
- && - c=a&&b; (c дорівнює true, якщо і a, і b дорівнюють true, інакше c дорівнюватиме false)

Тут у нас дві пари операцій | і || (а також & і &&) виконують схожі дії, проте ж вони не рівнозначні.

Вираз c=a|b; обчислюватиме спочатку обидва значення - a та b і на їхній основі виводитиме результат.

У виразі ж c=a||b; спочатку обчислюватиметься значення a, і якщо воно дорівнюватиме true, то обчислення значення b вже сенсу не має, оскільки в нас у будь-якому разі вже c дорівнюватиме true. Значення b буде обчислюватися тільки в тому випадку, якщо a дорівнює false

Те ж саме стосується пари операцій &/&&. У виразі c=a&b; обчислюватимуться обидва значення - a і b.

У виразі ж c=a&&b; спочатку обчислюватиметься значення a, і якщо воно дорівнюватиме false, то обчислення значення b уже не має сенсу, оскільки значення c у будь-якому разі дорівнюватиме false. Значення b буде обчислюватися тільки в тому випадку, якщо a дорівнює true

Таким чином, операції || і && зручніші в обчисленнях, даючи змогу скоротити час на обчислення значення виразу і тим самим підвищуючи продуктивність. А операції | і & більше підходять для виконання порозрядних операцій над числами.

Перетворення базових типів даних

Кожен базовий тип даних займає певну кількість байт пам'яті. Це накладає обмеження на операції, до яких залучені різні типи даних. Розглянемо такий приклад:

```
int a = 4;
byte b = a; // ! Помилка
```

У цьому коді ми зіткнемося з помилкою. Хоча і тип byte, і тип int представляють цілі числа. Ба більше, значення змінної a, яке присвоюється змінній типу byte, цілком укладається в діапазон значень для типу byte (від -128 до 127). Проте ми стикаємося з помилкою на етапі компіляції. Оскільки в даному випадку ми намагаємося присвоїти деякі дані, які займають 4 байти, змінній, яка займає всього один байт.

Проте в програмі може знадобитися, щоб подібне перетворення було виконано. У цьому випадку необхідно використовувати операцію перетворення типів (операція ()):

```
int a = 4;
byte b = (byte)a;  // перетворення типів: від типу int до типу byte
System.out.println(b); // 4
```

Операція перетворення типів передбачає зазначення в дужках того типу, до якого треба перетворити значення. Наприклад, у випадку операції (byte)a, відбувається перетворення даних типу int у тип byte. У підсумку ми отримаємо значення типу byte.

Явні та неявні перетворення

Коли в одній операції залучені дані різних типів, не завжди необхідно використовувати операцію перетворення типів. Деякі види перетворень виконуються неявно, автоматично.

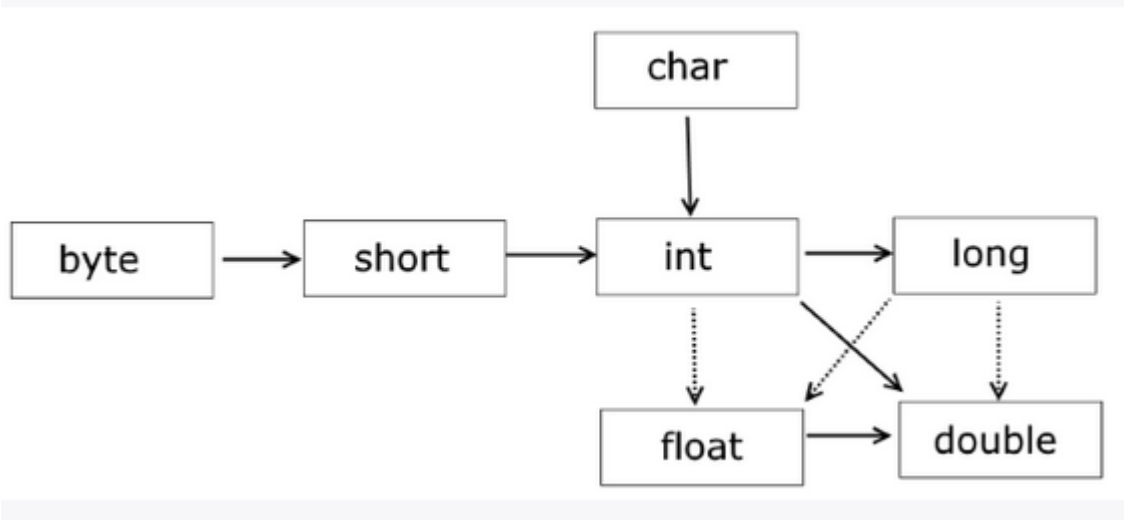


Рисунок 2.1 – Автоматичне перетворення

Стрілками на рисунку показано, які перетворення типів можуть виконуватися автоматично. Пунктирними стрілками показано автоматичні перетворення з втратою точності.

Автоматично без будь-яких проблем виконуються розширювальні перетворення (widening) - вони розширюють представлення об'єкта в пам'яті. Наприклад:

```
byte b = 7;
int d = b;  // перетворення від byte до int
```

У цьому випадку значення типу byte, яке займає в пам'яті 1 байт, розширюється до типу int, яке займає 4 байти.

Автоматичні перетворення, що розширюють, представлені такими ланцюжками:

byte -> short -> int -> long

int -> double

short -> float -> double

char -> int

Автоматичні перетворення з втратою точності

Деякі перетворення можуть здійснюватися автоматично між типами даних однакової розрядності або навіть від типу даних з більшою розрядністю до типу з меншою розрядністю. Це такі ланцюжки перетворень: int -> float, long -> float і long -> double. Вони виконуються без помилок, але при перетворенні ми можемо зіткнутися з втратою інформації.

Наприклад:

```
int a = 2147483647;
float b = a;            // від типу int до типу  float
System.out.println(b);  // 2.14748365E9
```

Явні перетворення

У всіх інших перетвореннях примітивних типів явним чином застосовується операція перетворення типів. Зазвичай це звужувальні перетворення (narrowing) від типу з більшою розрядністю до типу з меншою розрядністю:

```
long a = 4;
int b = (int) a;
```

Втрата даних під час перетворення

При застосуванні явних перетворень ми можемо зіткнутися з втратою даних. Наприклад, у наступному коді у нас не виникне жодних проблем:

```
int a = 5;
byte b = (byte) a;
System.out.println(b);      // 5
```

Число 5 цілком укладається в діапазон значень типу byte, тому після перетворення змінна b дорівнюватиме 5. Але що буде в наступному випадку:

```
int a = 258;
byte b = (byte) a;
System.out.println(b);      // 2
```

Результатом буде число 2. У цьому випадку число 258 поза діапазоном для типу byte (від -128 до 127), тому відбудеться усічення значення. Чому результатом буде саме число 2?

Число a, яке дорівнює 258, у двійковій системі дорівнюватиме 00000000 00000000 00000001 00000010. Значення типу byte займають у пам'яті тільки 8 біт. Тому двійкове подання числа int усікається до 8 правих розрядів, тобто 00000010, що в десятковій системі дає число 2.

Усічення раціональних чисел до цілих

При перетворенні значень з плаваючою крапкою до цілочисельних значень, відбувається усічення дробової частини:

```
double a = 56.9898;
int b = (int)a;
```

Тут значення числа b дорівнюватиме 56, незважаючи на те, що число 57 було б ближчим до 56.9898. Щоб уникнути подібних казусів, треба застосовувати функцію округлення, яка є в математичній бібліотеці Java:

```
double a = 56.9898;
int b = (int)Math.round(a);
```

Перетворення під час операцій

Непоодинокі ситуації, коли доводиться застосовувати різні операції, наприклад, додавання і добуток, над значеннями різних типів. Тут також діють деякі правила:

- якщо один з операндів операції належить до типу double, то і другий операнд перетворюється до типу double;
- якщо попередня умова не дотримана, а один з операндів операції належить до типу float, то і другий операнд перетворюється до типу float;
- якщо попередні умови не дотримано, один з операндів операції відноситься до типу long, то і другий операнд перетворюється до типу long;
- інакше всі операнди операції перетворюються до типу int.

Приклади перетворень:

```
int a = 3;
double b = 4.6;
```

```
double c = a+b;
```

Оскільки в операції бере участь значення типу double, то й інше значення приводиться до типу double, і сума двох значень a+b буде представляти тип double.

Інший приклад:

```
byte a = 3;
short b = 4;
byte c = (byte)(a+b);
```

Дві змінні типу byte і short (не double, float або long), тому під час додавання вони перетворюються до типу int, і їхня сума a+b представляє значення типу int. Тому якщо потім ми присвоюємо цю суму змінній типу byte, то нам знову треба зробити перетворення типів до byte.

Якщо в операціях беруть участь дані типу char, то вони перетворюються в int:

```
int d = 'a' + 5;

System.out.println(d);  // 102
```

Умовні конструкції

Одним із фундаментальних елементів багатьох мов програмування є умовні конструкції. Ці конструкції дають змогу спрямувати роботу програми одним зі шляхів залежно від певних умов.

У мові Java використовуються такі умовні конструкції: if..else і switch..case

Конструкція if/else

Вираз if/else перевіряє істинність деякої умови і залежно від результатів перевірки виконує певний код:

```
int num1 = 6;
int num2 = 4;
if(num1>num2){
    System.out.println("Перше число більше другого");
}
```

Після ключового слова if ставиться умова. І якщо ця умова виконується, то спрацьовує код, який поміщено далі в блоці if після фігурних дужок. В якості умов виступає операція порівняння двох чисел.

Але що, якщо ми захочемо, щоб у разі недотримання умови також виконувалися якісь дії? У цьому випадку ми можемо додати блок else:

```
int num1 = 6;
int num2 = 4;
if(num1>num2){
    System.out.println("Перше число більше другого");
}
else{
    System.out.println("Перше число менше другого");
}
```

Але під час порівняння чисел ми можемо нарахувати три стани: перше число більше за друге, перше число менше за друге і числа рівні. За допомогою виразу else if, ми можемо обробляти додаткові умови:

```
int num1 = 6;
int num2 = 8;
if(num1>num2){
    System.out.println("Перше число більше другого ");
}
else if(num1<num2){
    System.out.println("Друге число більше першого");
}
else{
    System.out.println("Числа рівні");
}
```

Також ми можемо з'єднати відразу кілька умов, використовуючи логічні оператори:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2 && num1>7){
    System.out.println("Перше число більше другого та більше 7");
}
```

Тут блок if виконуватиметься, якщо num1 > num2 дорівнює true і одночасно num1>7 дорівнює true.

Конструкція switch

Конструкція switch/case аналогічна конструкції if/else, оскільки дає змогу обробити одразу кілька умов:

```
int num = 8;
switch(num){
    case 1:
        System.out.println("число дорівнює 1");
        break;
    case 8:
        System.out.println("число дорівнює 8");
        num++;
        break;
    case 9:
        System.out.println("число дорівнює 9");
        break;
    default:
        System.out.println("число не дорівнює 1, 8, 9");
}
```

Після ключового слова switch у дужках йде порівнюваний вираз. Значення цього виразу послідовно порівнюється зі значеннями, поміщеними після операторів case. І якщо збіг знайдено, то буде виконує відповідний блок case.

Наприкінці блоку case ставиться оператор break, щоб уникнути виконання інших блоків. Наприклад, якби прибрали оператор break у такому випадку:

```
case 8:
    System.out.println("число дорівнює 8");
    num++;
case 9:
    System.out.println("число дорівнює 9");
    break;
```

то виконався б блок case 8, (оскільки змінна num дорівнює 8). Але оскільки в цьому блоці оператор break відсутній, то почав би виконуватися блок case 9.

Якщо ми хочемо також обробити ситуацію, коли збігу не буде знайдено, то можна додати блок default, як у прикладі вище. Хоча блок default необов'язковий.

Також ми можемо визначити одну дію відразу для кількох блоків case поспіль:

```
int num = 3;
int output = 0;
switch(num){
    case 1:
        output = 3;
        break;
    case 2:
    case 3:
    case 4:
        output = 6;
        break;
    case 5:
        output = 12;
        break;
    default:
        output = 24;
}
System.out.println(output);
```

Тернарна операція

Тернарна операція має такий синтаксис: [перший операнд - умова] ? [другий операнд] : [третій операнд]. Таким чином, у цій операції беруть участь одразу три операнди. Залежно від умови тернарна операція повертає другий або

третій операнд: якщо умова дорівнює true, то повертається другий операнд; якщо умова дорівнює false, то третій. Наприклад:

```
int x=3;
int y=2;
int z = x<y? (x+y) : (x-y);
System.out.println(z);
```

Тут результатом тернарної операції є змінна z. Спочатку перевіряється умова x<y. І якщо вона виконується, то z дорівнюватиме другому операнду - (x+y), інакше z дорівнюватиме третьому операнду.

Цикли

Ще одним видом керуючих конструкцій є цикли. Цикли дозволяють залежно від певних умов виконувати певну дію безліч разів. У мові Java є такі види циклів:

- for
- while
- do...while

Цикл for

Цикл for має таке формальне визначення:

for ([ініціалізація лічильника]; [умова]; [зміна лічильника])

{

  // дії

}

Розглянемо стандартний цикл for:

```
for (int i = 1; i < 9; i++){
    System.out.printf("Квадрат числа %d дорівнює %d \n", i, i * i);
}
```

Перша частина оголошення циклу - int i = 1 створює та ініціалізує лічильник i. Лічильник необов'язково повинен представляти тип int. Це може бути і будь-який інший числовий тип, наприклад, float. Перед виконанням циклу значення лічильника дорівнюватиме 1. У цьому випадку це те ж саме, що й оголошення змінної.

Друга частина - умова, за якої буде виконуватися цикл. У цьому випадку цикл буде виконуватися, поки i не досягне 9.

І третя частина - збільшення лічильника на одиницю. Знову ж таки нам необов'язково збільшувати на одиницю. Можна зменшувати: i--.

У підсумку блок циклу спрацює 8 разів, поки значення і не стане рівним 9. І щоразу це значення буде збільшуватися на 1.

Нам необов'язково вказувати всі умови під час оголошення циклу. Наприклад, ми можемо написати так:

```
int i = 1;
for (; ;){
    System.out.printf("Квадрат числа %d дорівнює %d \n", i, i * i);
}
```

Визначення циклу залишилося тим самим, тільки тепер блоки у визначенні в нас порожні: for (; ;). Тепер немає ініціалізованої змінної-лічильника, немає умови, тому цикл працюватиме вічно - нескінченний цикл.

Або можна опустити низку блоків:

```
int i = 1;
for (; i<9;){
    System.out.printf("Квадрат числа %d дорівнює %d \n", i, i * i);
    i++;
}
```

Цей приклад еквівалентний першому прикладу: у нас також є лічильник, тільки створений він поза циклом. У нас є умова виконання циклу. І є приріст лічильника вже в самому блоці for.

Цикл for може визначати відразу кілька змінних і керувати ними:

```
int n = 10;
for(int i=0, j = n - 1; i < j; i++, j--){
    System.out.println(i * j);
}
```

Цикл do

Цикл do спочатку виконує код циклу, а потім перевіряє умову в інструкції while. І поки ця умова істинна, цикл повторюється. Наприклад:

```
int j = 7;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```

У цьому випадку код циклу спрацює 7 разів, поки j не виявиться рівним нулю. Важливо зазначити, що цикл do гарантує хоча б одноразове виконання дій, навіть якщо умова в інструкції while не буде істинною. Так, ми можемо написати:

```
int j = -1;
```

```
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```

Хоча змінна j від початку менша за 0, цикл все одно один раз виконається.

Цикл while

Цикл while відразу перевіряє істинність деякої умови, і якщо умова істинна, то код циклу виконується:

```
int j = 6;
while (j > 0){
    System.out.println(j);
    j--;
}
```

Оператори continue і break

Оператор break дозволяє вийти з циклу в будь-який його момент, навіть якщо цикл не закінчив свою роботу:

Наприклад:

```
for (int i = 0; i < 10; i++){
    if (i == 5)
        break;
    System.out.println(i);
}
```

Коли лічильник стане рівним 5, спрацює оператор break, і цикл завершиться.

Тепер зробимо так, щоб якщо число дорівнює 5, цикл не завершувався, а просто переходив до наступної ітерації. Для цього використовуємо оператор continue:

```
for (int i = 0; i < 10; i++){
    if (i == 5)
        continue;
    System.out.println(i);
}
```

У цьому випадку, коли виконання циклу дійде до числа 5, програма просто пропустить це число і перейде до наступного.

Масиви

Масив являє собою набір однотипних значень. Оголошення масиву схоже на оголошення звичайної змінної, яка зберігає одиночне значення, причому є два способи оголошення масиву:

тип_даних назва_масиву[];

// або

тип_даних [] назва_масиву;

Наприклад, визначимо масив чисел:

```
int nums[];
int[] nums2;
```

Після оголошення масиву ми можемо ініціалізувати його:

Створення масиву здійснюється за допомогою такої конструкції: new тип_даних[кількість_елементів], де new - ключове слово, що виділяє пам'ять для зазначеної в дужках кількості елементів. Наприклад, nums = new int[4]; - у цьому виразі створюється масив із чотирьох елементів int, і кожен елемент матиме значення за замовчуванням - число 0.

Також можна відразу при оголошенні масиву ініціалізувати його:

```
int nums[] = new int[4];     // масив з 4 чисел
int[] nums2 = new int[5];    // масив з 5 чисел
```

При подібній ініціалізації всі елементи масиву мають значення за замовчуванням. Для числових типів (у тому числі для типу char) це число 0, для типу boolean це значення false, а для решти об'єктів це значення null. Наприклад, для типу int значенням за замовчуванням є число 0, тому вище визначений масив nums складатиметься з чотирьох нулів.

Однак також можна задати конкретні значення для елементів масиву при його створенні:

```
int[] nums = new int[] { 1, 2, 3, 5 };
int[] nums2 = { 1, 2, 3, 5 };
```

Варто зазначити, що в цьому випадку в квадратних дужках не вказується розмір масиву, оскільки він обчислюється за кількістю елементів у фігурних дужках.

Після створення масиву ми можемо звернутися до будь-якого його елемента за індексом, який передається в квадратних дужках після назви змінної масиву:

```
int[] nums = new int[4];
nums[0] = 1;
nums[1] = 2;
nums[2] = 4;
nums[3] = 100;
System.out.println(nums[2]);     // 4
```

Індексація елементів масиву починається з 0, тому в цьому випадку, щоб звернутися до четвертого елемента в масиві, нам треба використовувати вираз nums[3].

І оскільки в нас масив визначений тільки для 4 елементів, то ми не можемо звернутися, наприклад, до шостого елемента: nums[5] = 5;. Якщо ми так спробуємо зробити, то ми отримаємо помилку.

Довжина масиву

Найважливішою властивістю, яку мають масиви, є властивість length, що повертає довжину масиву, тобто кількість його елементів:

```
int[] nums = {1, 2, 3, 4, 5};
int length = nums.length;   // 5
```

Нерідко буває невідомим останній індекс, і щоб отримати останній елемент масиву, ми можемо використовувати цю властивість:

```
int last = nums[nums.length-1];
```

Багатовимірні масиви

Раніше ми розглядали одновимірні масиви, які можна представити як ланцюжок або рядок однотипних значень. Але крім одновимірних масивів також бувають і багатовимірними. Найвідоміший багатовимірний масив - таблиця, що представляє двовимірний масив:

```
int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };
int[][] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Оскільки масив nums2 двовимірний, він являє собою просту таблицю. Його також можна було створити таким чином: int[][] nums2 = new int[2][3];. Кількість квадратних дужок вказує на розмірність масиву. А числа в дужках - на кількість рядків і стовпців. І також, використовуючи індекси, ми можемо використовувати елементи масиву в програмі:

```
// встановимо елемент першого стовпця другого рядка
nums2[1][0]=44;
System.out.println(nums2[1][0]);
```

Оголошення тривимірного масиву могло б виглядати так:

```
int[][][] nums3 = new int[2][3][4];
```

Зубчастий масив

Багатовимірні масиви можуть бути також представлені як "зубчасті масиви". У вищенаведеному прикладі двовимірний масив мав 2 рядки і три стовпці, тому в нас виходила рівна таблиця. Але ми можемо кожному елементу в двовимірному масиві присвоїти окремий масив із різною кількістю елементів:

```
int[][] nums = new int[3][];
nums[0] = new int[2];
nums[1] = new int[3];
nums[2] = new int[5];
```

foreach

Спеціальна версія циклу for призначена для перебору елементів у наборах елементів, наприклад, у масивах і колекціях. Вона аналогічна дії циклу foreach , який є в інших мовах програмування. Формальне її оголошення:

```
for (тип_даних назва_змінної : контейнер){
    // дії
}
```

Наприклад:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i : array){
    System.out.println(i);
}
```

Як контейнер у цьому випадку виступає масив даних типу int. Потім оголошується змінна з типом int

Те ж саме можна було б зробити і за допомогою звичайної версії for:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < array.length; i++){
    System.out.println(array[i]);
}
```

Водночас ця версія циклу for гнучкіша порівняно з for (int i : array). Зокрема, у цій версії ми можемо змінювати елементи:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i=0; i<array.length;i++){
    array[i] = array[i] * 2;
    System.out.println(array[i]);
}
```

Перебір багатовимірних масивів у циклі

```
int[][] nums = new int[][]
{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
for (int i = 0; i < nums.length; i++){
    for(int j=0; j < nums[i].length; j++){

        System.out.printf("%d ", nums[i][j]);
    }
```

```
            System.out.println();
        }
```

Спочатку створюється цикл для перебору за рядками, а потім усередині першого циклу створюється внутрішній цикл для перебору за стовпцями конкретного рядка. Подібним чином можна перебрати і тривимрні масиви та набори з великою кількістю розмірностей.

Завдання до виконання

Виконати завдання згідно власного варіанту (таб. 2.1). Всі проміжні результати надрукувати.

Таблиця 2.1 – Варіанти завдань

№	Завдання
1.	Напишіть програму, яка виводить усі натуральні числа у зворотному порядку
2.	Заданий масив X=(x1,x2,...,xn), в якому можуть бути однакові числа. Знайти максимальний і мінімальний елементи серед неповторюваних чисел.
3.	Створіть програму, що виводить на екран перші 20 елементів послідовності 2 4 8 8 16 32 64 64 128 ...
4.	Напишіть програму, яка буде вводити числа до тих пір, поки цього бажає користувач, а в кінці програма повинна вивести найбільше і найменше введене число.
5.	Елементи масиву X = (x1,x2,...,xn) – це послідовність цифр цілого числа. Переставити цифри числа у зворотному порядку
6.	Напишіть програму на Java, яка знаходить значення, що повторюються у масиві рядкових значень.
7.	За однократний перегляд масиву знайти його максимальний позитивний елемент X_{max}
8.	Напишіть програму на Java, яка обчислює середнє значення масиву цілих чисел за винятком найбільшого та найменшого значень.
9.	У масиві X=(x1,x2,...,xn) поміняти місцями перший і другий позитивні елементи, третій і четвертий позитивні елементи тощо.
10.	Масив X=(x1,x2,...,xn) містить велику кількість нульових елементів. Визначити положення і розмір найбільш довгої серії ненульових елементів.

Приклад виконання

Заданий масив цілих чисел X=(x1,x2,...,xn). Сформувати масив Y=(y1,y2,...,ym), помістивши в нього в порядку убування всі позитивні числа, що входять у масив X.

```
public static void main(String[] args) {
    int[] x = new int[20];
    Random rand = new Random();
    int counter = 0;
    System.out.println("X array:");
    for(int i = 0; i < x.length; i++) {
        x[i] = rand.nextInt( origin: -50,  bound: 50);
        if (x[i] > 0)
            counter++;
        System.out.print(x[i]);
        System.out.print(',');
    }
    int buff = 0;
    for(int i = 0; i < x.length; i++) // сортування бульбашкою в порядку убування
        for(int j = 0; j < x.length-1; j++){
            if (x[j] < x[j + 1]){
                buff = x[j];
                x[j] = x[j + 1];
                x[j + 1] = buff;
            }
        }
    System.out.println("\nY array:");
    int[] y = new int[counter];
    for(int i = 0; i < y.length; i++){
        y[i] = x[i];
        System.out.print(y[i]);
        if(i != y.length -1)
            System.out.print(',');
    }
}
```

Рисунок 2.2 – Приклад виконання

Контрольні питання

1. Які особливості мови програмування Java вам відомі?
2. Що розуміється під терміном "тип даних", і які типи даних існують у мові програмування Java?
3. Що означає термін "префіксний" та "постфіксний інкремент" у контексті програмування?
4. Що означає термін "префіксний" та "постфіксний декремент" у програмуванні?
5. Який пріорітет арифметичних операцій у мові програмування Java?
6. Які операції порівняння існують в мові програмування Java?
7. Що представляє собою поняття "масив" у мові програмування Java?
8. Яка різниця між циклом foreach та циклом for у мові програмування Java?
9. В чому полягає різниця між циклами do...while та while у мові програмування Java?
10. Як використовуються оператори continue та break для управління складною логікою циклу у мові програмування Java?

ЛАБОРАТОРНА РОБОТА. РОБОТА З МЕТОДАМИ. ВИВЧЕННЯ РОБОТИ ЗІ СТРУКТУРАМИ ДАНИХ ТА КОЛЕКЦІЯМИ

Мета роботи: ознайомитись з роботою методів, опанувати структури даних та колекції.

Основні теоретичні відомості

Методи у Java

Якщо змінні та константи зберігають деякі значення, то методи містять набір операторів, які виконують певні дії.

Загальне визначення методів виглядає таким чином:

```
[модифікатори] тип_поверненого_значення назва_методу ([параметри]){
    // тіло методу
}
```

Модифікатори та параметри необов'язкові.

За замовчуванням головний клас будь-якої програми на Java містить метод main, який слугує точкою входу в програму:

```
public static void main(String[] args) {
    System.out.println("Привіт!");
}
```

Ключові слова public і static є модифікаторами. Далі йде тип значення, що повертається. Ключове слово void вказує на те, що метод нічого не повертає.

Потім ідуть назва методу - main і в дужках параметри методу - String[] args. І у фігурні дужки укладено тіло методу - усі дії, які він виконує.

Створимо ще кілька методів:

```
public class Program{
    public static void main (String args[]){

    }
    void hello(){

        System.out.println("Hello");
    }
    void welcome(){

        System.out.println("Welcome my dear friend!");
    }
}
```

Тут визначено два додаткові методи: hello і welcome, кожен з яких виводить деякий рядок на консоль. Методи визначаються всередині класу - у цьому випадку всередині класу Program, у якому визначено метод main.

Але якщо ми скомпілюємо і запустимо цю програму, то ми нічого не побачимо на консолі. У прикладі вище ми визначили два методи, але ми їх ніде не викликаємо. За замовчуванням у програмі Java виконується тільки метод main і весь його вміст. Тому, якщо ми хочемо, щоб інші методи теж виконувалися, їх треба викликати в методі main.

Виклик методу здійснюється у формі:

Після імені методу вказуються дужки, в яких перераховуються аргументи - значення для параметрів методу.

Наприклад, визначимо і виконаємо кілька методів:

```
public class Program{
    public static void main (String args[]){
         hello();
         welcome();
         welcome();
    }
    static void hello(){

        System.out.println("Hello");
    }
    static void welcome(){

        System.out.println("Welcome to Java 10");
    }
}
```

У методі main викликається один раз метод hello і два рази метод welcome. У цьому й полягає одна з переваг методів: ми можемо винести деякі загальні дії в окремий метод і потім викликати багаторазово їх у різних місцях програми. Оскільки обидва методи не мають жодних параметрів, то після їхньої назви під час виклику ставляться порожні дужки.

Також слід зазначити, що щоб викликати в методі main інші методи, які визначені в одному класі з методом main, вони повинні мати модифікатор static.

Параметри методів

За допомогою параметрів ми можемо передати в методи різні дані, які будуть використовуватися для обчислень. Наприклад:

```
static void sum(int x, int y){
   int z = x + y;
   System.out.println(z);
}
```

Ця функція приймає два параметри - два числа, додає їх і виводить їхню суму на консоль.

А при виклику цього методу в програмі нам необхідно передати на місце параметрів значення, які відповідають типу параметра:

Оскільки метод sum приймає два значення типу int, то на місце параметрів треба передати два значення типу int. Це можуть бути і числові літерали, і змінні типів даних, які представляють тип int або можуть бути автоматично перетворені в тип int. Значення, які передаються на місце параметрів, ще називаються аргументами. Значення передаються параметрам за позицією, тобто перший аргумент першому параметру, другий аргумент - другому параметру і так далі.

Розглянемо інший приклад:

```
public class Program{

    public static void main (String args[]){

        display("Tom", 34);
        display("Bob", 28);
        display("Sam", 23);
    }
    static void display(String name, int age){

        System.out.println(name);
        System.out.println(age);
    }
}
```

Метод display приймає два параметри. Перший параметр представляє тип String, а другий - тип int. Тому при виклику методу спочатку в нього треба передати рядок, а потім число.

Параметри змінної довжини

Метод може приймати параметри змінної довжини одного типу. Наприклад, нам треба передати в метод набір чисел і обчислити їхню суму, але ми точно не знаємо, скільки саме чисел буде передано - 3, 4, 5 або більше. Параметри змінної довжини дають змогу вирішити це завдання:

```
public class Program{

    public static void main (String args[]){

        sum(1, 2, 3);           // 6
        sum(1, 2, 3, 4, 5);     // 15
        sum();                  // 0
    }
    static void sum(int ...nums){

        int result =0;
        for(int n: nums)
```

```
            result += n;
        System.out.println(result);
    }
}
```

Трикрапка перед назвою параметра int ...nums вказує на те, що він буде необов'язковим і представлятиме масив. Ми можемо передати в метод sum одне число, кілька чисел, а можемо взагалі не передавати жодних параметрів. Причому, якщо ми хочемо передати кілька параметрів, то необов'язковий параметр повинен вказуватися в кінці.

```
public static void main(String[] args) {

    sum("Welcome!", 20,10);
    sum("Hello World!");
}
static void sum(String message, int ...nums){

    System.out.println(message);
    int result =0;
    for(int x:nums)
        result+=x;
    System.out.println(result);
}
```

Оператор return. Результат методу

Методи можуть повертати деяке значення. Для цього застосовується оператор return.

```
return повернуте_значення;
```

Після оператора return вказується значення, що повертається, яке є результатом методу. Це може бути літеральне значення, значення змінної або якогось складного виразу.

Наприклад:

```
public class Program{

    public static void main (String args[]){

        int x = sum(1, 2, 3);
        int y = sum(1, 4, 9);
        System.out.println(x);  // 6
        System.out.println(y);  // 14
    }
    static int sum(int a, int b, int c){

        return a + b + c;
    }
}
```

У методі як тип значення, що повертається, замість void використовується будь-який інший тип. У цьому випадку метод sum повертає значення типу int, тому цей тип вказується перед назвою методу. Причому якщо як тип, що повертається, для методу визначено будь-який інший, відмінний від void, то

метод обов'язково повинен використовувати оператор return для повернення значення.

При цьому значення, що повертається, завжди повинно мати той самий тип, що значиться у визначенні функції. І якщо функція повертає значення типу int, то після оператора return стоїть цілочисельне значення, яке є об'єктом типу int. Як у даному випадку це сума значень параметрів методу.

Метод може використовувати кілька викликів оператора return для повернення різних значень залежно від деяких умов:

```
public class Program{
    public static void main (String args[]){
        System.out.println(daytime(7));     // Good morning
        System.out.println(daytime(13));    // Good after noon
        System.out.println(daytime(18));    // Good evening
        System.out.println(daytime(2));     // Good night
    }
    static String daytime(int hour){
        if (hour >24 || hour < 0)
            return "Invalid data";
        else if(hour > 21 || hour < 6)
            return "Good night";
        else if(hour >= 15)
            return "Good evening";
        else if(hour >= 11)
            return "Good after noon";
        else
            return "Good morning";
    }
}
```

Тут метод daytime повертає значення типу String, тобто рядок, і залежно від значення параметра hour рядок, що повертається, буде відрізнятися.

Вихід із методу

Оператор return застосовується не тільки для повернення значення з методу, а й для виходу з методу. У подібній якості оператор return застосовується в методах, які нічого не повертають, тобто мають тип void:

```
public class Program{
    public static void main (String args[]){
        daytime(7);     // Good morning
        daytime(13);    // Good after noon
        daytime(32);    //
        daytime(56);    //
        daytime(2);     // Good night
    }
```

```
    static void daytime(int hour){
        if (hour >24 || hour < 0)
            return;
        if(hour > 21 || hour < 6)
            System.out.println("Good night");
        else if(hour >= 15)
            System.out.println("Good evening");
        else if(hour >= 11)
            System.out.println("Good after noon");
        else
            System.out.println("Good morning");
    }
}
```

Якщо передане в метод datetime значення більше 24 або менше 0, то просто виходимо з методу. Значення, що повертається, після return вказувати в цьому випадку не потрібно.

Перевантаження методів

У програмі ми можемо використовувати методи з одним і тим самим іменем, але з різними типами та/або кількістю параметрів. Такий механізм називається перевантаженням методів (method overloading).

Наприклад:

```
public class Program{
    public static void main(String[] args) {
        System.out.println(sum(2, 3));          // 5
        System.out.println(sum(4.5, 3.2));      // 7.7
        System.out.println(sum(4, 3, 7));       // 14
    }
    static int sum(int x, int y){
        return x + y;
    }
    static double sum(double x, double y){
        return x + y;
    }
    static int sum(int x, int y, int z){
        return x + y + z;
    }
}
```

Тут визначено три варіанти або три перевантаження методу sum(), але під час його виклику залежно від типу та кількості переданих параметрів система вибере саме ту версію, яка найбільше підходить.

Варто зазначити, що на перевантаження методів впливають кількість і типи параметрів. Однак відмінність у типі значення, що повертається, для перевантаження не мають жодного значення. Наприклад, у такому випадку методи розрізняються за типом значення, що повертається:

```
public class Program{
    public static void main(String[] args) {

        System.out.println(sum(2, 3));
        System.out.println(sum(4, 3));
    }
    static int sum(int x, int y){
```

```
        return x + y;
    }
    static double sum(int x, int y){
        return x + y;
    }
}
```

Однак перевантаженням це не вважатиметься. Ба більше, така програма некоректна і просто не скомілюється, оскільки метод з однією і тією ж кількістю і типом параметрів визначено кілька разів.

Рекурсивні функціїї

Окремо розглянемо рекурсивні функції. Головна відмінність рекурсивних функцій від звичайних методів полягає в тому, що вони рекурсивна функція може викликати саму себе.

Наприклад, розглянемо функцію, яка обчислює факторіал числа:

```
static int factorial(int x){
    if (x == 1){
        return 1;
    }
    return x * factorial(x - 1);
}
```

Спочатку перевіряється умова: якщо число, що вводиться, не дорівнює 1, то ми множимо це число на результат цієї самої функції, в яку як параметр передається число x-1. Тобто відбувається рекурсивний спуск. І так далі, поки не дійдемо того моменту, коли значення параметра не дорівнюватиме одиниці.

Рекурсивна функція обов'язково повинна мати деякий базовий варіант, який використовує оператор return і який поміщається на початку функції. У випадку з факторіалом це if (x == 1) return 1;. І всі рекурсивні виклики мають звертатися до підфункцій, які зрештою сходяться до базового варіанта. Так, у разі передачі у функцію додатного числа під час подальших рекурсивних викликів підфункцій у них передаватиметься щоразу число, менше на одиницю. І врешті-решт ми дійдемо до ситуації, коли число дорівнюватиме 1, і буде використано базовий варіант.

Хоча в даному випадку потрібно зазначити, що для визначення факторіалу є більш оптимальні рішення на основі циклів:

```
static int factorial(int x){
    int result=1;
    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    return  result;
```

```
    }
```

Ще одним поширеним прикладом рекурсивної функції слугує функція, що обчислює числа Фібоначчі. У теорії n-й член послідовності Фібоначчі визначається за формулою: f(n)=f(n-1) + f(n-2), причому f(0)=0, а f(1)=1.

```
    static int fibonachi(int n){
        if (n == 0){
            return 0;
        }
        if (n == 1){
            return 1;
        }
        else{
            return fibonachi(n - 1) + fibonachi(n - 2);
        }
}
```

Структури даних та колекції

Для зберігання наборів даних у Java призначені масиви. Однак їх не завжди зручно використовувати, насамперед тому, що вони мають фіксовану довжину. Цю проблему в Java вирішують колекції. Однак суть не тільки в гнучких за розміром наборах об'єктів, а й у тому, що класи колекцій реалізують різні алгоритми і структури даних, наприклад, такі як стек, черга, дерево і низка інших.

Вибір певного класу для роботи з колекціями визначає набір методів, які будуть доступні для об'єкта цього класу. Наприклад, якщо використовується список (який визначає інтерфейс List), то існує багатий вибір для його реалізації: ArrayList, LinkedList, Vector, Stack. Конкретний вибір реалізації списку позначається на ефективності маніпуляцій з об'єктами списку. Так, ArrayList зберігає елементи у вигляді масиву, а отже, доступ і заміна буде виконуватися відносно швидко. Водночас LinkedList зберігає елементи у вигляді зв'язного списку, що тягне за собою відносно повільний пошук елементів і швидку операцію додавання/видалення. Рекомендується ознайомитися з описами наступних колекцій з JSDK-документації:

- java.util.Collection ;
- java.util.ArrayList;
- java.util.HashMap;
- java.util.HashSet.

Важливе також знання основних класів для роботи з колекціями. На особливу увагу при вивченні колекцій заслуговують класи Comporator, Collections, Iterator.

Нижче наведено стислий розбір найважливіших класів під час роботи з колекціями, а також розв'язання одного з індивідуальних завдань.

Інтерфейс Collection

Інтерфейс Collection містить набір загальних методів, які використовуються в більшості колекцій. Розглянемо основні з них:

- add(Object item) - додає в колекцію новий елемент, якщо елементи колекції якимось чином упорядковані, новий елемент додається в кінець колекції;
- clear() - видаляє всі елементи колекції;
- contains(Object obj) - повертає true, якщо об'єкт obj міститься в колекції і false, якщо ні;
- isEmpty() - перевіряє, чи порожня колекція;
- remove(Object obj) - видаляє з колекції елемент obj, повертає false, якщо такого елемента в колекції не знайшлося;
- size() - повертає кількість елементів колекції.

Інтерфейс List

Інтерфейс List описує впорядкований список. Елементи списку пронумеровані, починаючи з нуля і до конкретного елемента можна звернутися за цілочисельним індексом. Інтерфейс List є спадкоємцем інтерфейсу Collection, тому містить усі його методи і додає до них кілька своїх:

- add(int index, Object item) - вставляє елемент item у позицію index, водночас список розсувається (усі елементи, починаючи з позиції index, збільшують свій індекс на 1);
- get(int index) - повертає об'єкт, що знаходиться в позиції index;
- indexOf(Object obj) - повертає індекс першої появи елемента obj у списку;

- lastIndexOf(Object obj) - повертає індекс останньої появи елемента obj у списку;
- add(int index, Object item) - замінює елемент, що міститься в позиції index об'єктом item;
- subList(int from, int to) - повертає новий список, що являє собою частину даного (починаючи з позиції from до позиції to-1 включно).

Інтерфейс Set

Інтерфейс Set описує множину. Елементи множини не впорядковані, множина не може містити двох однакових елементів. Інтерфейс Set успадкований від інтерфейсу Collection, але жодних нових методів не додає. Змінюється тільки сенс методу add(Object item) - він не додає об'єкт item, якщо він уже присутній у множині.

Інтерфейс Queue

Інтерфейс Queue описує чергу. Елементи можуть додаватися в чергу тільки з одного кінця, а витягуватися з іншого (аналогічно черзі в магазині). Інтерфейс Queue так само успадкований від інтерфейсу Collection. Специфічні для черги методи:

poll() - повертає перший елемент і видаляє його з черги;

peek() - повертає перший елемент черги, не видаляючи його.

offer(Object obj) - додає в кінець черги новий елемент і повертає true, якщо вставка вдалася.

Клас ArrayList

Клас ArrayList - аналог класу Vector (застарілий клас). Він являє собою список і може використовуватися в тих самих ситуаціях. Основна відмінність у тому, що він не синхронізований і одночасна робота декількох паралельних процесів з об'єктом цього класу не рекомендується. У звичайних же ситуаціях він працює швидше.

Клас Stack

Stack – колекція, що об'єднує елементи в стек. Стек працює за принципом LIFO (останнім прийшов - першим пішов). Елементи кладуть у стек "один на одного", до того ж узяти можна тільки "верхній" елемент, тобто той, який було покладено в стек останнім. Для стека характерні операції, реалізовані в таких методах класу Stack:

- push(Object item) - поміщає елемент на вершину стека;
- pop() - витягує зі стека верхній елемент;
- peek() - повертає верхній елемент, не витягуючи його зі стека;
- empty() - перевіряє, чи не порожній стек;
- search(Object item) - шукає "глибину" об'єкта в стеку. Верхній елемент має позицію 1, той, що знаходиться під ним, - 2 і т.д. Якщо об'єкта в стеку немає, повертає -1.

Клас Stack є спадкоємцем класу Vector, тому має всі його методи (і, зрозуміло, реалізує інтерфейс List). Однак якщо в програмі потрібно моделювати саме стек, рекомендується використовувати тільки п'ять перерахованих вище методів.

Інтерфейс Iterator

Перевага використання масивів і колекцій полягає не тільки в тому, що можна помістити в них довільну кількість об'єктів і вилучати їх за потреби, а й у тому, що всі ці об'єкти можна комплексно обробляти. Наприклад, вивести на екран усі шашки, що містяться у списку checkers. У разі масиву ми користуємося циклом:

```
for (int i = 1; i < array.length; i++){
// обробляємо елемент array[i]
}
```

Маючи справу зі списком, ми можемо вчинити аналогічно, тільки замість array[i] писати array.get(i). Але ми не можемо вчинити так з колекціями, елементи яких не індексуються (наприклад, чергою або безліччю). А в разі індексованої колекції треба добре знати особливості її роботи: як визначити

кількість елементів, як звернутися до елемента за індексом, чи може колекція бути розрідженою (тобто чи можуть існувати індекси, з якими не пов'язано жодних елементів) тощо.

Для навігації по колекціях в Java передбачено спеціальне архітектурне рішення, яке отримало свою реалізацію в інтерфейсі Iterator. Ідея полягає в тому, що до колекції "прив'язується" об'єкт, єдине призначення якого - видати всі елементи цієї колекції в деякому порядку, не розкриваючи її внутрішню структуру.

Інтерфейс Iterator має всього три методи:

- next() - повертає черговий елемент колекції, до якої "прив'язаний" ітератор (і робить його поточним). Порядок перебору визначає сам ітератор;
- hasNext() - повертає true, якщо перебір елементів ще не закінчено;
- remove() - видаляє поточний елемент.

Інтерфейс Collection крім розглянутих раніше методів, має метод iterator(), який повертає ітератор для даної колекції, готовий до її обходу. За допомогою такого ітератора можна обробити всі елементи будь-якої колекції таким простим способом:

```
Iterator iter = coll.iterator(); // coll - колекція
while (iter.hasNext()) {
// обробляємо об'єкт, що повертається методом iter.next()
}
```

Для колекцій, елементи яких проіндексовані, визначено більш функціональний ітератор, що дає змогу рухатися як у прямому, так і в зворотному напрямку, а також додавати в колекцію елементи. Такий ітератор має інтерфейс ListIterator, успадкований від інтерфейсу Iterator, який доповнює його такими методами:

previous() - повертає попередній елемент (і робить його поточним);

hasPrevious() - повертає true, якщо попередній елемент існує (тобто поточний елемент не є першим елементом для цього ітератора);

add(Object item) - додає новий елемент перед поточним елементом;

set(Object item) - замінює поточний елемент;

nextIndex() і previousIndex() - служать для отримання індексів наступного і попереднього елементів відповідно.

В інтерфейсі List визначено метод listIterator(), що повертає ітератор ListIterator для обходу даного списку.

Інтерфейс Map

Інтерфейс Map з пакета java.util описує колекцію, що складається з пар "ключ - значення", які широко використовуються для зберігання налаштувань у файлах конфігурації (див., наприклад, /etc/services). У кожного ключа тільки одне значення, що відповідає математичному поняттю однозначної функції або відображення (Map). Інтерфейс Map містить такі методи, що працюють із ключами та значеннями:

- boolean containsKey (Object key) - перевіряє наявність ключа key;
- boolean containsValue (Object value) - перевіряє наявність значення value;
- Set entry Set() - представляє колекцію у вигляді множини, кожен елемент якої - пара з даного відображення, з якою можна працювати методами вкладеного інтерфейсу Map.Entry;
- Object get(Object key) - повертає значення, що відповідає ключу key; S et key S et() - представляє ключі колекції у вигляді множини.

Завдання до виконання

Виконати 2 завдання згідно свого варіанту (табл 3.1)

Таблиця 3.1 – Варіанти завдань

№	Завдання
1.	За заданим елементом написати програму, яка перевіряє, чи існує елемент (значення) в ArrayList.
	Напишіть програму, яка сортує HashMap за ключами.
2.	Написати програму, яка сортує масив ArrayList за спаданням.
	Напишіть програму, яка отримує значення на основі ключа з колекції HashMap
3.	Написати програму для видалення елементу з заданого індексу ArrayList.
	Написати програму для перевірки вставки дублікатів ключів у HashMap
4.	Напишіть програму для обходу ArrayList з використанням циклу for, while і поліпшеного циклу for
	Напишіть програму для ітерації об'єкта типу HashMap з використанням циклу while і поліпшеного циклу for
5.	Написати програму для перетворення хеш-набору в список/масив List/ArrayList
	За заданим елементом написати програму, яка перевіряє, чи існує елемент (значення) в ArrayList.
6.	Напишіть програму на Java, яка додає заданий елемент в кінець HashSet
	Напишіть програму на Java для перемішування елементів у зв'язаному списку.
7.	Написати програму для перевірки вставки дублікатів ключів у HashMap
	Напишіть програму для ітерації об'єкта типу HashMap з використанням циклу while і поліпшеного циклу for
8.	Написати програму, яка видаляє елементи HashSet, що містяться в інших HashSet
	Напишіть програму на Java для заміни елементу у зв'язаному списку.
9.	Напишіть програму, яка вставляє заданий елемент у PriorityQueue.
	Напишіть програму для підрахунку кількості конкретних слів у рядку, використовуючи HashMap
10.	Напишіть програму для перетворення хеш-набору в масив.
	Напишіть програму на Java, яка додає заданий елемент в кінець хеш-набору.

Приклад виконання

Написати програму на Java для сортування заданого масиву ArrayList. Напишіть програму на Java, яка копіює всі відображення з заданої карти на іншу карту.

```
public static void main(String[] args){
    // First task
    ArrayList<String> cities = new ArrayList<>();
    cities.add("Kyiv");
    cities.add("Dnipro");
    cities.add("Lutsk");
    cities.add("Lviv");
    cities.add("Donetsk");
    System.out.println("Before sorting:\n" + cities);
    Collections.sort(cities);
    System.out.println("After sorting:\n" + cities);
    // Second task
    HashMap<String, String> firstMap = new HashMap<>();
    firstMap.put("Kyiv", "Ukraine");
    firstMap.put("Paris", "France");
    firstMap.put("Dublin", "Ireland");
    firstMap.put("Warsaw", "Poland");
    System.out.println("FirstMap:\n" + firstMap);
    HashMap<String, String> secondMap = new HashMap<>(firstMap);
    System.out.println("SecondMap\n" + secondMap);
```

Рисунок 3.1 – Приклад виконання

Контрольні питання

1. Які основні переваги використання методів у програмуванні, і чому вони важливі для покращення коду?
2. Що таке структури даних в Java, і як вони відрізняються від звичайних змінних?
3. Як створити власний метод у Java, і які основні кроки для його визначення?
4. Які основні типи колекцій доступні у Java, і які це має значення для роботи з даними?
5. Які основні операції можна виконати з колекціями в Java, такі як додавання, видалення та пошук елементів?
6. Як визначити і викликати метод з параметрами у Java, і які значення можна передавати методам?
7. Як використовувати цикли для роботи з колекціями в Java, зокрема для ітерації через їхні елементи?
8. Які приклади задач можуть бути вирішені за допомогою структур даних та колекцій в Java, і як вони сприяють покращенню ефективності програм?
9. Які основні методи доступу до даних в колекціях, такі як ArrayList або HashMap, і як їх використовувати для отримання та зміни даних?

ЛАБОРАТОРНА РОБОТА. РОБОТА З КЛАСАМИ. ООП В JAVA

Мета роботи: ознайомитись з роботою з класами в java. Вивчити основні поняття ООП: успадкування, поліморфізм та інкапсуляцію.

Основні теоретичні відомості

Класи та об’єкти

Об’єктно-орієнтоване програмування (ООП) – одна з парадигм програмування, яка розглядає програму як множину “об’єктів”, що взаємодіють між собою. В ній використано декілька технологій від попередніх парадигм, зокрема успадкування, модульність, поліморфізм та інкапсуляцію. Незважаючи на те, що ця парадигма з’явилась в 1960-тих роках, вона не мала широкого застосування до 1990-тих.

Java є об'єктно-орієнтованою мовою, тому такі поняття як "клас" і "об'єкт" відіграють у ній ключову роль. Будь-яку програму на Java можна уявити як набір об'єктів, що взаємодіють між собою.

Шаблоном або описом об'єкта є клас, а об'єкт являє собою екземпляр цього класу. Можна ще провести таку аналогію. У нас у всіх є деяке уявлення про людину - наявність двох рук, двох ніг, голови, тулуба тощо. Є деякий шаблон - цей шаблон можна назвати класом. Реально ж існуюча людина (фактично екземпляр даного класу) є об'єктом цього класу.

Опис класу починається з ключового слова class, після якого записується ім'я класу. Загальноприйняті угоди (Code Conventions) рекомендують починати ім'я класу з великої літери, на відміну від імен методів і змінних, які починаються з маленької літери. Приклад:

```
class Person{
    String name;         // ім’я
    int age;             // вік
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
public class Program{
    public static void main(String[] args) {
       Person tom;
    }
}
```

Як правило, класи визначаються в різних файлах. У цьому випадку для простоти ми визначаємо два класи в одному файлі. Варто зазначити, що в цьому випадку тільки один клас може мати модифікатор public (у цьому випадку це клас Program), а сам файл коду має називатися на ім'я цього класу, тобто в цьому випадку файл має називатися Program.java.

Клас представляє новий тип, тому ми можемо визначати змінні, які представляють даний тип. Так, тут у методі main визначено змінну tom, яка представляє клас Person. Але поки що ця змінна не вказує ні на який об'єкт і за замовчуванням вона має значення null. За великим рахунком ми її поки не можемо використовувати, тому спочатку необхідно створити об'єкт класу Person.

Конструктори

Крім звичайних методів класи можуть визначати спеціальні методи, які називаються конструкторами. Конструктори викликаються при створенні нового об'єкта даного класу. Конструктори виконують ініціалізацію об'єкта.

Якщо в класі не визначено жодного конструктора, то для цього класу автоматично створюється конструктор без параметрів.

Вище визначений клас Person не має жодних конструкторів. Тому для нього автоматично створюється конструктор за замовчуванням, який ми можемо використовувати для створення об'єкта Person. Зокрема, створимо один об'єкт:

```
public class Program{
    public static void main(String[] args) {

        Person tom = new Person(); // створення об’єкту
        tom.displayInfo();
        // змінюємо вік та ім’я
        tom.name = "Tom";
        tom.age = 34;
        tom.displayInfo();
    }
}
class Person{
    String name;    // ім’я
    int age;        // вік
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Оператор new виділяє пам'ять для об'єкта Person. І потім викликається конструктор за замовчуванням, який не приймає жодних параметрів. У підсумку після виконання цього виразу в пам'яті буде виділено ділянку, де зберігатимуться всі дані об'єкта Person. А змінна tom отримає посилання на створений об'єкт.

Якщо конструктор не ініціалізує значення змінних об'єкта, то вони отримують значення за замовчуванням. Для змінних числових типів це число 0, а для типу string і класів - це значення null (тобто фактично відсутність значення).

Ключове слово this

Ключове слово this представляє посилання на поточний екземпляр класу. Через це ключове слово ми можемо звертатися до змінних, методів об'єкта, а також викликати його конструктори. Наприклад:

```
    Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
```

Ініціалізатори

Крім конструктора початкову ініціалізацію об'єкта цілком можна було проводити за допомогою ініціалізатора об'єкта. Ініціалізатор виконується до будь-якого конструктора. Тобто в ініціалізатор ми можемо помістити код, спільний для всіх конструкторів:

```
class Person{
    String name;    // ім’я
    int age;        // вік

    /*початок блоку ініціалізатора*/
    {
        name = "Undefined";
        age = 18;
    }
 /*кінець блоку ініціалізатора*/
```

Модифікатори доступу та інкапсуляція

Усі члени класу в мові Java - поля і методи - мають модифікатори доступу. У минулих темах ми вже стикалися з модифікатором public.

Модифікатори доступу дають змогу задати допустиму сферу видимості для членів класу, тобто контекст, у якому можна вживати цю змінну або метод.

У Java використовуються такі модифікатори доступу:

- public: публічний, загальнодоступний клас або член класу. Поля і методи, оголошені з модифікатором public, видно іншим класам із поточного пакета і з зовнішніх пакетів;
- private: закритий клас або член класу, протилежність модифікатору public. Закритий клас або член класу доступний тільки з коду в тому ж класі;
- protected: такий клас або член класу доступний з будь-якого місця в поточному класі або пакеті або в похідних класах, навіть якщо вони знаходяться в інших пакетах;
- Модифікатор за замовчуванням. Відсутність модифікатора у поля або методу класу передбачає застосування до нього модифікатора за замовчуванням. Такі поля або методи видно всім класам у поточному пакеті.

Розглянемо модифікатори доступу на прикладі такої програми:

```
public class Program{
    public static void main(String[] args) {
        Person andrii = new Person("Andrii", 20, "Baker Street", "+39095687");
        andrii.displayName();     // все гаразд, метод public
        andrii.displayAge();      // все гаразд, метод має модифікатор за замовчуванням
        andrii.displayPhone();    // все гаразд, метод protected
        // andrii.displayAddress();  // ! Помилка, метод private

        System.out.println(andrii.name);      // все гаразд, модифікатор за замовчуванням
        System.out.println(andrii.address);   // все гаразд, модифікатор public
        System.out.println(andrii.age);       // все гаразд, модифікатор protected
        //System.out.println(andrii.phone);   // ! Помилка, модифікатор private
    }
}
class Person{
    String name;
    protected int age;
    public String address;
    private String phone;

    public Person(String name, int age, String address, String phone){
        this.name = name;
        this.age = age;
        this.address = address;
        this.phone = phone;
    }
    public void displayName(){
        System.out.printf("Name: %s \n", name);
    }
    void displayAge(){
        System.out.printf("Age: %d \n", age);
    }
    private void displayAddress(){
```

```
        System.out.printf("Address: %s \n", address);
    }
    protected void displayPhone(){
        System.out.printf("Phone: %s \n", phone);
    }}
```

У цьому випадку обидва класи розташовані в одному пакеті - пакеті за замовчуванням, тому в класі Program ми можемо використовувати всі методи і змінні класу Person, які мають модифікатор за замовчуванням, public і protected. А поля і методи з модифікатором private у класі Program не будуть доступні.

Якби клас Program розташовувався б в іншому пакеті, то йому були б доступні тільки поля і методи з модифікатором public.

Модифікатор доступу має передувати решті частини визначення змінної або методу.

Інкапсуляція

Використання різних модифікаторів гарантує, що дані не будуть спотворені або змінені неналежним чином. Подібне приховування даних усередині деякої області видимості називається інкапсуляцією.

Інкапсуляція – механізм мови, який дозволяє об’єднати дані та методи, які працюють з одними і тими самими даними в єдиний об’єкт та приховати деталі реалізації від користувача. Слід зауважити, що поняття інкапсуляції та приховування даних йдуть пліч-о-пліч в мові Java.

Об'єкти як параметри методів

Об'єкти класів, як і дані примітивних типів, можуть передаватися в методи. Однак у цьому випадку є одна особливість - під час передачі об'єктів як значення передається копія посилання на ділянку в пам'яті, де розташований цей об'єкт. Розглянемо невеликий приклад. Нехай у нас є такий клас Person:

```
    public class Program{
        public static void main(String[] args) {
            Person andrii = new Person("Andrii");
            System.out.println(andrii.getName()); // Andrii
            changeName(andrii);
            System.out.println(andrii.getName()); // Alice
        }
        static void changeName(Person p){
            p.setName("Alice");
        }
    }
    class Person{
        private String name;
        Person(String name){
```

```
            this.name = name;
        }
        public void setName(String name){
            this.name = name;
        }
        public String getName(){
            return this.name;
        }
    }
```

Тут у метод changeName передається об'єкт Person, у якого змінюється ім'я. Оскільки в метод буде передаватися копія посилання на область пам'яті, в якій знаходиться об'єкт Person, то змінна andrii і параметр p методу changeName вказуватимуть на один і той самий об'єкт у пам'яті. Тому після виконання методу в об'єкта andrii, який передається в метод, буде змінено ім'я з "Andrii" на "Alice".

Успадкування

Одним із ключових аспектів об'єктно-орієнтованого програмування є успадкування.

Успадкування – механізм мови, який дозволяє створити новий клас на основі класа, що вже існує. Успадкування дозволяє скоротити об’єм коду, але в той самий час сприяє зв’язуванню коду. Воно необхідне в першу чергу для вибудовування ієрархії між класами, тобто дати дорогу поліморфізму.

Наприклад, є такий клас Person, що описує окрему людину:

```
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){
        this.name=name;
    }

    public void display(){
        System.out.println("Name: " + name);
    }
}
```

І, можливо, згодом ми захочемо додати ще один клас, який описує співробітника підприємства - клас Employee. Оскільки цей клас реалізує той самий функціонал, що й клас Person, оскільки працівник - це також і людина, то було б раціонально зробити клас Employee похідним (спадкоємцем, підкласом) від класу Person, який, своєю чергою, називається базовим класом, батьком або суперкласом:

```
class Employee extends Person{
    public Employee(String name){
        super(name);    // якщо базовий клас визначає конструктор
                        //  то похідний клас повинен його викликати
    }
}
```

Щоб оголосити один клас спадкоємцем від іншого, треба використати після імені класу-спадкоємця ключове слово extends, після якого йде ім'я базового класу. Для класу Employee базовим є Person, і тому клас Employee успадковує всі ті самі поля та методи, які є в класі Person. Якщо в базовому класі визначено конструктори, то в конструкторі похідного класу необхідно викликати один із конструкторів базового класу за допомогою ключового слова super.

Похідний клас має доступ до всіх методів і полів базового класу (навіть якщо базовий клас перебуває в іншому пакеті), крім тих, що визначені з модифікатором private. При цьому похідний клас також може додавати свої поля та методи

Перевизначення методів

Похідний клас може визначати свої методи, а може перевизначати методи, які успадковані від базового класу. Перед перевизначуваним методом вказується анотація @Override. Ця анотація в принципі необов'язкова. При перевизначенні методу він повинен мати рівень доступу не менший, ніж рівень доступу в базовому класу. Наприклад, якщо в базовому класі метод має модифікатор public, то і в похідному класі метод повинен мати модифікатор public.

Поліморфізм

Поліморфізм - принцип мови, коли програма може використовувати об'єкти з однаковими інтерфейсами без інформації про внутрішній стан.

Без успадкування та інкапсуляції неможливо уявити необхідну реалізацію поліморфізму.

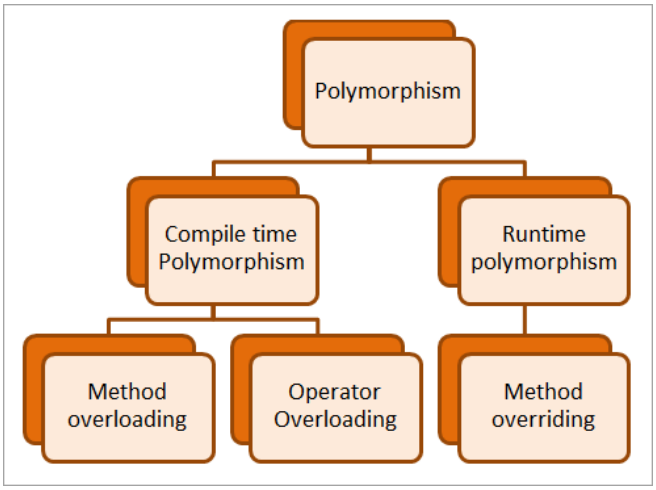


Рисунок 4.1 – Види поліморфізму

У Java є два типи поліморфізму:

- Перевантаження методу: це відбувається, коли в класі існує кілька методів з однаковим ім'ям, але різними параметрами. Правильний метод, що викликається, визначається під час компіляції на основі аргументів, переданих методу;
- Перевизначення методу: це відбувається, коли підклас надає нову реалізацію методу, який уже визначено в його суперкласі. Правильний метод, що викликається, визначається під час виконання залежно від типу об'єкта, на який робиться посилання.

У Java поліморфізм досягається за рахунок успадкування і використання типів інтерфейсу. Поліморфізм дає змогу розглядати об'єкти як об'єкти їхнього базового класу, що дає змогу писати загальний код, роблячи його гнучкішим і придатнішим для повторного використання. Це дає змогу створювати більш багаторазовий і зручний у супроводі код, а також підвищує читабельність коду.

Поліморфізм також забезпечує приховування даних або методів. Приведемо приклад:

```
class Parent {
	protected void test2{
		System.out.println(“parent test2 method”);
	}
}
class A extends Parent {
	public void test(){
		System.out.println(“test method”);
	}
	@Override
	public void test2(){
		this.Test(); //перевизначення
```

```
	}
}
class Program{
	public static void main(String [] args){
		Parent obj = new A();
		obj.test2() // апкаст
		obj.test() // помилка компіляції
	}
}
```

Завдання до виконання

Виконати завдання згідно свого варіанту (табл 4.1)**.** В цьому завданні дозволено використовувати звичайні класи. Кожен реалізований клас повинен містити методи: get, set, hashcode, equals, toString.

Таблиця 4.1 – Варіанти завдань

№	Завдання
1.	Напишіть програму для створення класу з назвою "TrafficLight" з атрибутами кольору та тривалості, а також методами для зміни кольору та перевірки на червоне чи зелене світло.
2.	Напишіть програму для створення класу з назвою "Restaurant" з атрибутами для пунктів меню, цін та оцінок, а також методами для додавання та видалення пунктів меню та обчислення середньої оцінки.
3.	Напишіть програму для створення класу Shape з абстрактними методами для обчислення площі та периметра, а також підкласів для прямокутника, кола та трикутника.
4.	Напишіть програму для створення класу під назвою "School" з атрибутами для учнів, вчителів та класів, методами для додавання та видалення учнів та вчителів, а також для створення класів.
5.	Напишіть програму для створення класу з назвою "Airplane" з атрибутами "номер рейсу", "пункт призначення" та "час відправлення", а також методами для перевірки статусу та затримки рейсу.
6.	Напишіть програму для створення класу з назвою "Employee" з атрибутами ім'я, зарплата та дата найму, а також метод для обчислення стажу роботи.
7.	Напишіть програму для створення класу з назвою "Movie" з атрибутами для назви, режисера, акторів та рецензій, а також методами для додавання та отримання рецензій.
8.	Напишіть програму для створення класу з назвою "Library" з колекцією книг та методами для додавання і видалення книг.
9.	Напишіть програму для створення класу з назвою "Bank" з колекцією рахунків і методами для додавання та видалення рахунків, а також для внесення та зняття грошей. Також визначте клас "Account", який буде зберігати дані про рахунки конкретного клієнта.
10.	Напишіть програму для створення класу з іменем Person з атрибутами ім'я та вік. Створіть два екземпляри класу Person, задайте їх атрибути з допомогою конструктора та виведіть їх ім'я та вік.

У таблиці 4.2 задано сімейство об'єктів, що мають деяку схожість (загальні ознаки). Необхідно виділити найбільш загальні риси об'єктів, на основі яких скласти базовий клас. На основі базового класу розробити ієрархію класів-нащадків, кінцевими гілками в якій будуть задані об'єкти. Ієрархія класів

повинна бути мінімум трирівнева, тобто між базовим класом і класами, що описують задані об'єкти, повинен бути хоча б один проміжний рівень ієрархії. Продемонструвати роботу з об'єктами, заданими за таблицею 4.2. У звіті привести діаграму класів. Також дозволено використання абстрактних класів та інтерфейсів.

Таблиця 4.2 – Варіанти завдань

№	*Завдання*
1.	собака, ведмідь, корова, акула, орел, кит, шпак, людина, тигр
2.	помідор, малина, полуниця, банан, картопля, виноград, яблуко, ківі
3.	солдат, майор, лейтенант, капітан, сержант, старший солдат, полковник, молодший лейтенант, старший лейтенант
4.	кухня, спальня, аудиторія, кабінет, ванна кімната, спортивний зал, гараж
5.	море, річка, океан, озеро, водосховище, ставок, затока, канал, калюжа
6.	пістолет, автомат, танк, граната, динаміт, ніж, револьвер
7.	меч, спис, шпага, рапіра, ніж, кинджал, сюрікени
8.	чоботи, кросівки, туфлі, босоніжки, валянки, черевики, сандалі, тапки
9.	пиріг, шашлик, розсольник, млинці, гуляш, рагу, плов, борщ, солянка
10.	флейта, гітара, саксофон, арфа, віолончель, скрипка

Приклад виконання

Напишіть програму на Java для створення класу з назвою "Inventory" з колекцією товарів і методами для додавання та видалення товарів, а також для перевірки наявності товарів на складі.

```
class Product{
    6 usages
    private String name;
    7 usages
    private int quantity;

    no usages
    public Product(String name, int quantity){
        this.name = name;
        this.quantity = quantity;
    }
    no usages
    public void setName(String name) { this.name = name; }
    2 usages
    public String getName() { return name; }
    no usages
    public void setQuantity(int quantity) { this.quantity = quantity; }
    2 usages
    public int getQuantity() { return quantity; }
    public String toString() { return "Name: " + getName() + ", quantity: " + quantity; }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Product product = (Product) o;
        return quantity == product.quantity && Objects.equals(name, product.name);
    }
    @Override
    public int hashCode() {
        return Objects.hash(name, quantity);
    }
}
```

Рисунок 4.1 – Створення класу Product

```java
import java.util.ArrayList;

class Inventory{
    4 usages
    private ArrayList<Product> products = new ArrayList<>();
    1 usage
    public Inventory() {}
    3 usages
    public void addProduct(Product product) {
        products.add(product);
    }
    1 usage
    public void removeProduct(Product product) {
        products.remove(product);
    }
    1 usage
    public void checkLowInventory() {
        for (Product product: products) {
            if (product.getQuantity() <= 100) {
                System.out.println(product.getName() + " закінчується. Поточна кількість: " + product.getQuantity());
            }
        }
    }
    @Override
    public String toString(){
        return products.toString();
    }
    public static void main(String[] args){
        Product cloth = new Product( name: "Cloth", quantity: 150);
        Product leather = new Product( name: "Leather", quantity: 59);
        Product bottleOfWater = new Product( name: "bottleOfWater", quantity: 101);
        Inventory products = new Inventory();
        products.addProduct(leather);
        products.addProduct(cloth);
        System.out.println("Products: " + products);
        products.addProduct(bottleOfWater);
        products.removeProduct(cloth);
        products.checkLowInventory();
    }
```

Рисунок 4.2 – Створення класу Inventory

Варіант другого завдання: абітурієнт, студент, першокурсник, бакалавр, спеціаліст, магістр.

```java
//абітурієнт, студент, першокурсник, бакалавр, спеціаліст, магістр
5 usages  1 inheritor
class Applicant{
    9 usages
    private String name;
    9 usages
    private String surname;
    9 usages
    private int age;
    2 usages
    public Applicant(){this.age = 0; this.name = ""; this.surname = "";}
    3 usages  1 override
    public void add(){
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter your name:");
        this.name = scanner.nextLine();
        System.out.println("Enter your surname:");
        this.surname = scanner.nextLine();
        System.out.println("Enter your age:");
        this.age = scanner.nextInt();}
    1 override
    public String toString() {return "Applicant name: " + this.name + ", surname: " + this.surname + ", age: " + this.age;}
    no usages
    public String getName() {return name;}
    no usages
    public void setName(String name) { this.name = name;}
    no usages
    public String getSurname() {return surname;}
    no usages
    public void setSurname(String surname) {this.surname = surname;}

    no usages
    public int getAge() {return age;}
    no usages
    public void setAge(int age) {this.age = age;}
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Applicant applicant = (Applicant) o;
        return age == applicant.age && Objects.equals(name, applicant.name) && Objects.equals(surname, applicant.surname);
    }
    @Override
    public int hashCode() {return Objects.hash(name, surname, age);}
}
```

Рисунок 4.3 – Фрагмент виконання другого завдання

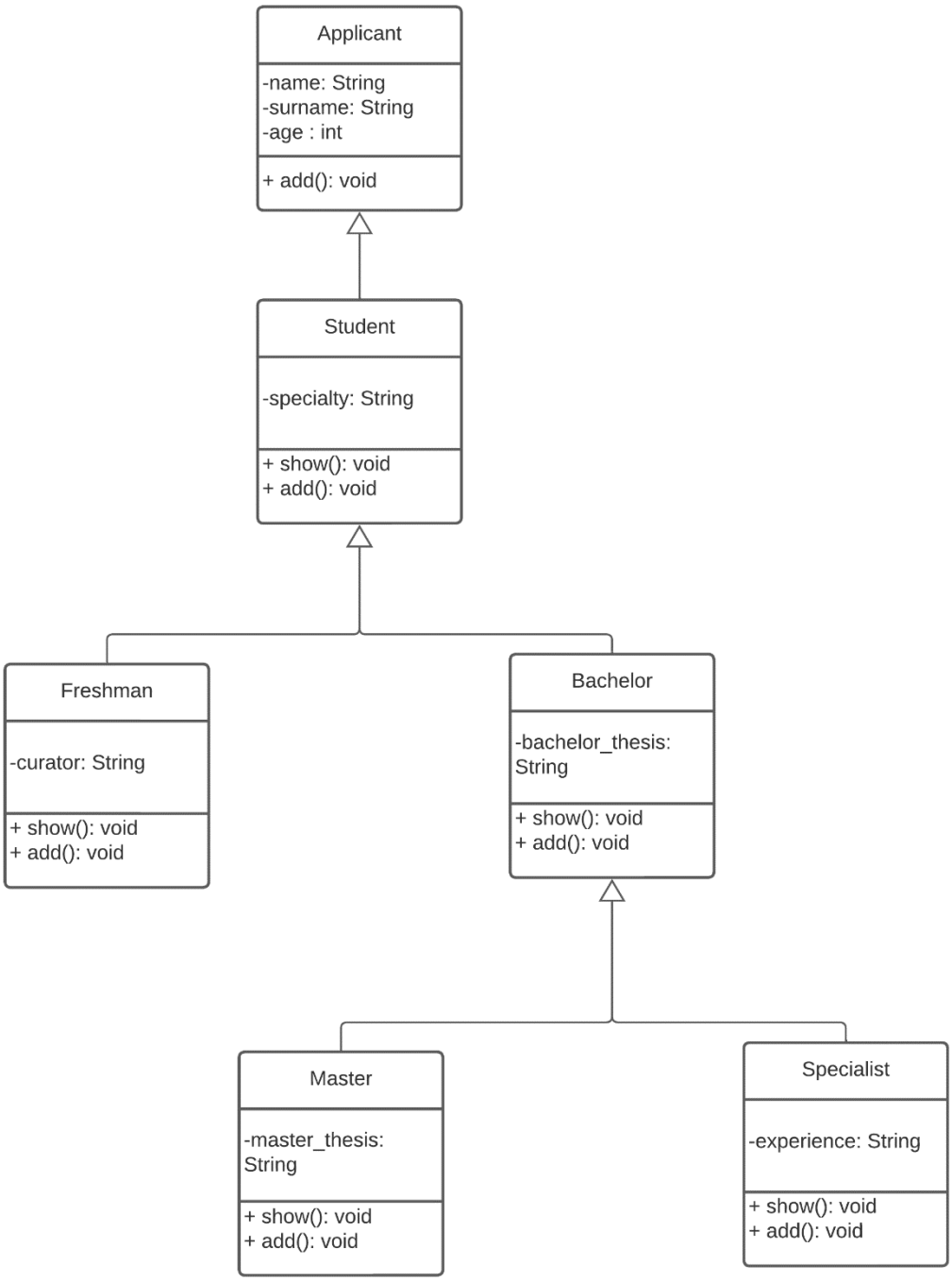


Рисунок 4.4 – Ієрархія класів

Контрольні питання

1. Надайте визначення поняттю «ООП» (об'єктно-орієнтоване програмування);
2. Надайте визначення поняттю «інкапсуляція» у контексті об'єктно-орієнтованого програмування;
3. Надайте визначення поняттю «поліморфізм» в рамках об'єктно-орієнтованого програмування;
4. Надайте визначення поняттю «успадкування» в об'єктно-орієнтованому програмуванні;
5. Для чого призначені конструктори в класі у мові програмування Java?
6. Які модифікатори доступу існують у мові програмування Java?
7. Які є недоліки у концепції успадкування в об'єктно-орієнтованому програмуванні?
8. Які два типи поліморфізму існують в мові програмування Java, і як їх можна охарактеризувати?

ЛАБОРАТОРНА РОБОТА. РОБОТА З РЯДКАМИ В JAVA

Мета роботи: ознайомитись з поняттям «рядок» та освоїти роботу з регулярними виразами.

Основні теоретичні відомості

Введення в рядки. Клас String

Рядок являє собою послідовність символів. Для роботи з рядками в Java визначено клас String, який надає низку методів для маніпуляції рядками. Фізично об'єкт String являє собою посилання на ділянку в пам'яті, в якій розміщені символи.

Для створення нового рядка ми можемо використати один із конструкторів класу String, або безпосередньо присвоїти рядок у подвійних лапках:

```
public static void main(String[] args) {

    String str1 = "Java";
    String str2 = new String(); // пустий рядок
    String str3 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});
    String str4 = new String(new char[]{'w', 'e', 'l', 'c', 'o', 'm', 'e'}, 3,
4);//3 -початковий індекс, 4 – кількість символів
    System.out.println(str1); // Java
    System.out.println(str2); //
    System.out.println(str3); // hello
    System.out.println(str4); // come
}
```

Під час роботи з рядками важливо розуміти, що об'єкт String є незмінним (immutable). Тобто при будь-яких операціях над рядком, які змінюють цей рядок, фактично буде створюватися новий рядок.

Основні методи класу String

Основні операції з рядками розкриваються через методи класу String, серед яких можна виділити такі:

- concat(): об'єднує рядки;
- valueOf(): перетворює об'єкт у рядковий вигляд;
- join(): з'єднує рядки з урахуванням роздільника;
- compareTo(): порівнює два рядки;

- charAt(): повертає символ рядка за індексом;
- getChars(): повертає групу символів;
- equals(): порівнює рядки з урахуванням регістру;
- equalsIgnoreCase(): порівнює рядки без урахування регістру;
- regionMatches(): порівнює підрядки в рядках;
- indexOf(): знаходить індекс першого входження підрядка в рядок;
- lastIndexOf(): знаходить індекс останнього входження підрядка в рядок;
- startsWith(): визначає, чи починається рядок із підрядка;
- endsWith(): визначає, чи закінчується рядок на певний підрядок;
- replace(): замінює в рядку один підрядок на інший;
- trim(): видаляє початкові та кінцеві пробіли;
- substring(): повертає підрядок, починаючи з певного індексу до кінця або до певного індексу;
- toLowerCase(): переводить усі символи рядка в нижній регістр;
- toUpperCase(): переводить усі символи рядка у верхній регістр.

Основні операції зі строками

З'єднання рядків

Для з'єднання рядків можна використовувати операцію додавання ("+"):

```
String str1 = "Java";
String str2 = "Hello";
String str3 = str1 + " " + str2;
System.out.println(str3); // Hello Java
```

При цьому якщо в операції додавання рядків використовується нерядковий об'єкт, наприклад, число, то цей об'єкт перетворюється на рядок:

```
String str3 = "Рік " + 2023;
```

Фактично ж під час додавання рядків з нерядковими об'єктами буде викликатися метод valueOf() класу String. Цей метод має безліч перевантажень і перетворює практично всі типи даних до рядка. Для перетворення об'єктів різних класів метод valueOf викликає метод toString() цих класів.

Інший спосіб об'єднання рядків представляє метод concat():

```
String str1 = "Java";
String str2 = "Hello";
str2 = str2.concat(str1); // HelloJava
```

Метод concat() приймає рядок, з яким треба об'єднати викликаючий рядок, і повертає з'єднаний рядок.

Ще один метод об'єднання - метод join() дозволяє об'єднати рядки з урахуванням роздільника. Наприклад, вище два рядки зливалися в одне слово "HelloJava", але в ідеалі ми б хотіли, щоб два підрядки були розділені пропуском. І для цього використовуємо метод join():

```
String str1 = "Java";
String str2 = "Hello";
String str3 = String.join(" ", str2, str1); // Hello Java
```

Метод join є статичним. Першим параметром йде роздільник, яким будуть розділятися підрядки в загальному рядку, а всі наступні параметри передають через кому довільний набір об'єднуваних підрядків - у цьому випадку два рядки, хоча їх може бути і більше.

Вилучення символів і підрядків

Для вилучення символів за індексом у класі String визначено метод char charAt(int index). Він приймає індекс, за яким треба отримати символів, і повертає витягнутий символ:

```
String str = "Java";
char c = str.charAt(2);
System.out.println(c); // v
```

Як і в масивах, індексація починається з нуля.

Якщо треба витягти відразу групу символів або підрядок, то можна використовувати метод getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin). Він приймає такі параметри:

- srcBegin: індекс у рядку, з якого починається вилучення символів
- srcEnd: індекс у рядку, до якого йде вилучення символів
- dst: масив символів, у який будуть витягуватися символи
- dstBegin: індекс у масиві dst, з якого треба додавати витягнуті з рядка символи

```
String str = "Hello world!";
int start = 6;
int end = 11;
char[] dst=new char[end - start];
str.getChars(start, end, dst, 0);
System.out.println(dst); // world
```

Порівняння рядків

Для порівняння рядків використовуються методи equals() (з урахуванням регістру) і equalsIgnoreCase() (без урахування регістру). Обидва методи як параметр приймають рядок, з яким треба порівняти:

```
String str1 = "Hello";
String str2 = "hello";
System.out.println(str1.equals(str2)); // false
System.out.println(str1.equalsIgnoreCase(str2)); // true
```

На відміну від порівняння числових та інших даних примітивних типів для рядків не застосовується знак рівності ==. Замість нього треба використовувати метод equals().

Ще один спеціальний метод regionMatches() порівнює окремі підрядки в межах двох рядків. Він має такі форми:

```
boolean regionMatches(int toffset, String other, int oofset, int len)
boolean regionMatches(boolean ignoreCase, int toffset, String other, int oofset,
int len)
```

Метод приймає такі параметри:

- ignoreCase: чи треба ігнорувати регістр символів під час порівняння. Якщо значення true, регістр ігнорується
- toffset: початковий індекс у викликаючому рядку, з якого почнеться порівняння
- other: рядок, з яким порівнюється той, що викликає
- oofset: початковий індекс у порівнюваному рядку, з якого почнеться порівняння
- len: кількість порівнюваних символів в обох рядках

Використовуємо метод:

```
String str1 = "Hello world";
String str2 = "I work";
boolean result = str1.regionMatches(6, str2, 2, 3);
System.out.println(result); // true
```

У цьому випадку метод порівнює 3 символи з 6-го індексу першого рядка ("wor") і 3 символи з 2-го індексу другого рядка ("wor"). Оскільки ці підрядки однакові, то повертається true.

І ще одна пара методів int compareTo(String str) і int compareToIgnoreCase(String str) також дають змогу порівняти два рядки, але водночас вони також дають змогу дізнатися, чи більший один рядок, ніж інший, чи ні. Якщо значення, що повертається, більше за 0, то перший рядок більший

за другий, якщо менше нуля, то, навпаки, другий більший за перший. Якщо рядки рівні, то повертається 0.

Для визначення більший чи менший один рядок, ніж інший, використовується лексикографічний порядок. Тобто, наприклад, рядок "A" менший, ніж рядок "B", оскільки символ 'A' в алфавіті стоїть перед символом 'B'. Якщо перші символи рядків рівні, то в розрахунок беруться наступні символи. Наприклад:

```
String str1 = "hello";
String str2 = "world";
String str3 = "hell";
System.out.println(str1.compareTo(str2)); // -15 - str1 меньше чем strt2
System.out.println(str1.compareTo(str3)); // 1 - str1 больше чем str3
```

Пошук у рядку

Метод indexOf() знаходить індекс першого входження підрядка в рядок, а метод lastIndexOf() - індекс останнього входження. Якщо підрядок не буде знайдено, то обидва методи повертають -1:

```
String str = "Hello world";
int index1 = str.indexOf('l'); // 2
int index2 = str.indexOf("wo"); //6
int index3 = str.lastIndexOf('l'); //9
```

Метод startsWith() дає змогу визначити, чи починається рядок з певного підрядка, а метод endsWith() дає змогу визначити, чи закінчується рядок на певний підрядок:

```
String str = "myfile.exe";
boolean start = str.startsWith("my"); //true
boolean end = str.endsWith("exe"); //true
```

Заміна в рядку

Метод replace() дозволяє замінити в рядку одну послідовність символів на іншу:

```
String str = "Hello world";
String replStr1 = str.replace('l', 'd'); // Heddo wordd
String replStr2 = str.replace("Hello", "Bye"); // Bye world
```

Обрізання рядка

Метод trim() дає змогу видалити початкові та кінцеві пробіли:

```
String str = "  hello world  ";
str = str.trim(); // hello world
```

Метод substring() повертає підрядок, починаючи з певного індексу до кінця або до певного індексу:

```
String str = "Hello world";
String substr1 = str.substring(6); // world
String substr2 = str.substring(3,5); //lo
```

Зміна регістру

Метод toLowerCase() переводить усі символи рядка в нижній регістр, а метод toUpperCase() - у верхній:

```
String str = "Hello World";
System.out.println(str.toLowerCase()); // hello world
System.out.println(str.toUpperCase()); // HELLO WORLD
```

Split

Метод split() дозволяє розбити рядок на підрядки за певним роздільником. Роздільник - який-небудь символ або набір символів передається як параметр у метод. Наприклад, розіб'ємо текст на окремі слова:

```
String text = "FIFA will never regret it";
String[] words = text.split(" ");
for(String word : words){
    System.out.println(word);
}
```

StringBuffer та StringBuilder

Об'єкти String є незмінними, тому всі операції, які змінюють рядки, фактично призводять до створення нового рядка, що позначається на продуктивності програми. Для розв'язання цієї проблеми, щоб робота з рядками відбувалася з меншими витратами, в Java було додано класи StringBuffer і StringBuilder. По суті вони нагадують розширюваний рядок, який можна змінювати без шкоди для продуктивності.

Ці класи схожі, практично двійники, вони мають однакові конструктори, одні й ті самі методи, які однаково використовуються. Єдина їхня відмінність полягає в тому, що клас StringBuffer синхронізований і потокобезпечний. Тобто клас StringBuffer зручніше використовувати в багатопотокових додатках, де об'єкт цього класу може змінюватися в різних потоках. Якщо ж мова про багатопотокові додатки не йде, то краще використовувати клас StringBuilder, який не є потокобезпечним, але при цьому працює швидше, ніж StringBuffer в однопотокових додатках.

StringBuffer та StringBuilder визначає чотири конструктори:

```
StringBuilder()
StringBuilder(int capacity)
StringBuilder(String str)
```

```
StringBuilder(CharSequence chars)
```

Третій і четвертий конструктори обох класів приймають рядок і набір символів, при цьому резервуючи пам'ять для додаткових 16 символів.

Класи StringBuffer та StringBuilder мають такі методи:

- charAt() – отримання символу по заданому індексу;
- setCharAt() – встановлення символу по заданому індексу;
- getChars()- отримання набору символів між заданими індексами;
- append()- додавання підрядка в кінець;
- insert() – додавання рядка або символу по визначеному індексу в StringBuffer;
- delete() – видаляє усі символи з визначеного індексу з визначеної позиції;
- substring() - обрізання рядка з певного індексу до кінця, або до певного індексу;
- setLength() - зміни довжини;
- replace() - заміна підрядка між певними позиціями в StringBuffer на інший підрядок;
- reverse() - зміна порядку у StringBuffer на зворотній змінює порядок у StringBuffer на зворотній.

Регулярні вирази

Регулярні вирази являють собою потужний інструмент для обробки рядків. Регулярні вирази дають змогу задати шаблон, якому має відповідати рядок або підрядок.

Деякі методи класу String приймають регулярні вирази і використовують їх для виконання операцій над рядками.

split

Для поділу рядка на підрядки застосовується метод split(). Як параметр він може приймати регулярний вираз, який представляє критерій поділу рядка.

Наприклад, розділимо речення на слова:

```
String text = "FIFA will never regret it";
String[] words = text.split("\\s*(\\s|,|!|\\.)\\s*");
```

```
for(String word : words){
    System.out.println(word);
}
```

Для розділення застосовується регулярний вираз "\\s*(\\s|,|!|\\.)\\s*". Підвираз "\\s" по суті представляє пробіл. Зірочка вказує, що символ може бути присутнім від 0 до нескінченної кількості разів. Тобто додаємо зірочку і ми отримуємо невизначену кількість пробілів, що йдуть підряд, - "\\s*" (тобто неважливо, скільки пробілів між словами). Причому пробілів може взагалі не бути. У дужках вказується група виразів, яка може йти після невизначеної кількості пробілів. Група дозволяє нам визначити набір значень через вертикальну риску, і підрядок має відповідати одному з цих значень. Тобто в групі "\\s|,|!|\\\." підрядок може відповідати пробілу, комою, знаком оклику або крапкою. Причому оскільки крапка являє собою спеціальну послідовність, то, щоб вказати, що ми маємо на увазі саме знак крапки, а не спеціальну послідовність, перед крапкою ставимо слеші.

Відповідність рядка. matches

Ще один метод класу String - matches() приймає регулярний вираз і повертає true, якщо рядок відповідає

цьому виразу. Інакше повертає false.

Наприклад, перевіримо, чи відповідає рядок номеру телефону:

```
String input = "+12343454556";
boolean result = input.matches("(\\+*)\\d{11}");
if(result){
    System.out.println("It is a phone number");
}
else{
    System.out.println("It is not a phone number!");
}
```

У цьому випадку в регулярному виразі спочатку визначається група "(\\+*)". Тобто спочатку може йти знак плюса, але також він може бути відсутнім. Далі дивимося, чи відповідають наступні 11 символів цифрам. Вираз "\\\d" представляє цифровий символ, а число у фігурних дужках - {11} - скільки разів цей тип символів має повторюватися. Тобто ми шукаємо рядок, де спочатку може йти знак плюс (або він може бути відсутнім), а потім йде 11 цифрових символів.

Клас Pattern

Більша частина функціональності по роботі з регулярними виразами в Java зосереджена в пакеті java.util.regex.

Сам регулярний вираз являє собою шаблон для пошуку збігів у рядку. Для задавання подібного шаблону і пошуку підрядків у рядку, які задовольняють даному шаблону, в Java визначено класи Pattern і Matcher.

Для простого пошуку відповідностей у класі Pattern визначено статичний метод boolean matches(String pattern, CharSequence input). Цей метод повертає true, якщо послідовність символів input повністю відповідає шаблону рядка pattern:

```
import java.util.regex.Pattern;

public class StringsApp {

    public static void main(String[] args) {

        String input = "Hello";
        boolean found = Pattern.matches("Hello", input);
        if(found)
            System.out.println("Найдено");
        else
            System.out.println("Не найдено");
    }
}
```

Але, як правило, для пошуку відповідностей застосовується інший спосіб - використання класу Matcher.

Клас Matcher

Розглянемо основні методи класу Matcher:

- boolean matches(): повертає true, якщо весь рядок збігається з шаблоном;
- boolean find(): повертає true, якщо в рядку є підрядок, який збігається з шаблоном, і переходить до цього підрядка;
- String group(): повертає підрядок, який збігся з шаблоном у результаті виклику методу find. Якщо збіг відсутній, то метод генерує виняток IllegalStateException;
- int start(): повертає індекс поточного збігу;
- int end(): повертає індекс наступного збігу після поточного;

- String replaceAll(String str): замінює всі знайдені збіги підрядком str і повертає змінений рядок з урахуванням замін.

Використовуємо клас Matcher. Для цього спочатку треба створити об'єкт Pattern за допомогою статичного методу compile(), який дозволяє встановити шаблон:

```
Pattern pattern = Pattern.compile("Hello");
```

Як шаблон виступає рядок "Hello". Метод compile() повертає об'єкт Pattern, який ми потім можемо використовувати в програмі.

У класі Pattern також визначено метод matcher(String input), який як параметр приймає рядок, де треба проводити пошук, і повертає об'єкт Matcher:

```
String input = "Hello world! Hello Java!";
Pattern pattern = Pattern.compile("hello");
Matcher matcher = pattern.matcher(input);
```

Потім у об'єкта Matcher викликається метод matches() для пошуку відповідностей шаблону в тексті:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class StringsApp {

    public static void main(String[] args) {

        String input = "Hello";
        Pattern pattern = Pattern.compile("Hello");
        Matcher matcher = pattern.matcher(input);
        boolean found = matcher.matches();
        if(found)
            System.out.println("Знайдено");
        else
            System.out.println("Не знайдено");
    }
}
```

Розглянемо більш функціональний приклад зі знаходженням не повної відповідності, а окремих збігів у рядку:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class StringsApp {

    public static void main(String[] args) {

        String input = "Hello Java! Hello JavaScript! JavaSE 8.";
        Pattern pattern = Pattern.compile("Java(\\w*)");
        Matcher matcher = pattern.matcher(input);
        while(matcher.find())
            System.out.println(matcher.group());
    }
}
```

Припустимо, ми хочемо знайти в рядку всі входження слова Java. У вихідному рядку це три слова: "Java", "JavaScript" і "JavaSE". Для цього застосуємо шаблон "Java(\\w*)". Цей шаблон використовує синтаксис регулярних виразів. Слово "Java" на початку говорить про те, що всі збіги в рядку повинні починатися на Java. Вираз (\\w*) означає, що після "Java" у збігу може бути будь-яка кількість алфавітно-цифрових символів. Вираз \w означає алфавітно-цифровий символ, а зірочка після виразу вказує на невизначену їхню кількість - їх може бути один, два, три або взагалі не бути. І щоб java не розглядала \w як ескейп-послідовність, як \n, то вираз екранується ще одним слешем.

Далі застосовується метод find() класу Matcher, який дозволяє переходити до наступного збігу в рядку. Тобто перший виклик цього методу знайде перший збіг у рядку, другий виклик знайде другий збіг тощо. Тобто за допомогою циклу while(matcher.find()) ми можемо пройтися по всіх збігах. Кожен збіг ми можемо отримати за допомогою методу matcher.group(). У підсумку програма видасть такий результат:

Завдання до виконання

Виконати завдання згідно свого варіанту (табл 5.1)**.**

Таблиця 5.1 – Варіанти завдань

№	Завдання
1	Написати регулярний вираз, що визначає, чи є цей рядок рядком "abcdefghijklmnopqrstuv18340" чи ні. - приклад правильних виразів: abcdefghijklmnopqrstuv18340. - приклад неправильних виразів: abcdefghijklmnoasdfasdpqrstuv18340.
2	Напишіть програму на Java для лексикографічного порівняння двох рядків. Два рядки вважаються лексикографічно рівними, якщо вони мають однакову довжину і містять однакові символи в однакових позиціях.
3	Написати регулярний вираз, що визначає, чи є цей рядок шістнадцятиричним ідентифікатором кольору в HTML. Де #FFFFFF для білого, #000000 для чорного, #FF0000 для червоного тощо. - приклад правильних виразів: #FFFFFFFF, #FFFF3421, #00ff00. - приклад неправильних виразів: 232323, f#fddee, #fd2.
4	Напишіть програму на Java, яка замінює заданий символ на інший символ
5	Написати регулярний вираз, що визначає, чи є цей рядок валідною E-mail адресою згідно з RFC під номером 2822. - приклад правильних виразів: user@example.com andrii@gmail.com - приклад неправильних виразів: bug@@@com.ru, @val.ru, Just Text2.
6	Напишіть програму на Java, яка повертає суму цифр, що містяться у заданому рядку. Якщо цифр немає, то сума дорівнює 0
7	Написати регулярний вираз, що визначає, чи є цей рядок валідною E-mail адресою згідно з RFC під номером 2822. - приклад правильних виразів: user@example.com, root@localhost - приклад неправильних виразів: bug@@@com.ru, @val.ru, Just Text2.
8	Напишіть програму на Java, яка реверсує кожне слово рядка.
9	Напишіть програму на Java, яка знаходить найдовший паліндромний підрядок у рядку.
10	Напишіть програму на Java, яка знаходить перший символ у рядку, що не повторюється.

Приклад виконання

Дано рядок A=”**imagine an imaginary menagerie manager managing an imaginary menagerie. “ – в кожному слові рядка розташувати слова в зворотному порядку, пробіли між словами замінити на порядковий номер пробілу в реченні.**

```
public static void main(String[] args){
    String input = "imagine an imaginary menagerie manager managing an imaginary menagerie";
    String[] words = input.trim().split( regex: "\\s+");
    System.out.println(Arrays.toString(words));
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < words.length - 1; i++) {
        StringBuilder buffer = new StringBuilder();
        buffer.append(words[i]);
        result.append(buffer.reverse());
        if (i > 0) {
            result.append(i);
        }
    }
    System.out.println(result);
```

Рисунок 5.1 – Приклад виконання

Контрольні питання

1. Зазначте визначення терміну "рядок" в мові програмування Java;
2. Виокреміть різницю між класами String та StringBuffer у мові програмування Java;
3. Визначте відмінності між класами StringBuffer та StringBuilder в контексті мови програмування Java;
4. Як у мові програмування Java можна проводити порівняння двох рядків? Розкрийте різницю між виразами str1 == str2 та str1.equals(str2) в Java;
5. Як здійснити реверс рядка в мові програмування Java?
6. Чи можна вважати рядки незмінними або кінцевими в Java? Якщо так, то які переваги надає незмінність рядків?
7. Визначте поняття "регулярні вирази" в контексті мови програмування Java.

ЛАБОРАТОРНА РОБОТА. РОБОТА З ФАЙЛОВОЮ СИСТЕМОЮ

Мета роботи: ознайомитись з роботою з файловою системою в java та з основним функціоналом роботи з потоками, який зосереджений у класах пакета java.io.

Основні теоретичні відомості

Поняття «потоку» при роботі з файловою системою

Відмінною рисою багатьох мов програмування є робота з файлами і потоками. У Java основний функціонал роботи з потоками зосереджений у класах із пакета java.io.

Ключовим поняттям тут є поняття **потоку**. Хоча поняття "**потік**" у програмуванні доволі перевантажене і може позначати безліч різних концепцій. У даному випадку стосовно роботи з файлами і введенням-виведенням ми будемо говорити про потік (stream), як про абстракцію, яка використовується для читання або запису інформації (файлів, сокетів, тексту консолі тощо).

Потік пов'язаний з реальним фізичним пристроєм за допомогою системи введення-виведення Java. У нас може бути визначений потік, який пов'язаний з файлом і через який ми можемо вести читання або запис файлу. Це також може бути потік, пов'язаний із мережевим сокетом, за допомогою якого можна отримати або надіслати дані в мережі. Усі ці завдання: читання і запис різних файлів, обмін інформацією мережею, введення-виведення в консолі ми будемо вирішувати в Java за допомогою потоків.

Об'єкт, з якого можна зчитати дані, називається потоком введення, а об'єкт, у який можна записувати дані, - потоком виведення. Наприклад, якщо треба зчитати вміст файлу, то застосовується потік введення, а якщо треба записати у файл - то потік виведення.

В основі всіх класів, які керують потоками байтів, лежать два абстрактні класи: **InputStream** (представляє потоки введення) і **OutputStream** (представляє потоки виведення).

Але оскільки працювати з байтами не дуже зручно, то для роботи з потоками символів були додані абстрактні класи **Reader** (для читання потоків символів) і **Writer** (для запису потоків символів).

Усі інші класи, що працюють із потоками, є спадкоємцями цих абстрактних класів.

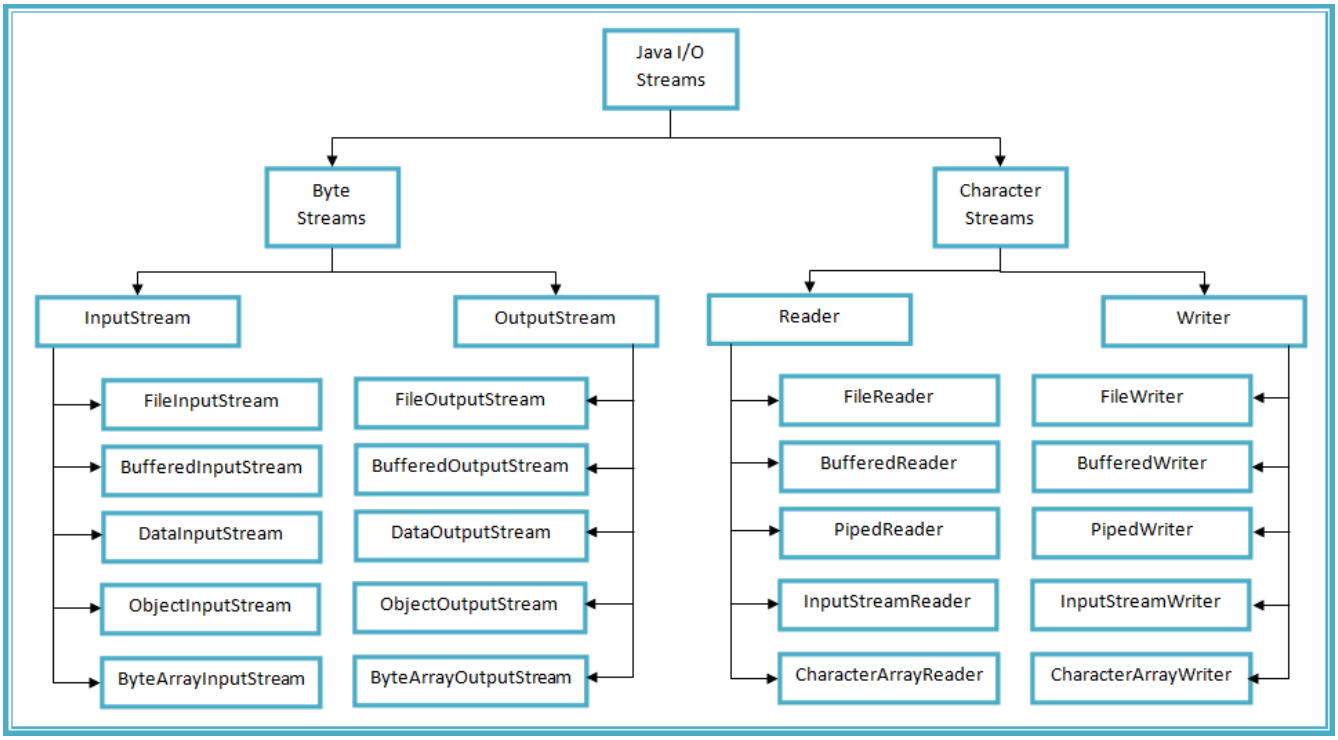


Рисунок 6.1 – Основні класи потоків

Клас OutputStream та InputStream

Клас OutputStream - це абстрактний клас, що визначає потоковий байтовий вивід.

Клас InputStream – це абстрактний клас, що визначаэ потокове байтовий введення.

У цій категорії містяться класи, що визначають, куди спрямовуються ваші дані: у масив байтів (але не безпосередньо в String; передбачається, що ви зможете створити їх із масиву байтів), у файл або канал.

- BufferedOutputStream (BufferedInputStream) – буферизований вихідний (вхідний) потік;

- ByteArrayOutputStream (ByteArrayInputStream) – створює буфер у пам'яті. Усі дані, що надсилаються в цей потік, розміщуються у створеному буфері;
- DataOutputStream (DataInputStream) – вихідний (вхідний) потік, що включає методи для запису стандартних типів даних Java;
- FileOutputStream (FileInputStream) - надсилання даних у файл на диску;
- ObjectOutputStream (ObjectInputStream) - вихідний потік для об'єктів;
- PipedOutputStream (PipedInputStream) - реалізує поняття вихідного (вхідного) каналу;
- FilterOutputStream (FilterInputStream) - абстрактний клас, що надає інтерфейс для класів-надбудов, які додають до наявних потоків корисні властивості.

Методи класу OutputStream:

- int close() - закриває вихідний потік. Наступні спроби запису передадуть виняток IOException;
- void flush() - фіналізує вихідний стан, очищаючи всі буфери виведення;
- abstract void write (int oneByte) - записує єдиний байт у вихідний потік;
- void write (byte[] buffer) - записує повний масив байтів у вихідний потік;
- void write (byte[] buffer, int offset, int count) - записує діапазон із count байт із масиву, починаючи зі зміщення offset.

Методи класу InputStream:

- int available(): повертає кількість байтів, доступних для читання в потоці;
- void close(): закриває потік;
- int read(): повертає цілочисельне представлення наступного байта в потоці. Коли в потоці не залишиться доступних для читання байтів, цей метод поверне число -1;
- int read(byte[] buffer): зчитує байти з потоку в масив buffer. Після читання повертає число зчитаних байтів. Якщо жодного байта не було зчитано, то повертається число -1;
- int read(byte[] buffer, int offset, int length): зчитує деяку кількість байтів, що дорівнює length, з потоку в масив buffer. При цьому зчитані байти

поміщаються в масиві, починаючи зі зміщення offset, тобто з елемента buffer[offset]. Метод повертає число успішно прочитаних байтів;

- long skip(long number): пропускає в потоці під час читання деяку кількість байт, яка дорівнює number.

Тепер розглянемо конкретні класи потоків.

BufferedOutputStream (BufferedInputStream)

Не надто відрізняється від класу OutputStream, за винятком додаткового методу flush(), що використовується для забезпечення запису даних у буферизований потік. Буфери виведення потрібні для підвищення продуктивності.

ByteArrayOutputStream (ByteArrayInputStream)

Використовує байтовий масив у вихідному потоці. Метод close() можна не викликати.

DataOutputStream (DataInputStream)

Дає змогу писати елементарні дані в потік через інтерфейс DataOutput, який визначає методи, що перетворюють елементарні значення на форму послідовності байтів. Такі потоки полегшують збереження у файлі двійкових даних.

Методи інтерфейсу DataOutput (DataInput):

- writeDouble(double value), readDouble();
- writeBoolean(boolean value), readBoolean();
- writeInt(int value), readInt();

FileOutputStream (FileInputStream)

Клас створюють об'єкти класу OutputStream (InputStream), який можна використовувати для запису байтів у файл. Створення нового об'єкта не залежить від того, чи існує заданий файл, оскільки він створює його перед

відкриттям. У разі спроби відкриття файлу, доступного тільки для читання, буде передано виняток.

Класи символьних потоків

Символьні потоки мають два основних абстрактних класи Reader і Writer, що керують потоками символів Unicode.

Reader

Методи класу Reader:

- abstract void close() - закриває вхідний потік. Наступні спроби читання передадуть виняток IOException;
- void mark(int readLimit) - поміщає мітку в поточну позицію у вхідному потоці;
- boolean markSupported() - повертає true, якщо потік підтримує методи mark() і reset();
- int read() - повертає цілочисельне подання наступного доступного символу наступного доступного символу викликаючого вхідного потоку. При досягненні кінця файлу повертає значення -1. Є й інші перевантажені версії методу;
- boolean ready() - повертає значення true, якщо наступний запит не чекатиме;
- void reset() - скидає покажчик введення в раніше встановлену позицію мітки;
- logn skip(long charCount) - пропускає вказану кількість символів введення, повертаючи кількість дійсно пропущених символів.

1. BufferedReader - буферизований вхідний символьний вхідний потік;
2. CharArrayReader - вхідний потік, який читає із символьного масиву;
3. FileReader - вхідний потік, який читає файл;
4. FilterReader - фільтрувальний читач;
5. InputStreamReader - вхідний потік, що транслює байти в символи;

6. LineNumberReader - вхідний потік, що підраховує рядки;
7. PipedReader - вхідний канал;
8. PushbackReader - вхідний потік, що дозволяє повертати символи назад у потік;
9. Reader - абстрактний клас, що описує символьне введення;
10. StringReader - вхідний потік, що читає з рядка.

Клас BufferedReader

Клас BufferedReader збільшує продуктивність за рахунок буферизації введення.

Клас CharArrayReader

Клас CharArrayReader використовує символьний масив як джерело.

Клас FileReader

Клас FileReader, похідний від класу Reader, можна використовувати для читання вмісту файлу. У конструкторі класу потрібно вказати або шлях до файлу, або об'єкт типу File.

Writer

Клас Writer - абстрактний клас, що визначає символьне потокове виведення. У разі помилок усі методи класу передають виняток IOException.

Методи класу:

- Writer append(char c) - додає символ у кінець вихідного потоку, що викликає. Повертає посилання на потік, що викликає;
- Writer append(CharSequence csq) - додає символи в кінець вихідного потоку, що викликає. Повертає посилання на потік, що викликає;
- Writer append(CharSequence csq, int start, int end) - додає діапазон символів у кінець вихідного потоку, що викликає. Повертає посилання на потік, що викликає;
- abstract void close() - закриває потік, що викликає;

- abstract void flush() - фіналізує вихідний стан так, що всі буфери очищаються
- void write(int oneChar) - записує єдиний символ у вихідний потік, що викликає. Є й інші перевантажені версії методу;

- BufferedWriter - буферизований вихідний символьний потік
- CharArrayWriter - вихідний потік, який пише в символьний масив
- FileWriter - вихідний потік, що пише у файл
- FilterWriter - фільтрувальний письменник
- OutputStreamWriter - вихідний потік, що транслює байти в символи
- PipedWriter - вихідний канал
- PrintWriter - вихідний потік, що включає методи print() і println()
- StringWriter - вихідний потік, що пише в рядок
- Writer - абстрактний клас, що описує символьне виведення

Клас BufferedWriter

Клас BufferedWriter - це клас, похідний від класу Writer, який буферизує виведення. З його допомогою можна підвищити продуктивність за рахунок зниження кількості операцій фізичного запису у вихідний пристрій.

Клас CharArrayWriter

Клас CharArrayWriter використовує масив для вихідного потоку.

Клас FileWriter

Клас FileWriter створює об'єкт класу, похідного від класу Writer, який ви можете застосовувати для запису файлу. Є конструктори, які дають змогу

```
FileInputStream(String filename) throws FileNotFoundException
FileOutputStream(String filename) throws FileNotFoundException
```

У filename потрібно вказати ім'я файлу, який ви хочете відкрити. Якщо під час створення вхідного потоку файл не існує, передається виняток FileNotFoundException. Аналогічно для вихідних потоків, якщо файл не може

бути відкритий або створений, також передається виняток. Сам клас виключення походить від класу IOException. Коли вихідний файл відкрито, будь-який раніше існуючий файл із тим самим ім'ям знищується.

Читання і запис файлів

Існує безліч класів і методів для читання та запису файлів. Найпоширеніші з них - класи FileInputStream і FileOutputStream, які створюють байтові потоки, пов'язані з файлами. Щоб відкрити файл, потрібно створити об'єкт одного з цих файлів, вказавши ім'я файлу як аргумент конструктора.

```
FileInputStream(String filename) throws FileNotFoundException
FileOutputStream(String filename) throws FileNotFoundException
```

У filename потрібно вказати ім'я файлу, який ви хочете відкрити. Якщо під час створення вхідного потоку файл не існує, передається виняток FileNotFoundException. Аналогічно для вихідних потоків, якщо файл не може бути відкритий або створений, також передається виняток. Сам клас виключення походить від класу IOException. Коли вихідний файл відкрито, будь-який раніше існуючий файл із тим самим ім'ям знищується.

Після завершення роботи з файлом, його необхідно закрити за допомогою методу close() для звільнення системних ресурсів. Незакритий файл призводить до витоку пам'яті.

У JDK 7 метод close() визначається інтерфейсом AutoCloseable і можна явно не закривати файл, а використовувати новий оператор try-з-ресурсами, що для Android поки що не надто актуально.

Щоб читати файл, потрібно викликати метод read(). Коли викликається цей метод, він читає єдиний байт із файлу і повертає його як ціле число. Коли буде досягнуто кінця файлу, то метод поверне значення -1. Приклади використання методів є в різних статтях на сайті.

Іноді використовують варіант, коли метод close() поміщається в блок finally. За такого підходу всі методи, які отримують доступ до файлу, містяться в межах блоку try, а блок finally використовується для закриття файлу. Таким чином, незалежно від того, як закінчиться блок try, файл буде закрито.

Оскільки виняток FileNotFoundException є підкласом IOException, то не обов'язково обробляти два винятки окремо, а залишити тільки IOException, якщо вам не потрібно окремо обробляти різні причини невдалого відкриття файлу. Наприклад, якщо користувач вводить вручну ім'я файлу, то більш конкретне виключення буде доречним.

Для запису у файл використовується метод write().

```
void write(int value) throws IOException
```

Метод пише у файл байт, переданий параметром value. Хоча параметр оголошено як цілочисельний, у файл записуються тільки молодші вісім біт. У разі помилки запису передається виняток.

У JDK 7 є спосіб автоматичного керування ресурсами:

```
try (специфікація ресурса) {
    // використання ресурса
}
```

Буферизоване читання з файлу з BufferedReader

Щоб відкрити файл для посимвольного читання, використовується клас FileInputReader; ім'я файлу задається у вигляді рядка (String) або об'єкта File. Прискорити процес читання допомагає буферизація введення, для цього отримане посилання передається в конструктор класу BufferedReader. Оскільки в інтерфейсі класу є метод readLine(), все необхідне для читання є у вашому розпорядженні. При досягненні кінця файлу метод readLine() повертає посилання null.

Виведення у файл - FileWriter

Об'єкт FileWriter записує дані у файл. При введенні/виведенні практично завжди застосовується буферизація, тому використовується BufferedWriter.

Коли дані вхідного потоку вичерпуються, метод readLine() повертає null. Для потоку явно викликається метод close(); якщо не викликати його для всіх вихідних файлових потоків, у буферах можуть залишитися дані, і файл вийде неповним.

Збереження та відновлення даних - PrintWriter

PrintWriter форматує дані так, щоб їх могла прочитати людина. Однак для виведення інформації, призначеної для іншого потоку, слід використовувати класи DataOutputStream для запису даних і DataInputStream для читання даних.

Єдиним надійним способом записати в потік DataOutputStream рядок так, щоб його можна було потім правильно зчитати потоком DataInputStream, є кодування UTF-8, що реалізується методами readUTF() і writeUTF(). Ці методи дають змогу змішувати рядки та інші типи даних, що записуються потоком DataOutputStream, бо ви знаєте, що рядки будуть правильно збережені в Юнікоді і їх буде просто відтворити потоком DataInputStream.

Метод writeDouble() записує число double у потік, а відповідний йому метод readDouble() потім відновлює його (для інших типів також існують подібні методи).

RandomAccessFile - Читання/запис файлів із довільним доступом

Робота з класом RandomAccessFile нагадує використання суміщених в одному класі потоків DataInputStream і DataOutputStream (вони реалізують ті самі інтерфейси DataInput і DataOutput). Крім того, метод seek() дає змогу переміститися до певної позиції та змінити значення, що там зберігається.

При використанні RandomAccessFile необхідно знати структуру файлу. Клас RandomAccessFile містить методи для читання і запису примітивів і рядків UTF-8.

RandomAccessFile може відкриватися в режимі читання ("r") або читання/запису ("rw"). Також є режим "rws", коли файл відкривається для операцій читання-запису і кожна зміна даних файлу негайно записується на фізичний пристрій.

Винятки введення/виведення

У більшості випадків у класів введення/виведення використовується виняток IOException. Друге виключення FileNotFoundException передається в тих випадках, коли файл не може бути відкритий. Дане виключення походить

від IOException, тому обидва виключення можна обробляти в одному блоці catch, якщо у вас немає потреби обробляти їх окремо.

Класс File

Клас File, визначений у пакеті java.io, не працює безпосередньо з потоками. Його завданням є управління інформацією про файли та каталоги. Хоча на рівні операційної системи файли і каталоги відрізняються, але в Java вони описуються одним класом File.

Залежно від того, що повинен представляти об'єкт File - файл або каталог, ми можемо використовувати один із конструкторів для створення об'єкта:

Клас File має низку методів, які дають змогу керувати файлами та каталогами. Розглянемо деякі з них:

- boolean createNewFile(): створює новий файл за шляхом, який передано в конструктор. У разі вдалого створення повертає true, інакше false;
- boolean delete(): видаляє каталог або файл за шляхом, який передано в конструктор. У разі вдалого видалення повертає true;
- boolean exists(): перевіряє, чи існує за вказаним у конструкторі шляхом файл або каталог. І якщо файл або каталог існує, то повертає true, інакше повертає false;
- String getAbsolutePath(): повертає абсолютний шлях для шляху, переданого в конструктор об'єкта;
- String getName(): повертає коротке ім'я файлу або каталогу;
- String getParent(): повертає ім'я батьківського каталогу;
- boolean isDirectory(): повертає значення true, якщо за вказаним шляхом розташовується каталог;
- boolean isFile(): повертає значення true, якщо за вказаним шляхом розташований файл;
- boolean isHidden(): повертає значення true, якщо каталог або файл є прихованими;
- long length(): повертає розмір файлу в байтах;

- long lastModified(): повертає час останньої зміни файлу або каталогу. Значення представляє кількість мілісекунд, що минули з початку епохи Unix;
- String[] list(): повертає масив файлів і підкаталогів, які знаходяться в певному каталозі;
- File[] listFiles(): повертає масив файлів і підкаталогів, які знаходяться в певному каталозі;
- boolean mkdir(): створює новий каталог і при вдалому створенні повертає значення true;
- boolean renameTo(File dest): перейменовує файл або каталог.

Завдання до виконання

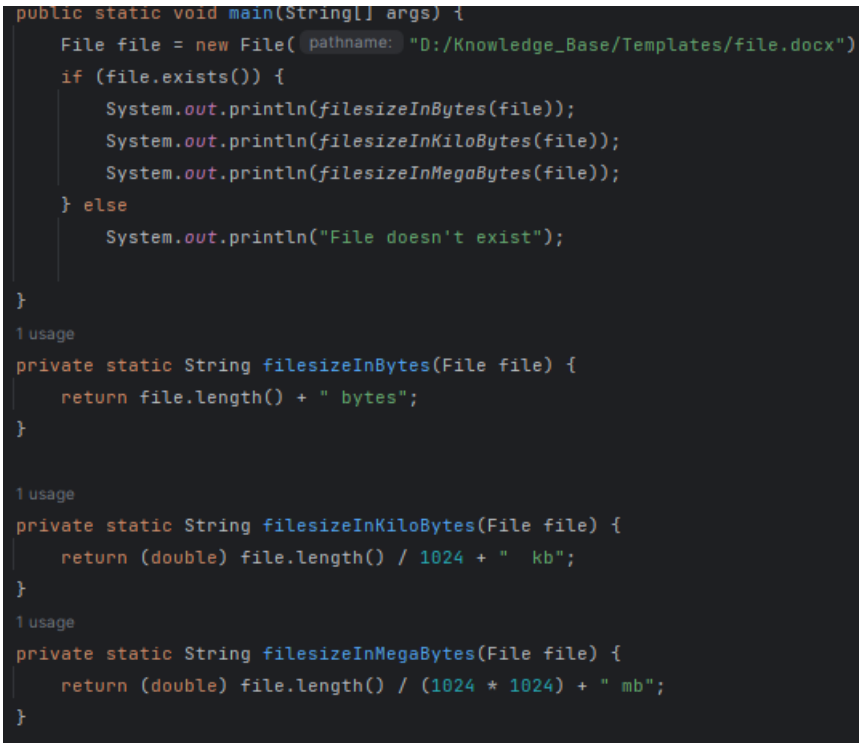
Виконати завдання згідно свого варіанту (табл 6.1)**.**

Таблиця 6.1 – Варіанти завдань

№	**Завдання**
1	Прочитати текст Java-програми і записати в інший файл у зворотному порядку символи кожного рядка.
2	Прочитати текст Java-програми і в кожному слові довше двох символів усі малі символи замінити прописними
3	У файлі, що містить прізвища студентів та їхні оцінки, записати великими великими літерами прізвища тих студентів, які мають середній бал понад 7.
4	З файлу видалити всі слова, що містять від трьох до п'яти символів, але при цьому цьому з кожного рядка має бути видалено тільки максимальну парну кількість таких слів.
5	Прочитати рядки з файлу і поміняти місцями перше й останнє слова у кожному рядку.
6	Вхідний файл містить сукупність рядків. Рядок файлу містить рядок квадратної матриці. Ввести матрицю у двовимірний масив (розмір матриці знайти). Вивести вихідну матрицю та результат її транспонування.
7	Напишіть програму на Java, яка знаходить найдовше слово у текстовому файлі.
8	Прочитати текст Java-програми і всі слова public в оголошенні атрибутів і методів класу замінити на слово private
9	Створити та заповнити файл випадковими цілими числами. Відсортувати вміст файлу за зростанням
10	Зберегти у файл, пов'язаний із вихідним потоком, записи про телефони та їхніх власників. Вивести у файл записи, телефони в яких починаються на k і на j.

Приклад виконання

Напишіть програму на Java, яка виводить розмір файлу в байтах, кілобайтах, мегабайтах.



```
public static void main(String[] args) {
    File file = new File( pathname: "D:/Knowledge_Base/Templates/file.docx");
    if (file.exists()) {
        System.out.println(filesizeInBytes(file));
        System.out.println(filesizeInKiloBytes(file));
        System.out.println(filesizeInMegaBytes(file));
    } else
        System.out.println("File doesn't exist");

}
1 usage
private static String filesizeInBytes(File file) {
    return file.length() + " bytes";
}

1 usage
private static String filesizeInKiloBytes(File file) {
    return (double) file.length() / 1024 + "  kb";
}
1 usage
private static String filesizeInMegaBytes(File file) {
    return (double) file.length() / (1024 * 1024) + " mb";
}
```

Рисунок 6.2 – Приклад виконання

Контрольні питання

1. Вкажіть класи, які представляють верхній рівень символьної ієрархії потоків введення-виводу в мові програмування Java;
2. Вкажіть класи, які представляють верхній рівень байтової ієрархії потоків введення-виводу в мові програмування Java;
3. Що мається на увазі під терміном "потік" при взаємодії з файловою системою?
4. Які відмінності між класами OutputStream, InputStream, Writer та Reader в мові програмування Java?
5. Які методи класу File вам відомі у мові програмування Java?
6. Для чого використовуються класи BufferedReader та BufferedWriter в контексті потоків введення-виводу в мові програмування Java?

ЛАБОРАТОРНА РОБОТА. ВИВЧЕННЯ БАЗОВОГО ФУНКЦІОНАЛУ STREAM API

Мета роботи: ознайомитись з поняттям Stream API в java та з роботою лямбда-виразів в java.

Основні теоретичні відомості

Вступ до лямбда-виразів

Ключем до розуміння лямбда-виразів слугують дві конструкції:

- лямбда-вираз;
- функціональний інтерфейс.

Почнемо з визначень кожного з цих понять. Лямбда-вираз - це, по суті, анонімний (тобто неіменований) метод. Однак сам цей метод ніколи не виконується. Він лише дає змогу призначити реалізацію коду методу, що визначається функціональним інтерфейсом. Таким чином, лямбда-вираз являє собою певну форму анонімного класу. Інший часто вживаний еквівалентний термін щодо лямбда-виразів - замикання.

Функціональний інтерфейс - це інтерфейс, який містить один і тільки один абстрактний метод. Зазвичай такий метод визначає передбачуване призначення інтерфейсу.

Тобто, функціональний інтерфейс представляє, як правило, якусь одну дію. Наприклад, стандартний інтерфейс Comparator є функціональним інтерфейсом, оскільки в ньому вказано тільки один метод -compare(), яким і визначається призначення інтерфейсу.

Про функціональні інтерфейси іноді говорять як про інтерфейси типу SAM, де SAM - скорочення від Single Abstract Method (одиночний абстрактний метод).

Лямбда-вирази вводять у мову Java новий синтаксис. У них використовується новий лямбда-оператор -> (інша назва - оператор-стрілка). Цей оператор розділяє лямбда-вираз на дві частини.

У лівій частині вказуються параметри, якщо цього вимагає лямбда-вираз.

У правій - тіло лямбда-виразу, яке описує дії, що виконуються лямбда-виразом. У Java підтримуються два різновиди тіл лямбда-виразів. Тіло одиночного лямбда-виразу складається з одного виразу, тіло блочного - з блоку коду.

Ми почнемо з розгляду лямбда-конструкцій, що визначають одиночний вираз.

Наведений нижче лямбда-вираз є, напевно, найпростішим із тих, які ми можемо використовувати. Він обчислює постійне значення:

```
() -> 77.7
```

Цей лямбда-вираз не приймає параметрів, тому список параметрів порожній.

Мається на увазі, що типом, який повертається, є double. Тобто, цей вираз еквівалентний такому методу:

```
double method () {
  return 77.7;
}
```

Метод, який визначається лямбда-виразом, не має імені.

Нижче наведено дещо цікавіший приклад лямбда-виразу.

```
() -> Math.random() * 666
```

Цей лямбда-вираз отримує псевдовипадкове значення за допомогою методу Math.random(), множить його на 666 і повертає результат. Він також не вимагає параметрів.

У разі лямбда-виразу, що приймає параметр, його вказують у списку параметрів ліворуч від лямбда-оператора:

```
(n) -> 1.0 / n;
```

Цей лямбда-вираз повертає результат, що являє собою зворотне значення параметра n. Тобто, якщо n дорівнює 2.0, то буде повернуто значення 0.5. Незважаючи на те, що тип параметра (у цьому прикладі параметра n) можна вказувати явно, цього часто можна не робити, якщо він легко встановлюється з контексту. Як і у випадку іменованих методів, у лямбда-виразі можна вказувати будь-яку необхідну кількість параметрів.

Як тип лямбда-виразу, що повертається, може використовуватися будь-який дійсний тип.

Наприклад, такий лямбда-вираз повертає значення true у разі парного значення параметра n і false - у разі непарного:

```
(n) -> (n % 2) == 0;
```

Таким чином, типом, що повертається, цього лямбда-виразу є boolean.

Якщо в лямбда-виразі є всього-лише один параметр, укладати його в дужки в лівій частині лямбда-оператора необов'язково. Наприклад, наведена нижче форма запису лямбда-виразу є абсолютно правильною.

```
n -> (n % 2) == 0;
```

Основи Stream API

Починаючи з JDK 8 у Java з'явився Stream API. Його завдання - спростити роботу з наборами даних, зокрема, спростити операції фільтрації, сортування та інші маніпуляції з даними. Уся основна функціональність цього API зосереджена в пакеті java.util.stream.

Ключовим поняттям у Stream API є потік даних. Взагалі сам термін "потік" є доволі перевантаженим у програмуванні загалом і в Java зокрема. В одному з попередніх розділів розглядалася робота з символьними і байтовими потоками при читанні-записі файлів. Стосовно Stream API потік являє собою канал передачі даних із джерела даних. Причому в якості джерела можуть виступати як файли, так і масиви та колекції.

Однією з відмінних рис Stream API є застосування лямбда-виразів, які дозволяють значно скоротити запис виконуваних дій.

При найближчому розгляді ми можемо знайти в інших технологіях програмування аналоги подібного API. Зокрема, у мові C# деяким аналогом Stream API буде технологія LINQ.

Деякі характеристики лямбда-виразів:

- є блоком коду з параметрами;
- використовувати потрібно, коли ми хочемо виконати блок коду в пізніший момент часу;
- можуть бути перетворені у функціональні інтерфейси;
- мають доступ до final змінних з навколишньої області видимості;

- посилання на метод і конструктор посилаються на методи або конструктори без їхнього виклику.

Для того, щоб працювати зі Stream API, потрібно виконати такий алгоритм:

- створити stream;
- виконати ланцюжок проміжних операцій з об'єктами stream;
- виконати одну термінальну операцію (об'єднання результату в колекцію, агрегатна функція тощо).

Створити stream можна кількома способами. Найпопулярніші з них:

- collection.stream()...
- Stream.of(значення1,... значенняN)...
- Arrays.stream(масив)...
- Files.lines(шлях до файлу)...
- "рядок".chars()...
- Stream.builder().add(...)....build()...
- Stream.iterate(початкова умова, вираз генерації)...

Приклади найпопулярніших методів при роботі з Stream API:

1) filter - відфільтровує записи, повертає тільки записи, що відповідають умові;

```
int[] filteredInts = IntStream.of(7, 12, 4, 666, 1)
  .filter(x -> x < 10) // залишаємо тільки елементи, менші за 10
  .toArray();
System.out.println(Arrays.toString(filteredInts));
// [7, 4, 1]
```

2) map - перетворює кожен елемент стріму;

```
      int[] mappedInts = IntStream.of(12, 666, 1)
        .map(x -> x + 7) // додаємо до кожного елемента елементу, що пройшов фільтр 7
        .toArray();
      System.out.println(Arrays.toString(mappedInts));
      // [19, 673, 8]
```

3) flatMap - схоже на map, але може створювати з одного елемента кілька. На практиці часто використовується, щоб перетворити масив, який містить всередині інші масиви, на звичайний масив з елементів;

```
int[] ints = IntStream.of(2, 5)
  .flatMap(x -> IntStream.range(0, x))
```

```
  .toArray();
System.out.println(Arrays.toString(ints));
// [0, 1, 0, 1, 2, 3, 4]
```

4) forEach - застосовує функцію до кожного об'єкта стріму, порядок під час паралельного виконання не гарантується;

```
IntStream.of(7, 666, 1)
  .filter(x -> x < 10) //
  .forEach(System.out::println);
  // 7
  // 1
```

5) collect(Collector collector) - подання результатів у вигляді колекцій та інших структур даних. Один із найпотужніших операторів Stream API. З його допомогою можна зібрати всі елементи в список, множину або іншу колекцію, згрупувати елементи за яким-небудь критерієм, об'єднати все в рядок тощо.

```
List<Integer> list = Stream.of(1, 2, 3)
  .collect(Collectors.toList());
// list: [1, 2, 3]
```

У класу Collectors також є методи toSet(), toMap() і деякі інші.

```
String s = Stream.of(1, 2, 3)
  .map(String::valueOf)
  .collect(Collectors.joining("-", "<", ">"));
// s: "<1-2-3>"
```

Додаткові методи Stream API

1) findFirst - повертає перший елемент зі стріму (повертає Optional).

```
int firstElem = IntStream.range(23, 666)
  .findFirst()
  .getAsInt();
// firstElem: 23
```

2) anyMatch - повертає true, якщо умова виконується хоча б для одного елемента;

```
boolean result = Stream.of(1, 2, 3, 4, 5)
  .anyMatch(x -> x == 3);
// result: true
```

3) skip - дає змогу пропустити N перших елементів;

```
int firstElem = IntStream.range(23, 666)
  .skip(1)
  .findFirst()
  .getAsInt();
// firstElem: 666
```

4) distinct - повертає стрім без дублікатів;

5) peek - повертає той самий стрім, але застосовує функцію до кожного його елемента;

6) limit - дозволяє обмежити вибірку певною кількістю перших елементів;

7) findAny - повертає будь-який відповідний елемент зі стріму (повертає Optional);

8) noneMatch - повертає true, якщо умова не виконується для жодного елемента;

9) allMatch - повертає true, якщо умова виконується для всіх елементів;

10) min - повертає мінімальний елемент, як умову використовується компаратор;

11) max - повертає максимальний елемент, як умову використовує компаратор;

12) sum - повертає суму всіх чисел;

13) count - повертає кількість елементів;

14) toArray - повертає масив зі значень стріму;

15) reduce - дає змогу виконувати агрегатні функції над усією колекцією і повертати один результат;

16) sorted - дозволяє сортувати значення або в натуральному порядку, або задаючи Comparator.

Тип Optional

Низка операцій, такі як min, max, reduce, повертають об'єкт Optional<T>. Цей об'єкт фактично обгортає результат операції. Після виконання операції за допомогою методу get() об'єкта Optional ми можемо отримати його значення:

```
import java.util.Optional;
import java.util.ArrayList;
import java.util.Arrays;
public class Program {

    public static void main(String[] args) {

        ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.addAll(Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8,9}));
        Optional<Integer> min = numbers.stream().min(Integer::compare);
        System.out.println(min.get());  // 1
    }
}
```

Але що, якщо потік не містить взагалі ніяких даних:

```
// список numbers пустой
ArrayList<Integer> numbers = new ArrayList<Integer>();
Optional<Integer> min = numbers.stream().min(Integer::compare);
System.out.println(min.get());  // java.util.NoSuchElementException
```

У цьому випадку програма видасть виняток java.util.NoSuchElementException. Що ми можемо зробити, щоб уникнути викидання винятку? Для цього клас Optional надає низку методів.

Найпростіший спосіб уникнути подібної ситуації - це попередня перевірка наявності значення в Optional за допомогою методу isPresent(). Він повертає true, якщо значення присутнє в Optional, і false, якщо значення відсутнє:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
Optional<Integer> min = numbers.stream().min(Integer::compare);
if(min.isPresent()){

    System.out.println(min.get());
}
```

orElse

Метод orElse() дає змогу визначити альтернативне значення, яке буде повертатися, якщо Optional не отримає з потоку якого-небудь значення:

```
// порожній список
ArrayList<Integer> numbers = new ArrayList<Integer>();
Optional<Integer> min = numbers.stream().min(Integer::compare);
System.out.println(min.orElse(-1)); // -1

// непорожній список
numbers.addAll(Arrays.asList(new Integer[]{4,5,6,7,8,9}));
min = numbers.stream().min(Integer::compare);
System.out.println(min.orElse(-1)); // 4
```

orElseGet

Метод orElseGet() дає змогу задати функцію, яка повертатиме значення за замовчуванням:

```
import java.util.Optional;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;

public class Program {

    public static void main(String[] args) {

        ArrayList<Integer> numbers = new ArrayList<Integer>();
        Optional<Integer> min = numbers.stream().min(Integer::compare);
        Random rnd = new Random();
        System.out.println(min.orElseGet(()->rnd.nextInt(100)));
    }
}
```

У цьому випадку значення, що повертається, генерується за допомогою методу nextInt класу Random, який повертає випадкове число.

OrElseThrow

Ще один метод - orElseThrow дозволяє згенерувати виняток, якщо Optional не містить значення:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
Optional<Integer> min = numbers.stream().min(Integer::compare);
// генеррация исключения IllegalStateException
System.out.println(min.orElseThrow(IllegalStateException::new));
```

Обробка отриманого значення

Метод ifPresent() визначає дії зі значенням в Optional, якщо значення є:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.addAll(Arrays.asList(new Integer[]{4,5,6,7,8,9}));
Optional<Integer> min = numbers.stream().min(Integer::compare);
min.ifPresent(v->System.out.println(v)); // 4
```

У метод ifPresent передається функція, яка приймає один параметр - значення з Optional. У цьому випадку отримане мінімальне число виводиться на консоль. Але якби масив numbers був би порожнім, і відповідно Optional не стримав би жодного значення, то ніякої помилки б не було.

Метод ifPresentOrElse() дає змогу визначити альтернативну логіку на випадок, якщо значення в Optional відсутнє:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
Optional<Integer> min = numbers.stream().min(Integer::compare);
min.ifPresentOrElse(
     v -> System.out.println(v),
    () -> System.out.println("Value not found")
);
```

У метод ifPresentOrElse передається дві функції. Перша обробляє значення в Optional, якщо воно присутнє. Друга функція представляє дії, які виконуються, якщо значення в Optional відсутнє.

Завдання до виконання

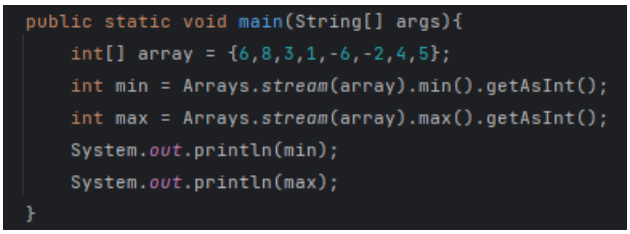
Виконати завдання згідно свого варіанту (табл 7.1)**.**

Таблиця 7.1 – Варіанти завдань

№	**Завдання**
1	Створіть список рядків, які представляють собою речення або тексти (через ввведення з клавіатури). Напишіть програму на Java для фільтрації рядків, які містять слово «Java» і збережіть їх у новий список.
2	Напишіть програму на Java для перетворення списку рядків у верхній або нижній регістр з використанням потоків.
3	Напишіть програму на Java, яка обчислює суму всіх парних та непарних чисел у списку, використовуючи потоки.
4	Напишіть програму на Java, яка видаляє всі унікальні елементи списку, залишаючи лише ті, що повторюються, використовуючи потоки.
5	Створіть клас товару з полями, такими як назва, ціна, категорія. Створіть список товарів та напишіть програму на Java, яка знаходить середню ціну товарів в категорії «електроніка».
6	Напишіть програму на Java, яка сортує список рядків в алфавітному порядку за зростанням та спаданням, використовуючи потоки.
7	Створіть клас користувача з полями, такими як ім`я, вік, статус тощо. Створіть список користувачів та напишіть програму на Java для фільтрації тих, хто має вік більше 25 років та статус «активний».
8	Напишіть програму на Java, яка знаходить другий найменший та найбільший елементи у списку цілих чисел, використовуючи потоки.
9	Напишіть програму на Java, яка підраховує кількість рядків у списку, що починаються на певну літеру, використовуючи потоки.
10	Напишіть програму на Java для обчислення середнього значення списку цілих чисел з використанням потоків.

Приклад виконання

Напишіть програму на Java, яка знаходить максимальне та мінімальне значення у списку цілих чисел, використовуючи потоки.



```
public static void main(String[] args){
    int[] array = {6,8,3,1,-6,-2,4,5};
    int min = Arrays.stream(array).min().getAsInt();
    int max = Arrays.stream(array).max().getAsInt();
    System.out.println(min);
    System.out.println(max);
}
```

Рисунок 7.1 – Приклад виконання

Контрольні питання

1. Охарактеризуйте концепцію лямбда-виразу в програмуванні, зазначте його основні властивості та призначення;
2. Визначте та розкрийте концепцію функціонального інтерфейсу в мові програмування Java, виокремте ключові характеристики цього підходу;
3. Надайте визначення Stream API в мові програмування Java та поясніть, як цей інтерфейс поліпшує обробку даних та взаємодію з колекціями;
4. Які методи є найбільш популярними при використанні Stream API для обробки даних в мові програмування Java?
5. Для чого призначений тип Optional в мові програмування Java та в яких випадках його використання є доцільним?
6. Розкрийте відмінності між колекцією та потоком в контексті мови програмування Java;
7. Чи можливе перетворення масиву на потік у мові програмування Java? Як це виконується?
8. Назвіть деякі вбудовані функціональні інтерфейси в мові програмування Java та поясніть їхнє призначення.

ЛАБОРАТОРНА РОБОТА. ОПАНУВАННЯ РОБОТИ З БАЗАМИ ДАНИХ З ВИКОРИСТАННЯМ JDBC

Мета роботи: ознайомитись з поняттям баз даних та отримати базові навички роботи підключення до СУБД використовуючи JDBC.

Основні теоретичні відомості

Поняття БД та JDBC

База даних – це організований набір структурованої інформації або даних, які зазвичай зберігаються в електронному вигляді в комп'ютерній системі. Зазвичай базою даних керує система управління базами даних (СУБД). Разом дані та СУБД, а також додатки, які з ними пов'язані, називаються системою баз даних, яку часто скорочують до просто бази даних.

Дані в найпоширеніших типах баз даних, що використовуються сьогодні, зазвичай моделюються у вигляді рядків і стовпців у серії таблиць, щоб зробити обробку та запити до даних ефективними. Тоді до даних можна легко отримати доступ, керувати ними, змінювати, оновлювати, контролювати та організовувати. Більшість баз даних використовують мову структурованих запитів (SQL) для запису і запитів до даних.

Для зберігання даних ми можемо використовувати різні бази даних - Oracle, MS SQL Server, MySQL, Postgres тощо. Усі ці системи управління базами даних мають свої особливості. Головне, що їх об'єднує, це взаємодія зі сховищем даних за допомогою команд SQL. І щоб визначити єдиний механізм взаємодії з цими СУБД у Java ще починаючи з 1996 року було введено спеціальний прикладний інтерфейс API, який називається JDBC.

Тобто якщо ми хочемо в додатку мовою Java взаємодіяти з базою даних, то необхідно використовувати функціональні можливості JDBC. Даний API входить до складу Java, зокрема, для роботи з JDBC в програмі Java досить підключити пакет java.sql. Для роботи в Java EE є аналогічний пакет javax.sql, який розширює можливості JDBC.

Однак не всі бази даних можуть підтримуватися через JDBC. Для роботи з певною СУБД також необхідний спеціальний драйвер. Кожен розробник

певної СУБД зазвичай надає свій драйвер для роботи з JDBC. Тобто якщо ми хочемо працювати з MySQL, то нам потрібен спеціальний драйвер для роботи саме MySQL. Як правило, більшість драйверів доступні у вільному доступі на сайтах відповідних СУБД. Зазвичай вони представляють JAR-файли. І перевага JDBC якраз і полягає в тому, що ми абстрагуємося від будови конкретної бази даних, а використовуємо уніфікований інтерфейс, який єдиний для всіх.

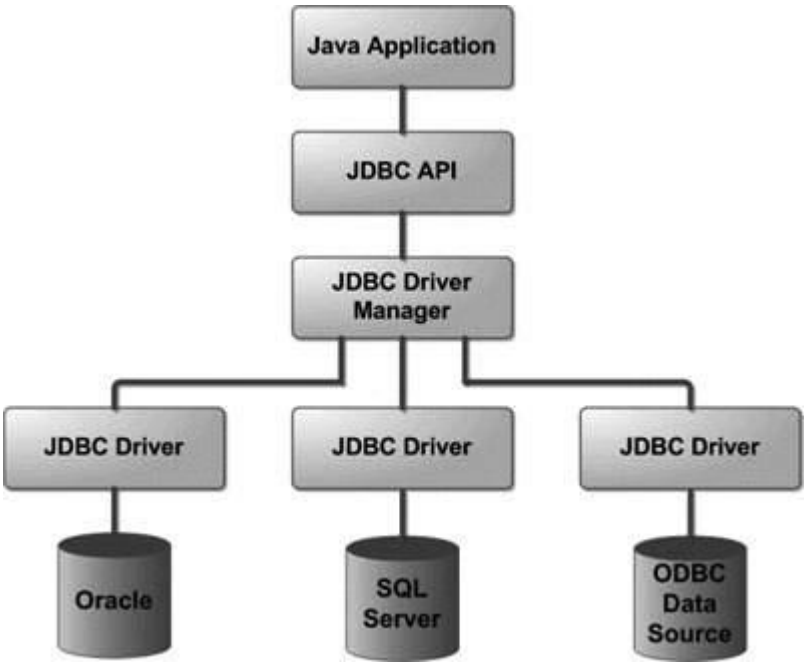


Рисунок 8.1 – Архітектура JDBC

Для встановлення драйвера ми використаємо Maven – фреймворк для автоматизації складання проектів. Для цього переходимо за посиланням https://mvnrepository.com/, в пошуку знаходимо драйвер PostgreSQL та копіюємо залежність знизу сторінки.

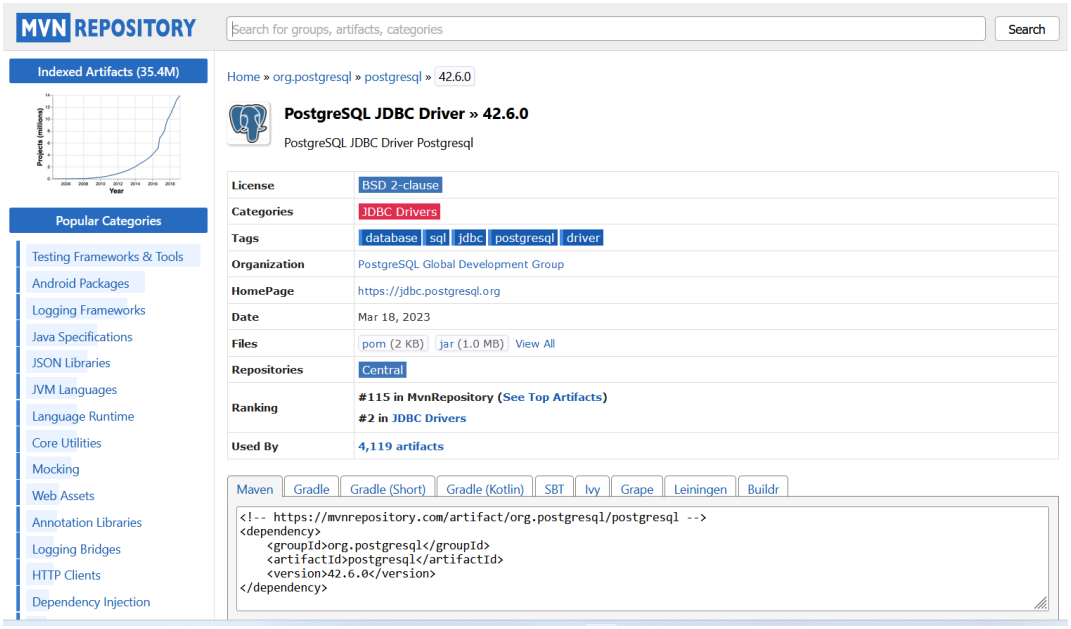


Рисунок 8.2 – Драйвер PostgreSQL в репозиторії Maven

Тепер створимо проект в Intellij IDEA, після цього переходимо до файлу pom.xml, та вставляємо залежність.

pom.xml (Test)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>ua.aharoo</groupId>
    <artifactId>Test</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>20</maven.compiler.source>
        <maven.compiler.target>20</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.6.0</version>
        </dependency>
    </dependencies>

</project>
```

Рисунок 8.3 – Файл pom.xml

Переконаємося, що ми можемо здійснювати взаємодію з PostgreSQL через цей драйвер. Для цього визначимо такий код програми:

```
public class Test{
    public static void main(String[] args) {
        try{

Class.forName("org.postgresql.Driver").getDeclaredConstructor().newInstance();
            System.out.println("Connection successful!");
        }
        catch(Exception ex){
            System.out.println("Connection failed!");
            System.out.println(ex);
        }
    }
}
```

Для завантаження драйвера тут застосовується рядок:

```
Class.forName("org.postgresql.Driver").getDeclaredConstructor().newInstance();
```

Метод Class.forName() як параметр приймає рядок, який представляє повний шлях до класу драйвера з урахуванням усіх пакетів. У випадку MySQL це шлях " org.postgresql.Driver". Таким чином, Метод Class.forName завантажує клас драйвера, який буде використовуватися.

Далі викликається метод getDeclaredConstructor(), який повертає конструктор даного класу. І в кінці викликається метод newInstance(), який створює за допомогою конструктора об'єкт даного класу. І після цього ми зможемо взаємодіяти з сервером PostgreSQL.

Підключення до бази даних

Для підключення до бази даних необхідно створити об'єкт java.sql.Connection. Для його створення застосовується метод:

```
Connection DriverManager.getConnection(url, username, password)
```

Метод DriverManager.getConnection як параметри приймає адресу джерела даних, логін і пароль. Як логін і пароль передаються логін і пароль від сервера PostgreSQL. Адреса локальної бази даних MySQL вказується в такому форматі: jdbc:postgresql://localhost/назва_бази даних

Приклад створення підключення до створеної вище локальної бази даних store:

```
try (Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/store",
"root", "password")){
}
```

Файли конфігурації

Ми можемо визначити всі дані для підключення безпосередньо в програмі. Однак що якщо якісь дані було змінено? У цьому разі знадобиться перекомпіляція програми. Іноді це не завжди зручно, наприклад, відсутній доступ до вихідних кодів, або перекомпіляція займе досить тривалий час. У цьому випадку ми можемо зберігати налаштування у файлі.

Так, створимо в папці програми новий текстовий файл database.properties, у якому визначимо налаштування підключення:

```
url = "jdbc:postgresql://localhost:5432/test";
username = "postgres";
password = "password";
```

Завантажимо ці налаштування в програмі:

```
public static void main(String[] args) {
    try {

Class.forName("org.postgresql.Driver").getDeclaredConstructor().newInstance();
        try (Connection conn = getConnection()) {

            System.out.println("Connection to Store DB succesfull!");
        }
    } catch (Exception ex) {
        System.out.println("Connection failed...");

        System.out.println(ex);
    }
}
public static Connection getConnection() throws SQLException, IOException {

    Properties props = new Properties();
    try (InputStream in =
Files.newInputStream(Paths.get("D:/Study/Test1/src/main/java/database.properties
"))) {
        props.load(in);
    }
    String url = props.getProperty("url");
    String username = props.getProperty("username");
    String password = props.getProperty("password");

    return DriverManager.getConnection(url, username, password);
}
```

Виконання команд. Метод executeUpdate

Для взаємодії з базою даних додаток надсилає серверу PostgreSQL команди мовою SQL. Щоб виконати команду, спочатку необхідно створити об'єкт Statement.

Для його створення в об'єкта Connection викликається метод createStatement():

```
Statement statement = conn.createStatement();
```

Для виконання команд SQL у класі Statement визначено три методи:

- executeUpdate: виконує такі команди, як INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE. Як результат повертає кількість рядків, зачеплених операцією (наприклад, кількість доданих, змінених або видалених рядків), або 0, якщо жодного рядка не зачеплено операцією або якщо команда не змінює вміст таблиці (наприклад, команда створення нової таблиці);
- executeQuery: виконує команду SELECT. Повертає об'єкт ResultSet, який містить результати запиту;
- execute(): виконує будь-які команди і повертає значення boolean: true - якщо команда повертає набір рядків (SELECT), інакше повертається false.

Метод executeQuery. Отримання даних

Для вибірки даних за допомогою команди SELECT застосовується метод executeQuery:

```
ResultSet executeQuery("Команда_SQL")
```

Метод повертає об'єкт ResultSet, який містить усі отримані дані. В об'єкті ResultSet ітератор встановлюється на позиції перед першим рядком. І щоб перемістити до першого рядка (і до всіх наступних) необхідно викликати метод next(). Поки в наборі ResultSet є доступні рядки, метод next повертатиме true. Типове переміщення по набору рядків:

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM Products");
while(resultSet.next()){
    // отримання вмісту рядків
}
```

Тобто поки в resultSet є доступні рядки, виконуватиметься цикл while, який переходитиме до наступного рядка в наборі.

Після переходу до рядка ми можемо отримати його вміст. Для цього у ResultSet визначено низку методів. Деякі з них:

- getBoolean() повертає значення boolean;
- getDate() повертає значення Date;
- getDouble() повертає значення double;
- getInt() повертає значення int;
- getFloat() повертає значення float;

- getLong() повертає значення long;
- getNString() повертає значення String (призначений для доступу до стовпців NCHAR, NVARCHAR та LONGNVARCHAR);
- getString() повертає значення String.

Залежно від того, дані якого типу зберігаються в тому чи іншому стовпці, ми можемо використовувати той чи інший метод. Кожен із цих методів має дві версії:

```
int getInt(int columnIndex)
int getInt(String columnLabel)
```

Перша версія отримує дані зі стовпця з номером columnIndex. Друга версія отримує дані зі стовпця з назвою columnLabel.

PreparedStatement

Крім класу Statement у java.sql ми можемо використовувати для виконання запитів ще один клас - PreparedStatement. Крім власне виконання запиту цей клас дозволяє підготувати запит, відформатувати його належним чином.

Наприклад, створимо таблицю яка має три стовпця:

```
CREATE TABLE products (
    Id SERIAL PRIMARY KEY,
    ProductName VARCHAR(20),
    Price INT
)
```

За допомогою PreparedStatement додамо в неї один об'єкт:

```
public static void main(String[] args) {
    try {
        Scanner scanner = new Scanner(System.in);

Class.forName("org.postgresql.Driver").getDeclaredConstructor().newInstance();
        System.out.println("Enter product name:");
        String name = scanner.nextLine();

        System.out.println("Enter product price:");
        int price = scanner.nextInt();

        try (Connection conn = getConnection()) {
            String sql = "INSERT INTO Products (ProductName, Price) Values (?,
?)";
            PreparedStatement preparedStatement = conn.prepareStatement(sql);
            preparedStatement.setString(1, name);
            preparedStatement.setInt(2, price);

            int rows = preparedStatement.executeUpdate();

            System.out.printf("%d rows added", rows);
        }
    } catch (Exception ex) {
```

```
            System.out.println("Connection failed...");

            System.out.println(ex);
        }
}
```

У цьому разі дані вводяться з консолі і потім додаються в базу даних. Для створення об'єкта PreparedStatement застосовується метод prepareStatement() класу Connection. У цей метод передається вираз sql INSERT INTO Products (ProductName, Price) Values (?, ?). Цей вираз може містити знаки запитання ? - знаки підстановки, замість яких будуть вставлятися реальні значення.

Щоб пов'язати окремі знаки підстановки з конкретними значеннями, у класі PreparedStatement визначено низку методів для різних типів даних. Усі методи, які постачають значення замість знаків підстановки, як перший параметр приймають порядковий номер знака підстановки (нумерація починається з 1), а як другий параметр - власне значення, яке вставляється замість знака підстановки.

Наприклад, перший знак підстановки ? у виразі sql представляє значення для стовпця ProductName, який зберігає рядок. Тому для зв'язку значення з першим знаком підстановки застосовується метод preparedStatement.setString(1, name).

Другий знак підстановки повинен передавати значення для стовпця Price, який зберігає цілі числа. Тому для вставки значення використовується метод preparedStatement.setInt(2, price)

Крім setString і setInt PreparedStatement має ще низку подібних методів, які працюють подібним чином. Деякі з них: setBigDecimal, setBoolean, setDate, setDouble, setFloat, setLong, setNull, setTime.

Для виконання запиту PreparedStatement має три методи:

- boolean execute(): виконує будь-яку SQL-команду;
- ResultSet executeQuery(): виконує команду SELECT, яка повертає дані у вигляді ResultSet;
- int executeUpdate(): виконує такі SQL-команди, як INSERT, UPDATE, DELETE, CREATE і повертає кількість змінених рядків.

При цьому на відміну від методів Statement ці методи не приймають SQL-вираз.

Завдання до виконання

Виконати завдання згідно свого варіанту (табл 8.1). В цьому завданні ви повинні використовувати обробку винятків для перевірки введення правильних типів даних. При необхідності додайте додаткові таблиці для виконання завдання. Примітка: Всі назви БД та таблиць повинні бути англійською мовою!

Додаткове завдання (необов'язково): спробуйте розгорнути СУБД з використанням контейнерів (docker) або розгорнути її на хмарі (cloud database).

Таблиця 8.1 – Варіанти завдань

№	Завдання
1	**Робота з базою даних магазину** Створіть базу даних для керування продуктами в інтернет-магазині. База даних повинна містити таблиці "Товари" (з полями: ID, Назва, Ціна, Кількість на складі) і "Замовлення" (з полями: ID, Дата, Ім'я покупця). Розробіть Java-додаток, який дає змогу додавати нові товари, змінювати їхню ціну та кількість на складі, а також оформлювати замовлення (створювати записи в таблиці "Замовлення") із зазначенням товарів і кількості кожного товару.
2	**Облік студентів в університеті** Створіть базу даних для обліку студентів в університеті. База даних повинна містити таблиці "Студенти" (з полями: ID, Ім'я, Прізвище, Рік вступу) і "Курси" (з полями: ID, Назва курсу, Викладач). Напишіть Java-додаток, який дає змогу додавати нових студентів, присвоювати їм курси та виводити список студентів на певному курсі.
3	**Облік завдань у проєкті** Створіть базу даних для обліку завдань у проекті. База даних повинна містити таблиці "Задачі" (з полями: ID, Назва, Опис, Статус) і "Співробітники" (з полями: ID, Ім'я, Посада). Розробіть Java-додаток, який дає змогу створювати нові завдання, призначати відповідальних співробітників, змінювати статус завдання і виводити список завдань на певного співробітника.
4	**Система обліку бібліотеки** Створіть базу даних для обліку книг у бібліотеці. База даних повинна містити таблиці "Книги" (з полями: ID, Назва, Автор, Рік видання) і "Читачі" (з полями: ID, Ім'я, Прізвище). Напишіть Java-додаток, який дає змогу додавати нові книжки, додати нового читача, видавати книжки читачам, відмічати повернення книжок і виводити список книжок, узятих певним читачем (додати нового читача)
5	**Система обліку фінансів** Створіть базу даних для обліку фінансової інформації. База даних повинна містити таблицю "Транзакції" (з полями: ID, Дата, Опис, Сума) і таблицю "Рахунки" (з полями: ID, Назва рахунку, Баланс). Розробіть Java-додаток, який дає змогу створювати нові транзакції, оновлювати баланс рахунків і виводити історію транзакцій для певного рахунку.
6	**Облік замовлень у ресторані** Створіть базу даних для обліку замовлень у ресторані. База даних повинна містити таблиці "Замовлення" (з полями: ID, Дата і час замовлення, Столик) і "Страви" (з полями: ID, Назва, Ціна). Розробіть Java-додаток, який дає змогу оформляти нові замовлення, додавати страви до замовлення, виводити страву, розраховувати загальну суму замовлення та виводити список усіх замовлень.
7	**Система обліку завдань у команді розробників** Створіть базу даних для обліку завдань у команді розробників. База даних повинна містити таблиці "Задачі" (з полями: ID, Назва, Опис, Статус) і "Розробники" (з полями: ID, Ім'я, Навички). Напишіть Java-додаток, який дає змогу створювати нові задачі, призначати розробників на задачі, змінювати статус задачі та виводити список завдань, призначених певному розробнику.
8	**Система обліку персоналу в компанії** Створіть базу даних для обліку співробітників у компанії. База даних повинна містити таблиці "Співробітники" (з полями: ID, Ім'я, Прізвище, Посада) і "Відділи" (з полями: ID, Назва відділу). Розробіть Java-додаток, який дає змогу додавати нових співробітників у відділи, змінювати інформацію про співробітників і виводити список співробітників певного відділу.
9	**Система обліку акцій на біржі** Створіть базу даних для обліку акцій на фінансовому ринку. База даних має містити таблиці "Акції" (з полями: ID, Назва, Ціна) та "Інвестори" (з полями: ID, Ім'я, Портфель). Напишіть Java-додаток, який дає змогу інвесторам купувати і продавати акції, оновлювати ціни акцій і виводити інформацію про портфелі інвесторів.
10	**Система обліку студентів у школі** Створіть базу даних для обліку студентів у школі. База даних повинна містити таблиці "Студенти" (з полями: ID, Ім'я, Прізвище, Дата народження) та "Вчителі" (з полями: ID, Ім'я, Предмет). Розробіть Java-додаток, який дає змогу додавати нових студентів, створити вчителя, призначати вчителів, змінювати інформацію про студентів і виводити список студентів певного вчителя.

Приклад виконання

Облік клієнтів і замовлень в інтернет-магазині

Створіть базу даних для обліку клієнтів і замовлень в інтернет-магазині. База даних повинна містити таблиці "Клієнти" (з полями: ID, Ім'я, Прізвище, Адреса) і "Замовлення" (з полями: ID, Дата замовлення, Статус замовлення, Сума замовлення).

Розробіть Java-додаток, який дає змогу додавати нових клієнтів, створювати замовлення для клієнтів, оновлювати статус замовлення та виводити список замовлень для певного клієнта.

Для початку створимо БД InternetShop та таблиці Clients та Orders.

```
CREATE TABLE Clients (
    ID INT SERIAL PRIMARY KEY,
    FirstName VARCHAR(255) NOT NULL,
    LastName VARCHAR(255) NOT NULL,
    Address VARCHAR(255)
);

CREATE TABLE Orders (
    ID INT SERIAL PRIMARY KEY,
    OrderDate DATE,
    Status VARCHAR(50),
    TotalAmount DECIMAL(10, 2),
    ClientID INT,
    FOREIGN KEY (ClientID) REFERENCES Clients(ID)
);
```

Рисунок 8.4 – Створення таблиць

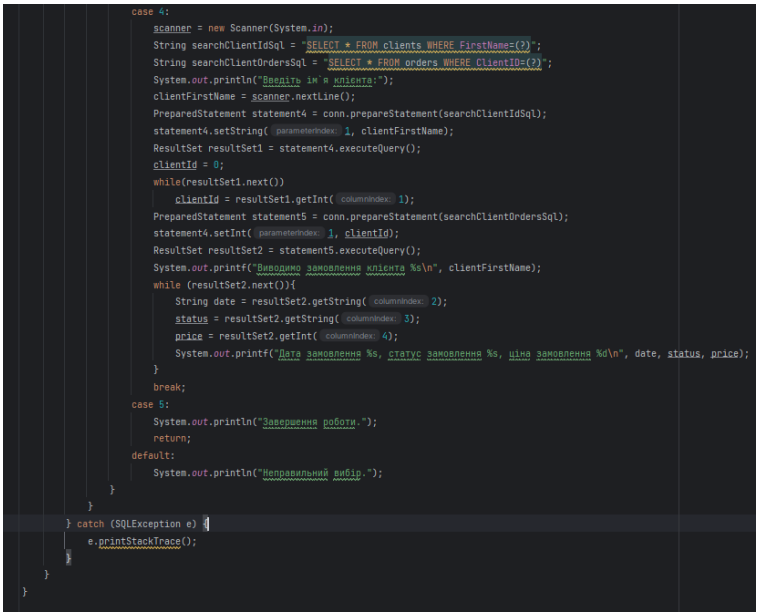
Тепер розробимо додаток.

```java
class InternetShopApp {
    1 usage
    static final String JDBC_URL = "jdbc:postgresql://localhost:5432/internetshop";
    1 usage
    static final String JDBC_USER = "postgres";
    1 usage
    static final String JDBC_PASSWORD = "a19041999";

    public static void main(String[] args) {
        try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER, JDBC_PASSWORD)) {
            System.out.println("З'єднання з базою даних успішно встановлено.");

            Scanner scanner = new Scanner(System.in);
            while (true) {
                System.out.println("Виберіть дію:");
                System.out.println("1. Додати клієнта");
                System.out.println("2. Створити замовлення");
                System.out.println("3. Оновити статус замовлення");
                System.out.println("4. Вивести список замовлень для клієнта");
                System.out.println("5. Вихід");
                int choice = scanner.nextInt();

                switch (choice) {
                    case 1:
                        scanner = new Scanner(System.in);
                        String addClientSql = "INSERT INTO Clients (FirstName, LastName, Address) VALUES (?,?,?)";
                        System.out.println("Введіть ім`я:");
                        String firstName = scanner.nextLine();
                        System.out.println("Введіть прізвище:");
                        String lastName = scanner.nextLine();
                        System.out.println("Введіть адресу:");
                        String address = scanner.nextLine();
                        PreparedStatement statement = conn.prepareStatement(addClientSql);
                        statement.setString( parameterIndex: 1, firstName);
                        statement.setString( parameterIndex: 2, lastName);
                        statement.setString( parameterIndex: 3, address);
                        statement.executeUpdate();
                        System.out.printf("Клієнта з ім`ям %s було додано.", firstName);
                        break;
                    case 2:
                        scanner = new Scanner(System.in);
                        String addOrderSql = "INSERT INTO Orders (OrderDate, Status, TotalAmount, ClientID) VALUES (?,?,?,?)";
                        String findClientSql = "SELECT * FROM Clients WHERE FirstName=(?)";
                        System.out.println("Введіть ім`я клієнта:");
                        String clientFirstName = scanner.nextLine();
                        System.out.println("Введіть ціну замовлення");
```

Рисунок 8.5 – Перша частина додатку

```java
                        scanner = new Scanner(System.in);
                        String addOrderSql = "INSERT INTO Orders (OrderDate, Status, TotalAmount, ClientID) VALUES (?,?,?,?)";
                        String findClientSql = "SELECT * FROM Clients WHERE FirstName=(?)";
                        System.out.println("Введіть ім`я клієнта:");
                        String clientFirstName = scanner.nextLine();
                        System.out.println("Введіть ціну замовлення");
                        int orderTotalAmount = scanner.nextInt();
                        PreparedStatement statement1 = conn.prepareStatement(addOrderSql);
                        statement1.setTimestamp( parameterIndex: 1, new Timestamp(System.currentTimeMillis()));
                        statement1.setString( parameterIndex: 2,  x: "IN PROCESS");
                        statement1.setInt( parameterIndex: 3, orderTotalAmount);
                        PreparedStatement statement2 = conn.prepareStatement(findClientSql);
                        statement2.setString( parameterIndex: 1, clientFirstName);
                        ResultSet resultSet = statement2.executeQuery();
                        int clientId = 0;
                        while(resultSet.next())
                            clientId = resultSet.getInt( columnIndex: 1);
                        statement1.setInt( parameterIndex: 4, clientId);
                        statement1.executeUpdate();
                        System.out.printf("Замовлення клієнта %s було додано.", clientFirstName);
                        break;
                    case 3:
                        scanner = new Scanner(System.in);
                        String changeOrderStatus = "UPDATE orders SET Status=(?) WHERE TotalAmount=(?)";
                        System.out.println("Напишіть статус замовлення:");
                        String status = scanner.nextLine();
                        System.out.println("Напишіть ціну замовлення:");
                        int price = scanner.nextInt();
                        PreparedStatement statement3 = conn.prepareStatement(changeOrderStatus);
                        statement3.setString( parameterIndex: 1, status);
                        statement3.setInt( parameterIndex: 2, price);
                        statement3.executeUpdate();
                        System.out.printf("Статус замовлення з ціною %d успішно змінено.", price);
                        break;
                    case 4:
                        scanner = new Scanner(System.in);
                        String searchClientIdSql = "SELECT * FROM clients WHERE FirstName=(?)";
                        String searchClientOrdersSql = "SELECT * FROM orders WHERE ClientID=(?)";
                        System.out.println("Введіть ім`я клієнта:");
                        clientFirstName = scanner.nextLine();
                        PreparedStatement statement4 = conn.prepareStatement(searchClientIdSql);
                        statement4.setString( parameterIndex: 1, clientFirstName);
                        ResultSet resultSet1 = statement4.executeQuery();
                        clientId = 0;
                        while(resultSet1.next())
```

Рисунок 8.6 – Друга частина додатку



```
                case 4:
                    scanner = new Scanner(System.in);
                    String searchClientIdSql = "SELECT * FROM clients WHERE FirstName=(?)";
                    String searchClientOrdersSql = "SELECT * FROM orders WHERE ClientID=(?)";
                    System.out.println("Введіть ім'я клієнта:");
                    clientFirstName = scanner.nextLine();
                    PreparedStatement statement4 = conn.prepareStatement(searchClientIdSql);
                    statement4.setString( parameterIndex: 1, clientFirstName);
                    ResultSet resultSet1 = statement4.executeQuery();
                    clientId = 0;
                    while(resultSet1.next())
                        clientId = resultSet1.getInt( columnIndex: 1);
                    PreparedStatement statement5 = conn.prepareStatement(searchClientOrdersSql);
                    statement4.setInt( parameterIndex: 1, clientId);
                    ResultSet resultSet2 = statement5.executeQuery();
                    System.out.printf("Виводимо замовлення клієнта %s\n", clientFirstName);
                    while (resultSet2.next()){
                        String date = resultSet2.getString( columnIndex: 2);
                        status = resultSet2.getString( columnIndex: 3);
                        price = resultSet2.getInt( columnIndex: 4);
                        System.out.printf("Дата замовлення %s, статус замовлення %s, ціна замовлення %d\n", date, status, price);
                    }
                    break;
                case 5:
                    System.out.println("Завершення роботи.");
                    return;
                default:
                    System.out.println("Неправильний вибір.");
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Рисунок 8.7 – Третя частина додатку

Вище наведено приклад виконання завдання, хотілося б зауважити, що даний варіант не містить в собі обробку винятків та обробку усіх можливих варіантів взаємодії з БД.

Контрольні питання

1. Як визначається термін "база даних"?
2. Якова природа системи управління базами даних?
3. Які конкретні системи управління базами даних вам відомі?
4. Як ви визначаєте технологію JDBC?
5. Розкажіть про алгоритм підключення до СУБД використовуючи мову Java.
6. Охарактеризуйте клас PreparedStatement. Яка відмінність PreparedStatement від Statement.
7. Яким чином можна виконати SQL запити в Java?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Naughton P. Java 2: The Complete Reference, Third Edition 3rd Edition / P. Naughton, H. Schildt . – Mcgraw-Hill Osborne Media, 1999. – 1008 p.
2. Mak G. Spring Enterprise Recipes: A Problem-Solution Approach / G. Mak, J. Long. - 2nd ed. – NY : Springer, 2009. – 1104 p.
3. Mike Keith, ol. Pro JPA 2. Mastering the Java™ Persistence API / Mike Keith, Merrick Schincari. – New York : Apress, 2009. – 238 p.
4. Calvert K. L. TCP/IP Sockets in Java Practical Guide for Programmers / K. L. Calvert, M. J. Donahoo. – 2nd ed. – Burlington : Morgan Kaufmann, 2007. – 193 p.
5. Heffelfinger D. Java EE 6 with GlassFish 3 Application Server / D. Heffelfinge. – Packt Pub Ltd, 2010. – 469 p.
6. Eckel Bruce Thinking in Java / Bruce Eckel. – 4 edition. - Prentice Hall, 2006. – 1150 p.
7. Schildt Herbert Java a Beginner's Guide 5/E (Beginner's Guide) / Herbert Schildt . – 5 edition. - McGraw-Hill Osborne Media, 2011. – 640 p.
8. Cadenhead Rogers Java 6 in 21 Days / Rogers Cadenhead, Laura Lemay. – Indiana : Sams Publishing, 2007. – 698 p.
9. Sierra Kathy Head First Java [Kindle Edition] / Kathy Sierra, Bert Bates. – 2 edition. - O'Reilly Media, 2012. – 688 p.
10. Linwood J. Beginning Hibernate / J. Linwood, D. Minter. - Second Edition. – NY : Apress, 2010. – 401 p.

ДОДАТОК А. ВИМОГИ ДО ОФОРМЛЕННЯ ЗВІТУ

Текст оформлюється у наступному стилі:

- шрифт Times New Roman текстового редактора Word;
- кегль – 14 п.;
- полуторний міжрядковий інтервал;
- вирівнювання по ширині;
- абзацний відступ 1,25 см.

В звіті повинно бути приведено:

- Титульний лист;
- Мета роботи;
- Номер варіанту та умови завдання;
- Програмний код з коментарями;
- Скріни роботи програми;
- Висновки.