МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ФАХОВИЙ КОЛЕДЖ ПРОМИСЛОВОЇ АВТОМАТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
ОДЕСЬКОГО НАЦІОНАЛЬНОГО ТЕХНОЛОГІЧНОГО УНІВЕРСИТЕТУ

ШВЕЦЬ Н.В.

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Конспект лекцій

Одеса 2022 р.

Обговорено та рекомендовано до видання цикловою комісією «Комп'ютерних наук та інженерії програмного забезпечення» ФКПАІТ ОНТУ відповідно до освітньо-професійної програми та структурно-логічної схеми підготовки фахового молодшого бакалавра зі *спеціальності 121 «Інженерія програмного забезпечення»* галузі знань 12 «Інженерія програмного забезпечення»

УКЛАДАЧ : викладач Фахового коледжу промислової автоматики та інформаційних технологій ОНТУ Наталія ШВЕЦЬ.

Обговорено та рекомендовано до видання цикловою комісією «Комп'ютерних наук та інженерії програмного забезпечення» ФКПАІТ ОНТУ

"___" _____ 2022 р. Протокол № ___

Зміст

Тема 1 Переваги і особливості мови програмування Java 5
1.1 Причини і тенденції розвитку мов програмування 5
1.2 Загальна характеристика мови Java 5
1.3 Організація типового середовища Java 10
Тема 2 Основні елементи мови Java 13
2.1 Найпростіші конструкції мови 13
2.2 Типи даних 16
2.3 Змінні 19
2.4 Область видимості змінних 20
2.5 Операції мови Java 21
2.6 Пріоритет і асоціативність 23
2.7 Операції і типи даних 23
Тема 3 Оператори мови 24
3.1 Організація розгалужень в програмі 24
3.2 Організація циклів 27
3.3 Оператори переходу 33
Тема 4 Масиви 36
4.1 Одновимірні масиви 36
4.2 Багатовимірні масиви 37
Тема 5 Класи та об’єкти 38
5.1 Структура класу 38
5.2 Дані і методи 39
5.3 Об’єкти класу 39
5.4 Список аргументів змінної довжини 40
5.5 Вкладені і внутрішні класи 43
5.6 Математичні функції 46
Тема 6 Рядки (клас String) 47
6.1 Оголошення і ініціалізація рядкової змінної. Операції над рядками 47
Тема 7. Структура програми мовою Java. Основи введення/виведення 49
7.1 Структура програми 49
7.2 Основи введення-виведення. Клас Scanner 50
Тема 8 Застосування командного рядка для запуска програм. JAR-файл 55
8.1 Застосування командного рядка 56
8.2 JAR-файл 58
Тема 9 Конструктори. Об’єкти як параметри. Рекурсія 61
9.1 Конструктори 63
9.2 Клас Stack 65
9.3 Перевантаження методів 66
9.4 Об’єкти як параметри 67
9.5 Передача аргументів за значенням і за посиланням 68
9.6 Рекурсія 68
9.7 Об’єкти як значення, що повертаються 69
9.8 Привласнення посилань 71
9.9 Приховування змінної екземпляра 72

Тема 10 Інкапсуляція. Обмеження доступу до елементів класу 72
10.1 Специфікатори доступу 72
10.2 Статичні дані та методи 74
10.3 Опис констант 76
10.4 Знищення об'єкта 77
Тема 11 Успадкування 77
11.1 Суперклас і підклас. Доступ до елементів класу при успадкуванні 78
11.2 Повторний розгляд масивів 79
11.3 Подальший розвиток класу Goods. Перекриття методів 80
11.4 Сумісність об'єктів в ієрархії успадкування 81
11.5 Порядок виклику конструкторів в ієрархії класів 83
Тема 12 Динамічне зв'язування. Поліморфізм. Абстрактні класи 84
12.1 Динамічне зв'язування 85
12.2 Поліморфізм 86
12.3 Абстрактні класи 86
12.4. Використання ключового слова final для запобігання спадкуванню 90
12.5 Клас Object 91
12.6 Методи valueOf() і toString() 92
Тема 13 Інтерфейси 93
13.1 Оголошення інтерфейсу 93
13.2 Реалізація інтерфейсу 94
13.3 Посилання на інтерфейс 97
13.4 Реалізація в одному класі декількох інтерфейсів 97
13.5 Спадкоємство інтерфейсів 99
13.6 Змінні інтерфейсу 100
Тема 14 Пакети 102
14.1 Призначення пакетів. Доступ до елементів класів 102
14.2 Визначення пакета 103
14.3 Оператор import 103
Тема 15 Обробка виняткових ситуацій 106
15.1 Класифікація виняткових ситуацій 107
15.2 Принцип обробки виняткових ситуацій. Оператори try і catch 109
15.3 Вкладені оператори try 113
15.4 Генерація виняткової ситуації 115
15.5 Створення власних класів виняткових ситуацій 116
15.6 Оголошення виняткових ситуацій, які може згенерувати метод. Оператори throws та finally 117
Тема 16 Каркас колекцій Collections Framework 121
16.1 Інтерфейси колекцій 122
16.2 Класи колекцій 125
ЛІТЕРАТУРА 127

Тема 1 Переваги і особливості мови програмування Java

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

1.1 Причини і тенденції розвитку мов програмування

Модернізація і розробка нових комп'ютерних мов обумовлені двома основними причинами:

- необхідністю адаптації до постійно змінюваних середовищ і галузей застосування;
- необхідністю реалізації поліпшень і удосконалень в області програмування.

Початок розробки мови Java було покладено в 1991 р. Джеймсом Гослінгом, Патріком Ноутоном, Крісом Вартом, Едом Франком і Майком Шериданом, які працювали в компанії Sun Microsystems, Inc. Спочатку мова отримала назву "Oak" ("Дуб"), але в 1995 р була перейменована в "Java".

Первинною причиною, що спонукала до створення мови Java, з'явилася потреба в незалежній від платформи (архітектурно нейтральній) мові, яку можна було б використовувати для створення програмного забезпечення, вбудованого в різні побутові електронні пристрої. В якості контролерів в таких системах використовується безліч різних типів процесорів. Проблема застосування існуючих на той момент мов, таких, наприклад, як C і C ++, полягала в тому, що написані на них програми повинні компілюватися для конкретної платформи, тобто потрібна наявність компілятора, призначеного для даного процесора. Для кожної конкретної архітектури розроблене програмне забезпечення доводилося компілювати заново. Але, як відомо, створення компілятора коштує дорого і вимагає значного часу.

Приблизно в той же час, коли визначилися основні характеристики мови Java, на сцену виступив другий, безсумнівно, більш важливий фактор, який повинен був зіграти вирішальну роль у долі цієї мови - World Wide Web. Якби формування Web не відбувалося майже одночасно з реалізацією Java, ця мова могла б залишитися корисною, але не поміченою мовою програмування побутових електронних пристроїв. Але з появою World Wide Web мова Java вийшла на передній рубіж проєктування комп'ютерних мов, оскільки веб також потребував програм, які можуть працювати на різних платформах. Розуміння цієї обставини змусило розробників мови Java перенести свою увагу з побутової електроніки на програмування для інтернету.

1.2 Загальна характеристика мови Java

Для розуміння того, чому мова Java так стрімко поширюється в середовищі програмістів, необхідно познайомитися з основними особливостями цієї мови.

В основі синтаксису Java лежить синтаксис мови C++. Однак для його практичного використання фахівцям необхідно розібратися в конструкціях і принципах програмування на мові Java.

Характерними особливостями Java є:

- простота;
- незалежність від архітектури;
- безпека;
- об'єктно-орієнтована організація;
- переносимість;
- розподіленість;
- висока продуктивність;
- багатопоточність;
- динамічність;
- це транслюючий інтерпретатор;
- стійкість до помилок.

Простота

Java *не використовує в явному вигляді покажчиків і арифметику покажчиків*, що є джерелом безлічі помилок при програмуванні на мові C++. Справа в тому, що за допомогою покажчиків можна обійти будь-яку систему безпеки.

Весь код в програмах інкапсулюється в одному або декількох класах.

Java не має таких понять, як глобальні змінні і глобальні функції. Не має деструкторів.

Неможливо оголосити беззнакові цілі.

Не має оператора delete.

В Java об'єкти передаються тільки за посиланням - немає передачі по значенню.

У мові Java немає структур, об'єднань, перевизначення операторів. При всьому при тому програміст не відчуває нестачі в засобах програмування.

Байт-код.Поняття: JVM, JRE, JDK

Віртуальна машина — модель обчислювальної машини, створеної шляхом віртуалізації обчислювальних ресурсів: процесора, оперативної пам'яті, пристроїв зберігання та вводу і виводу інформації. В даному випадку під віртуалізацією слід вважати програмну реалізацію машини (комп'ютера), яка виконує програми подібно до справжньої машини. Віртуальна машина, на

відміну від програми емуляції конкретного пристрою, забезпечує повну емуляцію фізичної машини чи середовища виконання (для програми).

Віртуальні машини поділяються на дві головні категорії, в залежності від їх використання та відповідності до реальної апаратури:

- системні (апаратні) віртуальні машини, що забезпечують повноцінну емуляцію всієї апаратної платформи і відповідно підтримують виконання операційної системи.
- прикладні віртуальні машини, які розробленні для виконання лише застосувань (прикладних програм), наприклад — віртуальна машина Java (JVM).

Прикладні віртуальні машини виконують звичайні програми всередині ОС. Вони зазвичай створюються коли програма запускається та знищуються після її завершення. Їхня ціль — забезпечити платформно-незалежне програмне середовище, яке дозволяє абстрагуватися від конкретної апаратури та операційної системи, на якій виконується програма.

JRE (Java Runtime Environment) - це мінімальна реалізація віртуальної машини, необхідна для виконання Java-застосувань, без компілятора та інших засобів розробки. Складається з віртуальної машини JVM та бібліотек Java-класів.

JDK (Java Development Kit) - це комплект розробника застосувань на мові Java, що включає компілятор, стандартні бібліотеки класів Java, приклади, документацію, різні утиліти і виконавчу систему JRE

Незалежність від архітектури

Транслятор мови Java генерує команди байт-коду (іноді також використовується термін псевдокод - машинно-незалежний код низького рівня, що генерується транслятором і виконується інтерпретатором). Більшість інструкцій байт-коду еквівалентні одній або кільком командам асемблера. Трансляція в байт-код займає проміжне положення між компіляцією в машинний код і інтерпретацією.

Програма на байт-коді зазвичай виконується інтерпретатором байт-коду — віртуальною машиною Java JVM. Віртуальна машина Java — віртуальна машина для виконання байт-коду Java. JVM нічого не знає про мову Java, вона просто вміє працювати з файлами формату .class, що містять інструкції для віртуальної машини Java та додаткову інформацію.

Байт-код - це компактне представлення програми, котра пройшла синтаксичний і семантичний аналіз. У ньому в явному вигляді закодовані типи, області видимості і інші конструкції. Кожна інструкція являє собою однобайтовий код операції від 0 до 255, за яким слідують такі параметри як регістри або адреси

пам'яті. Це в типовому випадку, але специфікація байт-коду істотно відрізняється в мові.

Перевага - в портіруємості, тобто один і той же байт-код може виконуватися на різних платформах і архітектурах. Оскільки байт-код зазвичай менш абстрактний, компактніший і більш «комп'ютерний» ніж початковий код, ефективність байт-коду зазвичай вище, ніж чиста інтерпретація початкового коду, призначеного для редагування людиною**.**

JVM обробляє байтовий код і передає інструкції обладнанню, як інтерпретатор, але з тією різницею, що байт-код обробляється значно швидше.

Гідність подібного способу виконання програм - в повній незалежності байт-коду від ОС і устаткування, що дозволяє виконувати Java-програми на будь-якому пристрої, який підтримує віртуальну машину.

Безпека

Іншою важливою особливістю технології Java є гнучка система безпеки. Завдяки тому, що виконання програми повністю контролюється JVM, будь-які операції, які перевищують встановлені повноваження програми (наприклад, спроба несанкціонованого доступу до даних або з'єднання з іншим комп'ютером) викликають негайне переривання. Це дозволяє користувачам завантажувати програми, написані на Java, на їх комп'ютери (або інші пристрої, наприклад, мобільні телефони) з невідомих джерел, при цьому не побоюючись зараження вірусами, зникнення цінної інформації і т.п.

Об’єктно-орієнтована організація

Java - повністю об'єктно-орієнтована мова. На відміну від C++ або Object Pascal перед розробниками Java не стояло завдання забезпечити сумісність з вихідним кодом інших мов, що дозволило повністю витримати принципи об'єктної організації мови. Разом з тим для підвищення швидкості обробки даних залишена можливість працювати з простими типами не як з об'єктами.

Накопичений досвід використання інших мов програмування дозволив скасувати множинне успадкування і віртуальні класи, які викликали проблеми з ієрархією об'єктів. У Java знайдено альтернативне рішення в вигляді інтерфейсів.

Переносимість

У Java фіксований розмір числових типів, що дозволяє уникати неприємностей, пов'язаних з виконанням програм на різних комп'ютерах.

Сучасний інструментальний набір Java для створення графічного інтерфейсу користувача практично не залежить від платформи.

Розподіленість

Java має велику бібліотеку програм, які підтримують передачу даних на базі протоколів TCP/IP. Зв'язок між розподіленими об'єктами забезпечується засобами виклику віддалених методів.

Висока продуктивність

Якщо для виконання байт-коду використовується інтерпретатор, то говорити про високу продуктивність не доводиться. Але при наявності синхронного компілятора здійснюється трансляція байт-коду в машиннозалежний код, який зберігається в пам'яті і використовується далі при необхідності. Синхронні компілятори працюють повільніше ніж традиційні машиннозалежні компілятори, але їхня технологія постійно вдосконалюється і вони мають потенційну можливість досягти і навіть перевищити характеристики звичайних компіляторів.

Багатопоточність

Мова Java підтримує багатопотокове програмування, забезпечуючи створення програм, що виконують одночасно багато потоків (якщо це дозволяє зробити операційна система). Ефективне вирішення проблеми багатопотокової синхронізації дозволяє позбавити програміста від багатьох клопотів, пов’язаних з управлінням багатозадачністю.

Динамічність

Мова Java, краще ніж багато інших мов, адаптується до середовища, що змінюється. Це виявляється в простоті додавання нових методів і об’єктів у бібліотеки і в тому, що Java-програми містять значний об’єм інформації про поведінку об’єктів у процесі виконання програми, що забезпечує безпечну і ефективну динамічну компоновку.

Інтерпретатор

Інтерпретатор мови Java може пересилатись на будь-яку машину і виконувати байт-код безпосередньо на цій машині. Це дає істотну перевагу при розробці програм користувачів та їхньому супроводі.

Стійкість до помилок

Мова Java належить до мов, що строго типізуються. Початковий код перевіряється на відповідність типів на етапі трансляції. Передбаченість поведінки написаного коду – одна з основних переваг мови Java. Багатьох помилок часу виконання вдається уникнути завдяки автоматичному виділенню і звільненню пам’яті, а також засобам об’єктно-орієнтованої обробки виняткових ситуацій.

1.3 Організація типового середовища Java

SDK - software development kit - комплект засобів розробки, який дозволяє фахівцям з програмного забезпечення створювати застосування. Постачальники SDK іноді підміняють слово software в словосполученні "software development kit" на більш точне слово. Наприклад, Oracle називає свій інструментарій Java Development Kit.

Пакет Java Development Kit (JDK)

Java Development Kit є основним компонентом середовища Java і надає всі інструменти, виконувані та бінарні файли, які потрібні для компіляції, налагодження та виконання програми Java. JDK є платформо-залежним програмним забезпеченням, тому є окремі інсталятори для Windows, Mac та Unix-систем. Можна сказати, що JDK є надбудовою JRE, оскільки він містить JRE з Java-компілятором, налагоджувачем та базовими класами.

Віртуальна машина Java (JVM)

JVM є серцем мови програмування Java. Коли ми запускаємо програму, JVM несе відповідальність за перетворення байт-коду в машинний код. JVM також залежить від платформи та надає основні функції, такі як управління пам'яттю Java, складання сміття, і т.д. Ми також можемо виділити певний обсяг пам'яті для JVM. JVM є віртуальною машиною, тому що забезпечує інтерфейс, який не залежить від операційної системи та апаратних засобів. Ця незалежність від апаратного забезпечення та операційної системи дає Java-програмам можливість виконуватися на будь-якому пристрої без необхідності внесення змін – «Write once, run anywhere» (Напиши раз - запускай де завгодно).

Java Runtime Environment (JRE)

JRE є реалізацією JVM, яка надає платформу для виконання Java програм. JRE складається з віртуальної машини Java, бінарних файлів та інших класів. JRE не містить інструментів для розробки (компілятор Java, відладчик і т.д.). Якщо ви хочете запустити будь-яку програму Java, ви повинні встановити JRE.

Just-in-time Compiler (JIT)

Just-in-time Compiler (JIT) є частиною JVM. Він оптимізує байт-код, зменшуючи загальний час, необхідний компіляції байт-коду в машинний код.

До складу JDK не входить інтегроване середовище розробки на Java, тому розробник, що використовує тільки JDK, змушений використовувати зовнішній текстовий редактор і компілювати свої програми, використовуючи утиліти командного рядка.

Всі сучасні інтегровані середовища розробки застосувань на Java спираються на послуги, що надаються JDK. Більшість з них для компіляції Java-програм використовують компілятор з комплекта JDK. Тому ці середовища розробки або включають в комплект поставки одну з версій JDK, або вимагають для своєї роботи попередньої інсталяції JDK на машині розробника.

Кращі IDE для Java це :

• краща безкоштовна IDE: NetBeans (повний опис, налогодження - https://stfalcon.com/ru/blog/post/netbeans--tips-and-tricks);
• краща комерційна IDE: IntellijJ IDEA;
• найпопулярніша IDE: Eclipse;
• універсальна IDE: JDeveloper;
• краща для Android: Android Studio.

Системи Java складаються з декількох компонентів:
• середовища;
• мови;
• програмного інтерфейсу додатків Java;
• різних бібліотек класів.

Послідовність розробки та виконання Java-програми схематично можна представити як на рисунку 1.1.

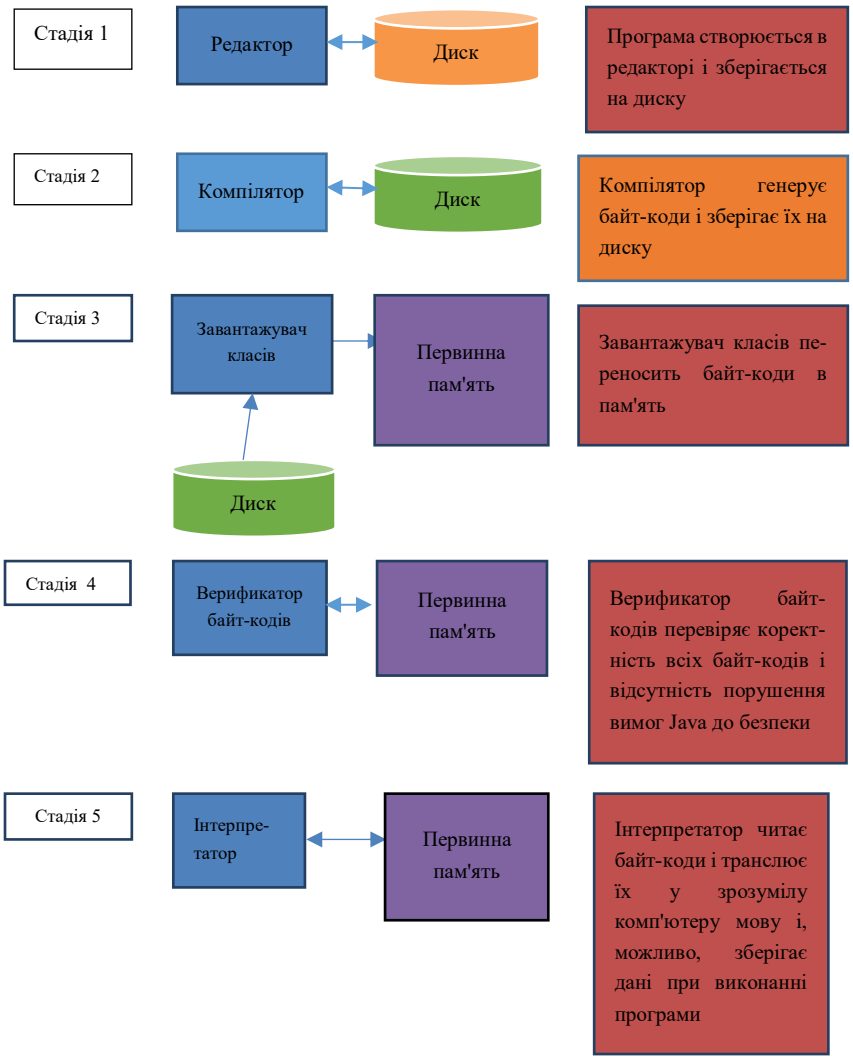


Рисунок 1.1 — Типове середовище розробки Java

Процес створення Java-програм включає в себе п'ять стадій:

- редагування;
- компіляцію;
- завантаження;
- верифікацію;
- виконання.

Сьогодні для доставки застосувань на клієнтський комп’ютер пропонується інша технологія – Java Web Start

Імена програмних файлів (вихідний код<=>початковий код) мають розширення .java.

На стадії компіляції програми програміст вводить в командному рядку команду javac. Компілятор Java транслює програму в байт-коди - мову, що розуміється інтерпретатором Java. Для компіляції програми з ім'ям welcome.java:

javac welcome.java

У разі успішної компіляції програми буде створено файл welcome.class. Саме в цьому файлі містяться байт-коди, які будуть інтерпретуватися на стадії виконання програми.

Стадія 3 називається завантаженням. Щоб програму Java можна було виконати, її необхідно спочатку помістити в оперативну пам'ять комп'ютера. Це робить завантажувач класів, котрий бере файл (або файли) з розширенням .class, в якому містяться байт-коди, і переносить його в пам'ять. Файл з розширенням .class може бути завантажений як з локального диска, так і по мережі - локальної або інтернет.

Застосування завантажуються в пам'ять і виконуються інтерпретатором командного рядка java-командою

java welcome

Ця команда ініціалізує інтерпретатор для застосування welcome і дає завдання завантажувачу класів завантажити інформацію, використовувану програмою welcome.

Усередині Java існує кілька основних сімейств технологій:

Java SE — Java Standard Edition, основне видання Java, містить компілятори, API, Java Runtime Environment; підходить для створення корпоративних застосувань, в першу чергу - для настільних систем.

Java EE — Java Enterprise Edition, є набір специфікацій до створення програмного забезпечення рівня підприємства. У 2017-му проект Java EE було передано Eclipse Foundation, після чого було перейменовано в Jakarta EE. Модулі Java EE видалені з Java SE, починаючи з 11-ї версії.

Java ME — Java Micro Edition, створена для використання в пристроях, обмежених за обчислювальною потужністю, наприклад, в мобільних телефонах, вбудованих системах;

Java Card — технологія надає безпечне середовище для застосувань, що працюють на смарт-картах та інших пристроях з дуже обмеженим об'ємом пам'яті та можливостями обробки.

JIT-компіляція (англ. Just-in-Time, компіляція «точно в потрібний час»), динамічна компіляція (англ. dynamic translation) - технологія збільшення продуктивності програмних систем, що використовують байт-код, шляхом компіляції байт-коду в машинний код або інший формат *безпосередньо під час роботи програми*. Таким чином досягається висока швидкість виконання порівняно з інтерпретованим байт-кодом (порівняючи з компілюваними мовами) за рахунок збільшення споживання пам'яті (для зберігання результатів компіляції) та витрат часу на компіляцію.

Тема 2 Основні елементи мови Java

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

2.1 Найпростіші конструкції мови

2.1.1 Ідентифікатори

Ідентифікатори визначають імена змінних, методів і класів. Ідентифікатор *є послідовністю букв і цифр, яка починається з букви*. Як букву в ідентифікаторах також можна використовувати знак долара і підкреслення. Мова розрізняє регістр букв.

Приклади допустимих ідентифікаторів: Set_of_char, y1, $diff, SysAdmin.

Приклади недопустимих ідентифікаторів: 1index, Alpha-Romeo, max/min.

Ідентифікатор не повинен починатися з цифри, щоб компілятор не переплутав його з числової константою.

2.1.2 Ключові слова

Ключовим словом називається ідентифікатор, який має для мови програмування відоме значення. У мові Java ключові слова є зарезервованими – програміст не може їх використовувати інакше, ніж це визначено мовою. Приклад ключових слів:

byte, class do, else, interface, switch.

Ключові слова const і goto зарезервовані, але не використовуються.

2.1.3 Роздільники

Роздільники – це символи, які дозволяють встановити межі для конструкцій мови.

Символи роздільників та випадки їхнього застосування наведені нижче.

Таблиця 2. 1 — Роздільники мови Java

Символ	Застосування
( )	У виразах для установки пріоритету обчислень, як межи списку параметрів методів, в операціях приведення типу
{ }	Для встановлення меж опису класів, методів, блоків коду, списків ініціалізації масивів
[]	Для оголошення типів масивів і запису індексів елементів масиву
,	Для розділення елементів в списках
;	Вказує на закінчення оператора
.	Відділення членів класів від імен класів

2.1.4 Коментарі

Коментар – це частина тексту програми, яка не обробляється компілятором (інтерпретатором) з метою створення коду. Коментарі складаються програмістами для програмістів.

У мові Java існують три типи коментарів:

- однорядкові – починаються з символів «//» і розповсюджуються до кінця рядка;
- багаторядкові – починаються з символів «/*» і закінчуються символами «*/»;
- документуючи коментарі - починаються символами «/**» і закінчуються символами «*/».

За допомогою документуючих коментарів у програму можна вставляти відомості власне про програму, про її автора, версію програми, про класи, про значення, що повертаються методами, посилання на інше джерело. За допомогою утиліти javadoc цю інформацію можна витягнути із програми і помістити у файл формату HTML. Таким чином, використовуючи документуючи коментарі при написанні тексту програми, можна одночасно готувати документацію до даної програми.

2.1.5 Пропуски

Програми на мові Java записуються у довільному форматі. В одному рядку можна записати декілька операторів або тільки один. Але для зручності читання і відладки програми не рекомендується в одному рядку розміщувати більш од-

ного оператора, а фігурні дужки, що обмежують класи, методи або блоки, розміщувати в окремих рядках. Наявність пропусків обов'язкова між елементарними конструкціями мови (лексемами), якщо вони не розділені роздільниками.

У мові Java пропусками є символи пробілу, табуляції і нового рядка.

2.1.6 Літерали

У Java постійне значення задається його літеральним поданням. Ось, наприклад, кілька литералів: 100 98.6 'X' "This is a test".

Перший літерал вказує цілочисельне значення, другий - значення з плаваючою точкою, третій - символьну константу, останній - строкове значення.

Літерал може використовуватися скрізь, де допустимо використання значень даного типу.

2.1.7 Цілочисельні літерали

Можна визначати цілочисельні літерали, використовуючи двійкову форму. Для цього перед значенням використовується префікс 0b або 0B. Наприклад, наступний код визначає цілочисельне значення десяткове 10 з використанням двійкового литерала:

```
int x = 0b1010;
```

Наявність двійкових литералів полегшує також введення значень, використовуваних як бітові маски. В такому випадку десяткове (або шістнадцяткове) уявлення значень числа не відображує візуально спосіб його використання, а двійковий літерал відображає.

Можна також впровадити в цілочисельний літерал один або кілька символів підкреслення. Це полегшує читання великих цілочисельних літералів. При компіляції символи підкреслення в цілочисельному літералі ігноруються. Наприклад:

```
int x = 123_456_789;
```

Значенням змінної x буде 123456789. Символи підкреслення будуть проігноровані. Символи підкреслення можуть використовуватися тільки для поділу цифр. Вони не можуть розташовуватися на початку або в кінці литерала, але цілком допустимо використання між двома цифрами декількох символів підкреслення. Наприклад:

```
int x = 123_456 ___ 789;
```

Використання символів підкреслення в цілочисельному літералі особливо корисно при написанні в коді таких елементів, як номери телефонів і т.п. Вони також корисні для візуальних угрупувань при визначенні двійкових литералів. Наприклад, двійкові значення часто візуально групуються в блоки по чотири цифри, наприклад:

```
int x = 0b1101_0101_001_1010;
```

2.1.8 Літерали з плаваючою точкою

Можна впровадити в літерал з плаваючою точкою один або кілька символів підкреслення. Це працює так само, як і для цілочисельних літералів. Наприклад:

double num = 9_423_497_862.0;

Значненням num буде 9432497862.0. Символи підкреслення полегшують читання великих литералів з плаваючою точкою.

Можна використовувати спеціальні символи підкреслення в дробовій частини числа:

double num = 9_423_497.1_0_9;

В даному випадку дрібна частина дорівнює .109

. 2.1.9 Булеві літерали

Тип boolean може мати тільки два значення: true і false. Ці значення не перетворюються в жодні числові уявлення. В Java true НЕ дорівнює 1, а false - 0. В Java логічні літерали можуть бути привласнені лише тим змінним, які оголошені як boolean або використовуватися у виразах з булевими операціями.

2.2 Типи даних

Тип даних визначає необхідне місце в пам’яті для запису цього даного, формат його представлення а також множину операцій, які можуть виконуватись над даним.

Java –типизована мова. Це означає, що:

- кожна змінна має тип і кожен тип строго визначений;
- всі привласнення - явні, або виконувані при передані параметрів у процесі виклику методів, перевіряються на відповідність типів.

Типи даних прийнято підрозділяти на:

- прості (елементарні), які визначені в мові Java;
- типи, визначені користувачем (сюди належать і типи (класи), описані в бібліотеці Java).

Прості типи підрозділяються на чотири групи:

- цілочислові;
- дійсні;
- символьні;
- логічні.

Прості типи не є класами і є винятком в об‘єктно-орієнтованій мові Java. Вони аналогічні простим типам в інших необ’єктно-орієнтованих мовах програмування. Розробники мови пішли на таке рішення з метою забезпечення високої продуктивності виконання операцій над даними.

2.2.1 Цілочислові типи

У Java визначені чотири цілочислові типи, використовувані для значень зі знаком (неможливо оголосити беззнакові цілі):

- byte;
- short;
- int;
- long.

Мови C і C++ допускають варіювання розмірів цілочисельних змінних в залежності від вимог середовища виконання. Однак в Java вони мають строго певниий діапазон значень. Наприклад, незалежно від конкретної платформи, значення типу int завжди є 32-бітовими.

У таблиці 2.2 приведені характеристики цілочислових типів.

Таблиця 2.2 — Цілочислові типи даних

Тип	Розмір в байтах	Діапазон значень
byte	1	від -128 до 127
short	2	від -32768 до 32767
int	4	від -2147483648 до 2147483648
long	8	від -92233720368547758008 до 92233720368547758008

Тип byte корисний при проєктуванні потоків даних (у файлах), коли конкретний тип може бути невідомий.

Тип short не рекомендується використовувати крім тих випадків, коли дане має займати точно два байти. Це визначається тим, що 16-розрядні комп'ютери практично вийшли з вживання.

Тип int є тим, що часто застосовується. Його рекомендується використовувати для організації циклів і індексації навіть тоді, коли необхідний діапазон значень значно менше, оскільки велика ймовірність саме 32-розрядного представлення чисел при реалізації програми.

Тип long використовується в тих випадках, коли типу int недостатньо для представлення деякого даного.

Константу цілого типу можна записати як:

- десяткову;
- вісімкову;
- шістнадцяткову.

Вісімкова константа починається з цифри 0 і не може містити цифр 8 або 9. Наприклад:

012, 0567.

Шістнадцяткова константа починається з комбінації символів 0x або 0X і може містити крім десяткових цифр букви A, B, C, D, E, F, що інтерпретуються в десятковій системі счислення як числа 10, 11, 12, 13, 14, 15 відповідно. Наприклад:

0x1010, 0xA9C.

Якщо записується константа типу long, то завжди в кінці числа слід записати символ L. Наприклад:

987654321L.

2.2.2 Дійсні типи

Дійсні типи використовуються тоді, коли потрібно записати число з дробовою частиною, або дуже велике число. У Java визначені два дійсних типи – double і float, характеристики яких наведені в табл. 2.3.

Таблиця 2.3 — Дійсні типи даних

Тип	Розмір в байтах	Діапазон значень
float	4	від -3.4e-38 до 3.4 e+38 (6-7 значущих цифр)
double	8	від -1.7e-308 до 1.7e+308 (15 значущих десяткових цифр)

Тип float дозволяє економити пам’ть порівняно з типом double. Що стосується швидкості обчислень, то тут все залежить від середовища реалізації програми. Оскільки всі трансцендентні математичні функції (sin(), exp(), sqrt() та ін.) повертають значення типу double, на практиці частіше використовують тип double.

Константу дійсного типу можна записувати так:

3.14; 2.5e-2; 3e-7; 0.0001; .5.

2.2.3 Символьний тип

У мові Java є один символьний тип, який представляє міжнародний символьний набір Unicode, що дозволяє використовувати будь-які мови світу (див. табл. 2.4):

Таблиця 2.4 — Символьний тип

Тип	Розмір в байтах	Діапазон значень
char	2	від 0 до 65535

Стандартний набір символів ASCII займає діапазон від 0 до 127, а розширений 8-бітовий символьний набір (ISO-Latin-1) знаходиться в межах від 0 до 255.

Для представлення символьної константи можна використовувати як символ, так і код. Наприклад, латинську букву “A” можна представити як ‘A’, або її кодом 65.

Символи, які не мають графічного відображення, можна представити Escape-послідовністю (див. табл. 2.5).

Таблиця 2.5 — Escape-послідовності

Escape-послідовність	Опис
\r	Повернення каретки
\n	Перехід на новий рядок
\f	Перехід но нову сторінку
\t	Горизонтальна табуляція
\b	Повернення на один символ

2.2.4 Логічний тип

Логічний тип має всього два значення: true і false. Призначення логічного типу – організація розгалужень у програмі. Логічні значення утворюються в результаті виконання операцій відношення.

2.3 Змінні

Дані, використовувані в програмі, можуть бути незмінними (константами), або такими, що змінюються в процесі виконання програми (змінні). Кожне дане має бути віднесено до певного типу. Якщо тип константи можна встановити за її значенням, то кожна змінна вимагає опису.

Опис змінної має відповідати такому правилу:

тип ідентифікатор [[=значення], ідентифікатор [=значення]…];

Наприклад:

```
byte a;
short b,k;
int d=5, e, f=-1;
long l;
float fl;
double d1=0.5, d2;
boolean bl=false;
char ch, ch1='*';
```

Мова Java дозволяє ініціалізацію змінних з використанням раніше оголошених змінних (динамічна ініціалізація). Наприклад:

```
int i=2, j=3;
…
int k=i*j;
```

Для описів змінної в програмі немає певного місця. Єдина вимога – опис має передувати її використовуванню.

2.4 Область видимості змінних

Кожна змінна оголошується усередині блока, який і визначає її область видимості. Блок починається з відкриваючої фігурної дужки і закінчується закриваючою фігурною дужкою. Якщо змінна є формальним параметром методу, то вона належить блоку, що є тілом методу. Особливості області видимості змінних, оголошених усередині класу розглянемо під час вивчення класів.

Блоки можуть бути вкладеними. Відповідно і області видимості можуть бути внутрішніми і охоплюючими. Якщо під час виконання програми відбувся вихід з внутрішнього блока, то всі *оголошені* в ньому змінні більше не існують. Повторне входження в блок не дозволяє використовувати значення змінних блока, які були в момент виходу з нього.

У мові Java не можна у внутрішньому блоці перевизначати змінні зовнішнього блока.

Наведений нижче фрагмент програми ілюструє правила визначення видимості змінних.

```
{
l=200000; //Помилка, використовування змінної раніше, ніж вона оголошена
int i=5;
double d=3.14;
long l;
…
        {
        int j=i+2;
        float d; //Помилка. Перевизначення змінної d, оголошеної в зовнішньому блоці
        d=2e3;
        ...
        }
        i=j-1;//Помилка. Змінна j у зовнішньому блоці не визначена
l=50000;
}
```

2.5 Операції мови Java

Якщо вираз містить декілька операцій, то порядок виконання дій визначається пріоритетом цих операцій. Крім пріоритету операції характеризуються кількістю операндів (одномісні, двомісні, тримісні), а також асоціативністю – порядком виконання при однакових пріоритетах (зліва направо або справа наліво).

У наведеній нижче таблиці 2.6 представлені всі операції мови Java в порядку убування пріоритету.

Таблиця 2.6 — Операції мови Java

Пріоритет	Знак операції	Кількість операндів	Асоціативність	Назва	Приклади
1	[]		→	Індексні дужки	arr1 [i], arr2 [j+1]
	( )		→	Виклик методу, зміна пріоритету	sin (x), a * (b + з)
	.		->	Доступ до члена класу	MyClass.d
2	!	1	←	Логічне НЕ	!(a>b)
	~	1	←	Побітове НІ	~ a

	++ --	1	←	Приріст, зменшення на 1	i ++, j - -
					int x=9,y;
					y=++x; //x=10; y=10;
					y=x++; //x= 10; y=9;

	( )		←	Приведення типів	int a=20; byte b = (byte)a;
	+ -	1	←	Зміна знака	a=-b
	new	2	-<	Виділення місця в пам'яті	double arr1[]=new double[10];
3	*	2	→	Множення	a*b

Пріоритет	Знак операції	Кількість операндів	Асоціативність	Назва	Приклади
	/	2	→	Ділення	a/b
	%	2	→	Остача	a % b (a і b – цілі або дійсні)
4	+	2	→	Додавання	a + b
	-	2	→	Віднімання	a-b
5	<< >>	2	→	Зрушення вліво, управо	a >> n, a << n (a і n - цілі)
	>>>		→	Зрушення із заповненням 0	a >>> n (a і n - цілі)
6	< <= > >=	2	→	Операції відношення	a < b, a > b, a >= b, a <= b
	instanceof	2	->	Перевірка належності об'єкта класу	if(a instanceof MyClass), де a – об'єкт, MyClass - клас
7	== !=	2	→	Операшї відношення	a == b, a!=b
8	&	2	→	Побітове і	a & b (a і b - цілі)
9	^	2	→	Побітове, що виключає, АБО	a ^ b (a і b - цілі)
10	|	2	→	Побітове АБО	a | b (a і b - цілі)
11	&&	2	→	Логічне І	a&&b
12	||	2	→	Логічне АБО	a||b
13	?:	3	→	Операція «?»	int x=5, y=10; int z=x > y ? 5 : 6; //z=6
14	=	2	←	Привласнення	a = b

Пріоритет	Знак операції	Кількість операндів	Асоціативність	Назва	Приклади
	+= _= *= /= %= != ^= <<= >>= >>>=	2	←	Виконання бінарної операції і привласнення	a += b (аналогічно a = a + b), a /= b (аналогічно a = a / b)

2.6 Пріоритет і асоціативність

Розглянемо оператор:

y=a+b*c/(f-g);

Тут обчислення виконується у такій послідовності: операція «()» має щонайвищій пріоритет. Це означає, що перш за все буде обчислений вираз у круглих дужках. Далі буде виконана операція «*», пріоритет якої вищий, ніж у операції «+ » і однаковий з операцією «/», але остання знаходиться правіше і відповідно до асоціативності виконуватиметься пізніше. Останньою буде виконана операція «+».

Наступний оператор містить помилку, спричинену незнанням пріоритетів:

int a=9;

boolean b=!a>7;

Операція «!» має найбільший пріоритет серед використовуваних в операторі операцій, проте вона повинна мати операнд логічного типу, тоді як змінна a має числовий тип. Очевидно правильним буде запис:

boolean b=!(a>7);

2.7 Операції і типи даних

Кожна операція (крім привласнення) розрахована на певні типи даних. Так, наприклад, всі побітові операції виконуються тільки з операндами цілого типу.

Операції «+», «-», «*», «/» визначені для операндів числових типів. При цьому результат операції може бути різним залежно від поєднання типів операндів. Тип результату буде цілим, якщо обидва операнди цілі і буде дійсним при будь якому іншому поєднанні типів операндів. Так, наприклад, результат виконання ділення 4/8 буде дорівнювати 0, а результат ділення 4.0/8 буде дорівнювати 0.5.

Типи даних можуть бути сумісними і несумісними. Сучасні типи автоматично перетворюються один в інший. Так, наприклад, змінній дійсного типу завжди можна привласнити ціле значення, змінній типу int можна привласнити значення типу byte.

Наведений нижче фрагмент програми ілюструє сумісність типів:

```
byte b;
short s;
int i;
```

```
long l;
float f;
double d;
…….
s=b;
i=b; i=s;
l=b; l=s; l=i;
f=b; f=s; f=i f=l;
d=b; d=s; d=i; d=f;
```

Якщо числові типи несумісні, то є можливість виконати перетворення одного типу в іншій шляхом приведення. При цьому значення даного може змінитися.

Операція приведення має синтаксис:

(тип_призначення)значення

де тип_призначення визначає тип, у який повинне відбутися перетворення.

Наступний фрагмент програми ілюструє приведення типів:

```
byte b1, b2;
int i=20, j=260;
double d=3.2;
……..
b1=(byte)i; //b1=20
i=(int)d; //i=3
b2=(byte)j; //b2=4
```

При перетворенні в ціле дробова частина дійсного числа відкидається.

При перетворенні цілого в байт визначається остача від ділення цілого на 256.

При перетворенні double в byte, дробна частина відкидається і значення зменшується до результату ділення по модулю 256.

Тема 3 Оператори мови

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

3.1 Організація розгалужень в програмі

Усі дії в програмі виконують оператори. Найпростіший оператор – оператор присвоювання ми вже розглядали раніше у зв’язку з описом змінних.

Якщо оператори завжди виконуються в одній і тій ж послідовності, то говорять, що програма або дялянка програми лінійна. Реальні програми мають нелінійну організацію. Для створення таких програм використовуються оператори розгалужень і цикли. Найчастіше для розгалужень використовують оператори if и switch.

3.1.1 Оператор if

Синтаксис оператора if визначається таким чином

```
if(вираз)
оператор1;
[else
оператор2]
```

де обчислене значення виразу має бути логічного типу, оператор1 і оператор2 можуть бути будь-якими операторами мови Java.

Порядок обчислення оператора такий. Спочатку обчислюється значення виразу. Потім, якщо набуте значення істинно, то виконується оператор1, інакше, якщо в операторі if присутня гілка else, то виконується оператор2, якщо гілки else немає, то не виконується нічого.

Наступний фрагмент програми ілюструє обчислення Y за формулами:

```
Y=a-x, якщо x<a;
Y=a,якщо x=a;
Y=a+x,якщо x>a.
if(x<a)
     Y=a-x;
else
     if(x==a)
          Y=a;
else
          Y=a+x;
```

Ми розглянули приклад використовування оператора if, коли при виконанні умови і при невиконанні вибирався тільки один внутрішній оператор. Проте часто вибір варіанта передбачає деякий ланцюжок дій. Припустимо, необхідно при значенні X>0 надати нові значення для Y і Z за відповідними формулами. У цьому випадку потрібно буде замінити внутрішнього оператора на блок. Блок – це послідовність операторів (і описів), узята у фігурні дужки. Нижче наведений оператор if, що містить блок.

```
if(X>0)
{
     Y=2;
     Z=X-1;
}
```

3.1.2 Оператор switch

Оператор switch використовують для організації в програмі множини гілок обчислень. *Якщо умова вибору варіанта може бути представлена константою*, то оператор switch представляє розгалуження в програмі значно простіше, ніж це можна реалізувати ланцюжком операторів if – else – if.

Синтаксис оператора має такий вигляд.

switch(вираз)
{
case *константа1* : *Послідовність операторів*
case *константа2* : *Послідовність операторів*
...
case *константаN* : *Послідовність операторів*

*[**default**: Послідовність операторів]*
}

Константи повинні мати тип, сумісний з типом виразу.

Робота оператора відбувається таким чином.

Обчислене значення виразу послідовно порівнюється зі всіма константами. Якщо знайдений збіг значень в деякій гілці, то відповідна послідовність операторів виконується. У цьому випадку оператор switch не передбачає вихід за свої межі, тому виконуватимуться всі послідовності операторів, розташовані нижче. Якщо такий алгоритм не відповідає розв'язуваній задачі, є можливість в кінці послідовностей операторів записати оператор break (розглянуто пізніше), який завершує роботу оператора switch.

Якщо значення виразу не збіглося ні з однією константою, то управління передається гілці default, а при її відсутності оператор switch не виконує ніяких дій.

Наступний приклад дозволяє вибрати один з п'яти варіантів обчислення Y залежно від значення деякої змінної P.

```
switch(P)
{
case 1 : Y=x+a; break;
case 2 : Y=x+a+b; break;
case 3 : Y=x+a+b+c; break;
case 4 : Y=x+a+b+c+d; break;
default : Y=0;
}
```

Для всіх версій Java до JDK7 вираз, що стоїть в дужках після switch, повинен мати тип byte, short, int, char. Починаючи з JDK7 вираз може мати тип String, як надано в листингу 3.1:

Листинг 3.1

```
class StringSwitch
{
public static void main(String args[])
{
String str="два";
switch(str)
{
case "один" : System.out.println("один");
              break;
case "два" : System.out.println("два");
              break;
case "три" : System.out.println("три");
              break;
default: System.out.println("немає відповідностей");
}
}
}
```

Однак краще застосовувати рядки тільки в тих випадках, коли контролюючі дані вже знаходяться в строковій формі. Переважно використовувати цілі числа.

3.2 Організація циклів

Ділянки програми, які можуть багато разів повторюватися в процесі одного запуску програми, називаються циклами. У кожному циклі має бути умова його повторення. Якщо ця умова перевіряється до входу в цикл, говорять, що це цикл з передумовою, а якщо умова перевіряється в кінці циклу, то такий цикл називається циклом з постумовою. Цикл з передумовою може жодного разу не виконатися, тоді як цикл з постумовою виконується як мінімум один раз.

3.2.1 Оператор while

Оператор while є циклом з передумовою. Нижче наведений синтаксис оператора:

while(умова)
оператор

Умовою має бути булевий вираз. Внутрішній оператор циклу (тіло циклу) виконується до тих пір, поки значенням *умова* буде істина. Звичайно цикл охоплює декілька операторів, тоді їх потрібно об'єднати в блок.

Наприклад, наведений далі фрагмент програми дозволяє визначити суму всіх парних чисел від 4 до 56.

```
int S=0,i=4;
while(i<=56)
{
     S+=i;
i+=2;
}
```

Тіло циклу while (або будь-якого іншого циклу в Java) може бути порожнім. Це обумовлено тим, що синтаксис Java допускає застосування порожнього оператора (що містить тільки символ крапка з комою). Розглянемо приклад програми (див. листинг 3.2).

Листинг 3.2

```
class NoBody
{
public static void main(String args[])
{
int i,j;
i=100;
j=200;
while(++i< --j);
     System.out.println("Середнє значення дорівнює "+i);
}
}
```

Ця програма обчислює середнє значення змінних i і j.

На екрані отримаємо:

Середнє значення дорівнює 150

Як видно, ніякої потреби в наявності тіла циклу немає. Всі дії виконуються всередині самого умовного виразу. *У професійно написаному коді Java короткі цикли часто не містять тіла, якщо сам по собі управляючий вираз може виконувати всі необхідні дії.*

3.2.2 Оператор do-while

Цикл do-while є циклом з постумовою. Нижче наведений синтаксис цього оператора.

```
do
оператор
while(умова)
```

Умова має бути булевого типу. Одне виконання оператора do-while завжди забезпечене. Подальші виконання визначаються значенням умови. Якщо воно істинне, то цикл виконується. Інакше виконання циклу завершується. Аналогічно іншим операторам управління як внутрішній оператор можна використовувати блок.

Для виконання попередньої задачі відповідний фрагмент програми з застосуванням оператора do-while має такий вигляд.

```
int S=0,i=4;
do
{
S+=i;
i+=2;
}
while(i<=56);
```

Цикл do-while зручний при виборі пункту меню, оскільки зазвичай в цьому випадку потрібно щоб тіло циклу меню виповнилося, щонайменше, один раз.

Розглянемо приклад програми (див. листинг 3.3), яка реалізує просту систему довідки по операторам вибору і циклу Java.

Листинг 3.3

```
class Menu
{
public static void main(String args[])
throws java.io.IOException
{
char choice;
do
{
System.out.println("Довідка по операторам: ");
System.out.println("\t1. if");
System.out.println("\t2. switch");
System.out.println("\t3. while");
System.out.println("\t4. do-while");
System.out.println("\t5. for");
```

```
System.out.println("Оберіть пункт меню:");
choice=(char)System.in.read();
}
while(choice>='1' || choice<='5');
System.out.println("\n");
switch(choice)
{
case '1' : System.out.println("\t if:\n");
          System.out.println("if(умова) оператор;");
          System.out.println("else оператор;");
          break;
case '2' : System.out.println("\t switch:\n");
          System.out.println("switch(вираз){");
          System.out.println("case          константа          :
послідовність_операторів");
          System.out.println("break;");
          System.out.println("//…");
          System.out.println('}');
          break;
case '3' : System.out.println("while: \n");
          System.out.println("while(умова) оператор;");
          break;
case '4' : System.out.println("do-while:\n");
          System.out.println("do{");
          System.out.println("оператор;");
          System.out.println("}while(умова);");
          break;
case '5' : System.out.println("for:\n");
          System.out.println("for(ініціалізація;           умова;
повторення)");
          System.out.println("оператор;");
          break;
}}}
```

Тут для зчитування з клавіатури використовується метод System.in.read(). Це одна з функцій консольного введення Java. Цей метод зчитує символи зі стандартного пристрою введення (*повертаються у вигляді цілочисельних значень - саме тому тип значення був приведений до типу char). За замовчуванням дані зі стандартного введення поміщаються в буфер построково, тому, щоб будь-які введені символи були надіслані програмі, необхідно натиснути [Enter]*.

Більшість реальних програм Java мають графічний інтерфейс і орієнтовані на роботу в віконному режимі, проте в даному випадку консольне введення зручно.

Оскільки ми використовуємо метод System.in.read (), програма повинна містити вираз throws java.io.IOException. Цей рядок необхідно вказувати для обробки помилок введення. Він є складовою частиною обробки виключень Java.

3.2.3 Оператор for

Цикли while і do-while можуть реалізувати всі запити користувача по організації циклічних обчислень. Проте в тих випадках, коли відома кількість повторень циклу, зручно використовувати цикл for. З циклом for звичайно зв'язане поняття управляючої змінної циклу, яка набуває початкового значення при вході в цикл, змінюється згідно із виразом, визначеним у заголовку циклу після кожного його виконання і порівнюється з деяким межовим значенням для вироблення рішення про продовження обчислень або їх завершення. Нижче наведений синтаксис циклу.

for(ініціалізація_управляючої_змінної; перевірка_умови_закінчення_циклу;зміна_значення_управляючої_змінної)

оператор

Як управляюча змінна використовуються змінні будь-яких числових типів.

Наприклад, наведений далі фрагмент програми дозволяє визначити суму всіх парних чисел від 2 до 38:

```
int i, S=0;
for(i=2; i<=38;i+=2)
S+=i;
```

3.2.4 Використання коми

Щоб дві або більше змінних могли управляти циклом for, Java дозволяє вказувати по кілька операторів як в ініціалізаційній, так і в итераційній частинах оператора for. Один від одного оператори відділяються комою. Розглянемо приклад програми (див. листинг 3.4).

Листинг 3.4

```
class Coma
{
public static void main(String args[])
{
int a,b;
for(a=1, b=4;a<b;a++,b--)
{
System.out.println("a="+a);
System.out.println("b="+b);
}}}
На екрані отримаємо:
a=1
b=4
a=2
b=3
```

3.2.5 Різновиди циклу for

Цикл for має кілька різновидів, які збільшують його можливості і підвищують застосовність. Гнучкість циклу for обумовлена тим, що його три частини не обов'язково використовувати тільки за прямим призначенням. Фактично кожен розділ циклу for можна застосувати в будь-яких цілях.

Розглянемо приклад програми (див. листинг 3.5).

Листинг 3.5

```
class ForVar
{
public static void main(String args[])
{
int i;
boolean done=false;
i=0;
for(;!done; )
{
System.out.println("i дорівнює "+i);
if(i==10)
     done=true;
i++;
}}}
```

У цьому прикладі *ініціалізуючий і ітераційний вираз винесено за межі циклу for*.

Ще один різновид цикла for:

```
for( ; ;)
{
//……
}
```

Залишаючи всі три частини оператора порожніми, можна навмисно створити нескінченний цикл. Нескінченні цикли в дійсності є цикли з особливими умовами переривання.

3.2.6 Версія for-each циклу for

В Java можна використовувати другу форму циклу for, що реалізує цикл в стилі "for-each" («для кожного»).

Цикл в стилі "for-each" призначений для строго послідовного виконання повторюваних дій по відношенню до колекції об'єктів, таких, наприклад, як масив. На відміну від деяких мов, подібних C#, в якому для реалізації циклу "for-each" використовується ключове слово foreach, в Java можливість застосування циклу "for-each" реалізована за рахунок удосконалення циклу for:

```
for(тип ітер_змінна: колекція)
    блок_операторів
```

Тут:

- `тип` – тип ітераційної змінної;
- `ітер_змінна` – ім'я ітераційної змінної, яка буде послідовно приймати значення з колекції, від першого до останнього. З циклом for можна застосовувати різні типи колекцій, але ми поки будемо використовувати масиви.

На кожній ітерації циклу програма витягує наступний елемент колекції і зберігає його у змінній `ітер_змінна`. Цикл виконується до тих пір, поки не будуть отримані всі елементи колекції.

Оскільки ітераційна змінна отримує значення з колекції, її тип повинен збігатися (або бути сумісним) з типом елементів, що зберігаються в колекції. Таким чином, при переборі масиву тип `ітер_змінна` повинен збігатися (або бути сумісним) з типом елементів масиву.

Щоб зрозуміти спонукальні причини застосування циклів в стилі "for-each", розглянемо тип циклу for, для заміни якого призначений цей стиль.

```
int nums[]={1,2,3,4,5,6,7,8,9,10};
int sum=0;
for(int i=0;i<10;i++)
     sum+=nums[i];
```

Тут читання всього масиву виконується строго послідовно за рахунок індексації масиву nums вручну по управляючій змінній циклу for.

Цикл for в стилі "for-each" дозволяє автоматизувати цей процес. Зокрема, застосування такого циклу дозволяє не встановлювати значення лічильника циклу за рахунок вказівки його початкового і кінцевого значень і виключає необхідність індексації масиву вручну. Натомість програма автоматично виконує цикл по всьому масиву, послідовно отримуючи значення кожного з його елементів, від першого до останнього. Попередній фрагмент можна переписати таким чином:

```
int nums[]={1,2,3,4,5,6,7,8,9,10};
int sum=0;
for(int x : nums)
     sum+=x;
```

При кожному проходженні циклу, змінній x автоматично присвоюється значення, рівне значенню наступного елемента масиву nums. При цьому не тільки спрощується синтаксис програми, але і *виключається можливість виходу за межі масиву*.

Хоча повторення циклу for в стилі "for-each" виконується до тих пір, поки не будуть оброблені всі елементи масиву, цикл можна перервати і раніше, використовуючи оператор break, наприклад, підсумувавши значення перших п'яти елементів масиву:

```
int nums[]={1,2,3,4,5,6,7,8,9,10};
int sum=0;
for(int x : nums)
{
     sum+=x;
     if(x==5)
          break;
}
```

System.out.println(“Сума перших п’яти елементів масиву дорівнює“+sum);

У разі використання циклу for в стилі "for-each" необхідно пам'ятати про наступне: його *ітераційна змінна є змінною «тільки для читання»*, оскільки

вона пов'язана тільки з вхідним масивом. Оператор присвоювання значення ітераційній змінній не робить ніякого впливу на вихідний масив. Інакше кажучи, вміст масиву не можна змінити, привласнюючи нового значення итераційній змінній. Розглянемо приклад програми (див. листинг 3.6).

Листинг 3.6

```
class NoChange
{
public static void main(String args[])
{
int nums[]={1,2,3,4,5,6,7,8,9,10};
for(int x : nums)
{
     System.out.print(x+" ");
     x=x*10;
}
System.out.println();
for(int x : nums)
     System.out.print(x+" ");
}}
```

Вигляд екрану має бути таким:

1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

3.3 Оператори переходу

3.3.1 Оператор break

Оператор break використовується у таких випадках:

- для виходу з оператора switch;
- для виходу з циклу;

Перший варіант використовування оператора break вже було розглянуто.

Для виходу з циклу оператор break розтошовують у циклі як внутрішній оператор умовного оператора if, як це зроблено у наступному прикладі, де оператор break дозволяє запобігти ділення на 0.

```
…
for(i=0;i<x;i++)
{
…
if(b==i)
     break;
Y=a/(b-i);
…}
```

3.3.2 break мітка

Розглянемо варіант використовування оператора break – вихід з блоку по мітці. За принципом дії цей оператор нагадує оператор goto. Оператор goto існує в багатьох мовах програмування. Він дозволяє передати управління в межах програмного модуля з однієї точки в будь-яку іншу. Кваліфіковані програмісти практично ніколи не використовують цього оператора з ряду причин. Застосування goto руйнує структурованість програми, робить її малозрозумілою, значно ускладнює відладку програми, утруднює оптимізацію коду компілятором. Творці Java позбавилися всіх перерахованих недоліків goto, ввівши наступні обмеження на використовування break. *Вихід дозволений тільки з внутрішнього блока у кінець зовнішнього блока.* Не можна передати управління блоку, якій не містить оператора break.

Використовування оператора передбачає дві конструкції. Власне оператор break та помічений блок:

мітка: {…if(…) break мітка; …}

Як мітка використовується звичний ідентифікатор. Мітка не вимагає опису. Найчастіше оператор break з міткою використовується для виходу з вкладених циклів.

Цей оператор застосовується всередині помічених операторів циклу, оператора варіанта або поміченого блоку для негайного виходу за ці оператори. Він повідомляє виконуючому середовищу, що слід припинити виконання іменованого блоку і передати управління оператору, наступному ЗА даним блоком.

3.3.3 Оператор continue

Оператор continue використовується тільки в циклах. Він перериває виконання операторів, що складають тіло циклу і повертає управляння заголовку циклу. При цьому заголовок працює так, ніби чергова ітерація завершилася. Конкретно в циклах while і do-while управління передається умові, а в циклі for – третьому елементу заголовка, змінючому управляючу змінну. У наступному прикладі, якщо b==i, обчислення Y не відбудеться, але виконання циклу буде продовжено.

```
for(i=0;i<x;i++)
{
…
if(b==i)
     continue;
Y=a/(b-i);
…
}
```

3.3.4 continue мітка

Як і в випадку з оператором break, в операторі continue можна задавати мітку, яка вказує, в якому з вкладених циклів ви хочете достроково припинити виконання поточної ітерації, тобто *continue мітка* використовується *тільки в*

разі декількох вкладених циклів для негайного переходу до чергової ітерації одного з об'емлюющіх циклів , а саме, позначеного циклу.

3.3.5 Оператор return

Оператор return служить для виходу з методу. Цей оператор треба використати у випадках, коли треба закінчити програму або деяку частину програми. При цьому оператор дозволяє передати результат обчислень у вигляді значення деякого типу.

Тема 4 Масиви

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

Масив – це структурований тип даних, що являє собою сукупність даних одного типа, елементи якої розрізняються індексами.

Масиви прийнято підрозділяти на :

- одновимірні (відповідають векторам в математиці):
- багатовимірні (відповідають матрицям).

Елементи масивів завжди займають послідовно розташовані комірки пам'яті. А ім'я масиву має значення адреси першого елемента масиву. У програмуванні змінна, значенням якої є адреса, називається покажчиком. Для доступу до будь-якого елемента масиву обчислюється його адреса, як зсув від адреси масива на добуток довжини одного елемента (визначається типом елемента) на значення індексу (для одновимірного масиву) або на певний вираз, що включає кілька індексів (для багатовимірних масивів).

4.1 Одновимірні масиви

Використовування масиву передбачає:

- його оголошення;
- виділення місця в пам'яті;
- привласнення його елементам конкретних значень.

Ці операції можуть бути записані окремо або частково чи повністю суміщені.

Оголошення масиву має такий синтаксис:

тип ім'я_масиву[];

або

тип [] ім'я_масиву;

Наприклад:

double arr1[];

int arr2[];

Фактично ім'я масиву є покажчиком. Після оголошення масиву пам'ть для масиву не виділена, тому значенням покажчика буде null (відсутність значення).

Пам'ять для масиву виділяється оператором new:

ім'я масиву=new тип[кількість_елементів];

Наприклад, для раніше оголошених масивів це виглядатиме так:

arr1=new double[10];

arr2=new int[8];

Після виділення місця під масив можна дістати доступ до будь-якого елементу масива, вказавши ім’я масиву і номер елемента (індекс). Нумерація елементів починається з 0.

Наприклад:

arr1[0]=1.2;

arr1[9]=12.7;

arr2[3]=-6;

Оголошення масиву і виділення для нього місця в пам’яті можна об’єднати:

double arr1[]=new double[10];

int arr2[]=new int[8];

Також можна виконати ініціалізацію масиву при його оголошенні. У цьому випадку виділення пам’яті виконується відповідно до вказаної кількості елементів у списку ініціалізації. Наприклад:

double arr3[]={0, 1.1, 2.2 3.3, 4.4};

Розглянемо фрагмент програми, в якій обчислюється сума значень елементів масиву дійсних чисел, визначаються найбільший і якнайменший елемент, а також кількість від’ємних чисел.

```
double m[]={1.2, -6,8.5, -2.8, 11,22,-5,0.7,9};
double max, min, S=0;
int i,k=0;
max=min=m[0];
for(i=1;i<9;i++)
{
   if(m[i]>max)
     max=m[i];
if(m[i]<min)
           min=m[i];
if(m[i]<0)
           k++;
S+=m[i];
}
```

4.2 Багатовимірні масиви

Багатовимірні масиви в Java розглядаються як масиви масивів. При оголошенні багатовимірного масиву потрібно записати стільки пар квадратних дужок, скільки потрібно вимірювань. Так, наприклад, оголошення двовимірного масиву цілих чисел матиме такий вигляд:

int Arrint[][];

Для виділення місця в пам’яті під масив потрібно вказати ціле число для кожного вимірювання. Для двовимірного це буде число рядків і стовбців у матриці:

Arrint[][]=new int[2][3];

Подібно одновимірним масивам, оголошення, виділення пам’яті і ініціалізацію двовимірного масиву можна сумістити в одній конструкції.

```
int Arrint[][]={
                {2,3,4},
 `              {5,6,7}
                };
```

При виділенні пам’яті під багатовимірний масив обов’язково потрібно вказати тільки перше вимірювання (число рядків у матриці). Числа, які визначають інші вимірювання, можна задати пізніше. Це дозволяє створювати нерегулярні багатовимірні масиви. Наведений нижче фрагмент програми створює «трикутний» двовимірний масив:

```
int i,j;
double m2[][];
m2=new double[3][];
for(i=0;i<3;i++)
{
m2[i]=new double[i+1];
     for(j=0;j<i+1;j++)
          m2[i][j]=i+j+1;
}
```

Нерегулярні багатовимірні масиви використовуються нечасто. Але існують задачі, для яких вони виявляються вельми ефективними, наприклад, для представлення розріджених матриць.

Тема 5 Класи та об’єкти

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

5.1 Структура класу

У загальному випадку клас може містити будь-яку кількість даних, названих об’єктними змінними, і методів (функцій).

Спрощений синтаксис класу:

```
class ім’я_класу
{
тип об’єктна_змінна1;
тип об’єктна_змінна2;
……….
тип об’єктна_змінна N;
тип ім’я_методу1(список_параметрів)
{
     тіло методу
}
тип ім’я_методу2(список_параметрів)
```

```
{
     тіло методу
}
тип ім'я_методуМ(список_параметрів)
{
     тіло методу
}
}
```

Розподіл обов'язків між членами класу такий. Об'єктні змінні зберігають дані класу, а методи управляють цими даними і таким чином забезпечують поведінку класу. В окремому випадку клас може не мати даних, або не мати методів.

Ім'я об'єкта – це, точніше, посилання на об'єкт. *Посилання являє собою адресу об'єкта в пам'яті, надану оператором new.* Якщо проводити порівняння з мовою С, то можна помітити, що посилання схожі із покажчиками. Але це на перший погляд. Основна різниця між ними – і основна властивість, яка забезпечує безпеку використання Java-програм – в тому, що *посиланнями неможливо маніпулювати як покажчиками: посилання на об'єкт не може вказувати на довільну комірку пам'яті, ним неможна маніпулювати як цілочисельним значенням*.

5.2 Дані і методи

Опис даних класу дуже схожий на опис змінних. Кожен метод має заголовок і тіло. Заголовок методу містить три обов'язкових елементи:

```
тип_значення_що_повертається ім'я _методу([список_формальних_параметрів])
```

Тип значення, що повертається, може бути будь-яким (простим або ім'ям класу). Призначення цього елемента заголовка полягає у такому. Якщо метод обчислює одне значення, то при виклику методу це значення передається в точку виклику і для його використовування потрібно знати його тип. Якщо метод нічого не повинен повертати, то типом значення, що повертається, є ключове слово void (порожньо). Кожен метод, що повертає не void, повинен у своєму тілі мати оператора return вигляду:

```
return вираз;
```

де саме вираз визначає значення, що повертається.

Список формальних параметрів є такою конструкцією:

```
тип ім'я_змінної [, тип  ім'я _змінної [, …]]
```

Через параметри метод одержує дані, необхідні для виконання своїх функцій.

5.3 Об'єкти класу

Приклад. Складемо клас Goods1 (товар) із змінними:

- price_w – ціна закупівлі;
- quantity – кількість;
- з методом getTotal(), що має повертати загальну вартість товару.

```
class Goods1
{
double price_w;
int quantity;
double getTotal()
{
     return price_w*quantity;
}
}
```

Застосування класу передбачає створення об'єкта цього класу, яке нагадує опис звичайної змінної. Але на відміну від змінної базових типів *для об'єкта треба виділити місце у пам'яті та надати усім даним конкретних значень*.

Якщо об'єкт класа створено, то можна з ним працювати, викликаючи методи класу, які тепер пов'язані з об'єктом.

Звертання до даних чи методів класу за межами його опису можливо тільки через ім'я класу чи об'єкту. Синтаксис такого звернення наступний:

Ім'я_класу.Ім'я_даного або Ім'я_об'єкта.Ім'я_даного

Ім'я_класу.Ім'я_методу(…..) або Ім'я_об'єкта.Ім'я_методу(…)

Напрклад, визначення об'єкту класу Goods1 та застосування його методу може мати наступний вигляд:

Goods1 g1=new Goods1();

g1.price_w=6.5;

g1.quantity=10;

double d=g1.getTotal();

Хоча g1 називаємо об'єктом, насправді це посилання на об'єкт. Якщо створити об'єкт g2 того ж класу, що й g1, то можна записати:

g2=g1;

У цьому випадку g2 посилатиметься на g1 . А якщо

g1=null;

то це означає, що g1 не посилаєься ні на що.

5.4 Список аргументів змінної довжини

У Java була додана нова функціональна можливість, яка спрощує створення методів, які приймають змінну кількість аргументів. Цей засіб отримав назву varargs (variable-length arguments). Метод, який приймає змінну кількість аргументів, називають методом зі змінною кількістю аргументів.

Як приклад, можна розглядати метод printf(), що входить до складу бібліотеки введення-виведення Java.

Раніше обробка списку аргументів змінної довжини могла виконуватися двома способами:

- якщо максимальна кількість аргументів була невеликою і відомою, можна було створювати перевантажені версії методів - по одній для кожного можливого способу виклику методу;
- коли максимальна кількість можливих аргументів була великою або невідомою, застосовувався підхід, при якому аргументи спочатку поміщувались в масив, а потім масив передавався методу.

Розглянемо приклад програми (див. листинг 5.1).

Листинг 5.1

```
class PassArray
{
static void vaTest(int v[])
{
     System.out.println("Кількість аргументів: "+v.length+"
Вміст:");
     for(int x : v)
        System.out.print(x+" ");
     System.out.println();
}
public static void main(String args[])
{
int n1[]={10};
int n2[]={1,2,3};
int n3[]={};
vaTest(n1);
vaTest(n2);
vaTest(n3);
}
}
```

На екрані отримаємо:

Кількість аргументів: 1 Вміст: 10

Кількість аргументів: 3 Вміст: 1 2 3

Кількість аргументів: 0 Вміст:

Можливість використання методів зі змінною кількістю аргументів забезпечує більш простий і ефективний підхід.

Для вказівки списку аргументів змінної довжини використовують три точки (...). Наприклад, ось як метод vaTest() можна записати з використанням списку аргументів змінної довжини:

static void vaTest(int ••• v)

{

…….

}

Ця синтаксична конструкція вказує компілятору, що метод vaTest () може бути викликан без аргументів або з декількома аргументами. В результаті v *неявно* оголошується як масив типу int [].

Таким чином, усередині методу vaTest() доступ до масиву v здійснюється з використанням *синтаксису звичайного масиву*.

Перепишемо попередню пограмму (див. листинг 5.1) із застосуванням методу із змінною кількістю аргументів.

Листинг 5.2

```
class VarArgs
{

static void vaTest( int ... v)
{
     System.out.println("Кількість аргументів: +v.length+ " Вміст:
");
     for(int x : v)
        System.out.print(x+" ");
     System.out.println();
}
public static void main(String args[])
{
vaTest(10);
vaTest(1,2,3);
vaTest();
}
```

Синтаксична конструкція (int ... v) просто вказує компілятору, що метод буде використовувати змінну кількість аргументів і що ці аргументи будуть зберігатися в масиві, на який посилається змінна v.

При виклику методу аргументи автоматично поміщаються в масив і передаються змінної v. У разі відсутності аргументів довжина масиву дорівнює нулю.

Поряд з параметром зі змінною кількістю аргументів, метод може містити «нормальні» параметри. Однак параметр зі змінною кількістю аргументів повинен бути останнім параметром в списку формальних параметрів. Наприклад:

```
int doIt(int a,int b,double c,int ... vals)
{
.......
}
```

Існує обмеження - метод повинен містити тільки один параметр із змінною кількістю аргументів.

Розглянемо змінену версію методу vaTest (), який приймає звичайний аргумент і список аргументів змінної довжини (див. листинг 5.3).

Листинг 5.3

```
class VarArgs2
```

```
{
     static void vaTest(String msg, int ... v)
{
     System.out.println(msg+v.length+" Вміст: ");
     for(int x : v)
        System.out.print(x+" ");
     System.out.println();
}
public static void main(String args[])
{
vaTest("Один параметр vararg ",10);
vaTest("Три параметра vararg ",1,2,3);
vaTest("Без параметрів vararg ");
}
}
```

Метод, який приймає список аргументів змінної довжини, можна перевантажувати.

5.5 Вкладені і внутрішні класи

Мова Java дозволяє визначати клас всередині іншого класу. Такі класи називають *вкладеними*. Область видимості вкладеного класу обмежена областю видимості зовнішнього класу. Якщо клас B визначено всередині класу A, клас B не може існувати незалежно від класу A. Вкладений клас має доступ до членів (*в тому числі і закритих*) класу, в який він вкладений. Однак, *зовнішній клас не має доступу до членів вкладеного класу*. Вкладений клас, який оголошений безпосередньо всередині області видимості свого зовнішнього класу, є його членом. Можно також оголошувати вкладені класи, які є локальними для блоку.

Існує два типи вкладених класів:

- статичні;
- нестатичні.

Статичний вкладений клас - це клас, до якого застосовано модифікатор static. Оскільки він є статичним, повинен звертатися до нестатичних членів свого зовнішнього класу через ім'я об'єкта. Тобто він не може безпосередньо посилатися на нестатичні члени свого зовнішнього класу. Через це обмеження статичні вкладені класи використовуються рідко.

Найбільш важливий тип вкладеного класу - *внутрішній клас*. Внутрішній клас - це *нестатичний вкладений клас*. Він має доступ до всіх змінних і методів свого зовнішнього класу і може безпосередньо на них посилатися так само, як це роблять інші нестатичні члени зовнішнього класу.

Розглянемо приклад (див. листинг 5.4), який демонструє використання внутрішнього класу.

Листинг 5.4

```
class Outer
{
```

```
int outer_x=100;
void test()
{
Inner inner=new Inner();
inner.display();
}
//Внутрішній клас
class Inner
{
void display()
{
System.out.println(“Виведення:  outer_x=“+outer_x);
}
}
}
class InnerClassDemo
{
public static void main(String args[])
{
Outer outer=new Outer();
outer.test();
}
}
```

На екрані отримаємо:

Виведення: outer_x=100

! Важливо розуміти, що екземпляр класу Inner може бути створений **тільки** *всередині* області видимості Outer.

Як ми вже говорили, внутрішній клас має доступ до всіх елементів свого зовнішнього класу, *але не навпаки*. Члени внутрішнього класу відомі тільки всередині області видимості внутрішнього класу і не можуть бути використані зовнішнім класом.

! Компіляція наступної програми буде неможлива.

Листинг 5.5

```
class Outer
{
int outer_x=100;
void test()
```

```
        {
        Inner inner=new Inner();
        inner.display();
        }

        class Inner
        {
        int y=10;
        void display()
        {
        System.out.println("Вивод: outer_x= "+outer_x);
        }
        }
        void showy()
        {
        System.out.println(y);// Помилка! Тут змінна y невідома
        }
}
class InnerClassDemo
{
public static void main(String args[])
{
Outer outer=new Outer();
outer.test();
}
}
```

Ми розглянули внутрішні класи, визначені в якості членів всередині області видимості зовнішнього класу. Але внутрішні класи можна *визначати* всередині області видимості *будь-якого блоку*. Наприклад, внутрішній клас можна визначити всередині блоку, визначеного методом, або навіть всередині тіла циклу for:

```
class Outer
{
        int outer_x=100;
        void test()
        {
                for(int i=0;i<10;i++)
                {
```

```
                class Inner
                {
                        void display()
                        {
                                System.out.println("outer_x="+outer_x);
                        }
                }
                Inner inner=new Inner();
                inner.display();
        }
    }
}
class InnerClassDemo
{
    public static void main(String args[])
    {
    Outer outer=new Outer();
    outer.test();
    }
}
```

На екрані буде 10 рядків виду:

Виведення: outer_x=100

Вкладені класи особливо зручні при обробці подій. При обробці подій ми також поговоримо про анонімні внутрішні класи - внутрішні класи без імені.

5.6 Математичні функції

Елементарні математичні функції мови Java реалізовані в класі Math. Найбільш часто використовувані математичні функції наведені в таблиці 5.1.

Всі функції (методи) класу Math повертають значення типу double і мають аргументи типу double.

Наведені математичні функції (методи) оголошені в класі Math як статичні – їх можна використовувати, не створюючи об'єктів класу Math. Наприклад:

```
double x=0.5, c;
c=Math.sin(x);
```

Таблиця 5.1 —Математичні функції

Тип функції	Виконувана дія
Math.sqrt(x)	корінь квадратний від x

Math.pow(x,a)	x^a
Math.sin(x)	sinx
Math.cos(x)	cos(x)
Math.tan(x)	tan(x)
Math.exp(x)	e^x
Math.log(x)	lnx
Math.random()	

Тема 6 Рядки (клас String)

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

Програмування практично неможливо без застосування бібліотек, у яких накопичені результати роботи багатьох програмістів. Бібліотекі об'єктно-орієнтованої мови Java містять класи.

Неможливо написати навіть найпростішу інтерактивну програму без використання рядків. При необхідності роботи з рядками використовують клас String із стандартної бібліотеки Java.

Будь-який створюваний рядок є об'єктом класу String. І рядкові константи є об'єктами класу String.

Об'єкти класу String є незмінними. Після того як він створений, його вміст не може змінюватися:

• якщо потрібно змінити рядок, завжди можна створити новий рядок, що містить всі зміни;

• в Java визначено клас StringBuffer, рівноправний класу String, що допускає зміну рядків, це дозволяє виконувати в Java всі звичайні маніпуляції з рядками.

6.1 Оголошення і ініціалізація рядкової змінної. Операції над рядками

Оголошення рядкової змінної подібно оголошенню змінної базового типу:

String str;

Рядку можна привласнити значення при ініціалізації. Це значення (рядковий літерал) представляє послідовність будь-яких символів, узяту у подвійні лапки.

String s1="перший рядок", s2="другий рядок";

Для рядків визначена операція конкатенації (знак «+»):

String s3=s1+" та "+s2;

До рядка *можна приєднати числа і значення інших простих типів, які автоматично перетворяться в рядок*

Наприклад:

int n=3;

char ch=’1’;

String s4=”Кількість днів в січні – “+n+ch;

Значенням s4 буде: “ Кількість днів в січні – 31”.

6.1.1 Виділення підрядка

З рядка можна виділити підрядок. Для цього можна використати метод substring() класу String. Аргументами методу є номер позиції першого, та останнього символів підрядка.

Нумерація символів починається з 0. Наприклад:

String s5=s1.substring(7,11);

Значенням s5 стане «рядок».

З рядка можна виділити окремий символ. Для цього використовується метод charAt() з класу String. Аргумент методу задає позицію символу, що виділяється. Наприклад:

char c=s4.charAt(0);

Значенням с стане «К».

Проте клас String не має методу, що дозволяє замінити певний символ з рядку. Обійти це обмеження можна шляхом виділення підрядка.

6.1.2 Порівняння рядків

Застосування операції «==» порівнює тільки перші символи рядків (порівняння посилань). Наприклад:

String s1=”Перший рядок”;

String s2=”Другий рядок”;

String s3=”Дошка“;

У результаті виконання операції s1==s2 буде одержано false, а у результаті виконання операції s2==s3 буде одержано true.

Метод equals() класу String з аргументом рядок, повертає логічне значення, якщо рядкі рівні (true), і false – в іншому випадку.

Наприклад:

boolean b=s1.equals(s2);

Значенням b буде false.

Існує і інший метод порівняння рядків – compareTo(), який дозволяє визначити взаємне розташування першого рядка (об’єкт) і рядка-аргументу в лексикографічному порядку. Метод виконує посимвольне порівняння рядків зліва направо. Результатом є число, яке виробляється при першому незбігу символів. Якщо рядок-аргумент передує першому рядку, число буде позитивним. У разі,

коли рядок-аргумент йде за першим рядком, результат буде негативний. При рівності рядків результат дорівнює 0. Наприклад:

String p1=”Березень”;

String p2=”Травень”;

int k=p1.compareTo(p2);

Значення k буде від’ємним.

У разі випадків дуже корисним може бути метод trim(), який видаляє з рядка всі попередні та замикаючі пропуски. Наприклад:

String p3=” рядок з пропусками на початку і у кінці “;

String p4=p3.trim();

Значенням p4 буде ”рядок з пропусками на початку і у кінці“;

Всього клас String містить більш ніж 50 методів. У випадках, коли необхідне безпосереднє редагування рядка шляхом доступу до будь-якого символу, Java пропонує клас StringBuffer, який відрізняється меншою швидкістю.

6.1.3 Пошук підрядка

Методи indexOf(String s), indexOf(String s,int ind) дозволяють виявити входження підрядка у рядок з першої позиції чи з позиції ind. Метод hasIndexOf(String s) знаходить останнє входження підрядка. Методи повертають номер позиції, чи «-1» у разі невдачі.

Тема 7 Структура програми мовою Java. Основи введення/виведення

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

7.1 Структура програми

Структуру простої програми на мові Java можна уявити так, як наведено на рис. 7.1:

Рисунок 7.1 — Структура простої Java-програми

Кожна програма на мові Java має використати, як мінімум, один клас – головний клас програми. В простих програмах головний клас може не мати даних.

Для створення об'єкту класу використовується спеціальний метод – *конструктор*. Про конструктори детальніше ми поговоримо пізніше, а поки що досить знати, що ім'я конструктора завжди збігається з ім'ям класу. Основне призначення конструктора – проініціалізувати дані – члени класу.

Робота програми починається з методу main() - система часу виконання викликає метод main().

7.2 Основи введення-виведення. Клас Scanner

Більшість реальних додатків Java не є текстовими консольними програмами. Замість цього вони є програмами з призначеним для користувача графічним інтерфейсом на базі бібліотек AWT (Abstract Window Toolkit) і Swing або веб-додатками.

Текстові консольні програми зазвичай використовуються в якості навчальних прикладів Проте текстове консольне введення цілком можна застосувати в реальному програмуванні на Java.

Класс Scanner

(import java.util.Scanner)

Клас Scanner - це додаток до класу Formatter. Він читає *форматований ввод* і перетворює його в бінарну форму. Клас Scanner може застосовуватися для введення:

- з консолі;
- з файлу;
- з рядка;

- або з іншого джерела, що реалізує інтерфейс Readable або ReadableByteChannel.

Об'єкт класу Scanner читає лексеми з деякого джерела введення, яке вказується при створенні об'єкта класу Scanner. З точки зору класу Scanner, лексема (token) - це порція введення, відокремлена набором роздільників, якими за замовчуванням є пробіли. Лексема читається відповідно до деякого регулярного вираза, що задає формат даних. Хоча клас Scanner дозволяє визначити специфічний тип шаблону, якому буде відповідати наступна операція введення, він включає безліч визначених шаблонів, відповідних елементарним типам, таким як int і double, а також рядкам. Іншими словами, найчастіше немає необхідності в зазначенні власних шаблонів.

У загальному випадку для використання класу Scanner необхідно виконати наступні дії:

- визначити, чи доступний специфічний тип введення викликом одного з методів Scanner.hasNextX(), де X - потрібний тип даних;
- якщо введення доступне, читати його одним з методів Scanner.nextX(), де X - потрібний тип даних;
- повторити процес до завершення введення;
- закрити об'єкт класу Scanner, викликаючи метод close().

Як випливає зі сказаного вище, клас Scanner визначає два набори методів, які дозволяють читати введення.

Перший - це методи hasNextX (), перераховані в таблиці 7.1. Ці методи визначають, чи доступний вказаний тип введення. Наприклад, виклик методу hasNextInt() повертає значення true тільки в тому випадку, якщо наступна лексема, що підлягає читанню, є цілим числом. Якщо необхідні дані доступні, вони зчитуються одним з методів Scanner.nextX() (описані в таблиці 7.2.). Наприклад, щоб прочитати таке ціле число, використовується метод nextInt().

Наведена послідовність показує, як читати список цілих чисел, що вводяться з клавіатури:

```
    Scanner conin=new Scanner(System.in);
    int i;
while(conin.hasNextInt())
    {
    i=conin.nextInt();
    …….
    }
```

Ось деякі методи типа hasNext классу Scanner.

Таблиця 7.1 — Методи класа Scanner типу hasNext

Метод	Призначення
boolean hasNext()	Повертає true, якщо доступна для читання наступна лексема будь-якого типу. В іншому випадку повертається false
boolean hasNext(Pattern шаблон)	Повертає true. якщо доступна для читання лексема, відповідна шаблону *шаблон.* В іншому випадку повертається false
boolean hasNext(String шаблон)	Повертає true, якщо доступна для читання лексема, відповідна шаблону *шаблон*. В іншому випадку повертається false
boolean hasNextBoolean()	Повертає true, якщо є для читання значення типу boolean. В іншому випадку повертається false
boolean hasNextByte()	Повертає true, якщо є для читання значення типу byte. В іншому випадку повертається false
boolean hasNextDouble()	Повертає true, якщо є для читання значення типу double. В іншому випадку повертається false
boolean hasNextFloat()	Повертає true, якщо є для читання значення типу float. В іншому випадку повертається false
boolean hasNextInt()	Повертає true, якщо є для читання значення типу int. В іншому випадку повертається false
boolean hasNextInt(int основание)	Повертає true, якщо є для читання значення типу int із зазначеною основою. В іншому випадку повертається false
boolean hasNextLine()	Повертає true, якщо доступний для читання рядок введення. В іншому випадку повертається false
boolean hasNextLong()	Повертає true. якщо є для читання значення типу long. В іншому випадку повертається false

Продовження таблиці 7.1

Метод	Призначення
boolean hasNextShort()	Повертає true, якщо є для читання значення типу short. В іншому випадку повертається false

Розглянемо деякі методи next класу Scanner, наведені в таблиці 7.2.

Таблиця 7.2. —Методи next класа Scanner

Метод	Призначення
String next()	Повертає наступну лексему будь-якого типу з вхідного джерела даних
String next(Pattern шаблон)	Повертає з вхідного джерела наступну лексему, відповідну шаблону
String next(String шаблон)	Повертає з вхідного джерела наступну лексему, відповідну шаблону
boolean nextBoolean()	Повертає наступну лексему як значення boolean
byte nextByte()	Повертає наступну лексему як значення byte
double nextDouble()	Повертає наступну лексему як значення double
float nextFloat()	Повертає наступну лексему як значення float
int nextInt()	Повертає наступну лексему як значення int
int nextInt(int основание)	Повертає наступну лексему як значення int по задані основі
String nextLine()	Повертає наступний рядок введення
short nextShort()	Повертає наступну лексему як значення short

Якщо метод next() не *може знайти тип даних*, який він очікує, передає виняток InputMismatchException. Виняток NoSuchElementException передається в разі, коли *більше немає доступного введення*. Тому спочатку краще перевірити, що дані необхідного типу доступні за допомогою методу hasNext(), перш ніж викликати відповідний йому метод next().

Розглянемо приклад застосування класу Scanner (див. листинг 7.1). Програма обчислює середнє арифметичне зі списку чисел, введених з клавіатури до тих пір, поки користувач не введе рядок "ready".

Листинг 7.1

```
package numbers;

import java.util.Scanner;
import java.io.*;
public class Numbers
{
    public static void main(String[] args)
    throws IOException
    {
    Scanner conin=new Scanner(System.in);
    int i=0;
    double summ=0.0;
   System.out.println("Введіть цілі значення. Для завершення введення
введіть рядок \"Ready\": ");
   while(conin.hasNext())
   {
   if(conin.hasNextInt())
      summ+= conin.nextInt();
   else
   {
       String str=conin.next();
       if( str.equals("Ready"))
           break;
       else
       {
           System.out.println("Помилка введення");
           return;
       }
   }
   }
       conin.close();
       System.out.println("S="+summ);
    }
}

Введіть цілі значення. Для завершення введення введіть рядок "Ready":
1
2
3
4
5
Ready
S=15.0
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 19 секунды)
```

```
run:
Введіть цілі значення. Для завершення введення введіть рядок "Ready":
3.4
Помилка введення
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 7 секунды)
```

Рисунок 7.2 — Результат виконання програми з листингу 7.1

Розглянемо ще один приклад введення/виведення даних — за допомогою класів BufferedReader і PrintWriter, про які ми детальніше поговоримо пізнише :

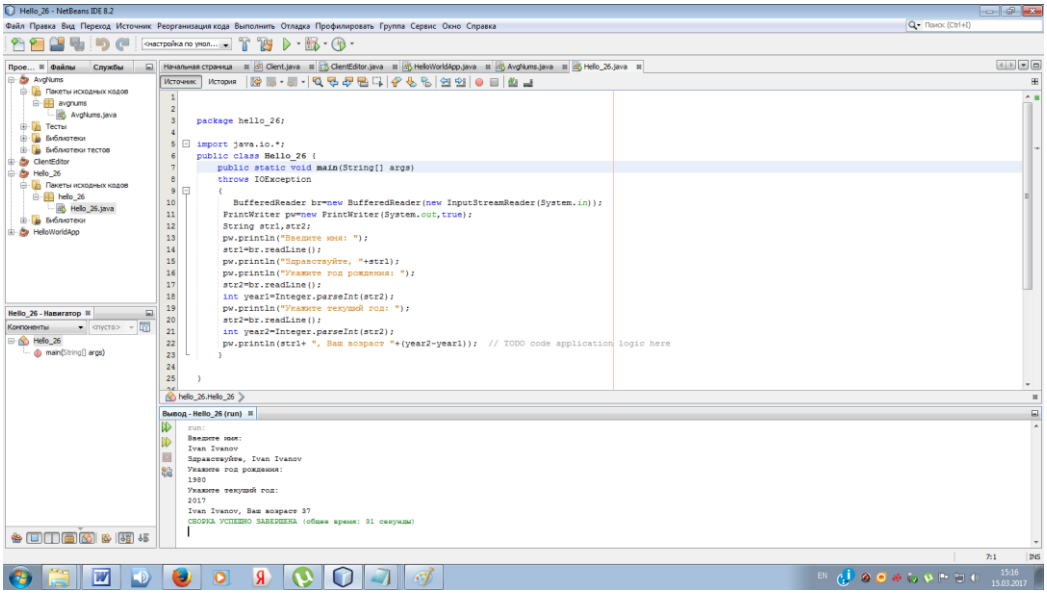


Рисунок 7.3 — Код та результат виконання програми

Тема 8 Застосування командного рядка для запуска програм. JAR-файл

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

8.1 Застосування командного рядка

Наша перша програма приймала початкові дані з консолі. Існує і інший простий спосіб передати дані в програму. Він полягає в запису початкових даних в командному рядку після імені виконуваної програми. Розглянемо приклад програми (рис. 8.1) , в якій обчислюється площа трикутника за формулою S=1/2absinα, где a,b - сторони трикутника, α – кут між ними.

Рисунок 8.1 — Отримання вхідних даних через аргументи командного рядка

Приклад запуска програми наведений на рисунку 8.2.

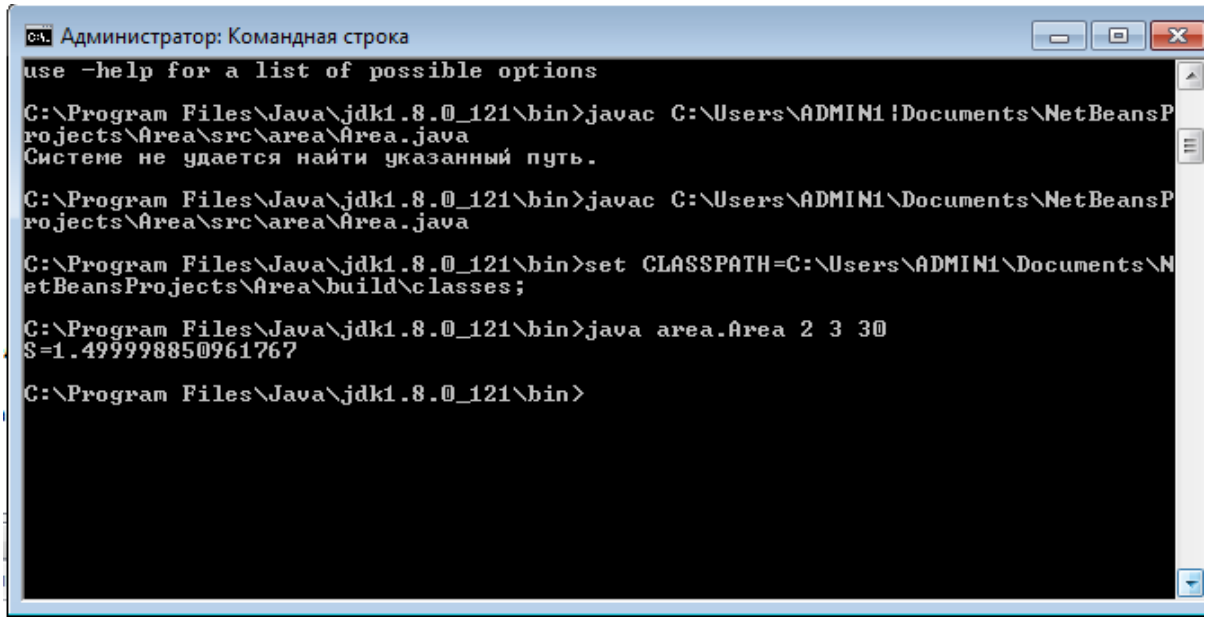
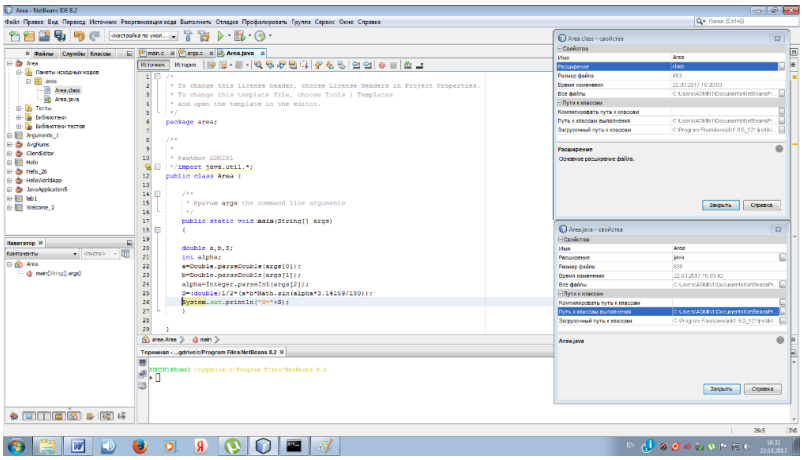


Рисунок 8.2 — Запуск програми і передача вхідних даних за допомогою командного рядка

Розглянемо ще один приклад – програма отримає вхідні дані через командний рядок. Користувач вводить рядки, які можуть мати символ ?. Треба визначити, який серед параметрів першим має символ ? (листинг кода програми див.



на рисунку 8.3). Компіляція і виконання програми в середовищі NetBeans наведені на рисунку 8.3. Приклад виконання програми в середовищі Intellij Idea наведений на рисунку 8.5.

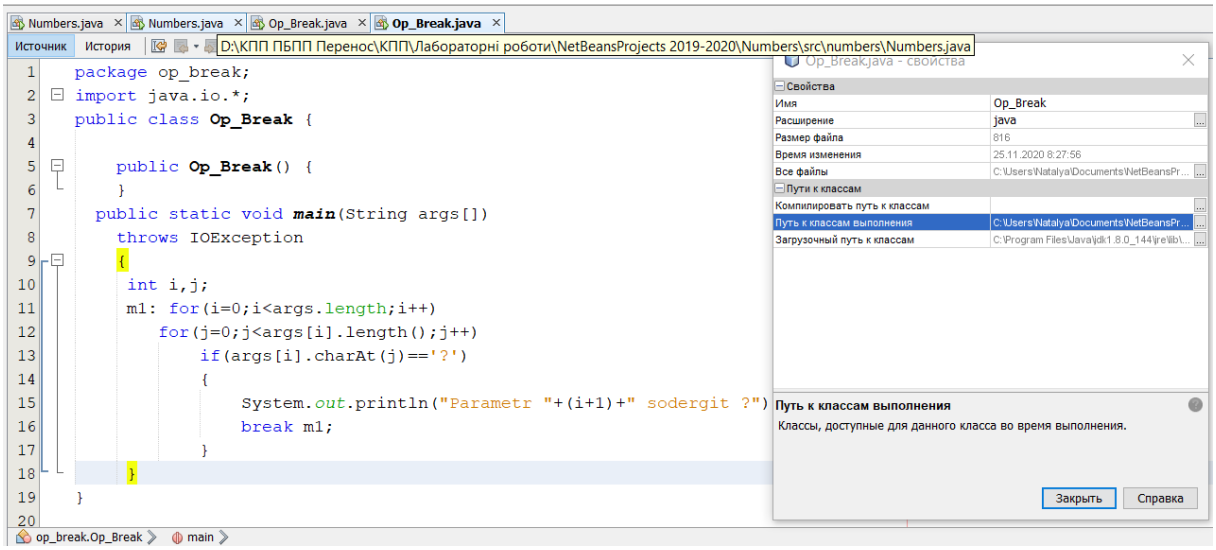


Рисунок 8.3 — Визначаємо шлях до файлу .java

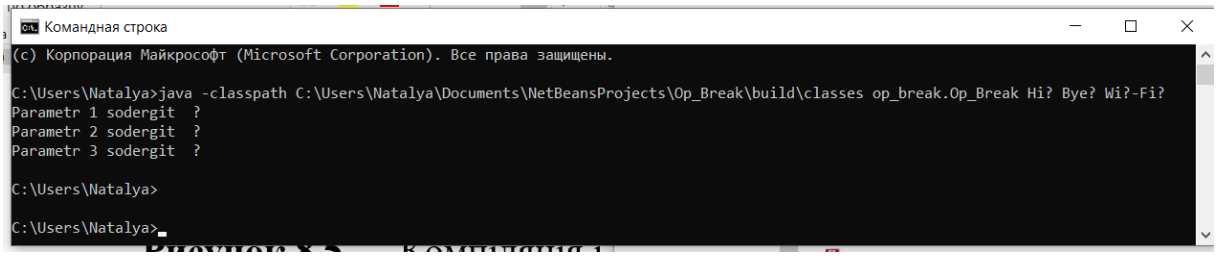
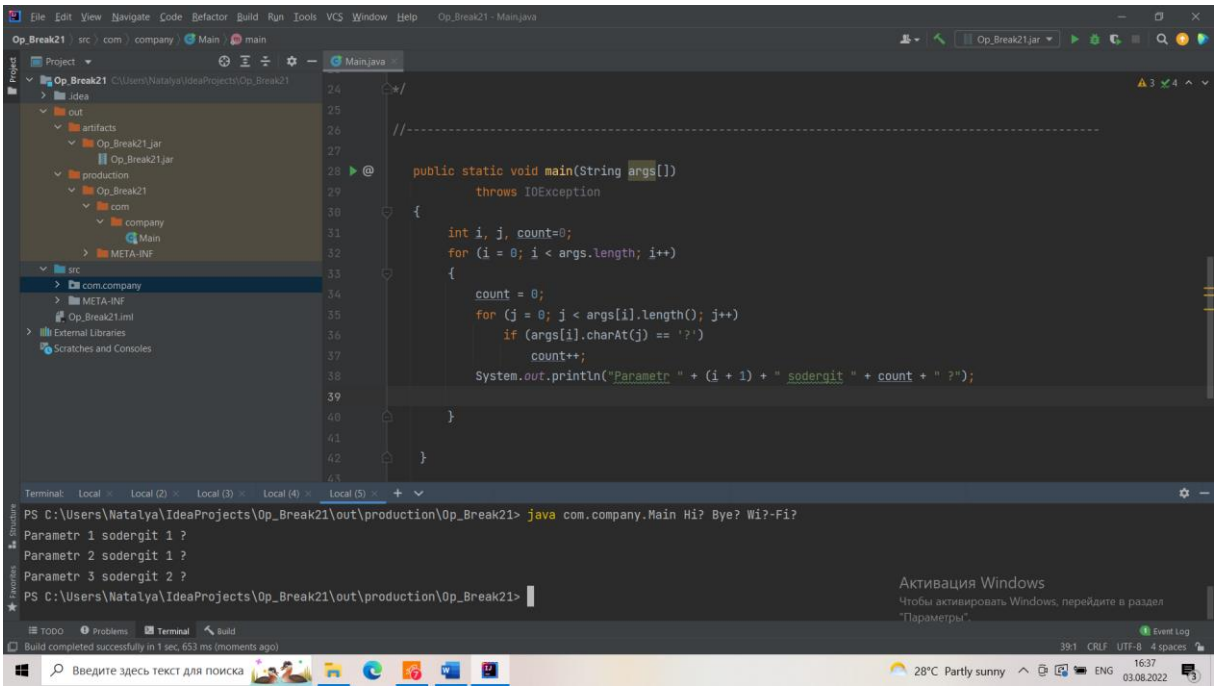


Рисунок 8.4 — Визначаємо шлях до файлу .class



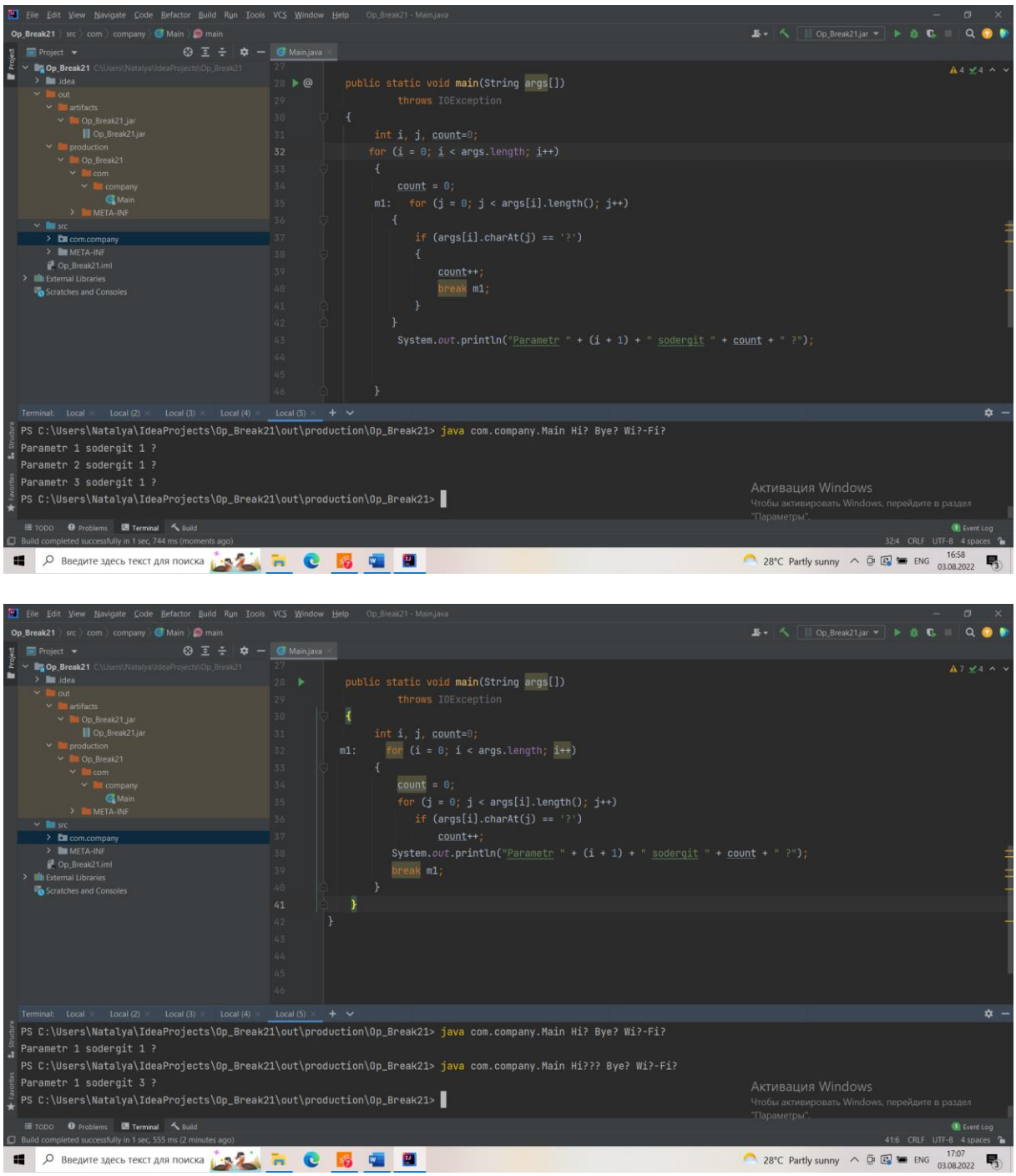


Рисунок 8.5 — Виконання програми в середовищі IntelliJ Idea

8.2 JAR-файл

Після складання і тестування програми для підготовки її до розгортання можна використовувати jar-файл. Це zip-архів, який містить файл маніфесту, вихідний (початковий) код (.java), class-файли, файли ресурсів.

8.2.1 Створенняе jar-файла в IntelliJ Idea

File -> Project Structure -> Artifacts -> Натискаємо + -> Із списку вибираємо JAR -> From modules with dependencies -> У з’явившемуся вікні натискаємо кнопку з трьома точками і отримуємо автоматично знайдений самою системою

головний клас Main -> OK -> В наступному вікні приймаємо усе за замовчуванням -> Apply -> OK.

Таким чином ми налагодили Intellij Idea на генерацію виконуваного jar-файла.

Тепер можна створити jar-файл:

Build -> Build Artifacts -> У з'явившемуся вікні обираємо -> Build -> Тепер IntelliJ Idea створює наш JAR-файл, який буде міститись в каталозі output/artifacts.

Варіанти запуску програми із викоримтанням jar-файла наведені на рисунках 8.7 – 8.10.

Так виконується запуск jar-файла з середовища розробки програми:

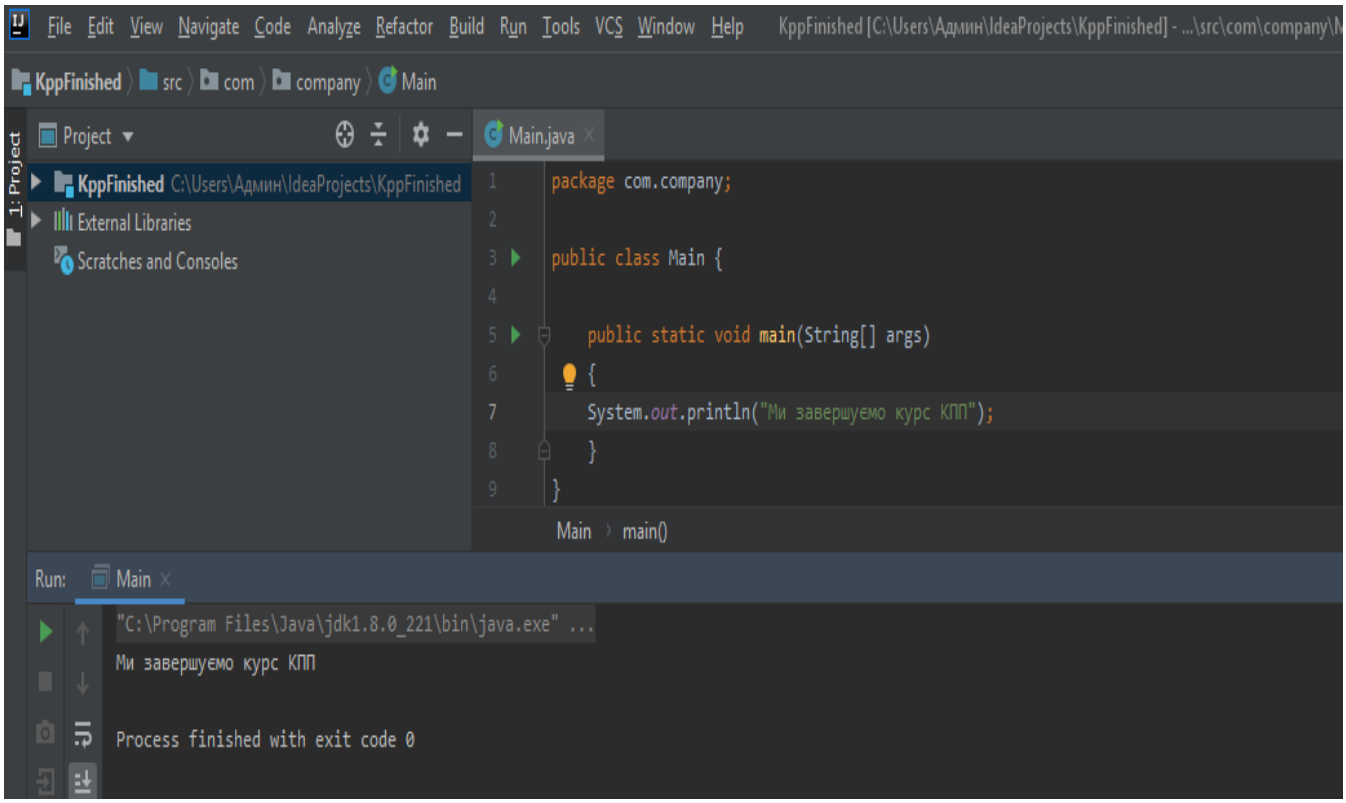


Рисунок 8.7 — Запуск jar-файла із середовища розробки

Із вікна термінала jar-файл можна запустити таким чином:

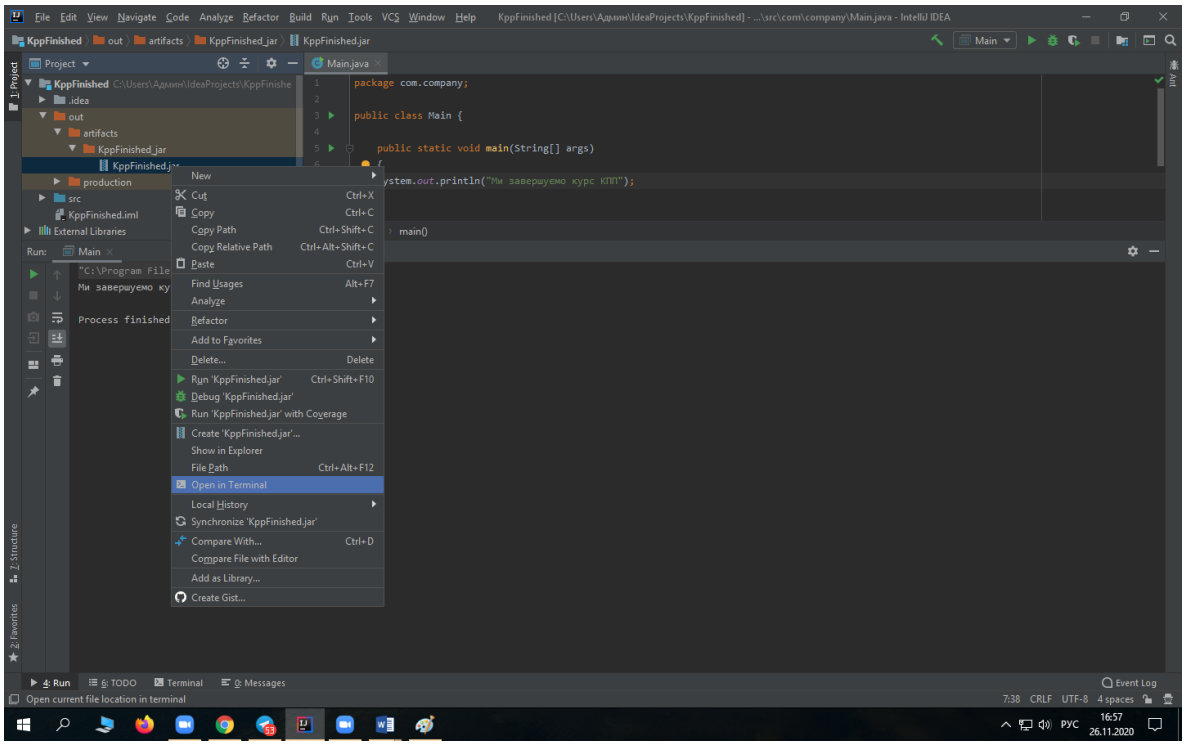


Рисунок 8.8 —Підготовка до запуску jar-файла з терміналу

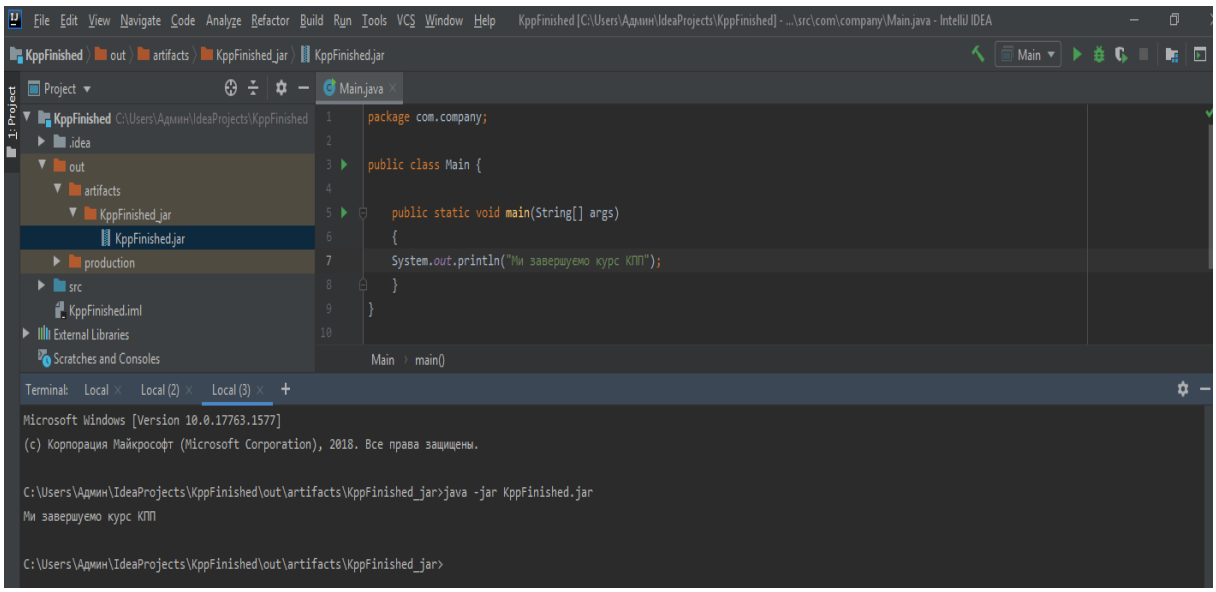


Рисунок 8.9 —Запуск jar-файла із терміналу

Jar-файл дозволяє розгорнути програму на будь-якому комп’ютері, але, зрозуміло, на ньому повинна бути встановленою JVM.

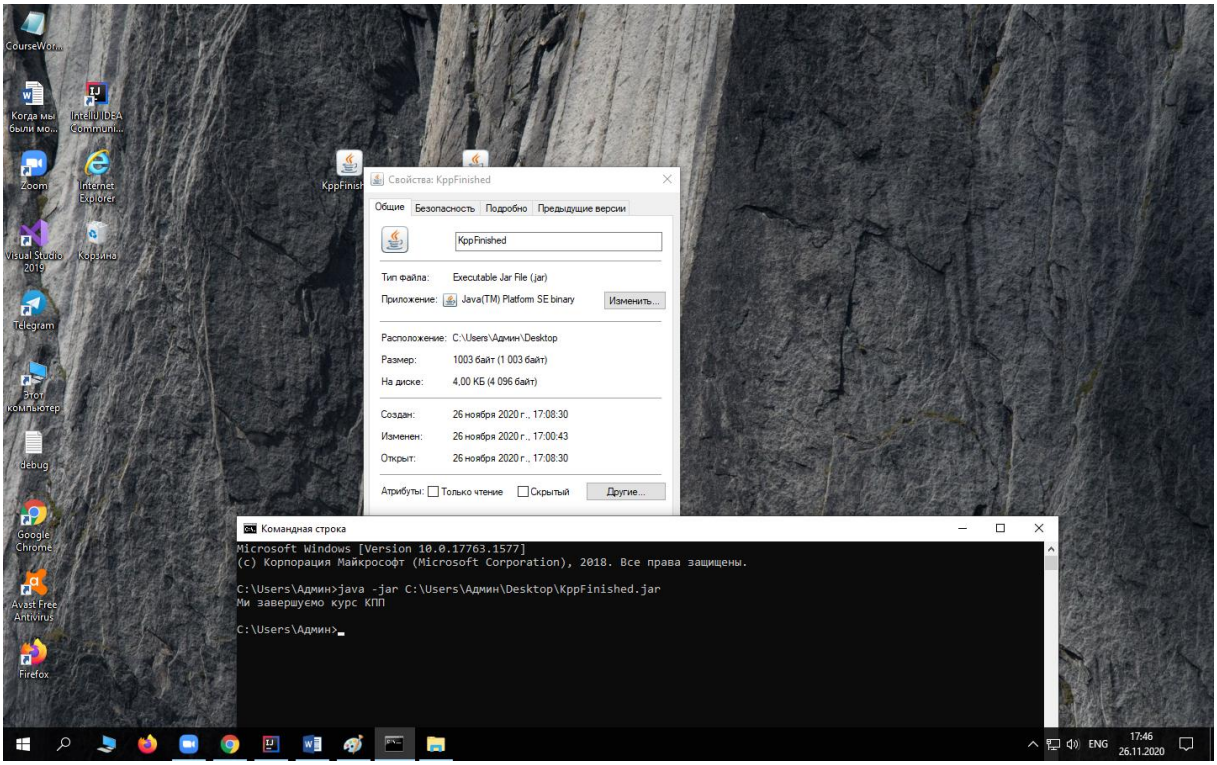


Рисунок 8.10— Запуск jar-файла на будь-якому комп’ютері

Тема 9 Конструктори. Об’єкти як параметри. Рекурсія

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

Створення програми з класом Goods2

Створемо більш вдосконалений новий клас Goods2. Товар повинен мати не тільки ціну та кількість, але й назву. Тому нашими об’єктними змінними будуть:

- name – назва товару;
- price_w – ціна закупівлі;
- quantity – кількість.

Які методи потрібно додати в новий клас?

Перш за все нам потрібен метод для *привласнення значень* об‘єктним змінним. Назвемо його setData(). Цей метод не має нічого повертати, але повинен мати три параметри для установки значень трьом змінним.

Для того, щоб побачити, *які саме значення мають об‘єктні змінні*, можна ввести метод print().

У процесі використовування об‘єкта *кількість товару може змінюватися*. Тому має сенс ввести метод setQuantity(), який повинен мати один параметр –

нова кількість товару. Може виникнути запитання – «Для чого потрібен цей метод, якщо вже є метод setData()? Річ у тому, що метод setData() має використовуватись один раз для створення об'єкта. Повторне використовування методу setData() може привести до зміни назви товару, що рівносильне знищенню існуючого об'єкту.

```
void setData(String n,double p,int q)
{
     name=n;
     price_w=p;
     quantity=q;
}
```

Для того, *щоб побачити , які значення мають об'єктні змінні*, можна ввести метод print(), який в точку виклику нічого не повинен повертати, а служить для виведення даних на екран. Ніяких параметрів цей метод одержувати не повинен. Всі необхідні дані є в об'єкті.

```
void print()
{
System.out.println("Товар "+name+" Ціна за одиницю "+price_w+ "
Кількість "+quantity);
}
```

У процесі використовування об'єкта кількість товару може змінюватися. Тому має сенс ввести метод setQuantity(), який повинен мати один параметр – нова кількість і нічого не повертає.

```
void setQuantity(int q)
{
quantity=q;
}
```

Для отримання загальної вартості партії товару збережемо метод getTotal() з класу Goods1.

Наш клас Goods2 буде описаний за межами класу Prog_class2, ім'я якого збігається з ім'ям файлу.

Листинг 9.1

```
package prog_goods2;
import java.io.*;
class Goods2
{
    String name;
    double price_w;
    int quantity;
    void setData(String n,double p,int q)
    {
        name=n;price_w=p;quantity=q;
    }
    void setQuantity(int q)
    {
        quantity+=q;
```

```
    }
    void print()
    {
        System.out.println("Товар:          "+name+"          Кіль-
кість:"+quantity+"Ціна закупівлі: "+price_w);
    }
    double getTotal()
    {
        return price_w*quantity;
    }
}

public class Prog_Goods2
{
    public static void main(String[] args)
 {
        Goods2 g1=new Goods2();
        g1.setData("PC",3500,20);
        g1.print();
        System.out.println("Товар:   "+g1.name+"Загальна  вартість
партії: "+ g1.getTotal() );
        g1.setQuantity(-10);
         g1.print();
    }
}
```

Використовування методів дозволяє відмовитися від безпосереднього маніпулювання об'єктними змінними. Як видно, доступ до методів класу аналогічний доступу до даних. Перед ім'ям методу потрібно вказати ім'я об'єкта. Метод без об'єкта не існує (винятком є статичні методи, див. далі).

9.1 Конструктори

У попередньому прикладі після створення об'єкту класу Goods2 ми виконували ініціалізацію даних об'єкту функцією setData(). Такий спосіб ініціалізації кращий, ніж безпосереднє привласнення значень об'єктним змінним, але все-таки має недоліки. По-перше, об'єкт – це модель чогось в реальному світі. Очевидно, що модель, у якій дані досконало випадкові, не може поводится адекватно модельованому об'єкту. Тому існування непрацездатного об'єкту є помилкою. Подруге, створивши об'єкт, можна забути проініціалізувати його дані, якщо створення і ініціалізація об'єкта не суміщені в часі.

Для усунення вказаних недоліків у мові Java існує спеціальний метод – *конструктор.* Синтаксис конструктора має ряд особливостей порівняно з іншими методами:

- ім'я конструктора завжди збігається з ім'ям свого класу;
- для конструктора не записується тип значення, що повертається.

Конструктор виконує ініціалізацію всіх об'єктних змінних класу не залежно від кількості своїх формальних параметрів.

Клас може мати декілька конструкторів. Всі вони повинні відрізнятися один від одного кількістю або типами параметрів.

Якщо клас має конструктор, то створення об’єкта класу можливе тільки шляхом виклику конструктора.

Створимо клас Goods3, додавши зміни в клас Goods2, зв’язані з використанням конструкторів. Перший конструктор буде дуже схожий на метод setData().

Вид другого конструктора визначається такими міркуваннями. Припустимо, що при створенні об’єктів класу Goods3 часто кількість товарів дорівнюватиме одиниці. Тоді буде виправдане створення другого конструктора з двома параметрами – без кількості.

Листинг 9.2

```
package prog_class3;
class Goods3
{
    String name;
    double price_w;
    int quantity;

    Goods3(String n,double p,int q)
    {
        name=n;
        price_w=p;
        quantity=q;
    }
    Goods3(String n,double p)
    {
        name=n;
        price_w=p;
        quantity=1;
    }
    void print()
    {
        System.out.println("Товар: "+name+" Ціна за одиницю:
"+price_w+" Кількість: "+quantity);
    }
    double getTotal()
    {
        return price_w*quantity;
    }
    void setQuantity(int q)
    {
        quantity+=q;
    }

}
public class Prog_Class3
{
    public static void main(String[] args)
{
        Goods3 g1=new Goods3("PC",5000,6);
        Goods3 g2=new Goods3("TV",10000);
        g1.print();
```

```
        g2.print();
        System.out.println("Товар: "+g1.name+" Загальна вартість партії:
        "+g1.getTotal());
        g2.setQuantity(10);
        g2.print();
    }

}
```

Ми розглянули найбільш затребуваний вид конструктора – конструктор з параметрами. Проте для деякіх класів має сенс створювати *конструктор за умовчанням. Такий конструктор не має параметрів і не має операторів. Для класу Goods3 він має вигляд:*

Goods3()

{ }

У результаті застосування конструктора за умовчанням:

- всі числові дані – члени класу набудуть значення нуль;
- всі рядкові – порожній рядок;
- логічні – значення false;
- а посилання на об’єкти – null.

Якщо ніякий явний конструктор не вказаний, Java автоматично надасть конструктор за умовчанням.

Існує і третій спосіб ініціалізації – використовування блоків ініціалізації. Проте блоки ініціалізації не мають яких-небудь переваг порівняно з конструкторами і рідко використовуються. Ми їх розглянемо при вивченні статичних даних і методів.

Які імена давати формальним параметрам методів? Можна рекомендувати називати формальні параметри іменами, які схожі на імена відповідних даних класу.

9.2 Клас Stack

Розглянемо приклад - клас Stack. Стек зберігає дані в порядку «першим увійшов, останнім вийшов». Стеки управляються двома операціями, які традиційно називаються:

- додавання (в стек);
- виштовхуванням (зі стека).

Розглянемо стек розміром до десяти цілочисельних значень.

```
class Stack
{
int stck[]=new int[10];
int tos;
// Ініціалізація верхівки стека
Stack()
{
tos=-1; // Вказує на порожній стек
```

```
}
void push(int item)
{
if(tos==9)
   System.out.println("Стек повний");
else
   stck[++tos]=item;
}
int pop()
{
if(tos<0)
{
   System.out.println("Стек порожній");
   return 0;
}
else
return stck[tos--];
}
}
```

Стек міг би зберігатися в більш складній структурі даних, ніж масив, наприклад, у вигляді зв'язного списку. Але інтерфейс, визначений методами push() і pop(), залишився б незмінним.

Напишемо клас TestStack, що демонструє застосування класу Stack.

```
class TestStack
{
public static void main(String args[])
{
Stack mystak1=new Stack();
Stack mystack2=new Stack();
for(int i=0;i<10;i++)
   mystack1.push(i);
for(int i=10;i<20;i++)
   mystack2.push(i);
System.out.println("Стек в mystack1: ");
for(int i=0;i<10;i++)
   System.out.println(mystack1.pop());
System.out.println("Стек в mystack2: ");
for(int i=0;i<10;i++)
   System.out.println(mystack2.pop());
}
}
```

9.3 Перевантаження методів

Мова Java дозволяє визначення усередині одного класу двох і більше методів з одним ім'ям, якщо тільки оголошення їх параметрів різні. В цьому випадку методи називають перевантаженими, а процес – перевантаженням методів.

При виклику перевантаженого методу для визначення потрібної версії Java використовує тип або кількість аргументів методу. Отже, перевантажені методи

повинні розрізнятися за типом або кількістю параметрів. Хоча повертані типи перевантажених методів можуть бути різні, самого повертаного типу недостатньо для розрізнення двох версій методу. Коли середовище виконання Java зустрічає виклик перевантаженого методу, воно просто виконує ту його версію, параметри якої відповідають аргументам, використаним при виклику.

При виклику перевантаженого методу Java шукає відповідність між аргументами, які були використані для виклику методу, і параметрами методу. Проте ця *відповідність необов'язково має бути повною*. В деяких випадках при пошуку перевантажених версій може використовуватися автоматичне перетворення типів Java.

У тих мовах, які не підтримують перевантаження методів, кожному методу має бути присвоєне унікальне ім'я. Проте частенько бажано реалізувати, по суті, один і той же метод для різних типів даних. Проте в мовах, які не підтримують перевантаження методів, існує три або більше за версії функції обчислення абсолютного значення з іменами (наприклад, в С це abs(), labs(), fabs()), що злегка розрізняються). У мові Java бібліотека класів містить метод обчислення абсолютного значення abs(). Перевантажені версії цього методу для обробки усіх числових типів визначені в класі Java Math. Вибір конкретної версії перевантаженого методу - завдання компілятора.

9.4 Об’єкти як параметри

У розглянутому класі Goods3 було два конструктори, які відрізнялися кількістю параметрів. Іноді потрібен конструктор іншого типу – створюючий об’єкт як копію вже існуючого об’єкта. Такий конструктор називається конструктором копіювання. Він має використовувати як параметр об’єкт цього ж класу.

Складемо конструктор копіювання для класу Goods3:

```
Goods3(Goods3 g)
{
 name=g.name;
 price_w=g.price_w;
 quantity=g.quantity;
}
```

Всім даним нашого об’єкта привласнюються значення відповідних даних об’єкта-параметра методу.

Розглянутий приклад показує, що параметрами методів можуть бути не тільки дані простих типів, але й об’єкти класів.

У попередніх прикладах програм ми побачили, як можна передати в метод дані через параметри і одержати результат роботи методу через значення, що повертається. Проте цим не обмежуються всі можливі способи обміну інформацією між методом і програмою, що його використовує. Двосторонній обмін може здійснюватися і за допомогою параметрів.

9.5 Передача аргументів за значенням і за посиланням

У більшості мов програмування існує два способи передачі аргументу підпрограмі:

- за значенням;
- за посиланням.

При виклику за значенням в методі створюється змінна, в яку копіюється параметр. Після цього ніякого зв'язку між фактичним параметром і його копією в методі немає.

При віиклику за посиланням копіювання фактичного параметра не відбувається. Тому є можливість як використовувати значення фактичного параметра, так і змінювати це значення, оскільки адреса параметра залишається у розпорядженні методу.

У мові Java всі параметри простих типів викликаються за значенням, а параметри-об'єкти викликаються за посиланням.

Коли посилання на об'єкт передається методу, саме посилання передається з використанням виклику за значенням. Проте оскільки передаване значення посилається на об'єкт, копія цього значення все одно посилатиметься на той же об'єкт, що і відповідний аргумент.

Додамо в наш клас Goods3 метод changeObj(), який використовує як параметр об'єкт класу Goods3. У методі порівнюється ім'я нашого об'єкта з ім'ям об'єкта-параметра. При збігу імен в об'єкті-параметрі змінюється ціна і кількість товару, як показано нижче:

```
void changeObj(Goods3 g)
{
if(g.name.equals(name))
{
g.price_w=(g.price_w+price_w)/2;
g.quantity+=quantity;
}
}
```

9.6 Рекурсія

У мові Java підтримується рекурсія.

```
class Factorial
{
int fact(int n)
{
int result;
if(n==1)
     return 1;
result=n*fact(n-1);
return result;
}
}
class Recursion
```

```
{
public static void main(String args[])
{
 Factorial f=new Factorial();
System.out.println("3!="+f.fact(3));
}
}
```

9.7 Об'єкти як значення, що повертаються

Ключове слово this

Якщо в деякому контексті треба зробити посилання на власний об'єкт, то слід застосовувати ключове слово this.

Метод класу може повертати не тільки значення простого типу, але й об'єкт.

Хай необхідно в класі Goods3 створити метод addPrice(), який на базі об'єкта this будує новий об'єкт з ціною, збільшеною на k%:

```
Goods3 addPrice(int k)
{
Goods3 obj=new Goods3(this);
obj.price_w+=obj.price_w*k/100;
return obj;
}
```

Змінимо попередню програму, розширюючи клас Goods3: додамо конструктор копіювання, метод changeObj() і метод addPrice(). Після цього назвемо наш клас Goods4. Крім цього, додамо дуже простий клас Show для ілюстрації виклику параметрів за значенням.

Листинг 9.3

```
package pack1.pack2;
  class Goods4
{
    String name;
    double price_w;
    int quantity;
    Goods4(String n,double p,int q)
    {
        name=n;
        price_w=p;
        quantity=q;
    }
    Goods4(String n,double p)
    {
        name=n;
        price_w=p;
        quantity=1;
    }
    Goods4(Goods4 g)
    {
        name=g.name;
        price_w=g.price_w;
        quantity=g.quantity;
    }
    void print()
```

```
    {
        System.out.println("Товар: "+name+" Ціна за одиницю: "+price_w+"
Кількість: "+quantity);
    }
    double getTotal()
    {
        return price_w*quantity;
    }
    void changeObj(Goods4 g)
    {
        if(g.name.equals(name))
        {
            g.price_w=(price_w+g.price_w)/2;
            g.quantity+=quantity;
        }
    }
    Goods4 addPrice(int k)
    {
        Goods4 g=new Goods4(this);
        g.price_w+=g.price_w*k/100;
        return g;
    }
}

public class Main {

    public static void main(String[] args)
    {
        Goods4 g1=new Goods4("PC",5000,6);
        Goods4 g2=new Goods4("TV",10000);
        Goods4 g3=new Goods4(g1);
        g1.print();
        g2.print();
        g3.print();
        System.out.println("Товар: "+g1.name+" Загальна ціна партії:
"+g1.getTotal());
        g3.changeObj(g1);
        g1.print();
        Show s=new Show();
        int i=4,j=3;
        System.out.println("В main() до виклику showParam(): i="+i+" j="+j);
        s.showParam(i,j);
        System.out.println("В main() після виклику showParam(): i="+i+" j="+j);
        Goods4 g4=g3.addPrice(20);
        g4.print();
    }
    }

class Show
{
    void showParam(int n,int m)
    {
        n++;
        m++;
        System.out.println("В showParam(): n="+n+" m="+m);
    }
}
```

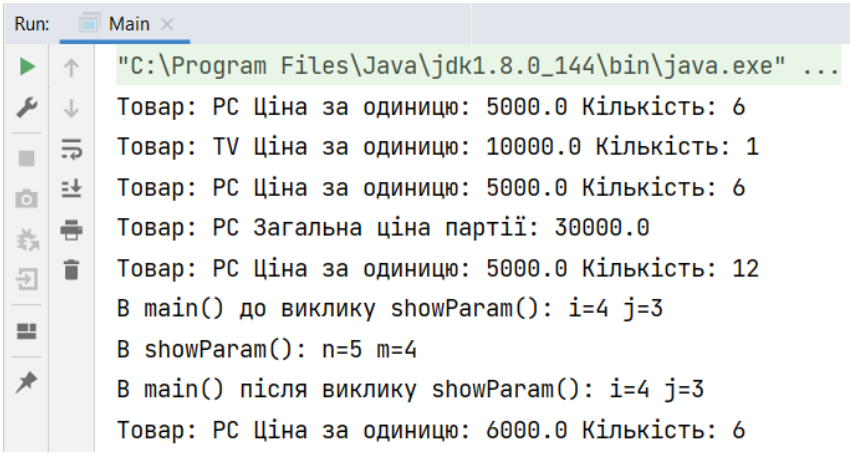


Рисунок 9.1 —Результат виконання програми

Після виконання програми ми побачимо, що значення змінних i та j, якими були, такими й залишилися: i=4, j=3.

9.8 Привласнення посилань

Нехай дан клас, що описує паралелепіпед.

```
 class Box
{
double width;
double height;
double depth;
Box(int w, int h, int d)
{width=w; height=h; depth=d;}
}
```

Тоді:

```
Box b1=new Box(2,3,4);
Box b2;
b2=b1;
```

Після виконання цього фрагменту коду обоє змінні, b1 і b2, посилатимуться на один і той же об'єкт.

Цей оператор не привів до резервування якоїсь області пам'яті або копіювання першого об'єкту - просто змінна b2 посилатиметься на той же об'єкт, що і b1. Таким чином, будь-які зміни, зроблені в об'єкті через змінну b2, вплинуть на об'єкт, на який посилається b1, оскільки це один і той же об'єкт (див. рисунок 9.1).

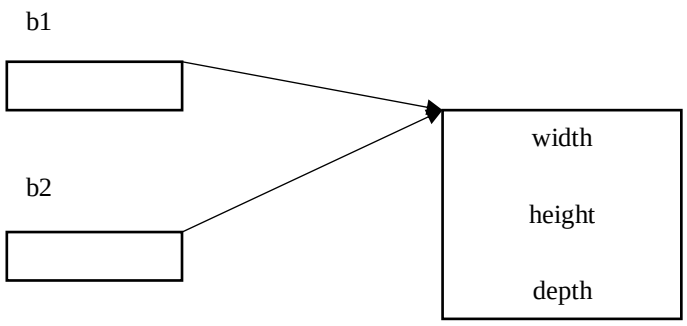


Рисунок 9.1— Привласнення посилань

Хоча змінні b1 і b2 посилаються на один і той же об'єкт, вони не пов'язані між собою ніяким іншим чином. Оператор привласнення

b1=null;

просто розірве зв'язок b1 з початковим об'єктом, не роблячи впливу на сам об'єкт або змінну b2. Змінна b2, як і раніше, вказує на початковий об'єкт.

9.9 Приховування змінної екземпляра

Як відомо, в мові Java не допускається оголошення двох локальних змінних з одним і тим же ім'ям в одній і тій же, або в тих, що включають одна іншу, зонах видимості. Треба відмітити, що можуть існувати локальні змінні, у тому числі формальні параметри методів, які співпадають з іменами *об'єктних* змінних (чи, змінних екземпляра) класу. Проте, коли ім'я локальної змінної співпадає з ім'ям змінної экземпляря, *локальна* змінна приховує змінну екземпляра.

Хоча зазвичай простіше використати різні імена для формальних параметрів і змінних екземпляра класу, існує і інший спосіб виходу з подібної ситуації. Оскільки ключове слово this дозволяє посилатися безпосередньо на поточний об'єкт, його можна використати для вирішення конфлікту простору імен. Ось так можна записати конструктор:

```
Box(double width, double height, double depth)
{
this.width=width;
this.height=height;
this.depth=depth;
}
```

Думки програмістів з приводу такого підходу різняться:

- одні - за заборону сокриття змінних;
- інші - вважають за доцільне для полегшення розуміння програм використати одні і ті ж імена.

Тема 10 Інкапсуляція. Обмеження доступу до елементів класу

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

10.1 Специфікатори доступу

Ми вже обговорювали вимоги до об’єктно-орієнтованого стилю програмування. Однією з основних вимог відзначалося обмеження доступу.

Як звісно, *інкапсуляція (encapsulation) - це приховування реалізації класу та відокремлення його внутрішнього уявлення від зовнішнього (інтерфейсу).*

При використанні об'єктно-орієнтованого підходу не прийнято застосовувати прямий доступ до властивостей будь-якого класу з інших класів.

У наших попередніх програмах ми не робили ніяких кроків щодо обмеження доступу, тобто обмеження встановлювалися за замовчуванням.

Може скластися враження, що ніяких обмежень не встановлювалося взагалі. Дійсно, у функції main() за межами класу ми зверталися до будь-якої об'єктної змінної і методу класу Goods3. Проте це не так.

Кожна програма на мові Java входить в пакет. Пакет можна організувати явно, або він буде утворений за замовчуванням. Якщо при оголошені методу, класу або об'єктної змінної не використовується ніякий специфікатор доступу, то встановлюється вільний доступ до даного елемента , але *тільки в межах пакета*. За межами пакета всі вказані імена стануть недоступними.

Зрозуміло, що доступ за замовчуванням не розв'язує задачу обмеження доступу.

У мові Java є три специфікатори доступу:

- public;
- protected;
- private.

Член класу, помічений специфікатором public, *доступний скрізь*.

Член класу, помічений специфікатором private, доступний *тільки для членів цього класу*.

Про специфікатор protected поговоримо пізніше, коли будемо розглядати успадкування.

Як використовувати специфікатори доступу?

Існує правило - об'єктні змінні (дані класу) мають бути доступні *тільки* членам цього ж класу, тоді як методи класу звичайно *відкриті* для використовування в інших класах.

Це правило не важко пояснити. Не можна маніпулювати даними, не знаючи специфіки цього процесу. Це може привести до серьозних наслідків. Задача методів – створити розумний механізм управління даними.

Наприклад, в наших класах Goods методи, встановлюючі і змінюючі дані, мають контролювати їхні значення: кількість товару і ціна не можуть бути негативними. Можливо, має сенс встановити і верхні межі для вказаних змінних. У будь-якому випадку розробник класу краще знає, як маніпулювати даними класу, ніж користувач цього класу.

Для обмеження доступу до об'єктних змінних тільки членами свого класу використовується специфікатор доступу private. Іноді його потрібно приписати деяким методам класу, якщо вони виконують *допоміжну* роль, *обслуговуючи інши методи* класу, і не використовуються для звернення до членів класу *ззовні*.

Для відкритих методів класу завжди потрібно застосовувати специфікатор public, якщо є ймовірність використовування даного класу за межами пакета, в якому він оголошений.

Перепишимо клас Goods4 , використовуючи специфікатори доступу.

```
  class Goods4
{
    private String name;
    private double price_w;
    private int quantity;
// В іншому вміст класу залишаємо незмінним
}
```

Виконаємо програму.

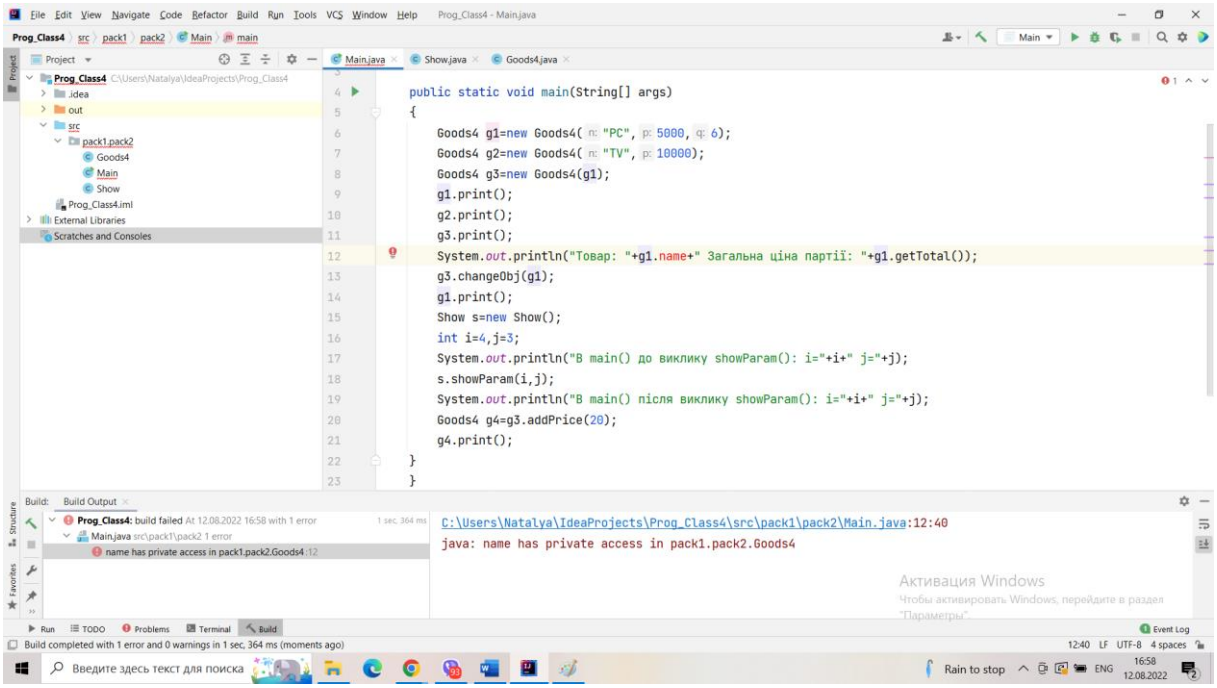


Рисунок 10.1 — Вплив специфікаторів на доступ до членів класу

Розглянемо рядок 12. Після введення специфікаторів доступу цей оператор приводить до помилки, дані класу Goods4 недоступні для зовнішнього використання.

10.2 Статичні дані та методи

Дані класів, яки використовувалися дотепер, були нерозривно пов’язані з об’єктами цих класів. Тому для них часто застосовується термін «об’єктна змінна».

Проте бувають випадки, коли деяке дане має обслуговувати всі об’єкти класу, будучи для них нібито глобальною змінною. Такі дані існують навіть тоді, коли не створено жодного об’єкта класу, для них існує спеціальний специфікатор *static* і назва – статичні дані.

Змінні екземплярів класа, оголошені як static, по суті є глобальними змінними для об’єктів цього класу.

Для роботи із статичними даними використовуються статичні методи (оголошуються як static). На відміну від звичних методів вони не пов'язані з об'єктами класу, і їх можна викликати, коли ще не створений жоден об'єкт класу.

На використовування статичних методів накладається ряд обмежень:

- статичні методи можуть використовувати тільки статичні дані;
- статичні методи можуть звертатися тільки до інших статичних методів;
- статичні методи не можуть посилатися на поточний об'єкт (this) і об'єкт базового класу.

У клас, що містить статичні дані, можна ввести спеціальний блок static, який виконуватиме обчислення за ініціалізацією статичних даних. Статичний блок виконується автоматично після запуску програми і тільки один раз.

Розглянемо приклад програми, в якій необхідно виконати розрахунок заробітної платні працівникам з почасовою оплатою праці.

Для визначення суми до видачі, слід відняти із запрацьованого прибутковий податок.

Вартість години праці для всіх працівників, що посідають однакову посаду, встановлюється працедавцем однаково і змінюється рідко.

Порогові значення зарплати і коефіцієнти для нарахування прибуткового податку також рідко змінюються.

Листинг 10.1

```
package pack1.pack2;
public class Worker
{
    private String sn;
    private int hours;
    private static double tariff=6.5;
    private static int z1=100;
    private static int z2=500;
    private static int v1=10;
    private static int v2=20;
    private static double k1;
    private static double k2;
    static
    {
        k1=(double)v1/100;
        k2=(double)v2/100;
    }
    public  Worker(String s)
    {
        sn=s;
        hours=0;
    }
    public void addHours(int h)
    {
        hours=h;
    }
    public void getSalary()
    {
```

```
        double Pn=0;
        double z;
        z=tariff*hours;
        if(z<=z1)
            Pn=0;
        if((z>z1)&&(z<=z2))
            Pn=(z-z1)*k1;
        if(z>z2)
            Pn=(z2-z1)*k1+(z-z2)*k2;
        System.out.println(sn+" Нараховано: "+z+" Податок на прибуток: "+Pn+ "
До видачі: "+(z-Pn));
        hours=0;
    }
    public static void setTax(int Z1,int Z2,double K1,double K2)
    {
        z1=Z1;z2=Z2;k1=K1;k2=K2;
    }
    public static void setTarif(double t)
    {
        tariff=t;
    }
}

public class Main {

    public static void main(String[] args)
    {
        Worker.setTarif(7);
        Worker w1=new Worker("Іванов ");
        w1.addHours(100);
        w1.getSalary();
        Worker.setTax(150, 600, 0.05, 0.2);
        w1.addHours(100);
        w1.getSalary();

    }
}
```

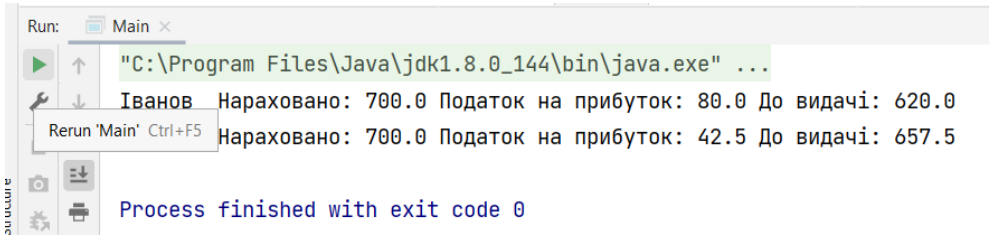


Рисунок 10.2 — Результат роботи програми

У дану програму включений статичний блок, оператори якого виконуються раніше, ніж будь який метод. Тому до обчислення прибуткового податку коефіцієнти k1 і k2 будуть вже розраховані. Використовуючи можливість статичних методів працювати із статичними даними класу до того, як був створений об’єкт класу, відбувається зміна тарифу за допомогою статичного методу setTarif().

10.3 Опис констант

Якщо перед описом змінної, що ініціалізується, поставити ключове слово *final*, то виключається можливість зміни її значення при роботі програми.

final double PI=3.14159;

final int DAYS_OF_JANUARY=31;

Звичайно імена констант записують великими літерами.

Ключове слово final використовується також при опису методів, але його застосування в цьому контексті пов'язане із спадкоємством (див. далі).

Крім полів, як final можуть бути оголошені параметри методу і локальні змінні. Оголошення параметра як final перешкоджає його зміні в межах методу. Оголошення як final локальної змінної перешкоджає присвоєнню їй значення більш ніж один раз.

10.4 Знищення об'єкта

У мові Java програміст не працює з покажчиками, не виділяє і не звільняє у явному вигляді пам'ять. Тому на відміну від інших мов, наприклад C++, в Java не передбачені деструктори для видалення об'єктів, які більше не потрібні. Задачу звільнення пам'яті, яка більше не використовується, в Java виконує складальник сміття.

Крім пам'яті об'єкти можуть використовувати і інші системні ресурси, наприклад, працювати з файлами, шрифтами та ін. Для звільнення *цих ресурсів* в класі можна використовувати спеціальний метод finalize(). Його формат:

```
protected void finalize()
{
//Оператори, що звільняють ресурси
}
```

Тут специфікатор доступу protected обмежує доступ до методу тільки членами класу (ієрархії класів).

Слід зазначити, що метод finalize() викликається *автоматично* безпосередньо перед роботою складальника сміття. Тому програміст не управляє цим процесом. Якщо необхідно звільнити ресурс негайно після його використовування, то слід написати свій код, зокрема додати метод dispose() для звільнення пам'яті.

Тема 11 Успадкування

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

Успадкування – один із найважливіших і часто використовуваних механізмів обєктно-орієнтованого програмування. Суть успадкування полягає в тому, що, використовуючи деякий клас як базовий, можна на його основі створити множину інших класів, які будуть містити *всі його змінні і методи*, а також нові члени класу, що відповідають новим вимогам. Завдякі успадкуванню значно пі-

двищується продуктивність праці програміста, є можливість створювати потужні засоби розробки програм, робити програмні продукти добре структурованими й, отже, зрозумілими і зручними в супроводі.

11.1 Суперклас і підклас. Доступ до елементів класу при успадкуванні

Базовий (батьківський) клас при програмуванні на Java прийнято називати *суперкласом,* похідні від нього (дочірні) класи – *підкласами.*

Схема створення підкласу дуже проста: якщо X – суперклас, його підклас у заголовку повинен мати фразу extends X:

```
class X
{
…….
}
class Y extends X
{
…..
}
```

Використовування спадкоємства розширює поняття про обмеження доступу до членів класу. Очевидно, що члени ієрархії класу повинні мати більше прав доступу, ніж інші об'єкти. Тому для них використовується додатковий специфікатор доступу protected, *який робить* ***відкритим*** *доступ до членів суперкласу в його підкласах*. Про доступ до членів класів, розміщених у різних пакетах, ми поговоримо пізніше.

Якщо в суперкласі використовується обмеження доступу типу private, то *слід гарантувати доступ до закритих членів класу за допомогою методів*. У результаті може значно збільшитися розмір класу за рахунок великої кількості методів установки і читання даних і знизитися швидкодія при роботі з об'єктами підкласу.

Наступний приклад ілюструє доступ до елементів класу в межах одного пакета.

```
class X
{
int a;
public int b;
private int c;
protected int d;
void setc(int c1)
{c=c1;}
int getc()
{return c;}
void setabd(int a1, int b1,int d1)
{a=a1;b=b1;d=d1;}
}
class Y extends X
{
int e;
```

```
void newc1(int c1)
{c=c1;} //Помилка! Немає доступу до змінної с
void newc2(int c1)
{setc(c1);} //Вірно! Доступ до с через метод суперкласу
void sete1()
{e=a+b+c+d;} // Помилка! Немає доступу до змінної с
void sete2()
{e=a+b+getc()+d;}
}//Вірно! Доступ до с через метод суперкласу.
```

У розглянутому прикладі в суперкласі змінна с оголошена як private. Для використання змінної с у підкласі є одна можливість – використати методи суперкласу.

Зазвичай методи забезпечують доступ до даних, які визначені класом, але це не обов'язково. Змінна об'єкту класу може бути відкритою, якщо на це є вагомі причини. Наприклад, в наших навчальних завданнях для простоти змінні екземпляра визначені як відкриті. Однак в більшості класів, застосовуваних в **реальних** програмах, маніпулювання даними повинно виконуватися **тільки** з використанням методів.

11.2 Повторний розгляд масивів

Маючи уявлення про масиви, можна зробити висновок - всі вони реалізовані як об'єкти. У зв'язку з цим існує спеціальний атрибут масиву - розмір масиву, тобто кількість елементів, які може містити масив. Він зберігається в змінній єкземпляра length.

Розглянемо вдосконалену версію класу Stack, яка дозволяє створювати стеки будь-якого розміру.

Листинг 11.1

```
class Stack
{
private int stck[];
private int tos;
//Ініціалізація верхівки стека
Stack(int size)
{
stck=new int[size];
tos=-1; //Вказує на пустий стек
}
void push(int item)
{
if(tos==stck.length-1)
   System.out.println(“Стек повний”);
else
   stck[++tos]=item;
}
int pop()
{
if(tos<0)
```

```
{
   System.out.println("Стек порожній");
   return 0;
}
else
return stck[tos--];
}
}
class TestStack2
{
public static void main(String args[])
{
Stack mystak1=new Stack(5);
Stack mystack2=new Stack(8);
for(int i=0;i<5;i++)
   mystack1.push(i);
for(int i=10;i<8;i++)
   mystack2.push(i);
System.out.println("Стек в mystack1: ");
for(int i=0;i<5;i++)
   System.out.println(mystack1.pop());
System.out.println("Стек в mystack2: ");
for(int i=0;i<8;i++)
   System.out.println(mystack2.pop());
// Ці оператори неприпустимі

mystack1.tos=-2;
mystack2.stck[3]=100;
}
}
```

11.3 Подальший розвиток класу Goods. Перекриття методів

Ми вже неодноразово використовували клас Goods з різними модифікаціями, які, однак, залишали незмінними змінні класу. Допустимо, виникла необхідність відобразити товар з погляду його продажу. Назва товара, його оптова ціна і кількість повністю підходять для нових завдань. Але цього недостатньо. Тепер додатково треба мати відомості про:

- дату виготовлення товару;
- гарантійний строк;
- роздрібну ціну.

Умовимся про спосіб представлення ціх даних. Оскільки ми ще не знаємо, як маніпулювати часом і датою, представимо дату виготовлення у вигляді рядка. Якщо домовитися, що запис дати буде виконуватися у форматі рр.мм.дд, то в нас навіть з'явиться можливість порівнювати дати. Гарантійний строк можна представити у вигляді змінної цілого типу (кількість місяців), а роздрібну ціну – змінною типу double.

Клас Goods є суперкласом, а Gods_of_sale – підклас.

Створюючи суперклас для даної програми, ми подбали про всі майбутні підкласи і зробили доступними для них змінні: name, price_w, quantity. Інших нововведень у суперкласі немає.

Розглянемо уважно перший конструктор підкласу Goods_of_sale. Типовою помилкою початкових програмістів є включення до списку параметрів конструктора тільки тих змінних, які ініціюють змінні, описані в підкласі. Потрібно пам'ятати, що підклас успадковує всі елементи свого суперкласу, тому в нашому конструкторі описані 6 параметрів.

Оскільки створюючи об'єкт підкласу необхідно ініціювати й змінні, оголошені і в суперкласі, то чому б не викликати для цієї мети конструктор класу, що вміє виконувати таке завдання. Ключове слово super у даному контексті підмінює ім'я суперкласу і позначає виклик конструктора . Оператор виклику конструктора суперкласу завжди повинен бути *першим* у тілі конструктора підкласу.

У підкласі Goods_of_sale є й конструктор копіювання, що також звертається до суперкласу для ініціалізації успадкованих змінних.

Для виводу на екран даних про об'єкт класу також довелося створювати новий метод. За своєю суттю він аналогічний методу print() суперкласу і тому одержав таке ім'я. Мова Java допускає *перекриття* членів суперкласу в підкласі. Однак для виводу на екран членів суперкласу бажано викликати метод print() суперкласу. Для усунення неоднозначності знову використане ключове слово super. Тепер воно указує, що метод print() належить суперкласу.

Проаналізуємо використання класів.

В одній програмі можна створювати скільки завгодно об'єктів різних класів. Об'єкт підкласу може використовувати рівною мірою методи, оголошені як у своєму класі, так і у суперкласі.

Наш ланцюжок успадкування складається тільки із двох класів, але його можна продовжувати, створивши, наприклад, підклас для продуктів живлення, де будуть представлені дані про строки й умови зберігання, виготовлювача продукту та інша необхідна інформація, а також відповідні методи.

11.4 Сумісність об'єктів в ієрархії успадкування

Ми встановили, що об'єкт підкласу містить усе, що є об'єктом суперкласу і *додатково* може мати свої дані й методи. Із цього положення випливає, що посиланню на об'єкт суперкласу завжди можна привласнити об'єкт будь-якого підкласу в ієрархії успадкування. Зворотна дія неприпустима, оскільки об'єкт суперкласу, не може забезпечити значеннями об'єкт підкласу.

Листинг 11.2

```
package pack1.pack2;
public class Goods
{
    protected String name;
    protected double price_w;
    protected int quantity;
    public Goods(String n,double p_w,int q)
    {
```

```
        name=n;price_w=p_w;quantity=q;
    }
    public Goods(String n,double p_w)
    {
        name=n; price_w=p_w; quantity=1;
    }
    public void print()
    {
        System.out.println("Товар: "+name+" Ціна закупівлі: "+price_w+"
Кількість товару: "+quantity);
    }
    public double getTotal()
    {
        return price_w*quantity;
    }
    public void setQuantity(int q)
    {
        quantity=q;
    }
    public void setPrice_w(double p)
    {
        price_w=p;
    }
    String getName()
    {
        return name;
    }

}


public class Goods_of_sale extends  Goods
{
    protected String date;
    protected double price_r;
    protected int guaranty;
    public Goods_of_sale(String n,double p_w,String d,double p_r,int g)
    {
        super(n,p_w);
        date=d; price_r=p_r;guaranty=g;
    }
    public Goods_of_sale(Goods_of_sale g)
    {
        super(g.name,g.price_w,g.quantity);
        date=g.date; price_r=g.price_r;guaranty=g.guaranty;
    }
    public Goods_of_sale(String n,double p_w,int q,String d,double p_r,int g)
    {
        super(n,p_w,q);
        date=d; price_r=p_r;guaranty=g;
    }
    public void print()
    {
        super.print();
        System.out.println("Дата виготовлення: "+date+"Роздрібна ціна:
"+price_r+ " Гарантійний строк: "+guaranty);
    }
    public double getProfit()
    {
        return (price_r-price_w)*quantity;
    }
    public void setPrice_r(double p_r)
    {
```

```
        price_r=p_r;
    }

}

public class Main {

    public static void main(String[] args)
    {
        Goods g1=new Goods("Personal computer",3000,3);
        g1.print();//1
        Goods_of_sale g2=new Goods_of_sale("Tv",1000,2,"09.01.19",1200,4);
        g2.print();
        Goods_of_sale g3=new Goods_of_sale(g2);
        g3.print();
        g2.setPrice_r(1150);
        g2.setQuantity(10);
        g2.print();
        System.out.println("Товар: "+g2.getName()+"Вартість партії:
"+g2.getTotal()+ " Очікуваний прибуток: "+g2.getProfit());
        System.out.println("Сумісність: ");
        Goods g; //посилання на тип класу Goods
        g=g1;
        g.print();//2

        g=g2;
        g.print();


        Goods_of_sale obj;
        // obj=g1; //Помилка

    }
}
```

В операторе `Goods g;` оголошується посилання на об’єкт суперкласу Goods, але ініціалізація об’єкта не виконується. Нижче `g` ініціалізується об’єктом `g1`. У коректності цієї операції можна переконатися за допомогою операторів, помічених коментарями 1 та 2:

```
g1.print();
g.print();
```

Далі в коді посиланню привласнюється

```
g=g2;
```

Виконання оператора

```
Goods_of_sale obj;
obj=g1;
```

приведе до помилки, оскільки посиланню на об’єкт підкласу неможна привласнювати значення посилання на об’єкт суперкласу.

11.5 Порядок виклику конструкторів в ієрархії класів

Розглядаючи ієрархію класів Goods і Goods_of_sale було сказано, що в тілі конструктора підкласу ключове слово super, що дозволяє викликати конструктор суперкласу, має бути першим. Загальне правило виклику конструкторів

таке: в ієрархії класів конструктори викликаються в порядку успадкування від суперкласу до підкласу.

Наприклад, код програми ConstructorsXYZ:

```
class X
{
X()
{
System.out.println("Конструктор класу X");
}
}
class Y extends X
{
Y()
{
System.out.println("Конструктор класу Y");
}
}
class Z extends Y
{
Z()
{
System.out.println("Конструктор класу Z");
}
}
class Prog_constructor
{
public Prog_constructor()
{  }
public static void main(String args[])
{
Z z=new Z();
}
}
```

На екрані побачимо:

Конструктор класу X

Конструктор класу Y

Конструктор класу Z

Якщо метод super() при реалізації конструктора підкласу не застосовується, програма використовує конструктор кожного суперкласу, заданий за замовчуванням або конструктор, який не містить параметрів.

Тема 12 Динамічне зв‘язування. Поліморфізм. Абстрактні класи

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

12.1 Динамічне зв'язування

Якщо метод є конструктором, статичним, закритим або термінальним (при оголошенні такого методу використовується ключове слово final), то *компілятор* точно знає, який метод потрібно викликати. Визначення посилань на методи в процесі *компіляції* називається *статичним зв'язуванням*.

Правила сумісності об'єктів при спадкуванні й перекриття методів лежать в основі динамічного зв'язування.

При *динімічному* зв'язуванні (тобто у *процесі* виконання програми) віртуальна машина повинна викликати одну з версій *перекритого* методу, що відповідає *фактичному* типу об'єкта, на який посилається змінна. Наприклад:

Листинг 12.1 Програма DynamicLinking

```
class X
{
void show()
{
System.out.println("Метод show() класу X");
}
}
class Y extends X
{
void show()
{
System.out.println("Метод show() класу Y");
}
}
class Z extends Y
{
void show()
{
System.out.println("Метод show() класу Z");
}
}
class Prog_dyn
{
public static void main(String args[])
{
X x1=new X();
Y y1=new Y();
Z z1=new Z();
X x2;
x2=x1;
x2.show();
x2=y1;
x2.show();
x2=z1;
x2.show();
}
}
```

На екрані побачимо:

Метод show() класу X

Метод show() класу Y

Метод show() класу Z

Відповідно до правила сумісності об’єктів при спадкуванні змінної x2 можна привласнити посилання на будь-який об’єкт свого класу або підкласу. Метод show() якої реалізації буде викликатись, залежить від того, на який об’єкт посилається змінна x2.

12.2 Поліморфізм

Динамічне зв’язування забезпечує реалізацію однієї з основних концепцій ООП – *поліморфізм.*

Сутність поліморфізму полягає в тім, що в суперкласі визначається ряд методів, які будуть ***перевизначатись (перекриватись)*** в підкласах. Чому ці методи не можна записати в такому вигляді, що не вимагав би перевизначення?

Хоча б тому, що розроблювач суперкласу не знає про всі можливі використання цього класу в подальшому. Крім того, потреб може бути дуже багато і тоді наш суперклас стане дуже величезним.

У підкласах суперкласу зазначені методи перевизначаються (перекриваються) залежно від потреб конкретного підкласу. У результаті посилання на суперклас, посилаючись на той, або інший об’єкт підкласів, «захоплює» різні методи й фактично вирішує завдання *спеціалізації* методів для різних класів ієрархії.

12.3 Абстрактні класи

Часто в основі різних ієрархій класів лежать класи, для яких не створюють об’єктів, але використання яких, безумовно, корисно.

Наприклад, поставимо за мету створити систему класів для представлення всіх людей, що забезпечують функціональність університету. У нас з’являться класи:

- «студент»;
- «викладач»;
- «обслуговуючий персонал»;
- «науковець»;
- і т. п.

При близькому розгляді перерахованих класів не важко виявити, що в них є загальні змінні і методи. Наприклад, змінні «ПІБ», «адреса», «рік народження», «номер телефону», а також методи «змінити адресу», «змінити номер телефону» є загальними для всіх зазначених класів.

Тому має сенс створити суперклас «людина», що буде містити загальні змінні та методи. На користь створення суперкласу говорять, принаймні, два міркування:

- підкласи стануть простіше;
- суперклас об’єднає інші класи в ієрархію, що завжди корисно для розуміння і подальшого розвитку складних систем.

Для запропонованого нами класа «людина» не має сенсу у системі керування університетом створювати об’єкти - вони не зможуть виконувати які небудь функції.

Отже, по суті клас «людина» у контексті керування університетом буде *абстрактним класом.* Однак існують і формальні ознаки абстрактного класу. Для цього треба використати модифікатор abstract у заголовку класу або його методів.

Наприклад:

```
abstract class Person
{
String Surname;
..............
public void setAddr(String str)
{..............}
public abstract String getInfo();
.................
}
```

Тут метод getInfo() оголошений абстрактним, оскільки тієї інформації, що нам може дати цей метод, явно недостатньо як відомості, наприклад, про студента або викладача.

Якщо метод оголошений абстрактним, його можна *не реалізовувати*. Вочевидь, що такий метод обов’язково потрібно буде перекрити (перевизначити) у підкласі. Разом з тим, абстрактний клас може містити методи, які будуть без перекриття переходити в підкласи, наприклад, setAddr() – визначення нової адреси.

Як приклад застосування абстрактних класів розглянемо таку ієрархію класів.

Суперкласом є клас «будова» (building). Вочевидь, кожна будова має:

- адресу (addr);
- займану площу (area);
- власника (owner);
- вартість (price).

Однак цих даних недостатньо для того, щоб, наприклад, купити будову, відремонтувати її або виконати інші подібні дії, пов’язані з реальним об’єктом. Тому має сенс вважати «будову» абстрактним класом.

Підкласом будови може бути «житловий будинок» (house). Якщо в цьому класі з’являться змінні, що встановлюють:

- кількість поверхів(floor);
- квартир усього (apart);
- що перебувають в експлуатації (occup);
- а також кількість мешканців (lodger).

то вже можна говорити про створення об’єктів такого класу.

Підкласом будови також може бути спортивне спорудження (sport). Будь-яке спортивне спорудження можна охарактеризувати видами спорту (types) і кількістю місць для глядачів (seat). У такому вигляді підклас «спортивна споруда» також буде абстрактним.

Підкласом «спортивної споруди» може стати реальний діючий стадіон, якщо додати назву спортивної споруди, назву спортивного заходу, кількість проданих квітків.

Листинг 12.2 Програма Abstract_cl

```
package pack1.pack2;

public abstract class building
{
    protected String addr;
    protected double area;
    protected double price;
    protected String owner;
    public building(String addr1, double area1,double price1, String owner1)
    {
        addr=addr1;
        area=area1;
        price=price1;
        owner=owner1;
    }
    public void setPrice(double price1)
    {
        price=price1;
    }
    public String getAddr()
    {
        return addr;
    }
    public abstract void getDescription();

}
public class house extends building {

    protected int apart;
    protected int floor;
    protected int occup;
    protected int lodger;

    public house(String addr1, double area1, double price1, String owner1, int
apart1, int floor1) {
        super(addr1, area1, price1, owner1);
        apart = apart1;
        floor = floor1;
        occup = 0;
        lodger = 0;
    }

    public void leaseApart(int occup1, int lodger1) {
        occup += occup1;
        lodger += lodger1;
    }

    public void freeApart(int occup1) {
        occup -= occup1;
    }
```

```
    public void getDescription() {
        System.out.println("Житловий будинок за адресою " + addr + " Власник: "
+ owner + " Ціна: " + price);
        System.out.println(floor + " поверхів, " + apart + " квартир, " +
lodger + " мешканців");
    }
}

public abstract class sport extends building
{
    protected String types;
    protected int seat;
    public sport(String addr1, double area1,double price1, String owner1,String
types1,int seat1)
    {
        super(addr1,area1,price1,owner1);
        types=types1;
        seat=seat1;
    }
    public abstract void getDescription();
    public void setType(String types1)
    {
        types=types1;
    }
}
public class stadium extends sport
{
    protected String type;
    protected String name;
    protected int spectator;
    stadium(String addr1, double area1,double price1, String owner1,String
types1,int seat1,String name1)
    {
        super(addr1,area1,price1, owner1,types1,seat1);
        type="Немає змагань";
        name=name1;
        spectator=0;
    }
    public void setContest(String type1,int spectator1)
    {
        type=type1;
        spectator=spectator1;
    }
    public void getDescription() {
        System.out.println("Стадион " + name + " за адресою: " + addr + ",
Кількість місць: " + seat + " Власник: " + owner + " Ціна: " + price);
        System.out.println("Змагання за видами спорту: " + types);
    }
}
public class Main {

    public static void main(String[] args)
    {
        house h1=new house("Фонтанська дорога, 50", 2500, 3000000,  "Банк
Південний", 60,12);
        h1.leaseApart(15,45);
        h1.getDescription();
        stadium st1=new stadium("Канатна, 62 ", 10000, 300000, "ЗАС Спорт для
всіх", "футбол, легка атлетика", 20000," Спартак ");
        st1.getDescription();

    }
}
```

12.4. Використання ключового слова final для запобігання спадкуванню

Із застосуванням ключового слова final для створення констант ми вже познайомилися раніше. Друге призначення цього ключового слова – заборонити перевизначення методів або успадкування класів. Схема програми, що наведена нижче, показує, як можна заборонити перекриття методів при спадкуванні.

```
class X
{
public final void method1()
{
.....................................
}
}
class Y extends X
{
public final void method1() //Помилка! Не можна перекрити метод
{
.....................................
}

}
```

Запис ключового слова final перед описом класу забороняє його успадкування.

```
final class X
{.................}
class Y extends X //Помилка. Клас X не може мати підкласів
{.................}
```

Метод і клас, для яких при оголошені застосоване ключеве слово final називаються відповідно *термінальним методом* і *термінальним класом.* Однак змінні термінального методу не є константами.

Для оголошення методу або класу термінальними існує дві причини:

- ефективність роботи програми. Якщо метод оголошений термінальним, то для нього використовується *статичне* зв'язування, що заощаджує час порівняно з динамічним зв'язуванням. Крім цього, оскільки термінальний метод не може бути перекритий методом підкласу, компілятор має можливість вставляти код цього методу в точку його виклику в програмі *(inline-метод*), що також може істотно скоротити час виконання програми;
- безпека. Механізм динамічного зв'язування в деяких ситуаціях може створити проблеми для програміста через неочевидність – який саме з перекритих методів буде викликаний у конкретному випадку. Якщо використовуваний метод оголосити термінальним, то такої ситуації ніколи не виникне.

12.5 Клас Object

У мові Java є *особливий* клас Object, що є суперкласом ***для всіх інших класів, у тому числі і тих, що ми створювали раніше***. Із сказаного випливає, що змінна типу Object може посилатися на об'єкт будь-якого класу.

Розглянемо основні методи класу Object, наведені в таблиці 12.1.

Таблиця 12.1 — Основні методи класу Object

Метод	Призначення
Object clone()	Створює новий об'єкт як дублікат об'єкта, для якого викликаний метод
boolean equals(Object об'єкт)	Порівнює два об'єкти
void finalize()	Викликається для звільнення ре–сурсів
final Class<?> getClass()	Визначає клас даного об'єкта
int hashCode()	Повертає хеш-код, пов'язаний з об'єктом
final void notify()	Відновляє виконання процесу, що очікує виклик об'єкта
final void notifyAll()	Відновляє виконання всіх процесів, що очікують виклик об'єкта
String toString()	Повертає рядок з описом класу об'єкта
final void wait()	Очікує виконання іншого процесу
final void wait(long мілісекунди)	
final void wait(long мілісекунди, int наносекунди)	

Методи getClass(), notify(), notifyAll(), wait() оголошені термінальними. Інші методи можуть бути перекритими. Найчастіше використовуються методи finalize(), equals() і toString().

Метод equals() класу Object перевіряє, *чи займають порівняні об'єкти ту саму область пам'яті.* Така перевірка не має практичної цінності, тому метод equals() звичайно перекривають у підкласах.

Метод toString() класу Object виводить на друкування ім'я класу і його адресу.

12.6 Методи valueOf() і toString()

При зчепленні символьних рядків в Java дані інших типів перетворюються *в їх строкове представлення* шляхом виклику одного з перевантажених варіантів методу valueOf() класу String.

Метод String.valueOf() викликається в тому випадку, *якщо потрібно строкове представлення деякого іншого типу даних*, наприклад, в операціях зчеплення символьних рядків. Метод valueOf() перетворює дані з внутрішнього представлення в легку для читання форму. Цей статичний метод перевантажується *в класі String для всіх вбудованих* в Java типів даних таким чином, *щоб кожен тип був правильно перетворений в символьний рядок*. ***Метод valueOf() перевантажується і для типу Object, тому об'єкт типу будь-якого створюваного класу також може використовуватися в якості аргументу.***

Нижче наведені деякі форми методу valueOf():

static String valueOf(double число)

static String valueOf(long число)

static String valueOf(Object об’єкт)

static String valueOf(char символи[])

Зокрема, для елементарних типів даних метод valueOf() повертає символьний рядок, що містить чіткий еквівалент того значення, з яким він був викликаний. А *для об'єктів метод valueOf() викликає метод toString()* класа цього об'єкта як зручний засіб для строкового подання об'єктів створюваних класів.

Метод toString() є в кожному класі, оскільки він визначений в класі Object. Але реалізація методу toString() за умовчанням рідко виявляється корисною. Тому у всіх найбільш важливих створюваних класах метод toString(), швидше за все, *доведеться перевизначати, щоб забезпечити в кожному з них строкове представлення об’єкта.*

Щоб реалізувати цей метод в своєму класі, досить повернути об'єкт типу String, що містить легкий для читання символьний рядок, який належним чином описує об'єкт даного класу.

Перевизначення методу toString() в створюваних класах дозволяє пов–ністю інтегрувати їх в середовище програмування на Java. Наприклад, перевизначені варіанти методу toString() можна застосовувати в printf() і println(), а також в операціях зчеплення символьних рядків з даними інших типів.

Наприклад:

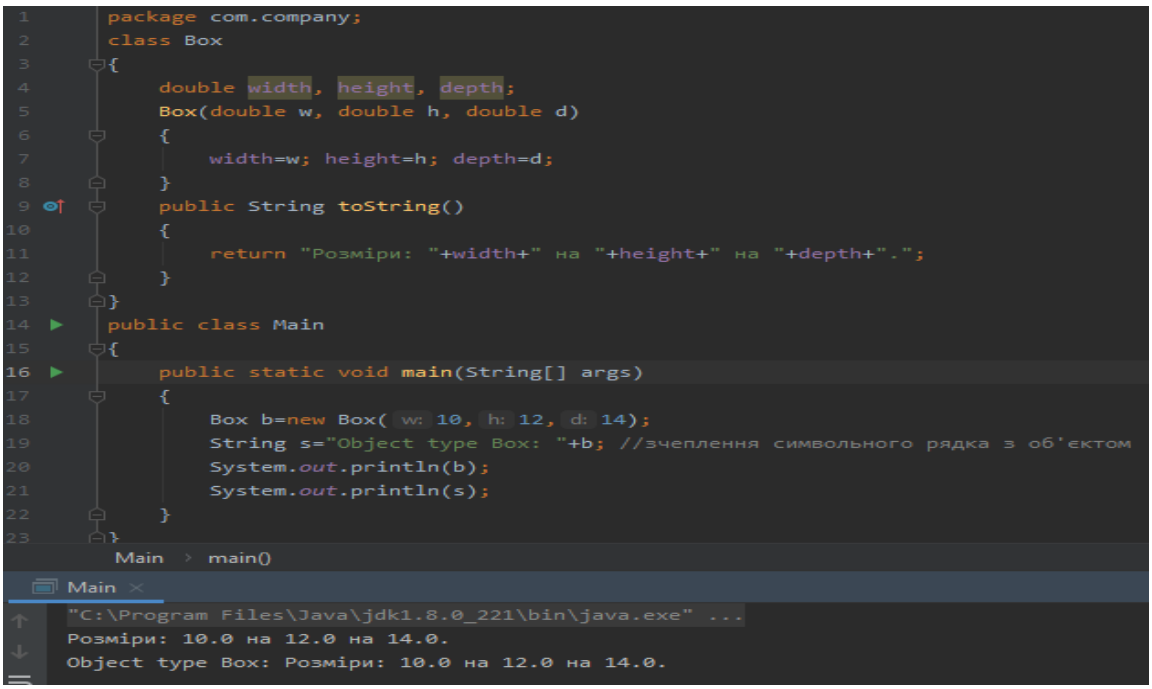


Рисунок 12.1 — Перевантаження метода toString() в класі Box

Тема 13 Інтерфейси

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

13.1 Оголошення інтерфейсу

Інтерфейси – один з найефективніших і цікавиших засобів програмування на мові Java. Деякою мірою інтерфейси поєднують в собі властивості абстрактних класів, множинного спадкоємства (якого немає в мові Java) і реалізують ідеї поліморфізму.

Сутність інтерфейсу полягає у тому, що є можливість створити конструкцію, подібну класу, але без реалізації методів. Методи передбачається реалізовувати надалі в класах, до яких підключатиметься даний інтерфейс. Кількість реалізацій одного інтерфейсу може бути будь-якою.

Синтаксис оголошення інтерфейсу:

специфікатор_доступу interface ім’я_інтерфейсу

{

тип кінцева_змінна1=значення;

тип кінцева_змінна2=значення;

……...

тип кінцева_змінnaN=значення;

тип ім’я_методу1(список_параметрів);

тип ім’я_методу2(список_параметрів);

……..

тип ім’я_методуM(список_параметрів);

}

Тут як специфікатор доступу можна записати *public, або нічого*. За наявності специфікатора public всі методи інтерфейсу також неявно мають специфікатор public. За відсутності специфікатора доступу інтерфейс можна використовувати тільки в межах того пакета (див. далі**)**, в якому він описаний.

Змінні, описані в інтерфейсі, *завжди* мають опис final і static. Їхні значення, задані в опису інтерфейсу, не можуть бути змінені при його реалізації.

13.2 Реалізація інтерфейсу

Реалізація інтерфейсу полягає у тому, що до деякого класу підключається інтерфейс, і у цьому разі в даному класі мають бути реалізовані методи, оголошені в інтерфейсі. Якщо методи реалізовані не повністю, то клас буде абстрактним.

Синтаксис класу, що реалізовує інтерфейс, має вигляд:

спеціфікатор_доступу class ім’я_класу [extends суперклас] implements ім’я_інтерфейсу[,ім’я_інтерфейсу …]

{

//опис власних змінних класу

……..

//опис власних методів класу

…….

//опис методів, оголошених в інтерфейсі

…….

}

Як приклад реалізації інтерфейсу розглянемо таку задачу.

Хай інтерфейс IntTurn задає дві операції для роботи з чергою цілих чисел:

- додавання елементу в чергу place();
- видалення елементу з черги goOut().

Клас FixedTurn реалізує чергу з фіксованим числом елементів, яке задається при створенні об’єкта класу. При переповнюванні черги видається відповідне повідомлення.

Клас DynTurn реалізує чергу, в якій переповнювання не відбувається за рахунок додаткового виділення пам’яті усякий раз, коли для чергового елементу, що додається, не вистачає місця (див. листинг 13.1).

Листинг 13.1:

```
interface intTurn
{
    void place(int el);
    int goOut();
```

```
}
class FixedTurn implements intTurn
{
    private int turn[];
    private int tail;
    public FixedTurn(int size)
    {
        turn=new int [size];
        tail=-1;
    }
    public void place(int el)
    {
        if(tail==turn.length-1)
            System.out.println("V ocheredi net mesta!");
        else
        turn[++tail]=el;
    }
    public  int goOut()
    {
        int i,k;
        if(tail<0)
        {
            System.out.println("Ochered pusta");
            return 0;
        }
        else
        {
            k=turn[0];
            for(i=1;i<=tail;i++)
                turn[i-1]=turn[i];
            tail--;
           return k;
        }
    }
}
    class DynTurn implements intTurn
    {
        private int turn[];
        private int tail;
        public DynTurn(int size)
        {
            turn=new int [size];
            tail=-1;
        }
        public void place(int el)
        {
            if(tail==turn.length-1)
            {
                int mas[]=new int[turn.length+1];
                for(int i=0;i<=turn.length-1;i++)
                    mas[i]=turn[i];
                turn=mas;
            }
```

```
            turn[++tail]=el;
        }
        public int goOut()
        {
            int i,k;
            if(tail<0)
            {
                System.out.println("Ochered pusta!");
                return 0;
            }
            else
            {
                k=turn[0];
                for(i=1;i<=tail;i++)
                    turn[i-1]=turn[i];
                tail--;
                return k;
            }
        }
    }



public class MyClass {

    public MyClass() {
    }
  public static void main(String args[])
  {
      FixedTurn t1=new FixedTurn(3);
      t1.place(1);
      t1.place(2);
      t1.place(3);
      t1.place(4);
      System.out.print(t1.goOut()+" ");
      t1.place(4);
      System.out.print(t1.goOut()+" ");
      System.out.print(t1.goOut()+" ");
      System.out.println(t1.goOut()+" ");
      DynTurn t2=new DynTurn(1);
      t2.place(100);
      t2.place(200);
      t2.place(300);
      t2.place(400);
      System.out.print(t2.goOut()+" ");
      System.out.print(t2.goOut()+" ");
      System.out.print(t2.goOut()+" ");
      System.out.print(t2.goOut()+" ");

  }
}
```

13.3 Посилання на інтерфейс

Не можна створити об'єкт типу інтерфейс, але можна створити посилання на інтерфейс. Корисною властивістю посилання на інтерфейс є можливість набувати значення будь-якого об'єкта класу, що реалізовує даний інтерфейс.

Перепишемо метод main() класу MyClass, увівши посилання p на інтерфейс intTurn. Посилання послідовно набуває значення об'єкту класа FixedTurn і DynTurn.

Результат виконання програми виявляється таким же, як і в попередньому прикладі.

```
public class MainClass {

    public MainClass() {
    }
  public static void main(String args[])
  {
intTurn p;
FixedTurn t1=new FixedTurn(3);
p=t1;
p.place(5);
p.place(10);
p.place(15);
p.place(20);
System.out.println(p.goOut()+" ");
p.place(20);
System.out.println(p.goOut()+" ");
System.out.println(p.goOut()+" ");
System.out.println(p.goOut()+" ");
DynTurn t2=new DynTurn(1);
p=t2;
p.place(100);
p.place(200);
p.place(300);
p.place(400);
System.out.println(p.goOut()+" ");
System.out.println(p.goOut()+" ");
System.out.println(p.goOut()+" ");

}
}
```

13.4 Реалізація в одному класі декількох інтерфейсів

Один клас може реалізовувати необмежену кількість інтерфейсів і це вирішує проблему відсутності в мові Java множинного успадкування.

Додамо в нашу програму інтерфейс з одним методом, який має визначити, чи відповідає задане ціле число деяким умовам.

Нижче наведена програма (див. листинг 13.2), в якій клас реалізує відразу два інтерфейси.

Листинг 13.2

```
interface intTurn
{
    void place(int el);
    int goOut();
}
interface Filter
{
    boolean filt(int el);
}

    class DynTurn implements intTurn, Filter
    {
        private int turn[];
        private int tail;
        public DynTurn(int size)
        {
            turn=new int [size];
            tail=-1;
        }
        public void place(int el)
        {
            if(filt(el))
          {
              if(tail==turn.length-1)
            {
                int mas[]=new int[turn.length+1];
                for(int i=0;i<=turn.length-1;i++)
                    mas[i]=turn[i];
                turn=mas;
            }
            turn[++tail]=el;
        }
      }
        public int goOut()
        {
            int i,k;
            if(tail<0)
            {
                System.out.println("Ochered pusta!");
                return 0;
            }
            else
            {
                k=turn[0];
                for(i=1;i<=tail;i++)
                    turn[i-1]=turn[i];
                tail--;
                return k;
            }
```

```
        }
  public      boolean filt(int el)
        {
            if(el>99 && el<1000)
                return true;
            else
            {
                System.out.println("Element    ne    dobavlen    v
ochered!");
                System.out.println("Element   "+el+"   vixodit   za
granitsi dopustimogo diapazona znacheniy");
                return false;
            }
        }

    }

public class MyClass {

    public MyClass () {
    }
   public static void main(String args[])
  {

      DynTurn t2=new DynTurn(1);
      t2.place(100);
      t2.place(99);
      t2.place(200);
      t2.place(199);
      t2.place(300);
      t2.place(299);
      t2.place(400);
      t2.place(1000);

       System.out.print(t2.goOut()+" ");
      System.out.print(t2.goOut()+" ");
      System.out.print(t2.goOut()+" ");
      System.out.print(t2.goOut()+" ");

  }
}
```

13.5 Спадкоємство інтерфейсів

Подібно класам один інтерфейс може успадковувати інший інтерфейс. В цьому випадку похідний інтерфейс одержує від базового інтерфейсу всі оголошені в ньому змінні і методи. Синтаксис спадкоємства інтерфейсів такий же, як і для спадкоємства класів.

Внесемо зміни в попередній приклад, створивши ієрархію інтерфейсів. Хай базовим інтерфейсом буде intTurn, а похідним - Filter. У цьому випадку клас

DynTurn реалізовуватиме один інтерфейс Filter. Якщо використовувати цей клас як ми це робили в попередньому прикладі, то результат буде ідентичним.

13.6 Змінні інтерфейсу

Як вже оголошувалося раніше, в інтерфейсі можна описувати змінні, які за умовчанням будуть final і static. Ці змінні можуть використовуватись як константи в класах, що реалізовують відповідний інтерфейс.

Опишемо в інтерфейсі Filter для нашого приклада дві змінні EXCEPT1 і EXCEPT2 з певними значеннями, які з деяких міркувань не можна поміщати в чергу. Створимо в класі DunTurn новий метод exept(), де порівнюватимо кожне число із значеннями EXCEPT1 і EXCEPT2 і внесемо відповідні зміни в метод place() (див. листинг 13.3).

Листинг 13.3

```
/*
 * Interface_all.java
*/

interface intTurn
{
    void place(int el);
    int goOut();
}
interface Filter extends intTurn
{
    boolean filt(int el);
    int EXCEPT1=199;
    int EXCEPT2=299;
}

    class DynTurn implements Filter
    {
        private int turn[];
        private int tail;
        public DynTurn(int size)
        {
            turn=new int [size];
            tail=-1;
        }
        public void place(int el)
        {
            if(filt(el) && exept(el))
          {
              if(tail==turn.length-1)
            {
                int mas[]=new int[turn.length+1];
                for(int i=0;i<=turn.length-1;i++)
                    mas[i]=turn[i];
                turn=mas;
            }
```

```
            turn[++tail]=el;
        }
      }
        public int goOut()
        {
            int i,k;
            if(tail<0)
            {
                System.out.println("Черга порожня!");
                return 0;
            }
            else
            {
                k=turn[0];
                for(i=1;i<=tail;i++)
                    turn[i-1]=turn[i];
                tail--;
                return k;
            }
        }
  public      boolean filt(int el)
        {
            if(el>99 && el<1000)
                return true;
            else
            {
                System.out.println("Елемент не може бути доданий в
чергу!");
                System.out.println("Елемент  "+el+"  виходить  за
межі припустимого диапазону значень");
                return false;
            }
        }
 public       boolean exept(int el)
        {
            if(el==EXCEPT1 || el==EXCEPT2)
            {
                System.out.println("Елемент не може бути доданий в
чергу!");
                System.out.println("Едемент "+el+" співпадає з не-
припустимим значенням");
                return false;
            }
            return true;
        }
    }

public class Interface_all {

    /** Creates a new instance of Interface_all */
    public Interface_all() {
    }
   public static void main(String args[])
```

```
{

    DynTurn t2=new DynTurn(1);
    t2.place(100);
    t2.place(99);
    t2.place(200);
    t2.place(199);
    t2.place(300);
    t2.place(299);
    t2.place(400);
    t2.place(1000);
         System.out.print(t2.goOut()+" ");
    System.out.print(t2.goOut()+" ");
    System.out.print(t2.goOut()+" ");
    System.out.print(t2.goOut()+" ");
     }}
```

Іноді має сенс створювати інтерфейси, які містять тільки описи змінних. Такі інтерфейси є *списками констант*, які можна підключати до різних класів.

Тема 14 Пакети

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

14.1 Призначення пакетів. Доступ до елементів класів

Пакети служать для ізольованого зберігання класів. Вони розв'язують дві важливі задачі. Поперше, *створюють простори імен класів, обмежені рамками пакета.* Це дозволяє програмісту, який розробляє деякий модуль програми, розташований в створених їм пакетах, не піклуватися про узгодження введених імен з іменами класів, що знаходяться в інших пакетах, і які знадобиться використовувати. Подруге, *обмежують доступ до елементів класу рамками пакета.* Природно припустити, що набір класів, що входять в один пакет, розробляється одним програмістом або групою програмістів, тісно взаємодіючих. Тому *елементи одного пакета мають право мати привілейований доступ до членів класів цього ж пакета.* Пакети особливо важливі при створенні бібліотек класів. Вони запобігають плутанині імен класів, визначених у бібліотеці, з іменами, визначеними в програмі. ***Пакети, по суті, це угрупування класів***. Пакети - це контейнери для класів. Вони використовуються для збереження ізоляції простору імен класу. Наприклад, пакет дозволяє створити клас List, не турбуючись про те, що в іншому пакеті теж може бути клас List. Всі варіанти доступу до членів класу в одному і різних пакетах наведені в таблиці 14.1.

Таблиця 14.1. — Прямий доступ до елементів класу

	private	без специфікатора	protected	public
Той же клас	Доступний	Доступний	Доступний	Доступний
Підклас в тому ж пакеті	Недоступний	Доступний	Доступний	Доступний
Не підклас в тому ж пакеті	Недоступний	Доступний	Доступний	Доступний
Підклас в іншому пакеті	Недоступний	Недоступний	Доступний	Доступний
Не підклас в іншому пакеті	Недоступний	Недоступний	Недоступний	Доступний

14.2 Визначення пакета

Для створення пойменованого пакета достатньо на початку файла програми записати оператора package у вигляді

package ім’я_пакета;

У даному випадку файл, що містить такий опис пакета, а також і інші файли, що належать цьому ж пакету, мають зберігатися в одному каталозі з ім’ям пакета.

Наприклад, файли, де визначені оператори вигляду

package Pack1;

усі повинні знаходитися в каталозі Pack1.

Є можливість створити ієрархію пакетів. Для цього потрібно записати послідовність пакетів, розділених крапками:

package ім’я_пакета[.ім’я_пакета[.ім’я_пакета[…]]];

Пакет, визначений як

package level1.level2.level3

відповідає ієрархії каталогів

level1\level2\level3

14.3 Оператор import

У Java всі вбудовані класи зберігаються в пойменованих пакетах. Оскільки при зверненні до класу необхідно вказати ім’я пакета, в якому він знаходиться, то ланцюжки з імен і крапок можуть бути достатньо довгими (3 – 4 імені). Оператор import дозволяє позбавити програміста від набору довгих імен. Оператор має такий формат:

import pkg1[.pkg2. …].ім’я_класа|*;

Тут метасимвол «|» розшифровується як символ «або». Якщо в кінці оператора стоїть ім'я, то це буде ім'я класу, що імпортується, до якого можна звертатися без префікса. Якщо в кінці стоїть символ «*», то це означає, що імпортуються всі класи пакета.

Всі *стандартні* класи Java зберігаються в пакеті з ім'ям java. Основні *функції мови* знаходяться в пакеті lang, який входить в пакет java (java.lang). Враховуючи те, що цей пакет використовується дуже часто, він неявно імпортується у всі програми компілятором.

При використовуванні операторів імпорту можуть виникати конфлікти імен. Якщо в двох імпортованих пакетах є класи з однаковими іменами, то при використовуванні такого класу компілятор видасть повідомлення про помилку. Для усунення конфлікту потрібно буде записати оператора з явною вказівкою використовуваного класу.

Для демонстрації пакетів і прав доступу розглянемо такий приклад програми (див. листинг 14.1).

У пакеті pack1 визначені три класи:

- Prot;
- Deriv;
- X.

Клас Deriv є підкласом класу Prot. Клас X не є ні підкласом, ні суперкласом.

У класі Prot визначені чотири змінні зі всіма можливими правами доступу. Класи Deriv і X використовують всі ці змінні наскільки це можливо.

У другому пакеті pack2 визначені три класи:

- Prot2;
- Demo;
- Y

Клас Prot2 є підкласом класу Prot, а клас Demo не залежить від Prot. Ці класи також у міру можливості використовують змінні класу Prot. Розглянемо приклад програми (див. листинг 14.1).

Листинг 14.1

Пакет pack1

Файл Prot.java

```
package pack1;
package pack1;
public class Prot
{
    int n=1;
    private int n_pri=2;
    protected int n_pro=3;
    public int n_pub=4;
```

```
    public Prot()
    {}
    public static void main(String[] args)
    {
        Prot ob1=new Prot();
        Deriv ob2=new Deriv();
        X ob3=new X();
        pack2.Demo ob4=new pack2.Demo();
    }
}
```

Файл Deriv.java

```
package pack1;
public class Deriv extends Prot
{
    Deriv()
    {
        System.out.println("Доступ з-під класу у тому ж пакеті:"+n+" "+n_pro+"
"+ n_pub);
    }
}
```

Файл X.java

```
package pack1;
public class X
{
    X()
    {
        Prot p=new Prot();
        System.out.println("Доступ із непідкласу в тому ж пакеті: "+p.n+"
"+p.n_pro+" "+p.n_pub);
    }
}
```

Пакет pack2

Файл Demo.java

```
package pack2;
public class Demo
{
    public Demo()
    {

        Prot2 ob1=new Prot2();
        Y ob2=new Y();
    }
}
```

Файл Prot2.java

```
package pack2;

public class Prot2 extends pack1.Prot
{
    Prot2()
    {
        System.out.println("Доступ з-під класу в іншому пакеті:"+n_pro+
                " "+ n_pub);
    }

}
```

Файл Y.java

```
package pack2;
public class Y
{
    Y()
    {
        pack1.Prot p=new pack1.Prot();
        System.out.println("Доступ із непідкласу в іншому пакеті:"+
                " "+ p.n_pub);
    }
}
```

Вигляд екрану працючого додатку буде таким, як наведено на рисунку 14.1:

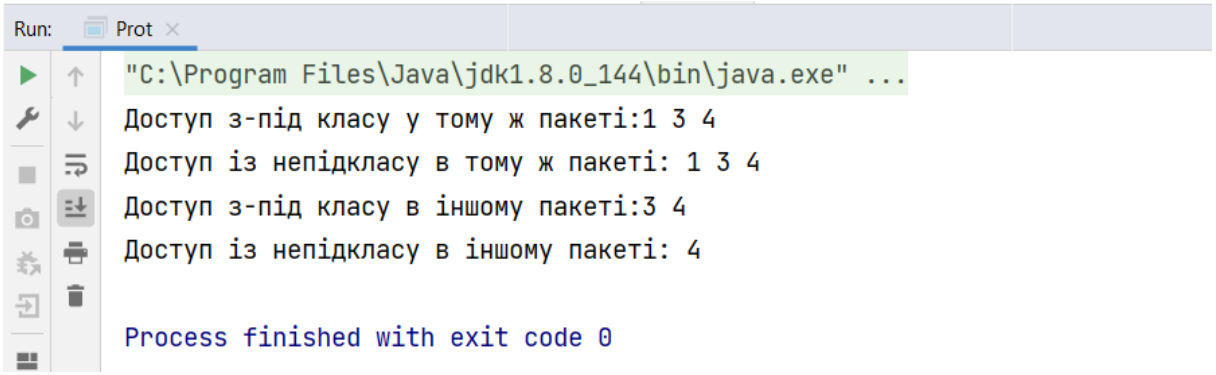


Рисунок 14.1 — Результат виконання програми

Тема 15 Обробка виняткових ситуацій

На основі матеріалів [1,2, 3, 4], зазначена інформація є авторським правом і належить [1,2, 3, 4]

Виняткова ситуація – це подія, що перериває нормальну роботу програми. При виникненні виняткової ситуації необхідно надати користувачу такі можливості:

- одержати повідомлення про виниклу помилку;
- зберегти результати роботи;
- коректно завершити роботу програми.

Обробка виняткової ситуації можлива тільки тоді, коли програміст передбачає можливі помилки, а також наслідки, до яких вони можуть привести. Найчастіше помилки при роботі програми викликаються такими причинами:

- помилки введення. Це може бути помилка в імені файлу, помилка в адресі при спробі з'єднання, помилково введене дане;
- помилки програмування. Це може бути вихід за межи масиву, непередбачений варіант обчислень, що приводить до виникнення ділення на нуль, некоректний виклик методу, спроба витягнути запис з порожнього стека та інші;

- збої устаткування. Це можуть бути збої мережевого устаткування, принтера та іншої апаратури;
- обмеження на ресурси. У процесі роботи може не виявитися досить місця в оперативній пам'яті, відбутися переповнювання диску.

15.1 Класифікація виняткових ситуацій

У мові Java виняткова ситуація, що виникла при виконанні програми, є *об'єктом* деякого класу. Усі класи виняткових ситуацій є підкласами вбудованого класу Throwable. У цього класу є два підкласи – Exception і Error.

До класу Exception належать ті виняткові ситуації, які *повинні бути перехоплені* програмою користувача.

У класі Error описуються ті виняткові ситуації, які *не можуть бути перехоплені* програмою користувача. Це *помилки в самій виконуючій системі Java.*

У свою чергу клас Exception має два підкласи – RuntimeException і IOException.

Виняткові ситуації типу *RuntimeException* виникають у наслідок *помилок програмування* (невірне приведення типів, вихід за допустимі межи масиву, спроба звернутися до об'єкта за посиланням null), їх причина завжди *може бути усунена*, якщо ретельніше писати код.

Виняткові ситуації типу IOException, в основному, виникають при введенні-виведенні даних і *не можуть бути усунені* на етапі програмування.

Фрагмент ієрархії класів виняткових ситуацій виглядає так, як наведено на рисунку 15.1.

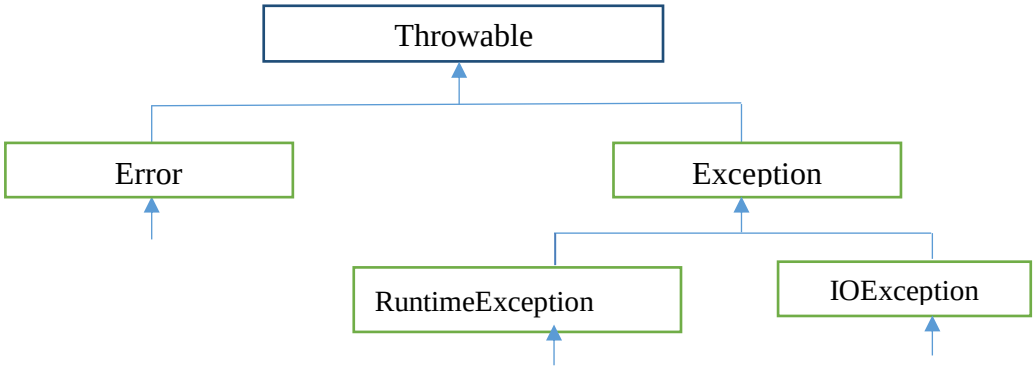


Рисунок 15.1— Фрагмент ієрархії класів виняткових ситуацій

Виняткові ситуації також прийнято підрозділяти на ті, що перевіряються (див. таблицю 15.2), і ті, що не перевіряються (див. таблицю 15.1). Мається на увазі, що компілятор Java для ряду виняткових ситуацій не перевіряє метод з погляду можливості виникнення і обробки цих ситуацій.

Для виняткових ситуацій, *що перевіряються*, компілятор вимагає використовування оператора *throws* з переліком виняткових ситуацій, можливих для даного методу (див. далі).

Таблиця 15.1— Підкласи RuntimeException, що не перевіряються

Виняткова ситуація	Опис ситуації
ArithmeticException	Арифметична помилка, наприклад, ділення на нуль
ArrayIndexOutOfBoundsException	Неправильний індекс при зверненні до масиву
ArrayStoreException	Привласнення елементові масиву невірного типу
ClassCostException	Помилка в приведенні типу
IllegalArgumentException	Помилковий аргумент при виклику методу
IllegalMonitorStateException	Невірна операція стеження, наприклад, очікування на розблокованому потоці
IllegalThreadStateException	Виконувана операція несумісна з поточним станом потоку
IndexOutOfBoundsException	Невірний тип масиву
NegativeArraySizeException	Спроба створити масив від’ємного розміру
NullPointerException	Неправильне використання посилання null
NumberFormatException	Помилка при перетворенні рядка в число
SecurityException	Спроба порушення захисту

Слід зазначити, що виняткові ситуації типу ArithmeticException не виникають при операціях над дійсними числами. Так, наприклад, ділення на 0.0 не породжує ситуацію «ділення на нуль», а результат операції при виведенні на монітор буде представлений словом infinity.

Таблиця 15.2 — Підкласи, що перевіряються

Виняткова ситуація	Опис ситуації
ClassNotFoundException	Клас не знайдений

CloneNotSupportException	Спроба створити копію об’єкта, який не відповідає інтерфейсу Cloneable
IllegalAccessException	Відмова в доступі до класу

Продовження таблиці 15.2

Виняткова ситуація	Опис ситуації
InstantiationException	Спроба створити об’єкт інтерфейсу або абстрактного класу
InterruptedException	Один потік був перерваний іншим

15.2 Принцип обробки виняткових ситуацій. Оператори try і catch

Перш за все необхідно виділити ту частину програми, в якій може виникнути виняткова ситуація.

Якщо у виділеній частині програми виникає виняткова ситуація, то її потрібно перехопити і обробити. Обробка полягає у виконанні деяких дій, відповідних виниклій ситуації. Крім цього, оскільки виділена частина програми виявилася незакінченою, а деякі дії в ній все ж таки були виконані до моменту виникнення виняткової ситуації, необхідно повернути програму користувача у стан, що існував до появи помилки, наприклад, закрити файли, видалити створені об’єкти.

Оператори try і catch дозволяють виділити блок операторів, де може виникнути виняткова ситуація, *або згенерувати виняткову ситуацію при виникненні помилки,* розпізнати виняткову ситуацію і виконати деякі дії для її подолання. Оператор catch розпізнає ситуацію і за допомогою наступних за ним операторів, *узятих у фігурні дужки,* намагається привести програму в робочий стан.

Синтаксис операторів такий:

```
try
{
оператор
……
оператор
}
catch(клас_виняткової_ситуації об’єкт)
{
оператор
……..
оператор
```

}

Не можна застосувати оператор try до окремого оператору програми. Метою правильно побудованих операторів catch є розв'язання виняткової ситуації та продовження роботи, як якщо б помилки не було.

Розглянемо приклад використовування операторів try і catch (див. листинг 15.1).

Програма приймає з командного рядку ряд чисел. Необхідно знайти результат ділення першого числа на суму всіх подальших. Виняткова ситуація може виникнути, якщо сума дорівнює нулю. Помилка ділення на нуль належить до класу ArithmeticException.

Листинг 15.1

```
class Except_try_catch1
{
public static void main(String args[])
{
int i,s=0,y,x=Integer.parseInt(args[0]);
for(i=1;i<args.length;i++)
   s+=Integer.parseInt(args[i]);
try
{
                y=x/s;
                System.out.println("y="+y);
}
catch(ArithmeticException e)
{
                System.out.println("Виникла ситуація ділення на 0");
}}}
```

Результат роботи програми наведено на рисунку 15.2

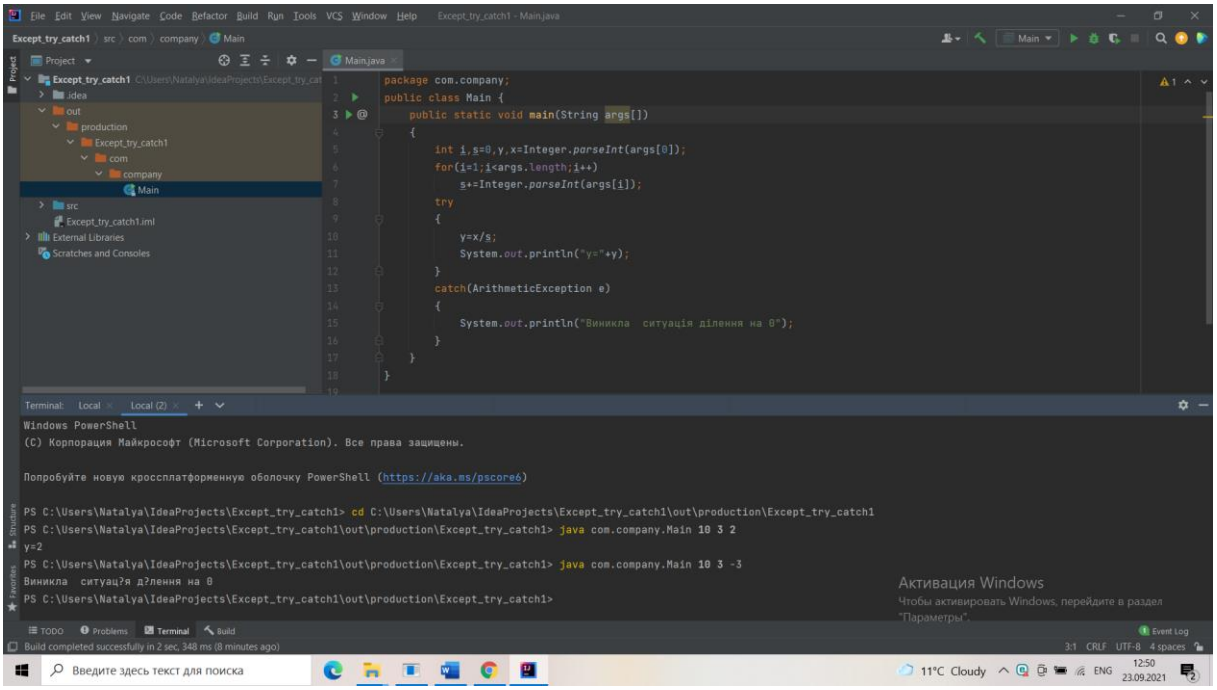


Рисунок 15.2— Принцип обробки виняткових ситуацій

Якщо в деякому фрагменті програми можуть виникнути декілька винятковых ситуацій різної природи, то для кожної з них можна утворити свій блок catch. Синтаксис такої конструкції:

```
try
{
        оператор
        ……
        оператор
}
catch(клас_виняткової_ситуації об’єкт)
{
        оператор
        ……
        оператор
}
catch(клас_виняткової_ситуації об’єкт)
{
        оператор
        ……
        оператор
}
```

Наступний приклад (див. рисунок 15.2) проілюструє виявлення двох виняткових ситуацій різного типу. Хай необхідно знайти середнє арифметичне чисел командного рядка, а потім записати в масив ті числа, які перевищують середнє арифметичне.

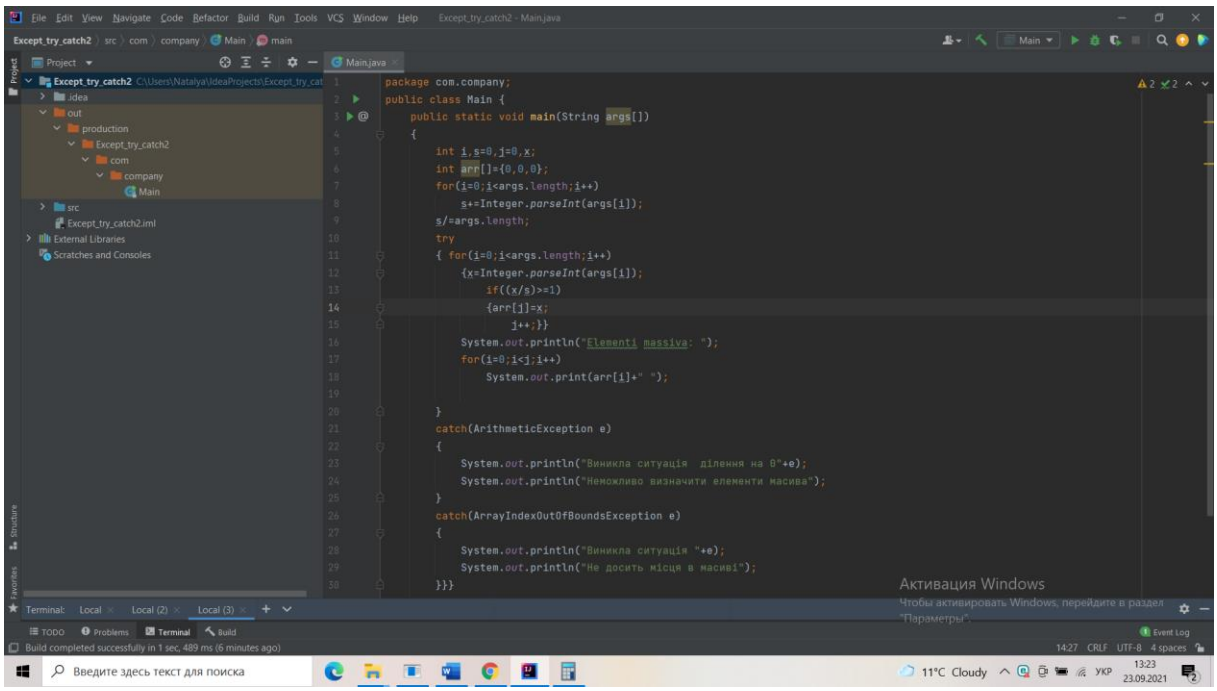


Рисунок 15.2— Виявлення декількох виняткових ситуацій

Можливі результати виконання програми наведені на рисунку 15.3:

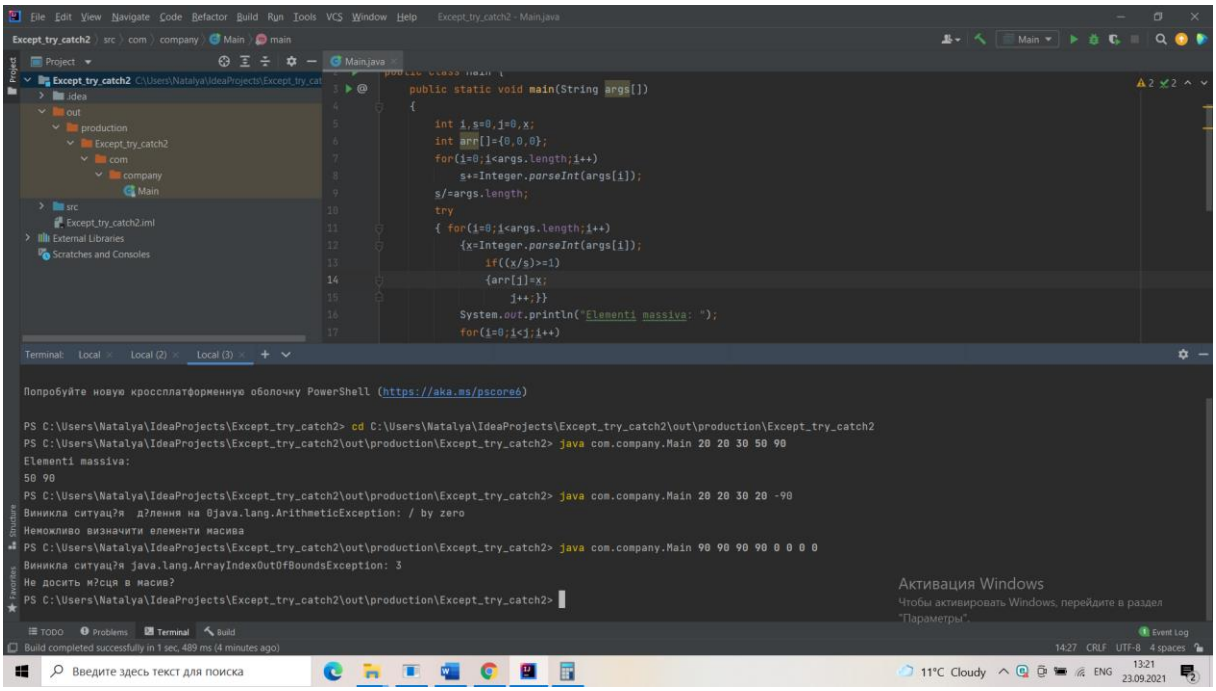


Рисунок 15.3— Результат роботи програми

Розглянемо оператор

```
System.out.println("Виникла ситуація "+e);
```

Об’єкт виняткової ситуації e автоматично перетворюється в рядок, що визначає виняткову ситуацію, крім цього, виведене число 3 указує індекс неіснуючого елемента, до якого була виконана спроба доступу.

Якщо відповідний оператор catch не буде знайдений, то виняток обробить система часу виконання Java - викликається стандартний обробник виключень і перериває програму. Він виводить трасіровку стека.

Клас Throwable перевизначає метод toString(), визначений в класі Object, таким чином, що він повертає рядок, що містить опис виняткової ситуації.

15.3 Вкладені оператори try

Блоки try можуть бути вкладеними. Внутрішні блоки можуть мати або не мати оператори catch. Якщо внутрішній блок try не має оператора catch, то об’єкт виняткової ситуації буде переданий в зовнішній блок і оброблений оператором catch цього рівня.

Як приклад використовування вкладених блоків try, також як і в попередній програмі, виконаємо обробку параметрів командного рядка (див. листинг 15.2). Проте зараз введемо новий клас Supplement, метод якого average() обчислюватиме середнє арифметичне чисел у командному рядку. Якщо командний рядок не містить аргументів, то виникає виняткова ситуація ділення на нуль. Ця ситуація обробляється внутрішнім блоком try, що входить у функцію average(). Зовнішній блок try включений у функцію main() і включає виклик функції average(). Його задача – розпізнати виняткову ситуацію, викликану виходом за межі індексів масиву.

Листинг 15.2

```
/*
 * Except_try_catch3.java
*/
class Supplement
 {
     public Supplement()
     {}
     public double average(String x[],int q)
     {
        int s=0;
         System.out.println("q="+q);
         try
         {
          for(int i=0;i<q;i++)
             s+=Integer.parseInt(x[i]);
          s=s/q;

         }
           catch(ArithmeticException e)
        {

            System.out.println("Виникла ситуація ділення на 0");
```

```
        }
         return s;
     }
}
 public class Except_try_catch3
 {
     public Except_try_catch3()
     {}
     public static void main(String args[])
     {
         double k;
         int i,j=0,x;
         int arr[]={0,0,0};
         try
         {
          Supplement sup=new Supplement();
          k=sup.average(args, args.length);
          for(i=0;i<args.length;i++)
          {
              x=Integer.parseInt(args[i]);
              if(x/k>=1)
              {
                  arr[j]=x;
                  j++;
              }
          }
          System.out.println("Elementi massiva: ");
          for(i=0;i<j;i++)
              System.out.print(arr[i]+" ");
         }
         catch(ArrayIndexOutOfBoundsException e)
         {
             System.out.println("Виникла ситуація "+e);
             System.out.println("Недосить місця для елементів ма-
сиву");
         }
     }
     }
```

Коли передається виключення, кожен оператор catch перевіряється по порядку, і перший з них, тип якого відповідає виключенню, виконується. Після того як виконається один з операторів catch, всі інші пропускаються і виконання програми продовжиться з місця, наступного за блоком try/catch.

Варіант А

Аргументи командного рядка: 20 20 30 50 90

Елементи масиву: 50 90

Варіант Б

Аргументи командного рядка:

```
Виникла ситуація ділення на 0
```

Елементи масиву:

Варіант В

Аргументи командного рядка: 90 90 90 90 0 0 0 0

```
Виникла ситуація ArrayIndexOutOfBoundsException: 3
Середнє арифметичне прирівнюватиме  до 0
Недосить місця для елементів масиву
```

15.4 Генерація виняткової ситуації

Програміст може не тільки передбачити обробку виняткової ситуації, що згенерує виконуюча система Java, але і виняткову ситуацію згенерувати тоді, коли на його думку це необхідно. Для цього служить оператор throw, який може бути використаний двома способами:

throw new клас_виняткової_ситуації();

або

Клас_виняткової_ситуації ім'я=new Клас_виняткової_ситуації();

throw ім'я;

Проілюструємо застосування оператора throw таким прикладом (див. листинг 15.3).

Хай необхідно знайти найбільше і якнайменше число в списку аргументів командного рядка. Вважатиме, що при кількості аргументів менше двох, задача втрачає сенс і таку ситуацію слід вважати помилковою.

У списку підкласів RuntimeException вибираємо клас IllegalArgumentException (невірний аргумент при виклику методу), який за назвою деякою мірою відповідає нашій винятковій ситуації.

Листинг 15.3

```
class Except_try_catch4
{
public static void main(String args[])
{
int i,n=args.length;
     double max,min,x;
     try
{
if(n<2) throw new IllegalArgumentException();
max=min=Double.parseDouble(args[0]);
for(i=1;i<n;i++)
{
     x=Double.parseDouble(args[i]);
     if(min>x)
          min=x;
if(max<x)
          max=x;
}
```

```
System.out.println("min="+min+" "+"max="+max);
}
catch(IllegalArgumentException e)
{
     System.out.println("Потрібно, як мінімум, два аргументи."+e);
}
}
```

Варіант А:

Аргументи командного рядка: 2.5 9.7 1.2

min=1.2 max=9.7

Варіант Б:

Аргументи командного рядка: 2.5

Потрібно, як мінімум, два аргументи. java.lang.IllegalArgumentException

15.5 Створення власних класів виняткових ситуацій

Вбудовані класи виняткових ситуацій дозволяють обробляти багато помилок, проте в деяких випадках цього не досить. Програміст завжди може створити свій власний клас виняткової ситуації, як підклас *вже існуючих* класів. Новий клас можна забезпечити конструктором за умовчанням і конструктором, що приймає повідомлення про тип помилки. Ім'я нового класу має закінчуватися суфіксом Exception.

Внесемо зміни в попередню програму (див. листинг 15.3). Оголосимо клас ListArgsException, похідний від класу IOException, з двома конструкторами.

Обробка нової виняткової ситуації не впливає на обробку стандартних виняткових ситуацій. Для ілюстрації цього в програмі (див. листинг 15.4) перехоплюється ситуація NumberFormatException (невірна конвертація рядка в чисельний формат).

Листинг 15.4

```
import java.io.*;
class ListArgsException extends IOException
{
     public ListArgsException()
{ }
public ListArgsException(String mess)
{
     super(mess);
}
}
public class Except_try_catch5
{
public Except_try_catch5()
{ }
public static void main(String args[])
{
          int i,j=0,n=args.length;
```

```
          double min,max,x;
try
{
     if(n<2) throw new ListArgsException("Мало аргументів");
     min=max=Double.parseDouble(args[0]);
     for(i=1;i<n;i++)
          {
x=Double.parseDouble(args[i]);
if(min>x)
     min=x;
if(max<x)
     max=x;
}
System.out.println("min="+min+" max="+max);
}
catch(ListArgsException e)
{
     System.out.println("Потрібно, як мінімум, два аргументи."+e);
}
catch(NumberFormatException e)
{
     System.out.println("Аргумент нечислового типу");
}
}
}
```

Варіант А:

Аргументи командного рядка: 2.5 9.7 1.2

min=1.2 max=9.7

Варіант Б:

Аргументи командного рядка: 2.5

Потрібно, як мінімум, два аргументи. Мало аргументів

Варіант В:

Аргументи командного рядка: 2.5 XX 1.2

Аргумент нечислового типу: java.lang.NumberFormatException: For input string: "XX"

15.6 Оголошення виняткових ситуацій, які може згенерувати метод. Оператори throws та finally

Метод може порушити виняткову ситуацію, коли виникнуть помилки, з якими він не може справитися. У цьому випадку потрібні засоби, що дозволяють обробити таку ситуацію при виклику методу. Таким засобом є оператор throws, в якому перераховуються всі можливі для методу виняткові ситуації. Враховуючи сказане, сінтакісі оголошення методу може бути розширений:

тип_значення_що_повертається ім'я_методу(параметри) throws список_виняткових_ситуацій

```
{
тіло_методу
}
```

У список виняткових ситуацій *не потрібно включати виняткові ситуації типу Error, на які програма не може впливати, а також не контрольовані виняткові ситуації, похідні від класу RuntimeException.* Звичайно замість оголошення виняткової ситуації її можна перехопити, як було показано раніше. У цьому випадку оператор throws не потрібен.

Якщо метод здатний порушити виключення, які він сам не обробляє, він повинен заявити про таку поведінку, щоб викликаючі методи могли захистити себе від цих винятків.

Якщо метод в явному вигляді (тобто за допомогою оператора throw) порушує виключення відповідного класу, тип класу винятків повинен бути зазначений в операторі throws в оголошенні цього методу.

Якщо генерується виняткова ситуація, то весь код від оператора, що створив виняткову ситуацію, до кінця блока try не виконується. Якщо до моменту виникнення виняткової ситуації були виділені деякі ресурси, то вони виявляться недоступними надалі. Наприклад, якщо на початку деякого метода відкривається файл, а в кінці методу він має бути закритий, то цього не відбудеться, що може спричинити серйозні наслідки. Для вирішення подібних проблем використовується оператор finally.

Схема використовування оператора:

```
try
{
 оператори
}
[catch(клас_виняткової_ситуації об’єкт)
 {
        оператори
 }]
finally
{
 оператори, що звільняють ресурси
}
```

Блок finally виконується незалежно від того, згенерована виняткова ситуація чи ні, а також була вона перехоплена оператором catch чи ні. Зокрема оператора finally можна використовувати і за відсутності оператора catch. Використання операторів throw та finally розглянемо на прикладі програми із листингу 15.5.

Листинг 15.5.

```
class Exc_finally
{
     static void methodA()
{
          try
          {
          System.out.println("Виконується метод methodA()");
          throw new ArithmeticException("Example");
          }
          catch(ArithmeticException e)
          {
          System.out.println("Перехоплена виняткова ситуація в ме-
          тоді methodA()");
          }
          finally
          {
          System.out.println("Виконується блок finally методу
          methodA()");
          }
     }
static void methodB()
{
          try
          {
          System.out.println("Виконується метод methodB()");
          throw new ArithmeticException("Example");
          }
          finally
          {
          System.out.println("Виконується  блок  finally   методу
          methodB()");
          }
}
static void methodC()
{
          try
          {
                    System.out.println("Виконується          метод
          methodC()");
          }
          finally
          {
          System.out.println("Виконується  блок  finally  методу
          methodC()");
          }
}
public static void main(String args[])
{
     methodA();
try
{
methodB();
}
```

```
catch(ArithmeticException e)
{
System.out.println("Перехоплена виняткова ситуація методу
methodB()  у методі main()");
}
methodC();
}
}
```

У наведеній програмі метод methodA() містить блок try, в якому генерується виняткова ситуація (вибрана ArithmeticException, але можна вибрати і іншу). Перехоплення виняткової ситуації здійснюється усередині методу methodA().

У методі methodB() виняткова ситуація генерується, але не перехоплюється. Перехоплення цієї виняткової ситуації виконується в методі main().

У методі methodC() виняткова ситуація не генерується.

У всіх трьох випадках виконуються блоки finally, як це наведено на рисунку 15.4.

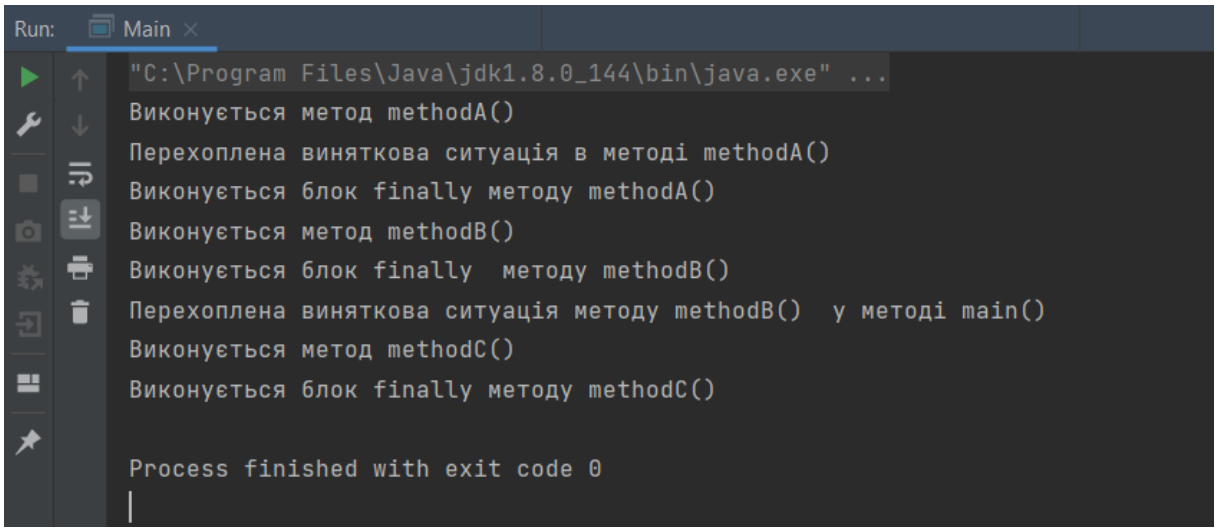


Рисунок 15.4 — Результат роботи програми із листингу15.5

Тема 16 Каркас колекцій Collections Framework

На основі матеріалів [1, 2,3, 4], зазначена інформація є авторським правом і належить [1, 2, 3, 4]

Каркас колекцій Collections Framework являє собою складну ієрархію інтерфейсів і класів, що реалізують сучасну технологію управління групами об'єктів. Основні колекції - динамічні масиви, пов’язані списки, дерева. Collections Framework представлений в пакеті java.util.

Каркас колекцій стандартизує способи управління групами об'єктів в прикладних програмах. До цього для зберігання і управління групами об'єктів в Java надавалися спеціальні класи, такі як Vector, Stack, Dictionary. І хоча ці класи були досить зручні, їм не вистачало загальної, об'єднуючої основи.

Каркас колекцій був розроблений для того, щоб забезпечити високу продуктивність, однакове функціонування колекцій. Колекції повинні були допускати просте розширення і/або адаптацію.

Весь каркас колекцій побудований на єдиному наборі стандартних інтерфейсів. Саме вони визначають саму сутність класів колекцій. А конкретні класи лише надають різні реалізації стандартних інтерфейсів. Алгорітми складають іншу важливу частину каркаса колекцій. Алгоритми оперують колекціями і визначені у вигляді статичних методів в класі Collections. Тому вони доступні всім колекціям і не вимагають реалізації їх власної версії в кожному класі колекції. Алгоритми надають стандартні засоби для маніпулювання колекціями.

Іншим елементом, тісно пов'язаним з каркасом колекцій, є інтерфейс Iterator, що визначає итератор, який забезпечує загальний, стандартизований спосіб почергового доступу до елементів колекцій. Кожна колекція надає свій ітератор, тому елементи будь-якого класу можуть бути доступні за допомогою методів, визначених у інтерфейсі Iterator. Пізніше був впроваджений інший різновид ітератора, званий ітератором-роздільником (Spliterator).

До змін, які суттєво вплинули на ефективність та спрощення застосування колекцій, відносять:

- впровадження узагальнень;
- автоматичну упаковку і розпаковку;
- організацію циклу for в стилі for each.

Всі колекції тепер є узагальненими і багато методів, які оперують колекціями, також приймають узагальнені параметри. Узагальнення внесли в колекції типову безпеку.

Автоупаковка і авторозпаковка спрощують збереження елементарних типів даних в колекціях. В колекціях, як звісно, можна зберігати тільки посилання на об'єкти, але не значення елементарних типів.

Всі класи в каркасі колекцій вдосконалені таким чином, щоб реалізовувати інтерфейс Iterable. Це означає, що вміст колекції можна перебирати, використовуючи цикл for в стилі for each. До цього для перебору колекцій потрібно було використовувати итератор.

16.1 Інтерфейси колекцій

Розглянемо інтерфейси, що підтримують колекції (див. таблицю 16.1)

Таблиця 16.1 — Інтерфейси колекцій

Інтерфейс	Призначення
Collection	Дозволяє працювати з групами об'єктів. Знаходиться на вершині ієрархії колекцій
Deque	Розширює інтерфейс Queue для організації двосторонніх черг
List	Розширює інтерфейс Collection для управління послідовностями (списками об'єктів)
NavigableSet	Розширює інтерфейс SortedSet для вилучення елементів по результатам пошуку найближчого збігу
Queue	Розширює інтерфейс Collection для управління спеціальними типами списків, де елементи видаляються тільки з початку списку
Set	Розширює інтерфейс Collection для управління множинами, які повинні містити однозначні елементи
SortedSet	Розширює інтерфейс Set для управління відсортованими множинами

Задля забезпечення максимальних зручностей застосування інтерфейсів колекцій, деякі методи в них можуть бути необов'язковими. Необов'язкові методи дозволяють видозмінювати вміст колекцій. Колекції, що підтримують такі методи, називають *змінними*, що не дозволяють - *незмінними*. Всі вбудовані колекції є змінними

16.1.1 Інтерфейс Collection

Служить основою, на якій побудований весь каркас колекцій, оскільки він повинен бути реалізований всіма класами колекцій. Його оголошення

```
interface Collection<E>,
де E – тип об'єктів, які буде містити колекція.
```

Інтерфейс Collection розширює інтерфейс Iterable. Це означає, що всі колекції можна перебирати, організувавши цикл for в стилі for each.

В інтерфейсі Collection оголошуються основні методи, які повинні мати всі колекції. Розглянемо основні з них (див. таблицю 16.2).

Таблиця 16.2 — Методи з інтерфейсу Collection

Метод	Призначення
boolean add(E об’єкт)	Вводить заданий об'єкт в визиваючу колекцію. Повертає true, якщо об'єкт успішно введений в колекцію. А якщо об'єкт вже присутній в колекції, яка не допускає дублювання об'єктів, то повертає false
boolean addAll(Collection<? extends c)	Вводить всі елементи заданої колекції с в викликаючу колекцію. Повертає true, якщо колекція змінена (тобто всі елементи введені), а інакше - false
void clear()	Видаляє всі елементи з викликаючої колекції
boolean contains(Object об’єкт)	Повертає true, якщо заданий об'єкт є елементом викликаючої колекції, а інакше - false
boolean containsAll(Collection<?> c)	Повертає true, якщо викликаюча колекція містить всі елементи колекції с, а інакше - false
boolean equals(Object об’єкт)	Повертає true, якщо викликаюча колекція і об'єкт рівнозначні, а інакше - false
boolean isEmpty()	Повертає true, якщо викликаюча колекція порожня, а інакше - false
Iterator<E> iterator()	Повертає ітератор для викликаючої колекції
boolean remove(Object об’єкт)	Видаляє один екземпляр об'єкта з викликаючої колекції. Повертає true, якщо об'єкт видалений, а інакше - false

Продовження таблиці 16.2

Метод	Призначення
boolean removeAll(Collection<?> c)	Видаляє всі елементи заданої колекції с з викликаючої колекції. Повертає true, в разі успішного видалення, а інакше - false
boolean retainAll(Collection<?> c)	Видаляє з викликаючої колекції всі елементи, крім елементів заданої колекції. Повертає true, в разі успішного видалення а інакше - false
int size()	Повертає кількість елементів, що містяться в коллкції
<T> T[] toArray(T масив[])	Повертає масив, що містить елементи викликаючої колекції. Елементи масиву є копіями елементів колекції

16.1.2 Інтерфейс List

Цей інтерфейс розширює інтерфейс Collection і визначає таку поведінку колекцій, яке зберігає послідовність елементів. Елементи можуть бути введені або видалені по індексу їх позиції в списку, починаючи з нуля. Список може містити елементи, що повторюються. Інтерфейс List є узагальненим і оголошується наступним чином:

```
interface List<E>
```

Крім методів, оголошених в інтерфейсі Collection, інтерфейс List визначає ряд власних методів, перерахованих в таблиці 5.3.

Таблиця 16.3 — Деякі методи інтерфейса List

Метод	Призначення
void add(int индекс, E об’єкт)	Вводить заданий об'єкт на позиції викликаючого списку за вказаним індексом. Будь-які введені раніше елементи зміщуються, починаючи з вказаної позиції і далі до початку списку. Це означає, що елементи в списку не перезаписуються

Продовження таблиці 16.3

Метод	Призначення
boolean addAll(int індекс, Collection<? extends E> c)	Вводить всі елементи колекції с на позиції викликаючого списку, починаючи з позиції за вказаним індексом. Будь-які введені раніше елементи зміщуються, починаючи з вказаної позиції і далі до початку списку. Це означає, що елементи в списку не перезаписуються
E get(int індекс)	Повертає об’єкт, який зберігається на позиціїї за вказаним індексом
int indexOf(E об’єкт)	Повертає індекс першого екземпляра об'єкта в викликаючому списку. Якщо заданий об'єкт відсутній в списку, повертається -1
ListIterator<E> listIterator(int індекс)	Повертає ітератор для обходу елементів викликаючого списку, починаючи з позиції за вказаним індексом
E remove(int індекс)	Видаляє елемент з викликаючого списку на позиції за вказаним індексом і повертає видалений елемент. Результуючий список ущільнюється - елементи, які йдуть за видаленим, зрущуються на одну позицію назад
E set(ind індекс, E об’єкт)	Присвоює заданий елемент об'єкту, що знаходиться в списку на позиції за вказаним індексом. Повертає колишнє значення

16.2 Класи колекцій

Розглянемо стандартні класи (див. таблицю 5.4), які реалізують описані вище інтерфейси. Одні з цих класів надають повну реалізацію відповідних інтерфейсів і можуть застосовуватися без змін. А інші є абстрактними, надаючи

тільки шаблонні реалізації відповідних інтерфейсів, які використовуються в якості відправної точки для створення конкретних колекцій.

Таблиця 16.4 — Деякі базові класи колекцій

Клас	Призначення
AbstractCollection	Реалізує більшу частину інтерфейсу Collection
AbstractList	Розширює клас AbstractCollection і реалізує більшу частину інтерфейсу List
ArrayList	Реалізує динамічний масив, розширюючи клас AbstratList

16.4.1 Клас ArrayList

Клас ArrayList розширює клас AbstractList і реалізує інтерфейс List. Клас ArrayList є узагальненим і оголошується так:

```
class ArrayList<E>,
де E - тип об'єктів , що зберігаються.
```

У класі ArrayList підтримуються списочні масиви, які можуть нарощуватися у міру потреби або зменшувати свій розмір. Списочні масиви створюються з деяким початковим розміром. Коли ж цього початкового розміру виявляється недостатньо, колекція автоматично розширюється. А коли з колекції видаляються об'єкти, вона може скорочуватися.

У класі ArrayList визначені наступні конструктори:

```
ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int ємність)
```

Перший конструктор створює порожній списочний масив, другий - списочний масив, що ініціалізується елементами із заданої колекції c, а третій - списочний масив, який має початкову ємність. Під ємністю тут мається на увазі розмір базового масиву, використовуваного для зберігання елементів даного виду колекції. Ємність нарощується автоматично під час введення елементів в масив.

ЛІТЕРАТУРА

1. Васильєв О. М. Програмування мовою Java / О. М. Васильєв - М. Тернопіль: Навчальна книга - Богдан, 2020. - 696 с Шилдт Г. Java 8. Полное руководство, Десятое издание / Г. Шилдт - Москва: ООО «И.Д. Вильямс», 2016. - 1488 с.
2. Кунгурцев О. Б. Основи програмування на мові Java. Середовище Net Beans: навчальний посібник для студентів вищих навчальних закладів / О. Б. Кунгурцев; за ред. Т.В. Ковалюк. - Одеса, 2016. - 183 с.