МІНІСТЕРСТВО ОСВІТИІ НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Ніщеменко Д.О., Жебка В.В., Аронов А.О.

Конструювання програмного забезпечення Java

Навчальний посібник (Частина 1)

Київ - 2025

УДК 004.42

Рекомендовано Вченою радою Державного університету інформаційно-комунікаційних технологій (протокол № 12 від 21 жовтня 2025 року)

Ніщеменко Д.О., Жебка В.В., Аронов А.О. Конструювання програмного забезпечення Java. – Навчальний посібник. – Київ: ДУІКТ, 2025. - с. 119

Навчальний посібник призначений для студентів, що вивчають дисципліну «Конструювання програмного забезпечення Java», та розглядає шлях від базового синтаксису до професійних підходів у розробці. Основний акцент зроблено на засвоєнні принципів об'єктно-орієнтованого дизайну SOLID та практичному застосуванні ключових патернів проєктування. Посібник охоплює розробку сучасного REST API з використанням Spring Framework, формуючи навички, необхідні для подальшого зростання в інженерії ПЗ.

ISBN 978-617-8521-02-8

ЗМІСТ

ВСТУП

Мова програмування Java вже понад два десятиліття займає провідні позиції у світі розробки програмного забезпечення, залишаючись однією з найпотужніших та найпопулярніших мов програмування. Її успіх ґрунтується на філософії "Write Once, Run Anywhere" ("Напиши один раз, запускай будь-де"), що забезпечує незалежність від платформи та легку переносимість програм. Завдяки своїй надійності, великій екосистемі бібліотек та фреймворків, а також широкій спільноті розробників , Java стала основою для створення широкого спектра програмних рішень: від складних корпоративних систем та банківських додатків до мобільних застосунків під керуванням ОС Android.

Цей навчальний посібник є фундаментальним курсом, що має на меті закласти основи професійного підходу до розробки програмних систем на мові Java. Він розрахований на читачів та студентів, які вже мають початкові навички програмування і прагнуть перейти від вивчення базового синтаксису до глибокого розуміння принципів якісного проєктування та архітектури.

Структура посібника відповідає двом логічним етапам навчання. Впродовж першої частини курсу (перший семестр) основний акцент робиться на засвоєнні ключових принципів об'єктно-орієнтованого проєктування SOLID та практичному застосуванні найпоширеніших патернів проєктування (GoF). Це дозволяє сформувати міцну теоретичну базу для створення гнучкого, масштабованого та підтримуваного коду.

У результаті вивчення дисципліни випускники набувають комплексних навичок, необхідних для позиції Junior Java Developer. Вони не лише володіють мовою Java та фреймворком Spring, але й розуміють принципи побудови якісної архітектури, вміють проєктувати системи з урахуванням майбутніх змін, писати тести, працювати з базами даних та контейнеризувати додатки за допомогою Docker.

Посібник розбито на лекції, кожна з яких містить теоретичне введення, приклади коду, а також завершується завданнями для практичної та лабораторної роботи для закріплення навичок і контрольними питаннями для самоперевірки засвоєного матеріалу.

ТЕМА 1. ВСТУП ДО МОВИ JAVA ТА ОСНОВИ ООП

Лекція 1. Основи платформи Java та синтаксис мови

Що таке Java: JVM, JRE, JDK. Принцип "Write Once, Run Anywhere"

Java — це високорівнева, об’єктно-орієнтована мова програмування, створена на основі класів. Вона була розроблена Джеймсом Гослінгом у компанії Sun Microsystems (яка зараз є частиною корпорації Oracle) і вперше представлена у 1995 році. Фундаментальним принципом, закладеним у філософію Java, є ідея "Write Once, Run Anywhere" (WORA), що перекладається як "Напиши один раз, запускай будь-де". Цей принцип підкреслює ключову перевагу мови — незалежність від платформи. Це означає, що програма, написана та скомпільована на одній операційній системі (наприклад, Windows), може без будь-яких змін виконуватися на будь-якій іншій (наприклад, macOS або Linux). Така універсальність стала можливою завдяки унікальній архітектурі виконання, що складається з кількох ключових компонентів: віртуальної машини Java (JVM), середовища виконання (JRE) та комплекту для розробки (JDK).

Серцем платформної незалежності Java є Віртуальна машина Java (JVM – Java Virtual Machine). JVM — це, по суті, абстрактна обчислювальна машина, яка створює стандартизоване середовище для виконання програм. Коли розробник компілює вихідний код Java, він перетворюється не в машинний код, специфічний для конкретного процесора, а в проміжний формат, який називається байт-кодом. Цей байт-код є універсальним набором інструкцій, який може зрозуміти будь-яка JVM. Сама ж JVM для кожної операційної системи та архітектури процесора є унікальною. Вона діє як перекладач, який на льоту інтерпретує байт-код і транслює його в машинні інструкції, зрозумілі конкретному пристрою. Завдяки такому підходу, розробнику достатньо скомпілювати програму лише один раз, щоб отримати універсальний байт-код, який можна запускати скрізь, де встановлена JVM.

Для того, щоб запустити готову Java-програму на комп'ютері, користувачеві необхідне Середовище виконання Java (JRE – Java Runtime Environment). JRE є пакетом, що містить усі необхідні компоненти для виконання скомпільованих програм. До його складу входить сама віртуальна машина Java (JVM) та Бібліотека класів Java (JCL – Java Class Library). JCL — це великий набір стандартних бібліотек, які надають програмістам готовий функціонал для виконання найпоширеніших завдань: робота з файлами, мережею, колекціями даних, графічним інтерфейсом, безпекою та багато іншого. Коли ви запускаєте програму, JRE об'єднує її байт-код з необхідними бібліотеками та передає на виконання віртуальній машині. Варто зазначити, що, починаючи з версії Java 11, JRE зазвичай не надається як окремий пакет для завантаження; для запуску програм тепер потрібно встановлювати JDK.

Нарешті, для створення програм мовою Java розробники використовують Комплект для розробки Java (JDK – Java Development Kit). JDK — це повноцінний набір інструментів, який включає в себе все, що є в JRE (тобто JVM і стандартні бібліотеки), а також додаткові утиліти для розробника. Найважливішим з них є компілятор javac, який перетворює написаний людиною вихідний код (файли з розширенням .java) у байт-код (файли з розширенням .class). Крім компілятора, JDK містить відладчик (debugger) для пошуку помилок, архіватор (jar) для пакування кількох .class файлів в єдиний архів, генератор документації (javadoc) та інші інструменти, необхідні на всіх етапах розробки програмного забезпечення.

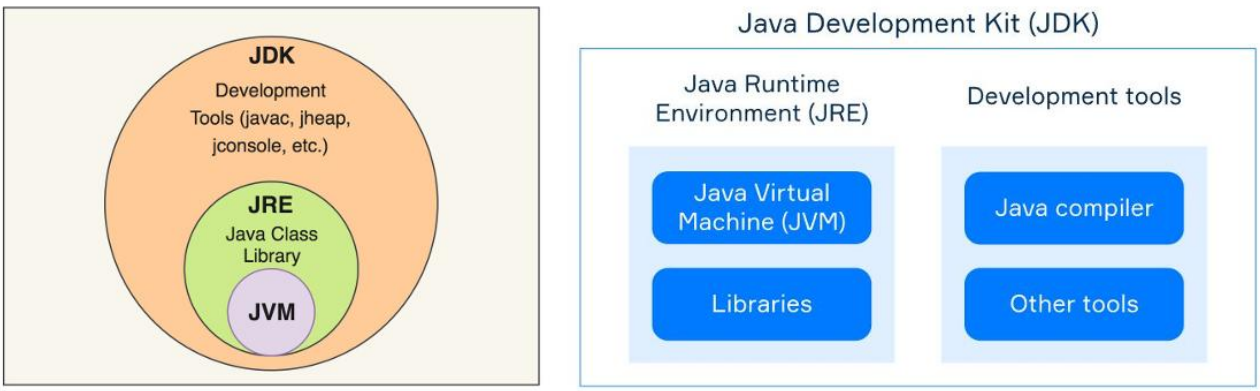

Рисунок 1.1 - Компоненти JDK

Таким чином, ці три компоненти утворюють ієрархічну структуру: JDK містить усе необхідне для розробки, включаючи JRE; JRE містить усе для запуску програм, включаючи JVM; а JVM є фундаментальним виконавчим механізмом, що забезпечує легендарну портативність Java.

Версіонування в Java: огляд ключових версій

З моменту свого створення Java пройшла довгий шлях еволюції. Спочатку нові версії виходили нерегулярно, раз на кілька років. Проте, починаючи з Java 9, Oracle змінила підхід, запровадивши швидкий цикл релізів: нова версія виходить кожні шість місяців. Водночас були введені версії з довготривалою підтримкою — LTS (Long-Term Support). Саме LTS-версії є найстабільнішими та рекомендованими для використання у комерційних проєктах, оскільки вони отримують оновлення безпеки та виправлення помилок протягом багатьох років.

Історія розвитку мови Java — це історія постійних інновацій та вдосконалень, що відображається у її версіонуванні. Розуміння еволюції версій є важливим для будь-якого розробника, оскільки воно визначає доступний функціонал, сумісність бібліотек та підходи до розробки. Протягом свого існування модель випуску нових версій Java зазнала значних змін, перейшовши від тривалих циклів розробки до швидкого та передбачуваного графіка.

Чому для ринку Java важливо знати різницю між версіями? На відміну від деяких новіших мов програмування, де екосистема швидко переходить на останню версію, у світі Java існує значна різноманітність версій, що одночасно використовуються в комерційних проєктах. Для розробника, особливо того, хто шукає роботу, розуміння відмінностей між цими версіями є не просто перевагою, а часто — критичною вимогою. Причина цього полягає в унікальній історії розвитку мови та консервативній природі великого бізнесу.

Ключову роль у формуванні поточної ситуації відіграла Java 8. Випущена у 2014 році, вона стала настільки революційною та стабільною, що на довгі роки закріпилася як промисловий стандарт. Впровадження лямбда-виразів та Stream API дозволило писати код у новому, функціональному стилі, що значно підвищило його читабельність та ефективність. Протягом майже чотирьох років (до виходу Java 9 у 2017) Java 8 була основною версією. За цей час на ній були написані мільйони рядків коду, створені величезні корпоративні системи, розроблені тисячі бібліотек та фреймворків.

Після 2017 року Oracle змінила підхід до випусків, перейшовши на швидкий шестимісячний цикл релізів. Для великих компаній, чиї системи обслуговують мільйони користувачів і де стабільність є найвищим пріоритетом, оновлювати всю інфраструктуру кожні півроку виявилося неможливим і ризикованим. Як відповідь на цю проблему, з'явилася модель Long-Term Support (LTS) — версій з довготривалою підтримкою.

Саме LTS-версії (Java 8, 11, 17, 21) стали новими "островами стабільності". Компанії вирішили ігнорувати проміжні релізи й планувати свої оновлення від однієї LTS-версії до іншої. Це призвело до того, що екосистема Java фрагментувалася. На ринку одночасно існують:

1. "Спадкові" (Legacy) системи: Величезні, надійні, але написані кілька років тому. Вони часто працюють на Java 8.
2. Системи на проміжній стадії: Проєкти, які встигли мігрувати на Java 11.
3. Сучасні проєкти: Нові сервіси та стартапи, які одразу починають розробку на останніх LTS-версіях, як-от Java 17 або Java 21, щоб використовувати найновіші можливості мови.

Розуміння цієї фрагментації та відмінностей між версіями безпосередньо впливає на вашу цінність як кандидата. Більшість вакансій, особливо у великих банках, страхових компаніях чи e-commerce гігантах, передбачають підтримку та розвиток існуючих систем. Якщо компанія використовує Java 8, ви повинні не просто знати синтаксис, а вміти писати "ідіоматичний" код для цієї версії: використовувати анонімні класи там, де ще не було лямбд (хоча вони є в 8-й), розуміти старий Date/Time API, знати, як обходитись без var і т.д.

Компанії постійно перебувають у процесі міграції зі старих версій на новіші (наприклад, з 8 на 17). Розробники, які глибоко розуміють відмінності між цими версіями, є надзвичайно цінними. Вони можуть не лише писати новий код, а й грамотно рефакторити старий, впроваджуючи нові можливості мови (наприклад, замінюючи громіздкі класи-конструктори на лаконічні Records або переписуючи складні if-else конструкції за допомогою pattern matching). Сучасні версії популярних фреймворків вимагають новіших версій Java. Наприклад, Spring Framework 6 (і, відповідно, Spring Boot 3) вимагає щонайменше Java 17. Якщо ви претендуєте на позицію, де розробляється новий продукт з використанням останніх технологій, знання лише Java 8 буде недостатньо.

На технічних співбесідах дуже часто ставлять питання, спрямовані на перевірку знання еволюції мови: "Які ключові нововведення з'явилися в Java 8?", "Що таке модульна система з Java 9?", "Для чого потрібні Records та Sealed Classes у Java 17?", "Що ви знаєте про віртуальні потоки в Java 21?". Відповіді на ці питання демонструють не лише ваші знання, а й вашу зацікавленість у професійному розвитку та розуміння сучасних трендів.

Таким чином, для успішної кар'єри в Java сьогодні недостатньо знати мову в якомусь одному її стані. Потрібно мати гнучкість: вміти підтримувати код, написаний за канонами Java 8, і водночас бути готовим проєктувати нові системи, використовуючи всі переваги та синтаксичні можливості Java 17 або 21. Це робить вас універсальним та конкурентоспроможним фахівцем на ринку.

Спочатку нові версії Java виходили з інтервалом у кілька років, і кожна з них приносила значні зміни. Наприклад, версія JDK 1.0 (1996) заклала основи мови, а вже Java 2 (J2SE 1.2) представила Collections Framework — фундаментальний набір інструментів для роботи зі структурами даних. Версія J2SE 5.0 (2004), також відома як "Tiger", стала революційною, додавши дженерики (generics), анотації, автобоксинг та покращений цикл for, що суттєво змінило спосіб написання коду.

Ключовим моментом в історії версіонування стало впровадження нового, швидкого циклу випусків, починаючи з Java 9 (2017). Корпорація Oracle, що опікується розвитком мови, перейшла на шестимісячний графік релізів. Це означає, що нова версія Java з'являється кожні пів року, у березні та вересні. Такий підхід дозволяє швидше впроваджувати нові можливості та отримувати зворотний зв'язок від спільноти.

Разом із цим було введено поняття версій з довготривалою підтримкою (Long-Term Support, LTS). LTS-версії виходять кожні два роки (раніше — кожні три роки) і отримують оновлення безпеки та виправлення помилок протягом тривалого періоду (зазвичай кілька років). Інші, проміжні версії, отримують підтримку лише до виходу наступного релізу, тобто протягом шести місяців.

Саме LTS-версії є стандартом для розробки у великих комерційних проєктах, оскільки вони забезпечують стабільність, надійність та передбачуваність. У силабусі вашого курсу, наприклад, для лабораторних робіт рекомендовано використовувати Java 21, яка є останньою LTS-версією на момент його створення.

Розглянемо ключові версії, що мали найбільший вплив на сучасну розробку.

Java 8 (LTS, 2014). Ця версія стала справжньою віхою в історії Java і досі є однією з найпопулярніших у світі. Вона представила елементи функціонального програмування: лямбда-вирази та Stream API, що кардинально змінило підходи до обробки колекцій даних, зробивши код більш лаконічним та виразним. Також у Java 8 з'явився новий Date/Time API та Optional для кращої обробки null.

Java 9 (2017). Перша версія у новому шестимісячному циклі. Її головним нововведенням стала модульна система (Project Jigsaw), яка дозволила розбивати великі монолітні додатки на менші, керовані модулі. Це покращило інкапсуляцію та продуктивність.

Java 11 (LTS, 2018). Перша LTS-версія після переходу на новий графік. Вона закріпила модульність, представила новий HTTP-клієнт, а також дозволила запускати файли з вихідним кодом без попередньої компіляції. Важливою зміною стало те, що JDK більше не постачався з окремим JRE, що підкреслило орієнтацію на розробників.

Java 17 (LTS, 2021). Ця версія принесла подальші вдосконалення мови, такі як запечатані класи (Sealed Classes), які надають розробникам кращий контроль над ієрархією успадкування, та відповідність шаблонів (Pattern Matching) для instanceof, що спрощує перевірку типів та приведення.

Java 21 (LTS, 2023). Остання на сьогодні версія з довготривалою підтримкою, яка продовжує розвиток мови в напрямку спрощення асинхронного програмування. Ключовими нововведеннями стали віртуальні потоки (Virtual Threads, Project Loom), які значно спрощують написання високопродуктивних паралельних додатків, а також подальше розширення відповідності шаблонів для switch та записи.

Для розробника-початківця важливо орієнтуватися на останні LTS-версії (11, 17, 21), оскільки саме вони використовуються в більшості сучасних проєктів і містять актуальний набір можливостей, що вимагаються на ринку праці.

Структура Java-програми. Компіляція та запуск

Будь-яка, навіть найпростіша, програма на Java має чітку структуру. В основі лежить поняття класу — це фундаментальний будівельний блок, який слугує "кресленням" для майбутніх об'єктів. Весь виконуваний код у Java повинен знаходитись всередині класів.

Розглянемо класичну програму "Hello, World!"

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

public class HelloWorld цей рядок оголошує новий клас з назвою HelloWorld. Ключове слово public означає, що цей клас доступний з будь-якого іншого класу. В одному файлі .java може бути лише один публічний клас, і його назва повинна збігатися з назвою файлу (у нашому випадку, файл має називатися HelloWorld.java).

public static void main(String[] args) це головний метод програми, або її точка входу. Саме з цього методу JVM починає виконання програми. Його сигнатура (назва, параметри та модифікатори) є строго визначеною:

public детод доступний для виклику ззовні класу.

static метод належить класу, а не об'єкту. Це дозволяє JVM викликати його без створення екземпляра класу.

void: метод нічого не повертає після свого завершення.

main: спеціальне ім'я, яке JVM шукає для запуску.

String[] args параметр, який дозволяє передавати програмі аргументи командного рядка у вигляді масиву рядків.

System.out.println("Hello, World!"); це інструкція, яка виконує дію — виводить рядок "Hello, World!" на консоль.

Щоб виконати написану програму, необхідно пройти два основні етапи, які зазвичай автоматизуються в сучасних середовищах розробки (IDE), але їх важливо розуміти.

1. Компіляція (перетворення вихідного коду на байт-код). Ви створюєте файл з вихідним кодом і зберігаєте його з розширенням .java, наприклад, HelloWorld.java.

Далі, за допомогою компілятора з JDK, який викликається командою javac, ви компілюєте цей файл:

```
javac HelloWorld.java
```

Якщо в коді немає помилок, компілятор створить новий файл з назвою HelloWorld.class. Цей файл містить не машинний код, а проміжний байт-код — набір інструкцій, зрозумілих для віртуальної машини Java (JVM). Якщо у вашому .java файлі було декілька класів, для кожного з них буде створено окремий .class файл.

Запуск (виконання байт-коду віртуальною машиною). Тепер, коли у вас є скомпільований байт-код, ви можете виконати його за допомогою команди java. Ця команда запускає JVM.

Ви вказуєте ім'я класу, що містить головний метод main, без розширення .class:

```
java HelloWorld
```

JVM завантажує вказаний .class файл, знаходить у ньому метод public static void main(String[] args) і починає його виконання. Якщо такий метод не знайдено, JVM видасть помилку.

Цей двохетапний процес — компіляція в універсальний байт-код і подальша його інтерпретація віртуальною машиною — є основою принципу "Write Once, Run Anywhere".

Якщо в одному файлі .java описано кілька класів, то при компіляції кожен клас буде скомпільовано в окремий файл .class.

У Java, кожен публічний клас (якщо є) повинен мати таку ж саму назву, як ім'я файлу. Наприклад, якщо у файлі MyClass.java є два класи: MyClass і MyOtherClass, то після компіляції ви отримаєте два окремі файли: MyClass.class та MyOtherClass.class.

При цьому, навіть якщо файл містить декілька класів, лише один клас може бути публічним (і цей клас буде мати таку саму назву, як файл). Інші класи будуть мати пакетний доступ або доступ за замовчуванням (package-private) і скомпільовані в окремі файли

У Java компілятор не має потреби знати точку входу, адже він лише перетворює вихідний код у байт-код. Точка входу визначається саме під час запуску програми, коли її завантажує і виконує JVM. При запуску програми JVM шукає метод main() із точною сигнатурою public static void main(String[] args) у класі, який вказується як точка входу. Клас, що містить main, не обов'язково має називатися певним чином або мати спеціальне призначення, але саме в ньому JVM шукатиме точку входу.

Коли ви запускаєте програму командою, наприклад:

```
java MyProgram
```

то JVM шукає метод main у класі MyProgram. Якщо метод main у такій формі не буде знайдений, JVM видасть помилку, наприклад: Error: Main method not found in class MyProgram, please define the main method as: public static void main(String[] args)

У програмі може бути кілька класів із методом main, але JVM виконає лише той main, клас якого ви вкажете при запуску. Якщо у великій програмі потрібно мати кілька точок входу, вони можуть бути реалізовані в різних класах із власними методами main. Отже, точка входу визначається JVM під час виконання на основі вказаного класу, а не компілятором під час компіляції.

Базовий синтаксис: змінні, примітивні типи даних, оператори

Основою будь-якої мови програмування є її синтаксис — набір правил, що визначає, як писати коректні та зрозумілі для компілятора інструкції. У Java базовими будівельними блоками для роботи з даними є змінні, типи даних та оператори. Розуміння цих концепцій є першим і найважливішим кроком до опанування мови.

Кожна змінна в Java має бути оголошена перед її використанням. Оголошення змінної має чітку структуру:

```
DataType variableName = initialization;
```

Тут DataType визначає, якого роду інформація може зберігатися у змінній (наприклад, ціле число або рядок тексту). variableName — це унікальне ім'я, за яким ви будете звертатися. За загальноприйнятою конвенцією Java, назви змінних пишуться у стилі camelCase, де перше слово починається з маленької літери, а кожне наступне — з великої (наприклад, myVariable або userAge).

Ініціалізація — це процес присвоєння змінній початкового значення за допомогою оператора =. Самі ж конкретні значення, такі як число 10, рядок "hello" або символ 'A', називаються літералами.

Java є строго типізованою мовою, що означає, що кожна змінна повинна мати визначений тип. В основі системи типів Java лежать примітивні типи — найпростіші види даних, вбудовані безпосередньо в мову. Існує вісім примітивних типів, які можна згрупувати за призначенням.

1. **Цілі числа**

Для зберігання цілих чисел без дробової частини Java надає чотири типи, що відрізняються розміром та діапазоном значень. Найменшим є byte, що займає 8 біт і може зберігати числа від -128 до 127. Далі йде short, який займає 16 біт (від -32 768 до 32 767). Найбільш уживаним цілочисловим типом є int, що займає 32 біти і покриває величезний діапазон значень (приблизно від -2 мільярдів до +2 мільярдів). Для роботи з дуже великими числами, наприклад, у фінансових розрахунках або наукових обчисленнях, призначений тип long, що займає 64 біти.

2. **Числа з плаваючою комою**

Для представлення чисел з дробовою частиною існують два типи. Тип float (32 біти) має меншу точність і вимагає додавання суфікса f до літерала (наприклад, 2.71828f). Тип double (64 біти) пропонує подвійну точність і є стандартним вибором для більшості обчислень з дробовими числами. На практиці, зазвичай, перевага надається саме типу double.

3. **Символи**

Для представлення окремих символів, таких як літери, цифри або знаки пунктуації, використовується тип char. Він займає 16 біт, оскільки Java

використовує кодування Unicode, що дозволяє представляти символи практично всіх мов світу. Символьні літерали завжди беруться в одинарні лапки (наприклад, 'A' або '$').

4. **Булевий тип**

Особливий тип boolean може зберігати лише два можливих значення: true (істина) або false (хибність). Цей тип є основою для побудови логічних виразів та керування потоком виконання програми, наприклад, у розгалуженнях та циклах.

Оператори — це спеціальні символи, які виконують операції над одним або декількома операндами (змінними або літералами).

Арифметичні оператори: До них належать додавання (+), віднімання (-), множення (*), ділення (/) та отримання залишку від ділення (%).

Оператори інкременту та декременту: Оператор інкременту (++) збільшує значення змінної на одиницю, а декременту (--) — зменшує на одиницю. Важливо розрізняти їх префіксну та постфіксну форми. Префіксна форма (++n) спочатку змінює значення змінної, а потім використовує це нове значення у виразі. Постфіксна форма (n++) спочатку використовує поточне значення змінної у виразі, і лише після цього збільшує його.

Оператори відношення: Використовуються для порівняння двох значень і завжди повертають результат типу boolean. До них належать: дорівнює (==), не дорівнює (!=), більше (>), менше (<), більше або дорівнює (>=) та менше або дорівнює (<=).

Логічні оператори: Призначені для комбінування булевих виразів. Основні логічні оператори: логічне НЕ (!), логічне І (&&), логічне АБО (||) та виключне АБО (^).

Іноді виникає необхідність присвоїти значення одного типу змінній іншого типу. Цей процес називається **приведенням типів** (Type Casting) і буває двох видів.

```
short shortNum = 100;
int num = shortNum; // 100
```

```
double d = 2.00003;

// it loses the fractional part
long l =  (long) d; // 2
```

Рисунок 1.2 - Приклад неявного та явного приведення типів

Неявне приведення (або розширююче перетворення) відбувається автоматично, коли ми присвоюємо значення меншого типу змінній більшого типу. Наприклад, значення типу short можна без проблем присвоїти змінній типу int, оскільки при цьому не відбувається втрата даних.

```
short shortNum = 100;
int num = shortNum; // Неявне приведення
```

Явне приведення (або звужуюче перетворення) потрібне тоді, коли існує ризик втрати даних, наприклад, при спробі присвоїти значення double змінній типу long. У цьому випадку програміст повинен явно вказати, до якого типу він хоче привести значення, взявши його в круглі дужки. При такому перетворенні може бути втрачена точність або частина даних.

```
double d = 2.99;
long l = (long) d; // Явне приведення, l буде дорівнювати 2, дробова частина відкидається.
```

Java використовує Unicode для представлення символів, UTF-16 для зберігання рядків у пам'яті та UTF-8 для зберігання текстових даних у .class файлах. Також Java підтримує ASCII як частину UTF-кодувань для сумісності з іншими системами.

Java повністю підтримує стандарт Unicode, що дозволяє їй обробляти текст у різних мовах і з символами будь-яких алфавітів, це частина стандарту Java з моменту її створення.

UTF-16 використовує клас String для зберігання тексту в пам'яті. Це означає, що більшість символів займають 2 байти, але деякі символи (наприклад, емодзі) можуть займати 4 байти.

У .class файлах, де Java зберігає байт-код, текстові дані (імена класів, методів тощо) кодуються в UTF-8 для ефективного зберігання.

Для вводу/виводу (наприклад, читання і запису файлів або мережевих операцій) UTF-8 є стандартним вибором для роботи з текстом, адже він компактний і сумісний з ASCII.

Керуючі конструкції: if-else, switch-case

За замовчуванням програма на Java виконується послідовно, інструкція за інструкцією, зверху вниз. Однак для створення складних та корисних програм необхідно мати можливість змінювати цей лінійний потік виконання. Для цього існують керуючі конструкції, які дозволяють програмі "приймати рішення" і виконувати різні блоки коду залежно від певних умов. Найбільш фундаментальними конструкціями для прийняття рішень є if-else та switch-case.

Конструкція if-else є основним інструментом для реалізації розгалужень у коді. Вона дозволяє програмі перевірити певну умову і, залежно від її істинності, виконати той чи інший блок коду.

У найпростішому вигляді використовується лише оператор if. Він перевіряє умову, що міститься в круглих дужках. Якщо ця умова є істинною (true), то виконується блок коду, що знаходиться у фігурних дужках. Якщо ж умова хибна (false), цей блок коду просто ігнорується, і програма продовжує виконання з наступної інструкції після блоку if.

```
int userAge = 20;
if (userAge >= 18) {
    System.out.println("Доступ дозволено.");
}
```

Часто необхідно визначити альтернативну дію, яка має виконуватися, якщо основна умова не справдилася. Для цього використовується ключове слово else. Блок else виконується тільки тоді, коли умова в if є хибною.

```
int userAge = 16;
if (userAge >= 18) {
    System.out.println("Доступ дозволено.");
} else {
   System.out.println("Доступ заборонено. Вам ще немає 18 років.");
}
```

Для обробки більш складних сценаріїв з кількома можливими умовами можна вибудовувати ланцюжки за допомогою конструкції else if. Програма послідовно перевіряє кожну умову, починаючи з першого if. Як тільки одна з умов виявляється істинною, виконується відповідний їй блок коду, а всі наступні else if та else у ланцюжку ігноруються. Блок else в кінці такого ланцюжка є необов'язковим і спрацьовує, якщо жодна з попередніх умов не була виконана.

Розглянемо приклад, що визначає оцінку студента. Коли потрібно порівняти одну змінну з великою кількістю можливих постійних значень, ланцюжок if-else if може стати громіздким та менш читабельним. У таких ситуаціях кращою альтернативою є оператор switch-case. Він перевіряє значення змінної та передає управління тому блоку case, константа якого збігається зі значенням цієї змінної.

Структура switch-case виглядає так:

```
int dayOfWeek = 3;
String dayName;
switch (dayOfWeek) {
    case 1:
        dayName = "Понеділок";
        break;
    case 2:
        dayName = "Вівторок";
        break;
    case 3:
        dayName = "Середа";
        break;
    // ... інші дні
    default:
        dayName = "Невідомий день";
        break;
```

```
}
System.out.println(dayName); // Виведе "Середа"
```

Розберемо ключові елементи switch.

Кожен case позначає конкретне значення, з яким порівнюється змінна.

break; — оператор вказує на необхідність негайно вийти з конструкції switch після виконання коду в поточному case. Якщо break пропустити, відбудеться так зване "провалювання" (fall-through), і програма продовжить виконувати код наступних case аж доки не зустріне break або не дійде до кінця switch. Це є поширеною причиною помилок.

default: — цей блок є необов'язковим і виконується, якщо значення змінної не збіглося з жодним із значень case. Він аналогічний фінальному else у конструкції if-else if.

Сучасні версії Java (починаючи з Java 14) пропонують покращений, більш лаконічний та безпечний синтаксис для switch, який можна використовувати як вираз, що повертає значення. У ньому використовуються стрілки -> і не потрібен оператор break, оскільки "провалювання" відсутнє за замовчуванням.

```
String dayName = switch (dayOfWeek) {
    case 1 -> "Понеділок";
    case 2 -> "Вівторок";
    case 3 -> "Середа";
    // ...
    default -> "Невідомий день";
};
```

Цей новий синтаксис є кращим для використання, оскільки він зменшує ризик помилок і робить код чистішим.

На завершення, вибір між if-else та switch залежить від задачі. if-else є універсальним і підходить для перевірки діапазонів, складних логічних умов та порівняння різних змінних. switch ідеально підходить для ситуацій, коли потрібно порівняти одну змінну з набором конкретних, заздалегідь відомих констант, роблячи код більш структурованим та легким для читання.

Цикли: for, while, do-while. Масиви

Керуючі конструкції, такі як if-else, дозволяють програмі обирати між різними шляхами виконання. Однак не менш важливою є здатність програми повторювати певні дії багато разів. Для цього в програмуванні використовуються цикли. Цикли дозволяють виконувати блок коду знову і знову, доки виконується певна умова, що значно скорочує кількість коду та робить програми більш потужними. У Java є три основні види циклів: while, do-while та for.

Цикл **while** є найпростішим з точки зору синтаксису. Він складається з умови та блоку коду, і його можна описати як "поки умова істинна, виконувати

цей код". Перевірка умови відбувається перед кожною ітерацією (повторенням) циклу. Якщо умова є істинною (true), блок коду виконується. Після цього умова перевіряється знову. Цей процес повторюється, доки умова не стане хибною (false), після чого цикл завершується, і програма продовжує виконання з наступної інструкції. Оскільки перевірка відбувається на початку, такий цикл ще називають циклом з передумовою. Важливою особливістю є те, що якщо умова є хибною з самого початку, то блок коду всередині while не виконається жодного разу.

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

Цикл **do-while** є варіацією циклу while, але з однією ключовою відмінністю: перевірка умови відбувається після виконання блоку коду. Це означає, що тіло циклу do-while гарантовано виконається принаймні один раз, незалежно від того, істинна умова чи ні. Такий цикл називають циклом з постумовою. Він особливо корисний у ситуаціях, коли потрібно спочатку виконати дію, а потім перевірити, чи варто її повторювати. Наприклад, при отриманні введення від користувача.

```
import java.util.Scanner;
// ...
Scanner scanner = new Scanner(System.in);
int value;
do {
    System.out.println("Введіть число (0 для виходу):");
    value = scanner.nextInt();
    System.out.println("Ви ввели: " + value);
} while (value != 0);
// Цей цикл буде виконуватися, доки користувач не введе 0.
```

Коли кількість повторень відома заздалегідь або легко обчислюється, найзручнішим та найструктурованішим є цикл **for**. Його синтаксис є більш компактним, оскільки він об'єднує три ключові компоненти керування циклом в одному рядку:

for (ініціалізація; умова; модифікація) { ... }

Ініціалізація: Ця частина виконується один раз на самому початку циклу. Зазвичай тут оголошують та ініціалізують змінну-лічильник.

Умова: Цей логічний вираз перевіряється перед кожною ітерацією. Цикл продовжується, доки умова є істинною.

Модифікація: Ця інструкція виконується в кінці кожної ітерації. Зазвичай тут змінюють лічильник (наприклад, i++).

Ця структура робить код більш читабельним і знижує ризик створення нескінченного циклу, оскільки вся логіка керування зібрана в одному місці.

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
// Результат буде таким самим, як і в прикладі з while
```

Часто в програмах потрібно працювати не з окремими змінними, а з цілою групою однотипних даних. Для зберігання таких колекцій у Java використовується структура даних під назвою масив. **Масив** — це контейнер фіксованого розміру, який може зберігати послідовність значень одного й того ж типу.

Оголошення масиву виглядає так:

```
ТипДаних[] назваМасиву;.
int[] numbers;.
```

Після оголошення масив потрібно створити (ініціалізувати), вказавши його розмір. Розмір масиву не можна змінити після створення.

```
// Створення масиву цілих чисел на 5 елементів
int[] numbers = new int[5];
// Також можна одразу заповнити масив значеннями при створенні
String[] names = {"Аліса", "Богдан", "Вікторія"};
```

Доступ до елементів масиву здійснюється за допомогою індексу — його порядкового номера. Важливо пам'ятати, що індексація в масивах Java, як і в багатьох інших мовах, починається з нуля. Отже, перший елемент має індекс 0, другий — 1, і так далі. Останній елемент масиву розміром N матиме індекс N-1. Спроба звернутися до елемента за межами цього діапазону призведе до помилки ArrayIndexOutOfBoundsException.

Цикли є підходящим інструментом для роботи з масивами. Найчастіше для перебору всіх елементів масиву використовується цикл for, оскільки кількість елементів (ітерацій) відома — це довжина масиву, яку можна отримати через властивість .length.

```
String[] names = {"Аліса", "Богдан", "Вікторія"};
// Використання циклу for для виведення всіх елементів
for (int i = 0; i < names.length; i++) {
   System.out.println("Елемент з індексом " + i + ": " + names[i]);
}
```

Для випадків, коли потрібно просто перебрати всі елементи колекції без необхідності знати їхній індекс, Java пропонує спрощену версію циклу for, яка називається "enhanced for" або "for-each". Його синтаксис більш лаконічний та читабельний:

```
// Використання циклу for-each
for (String name : names) {
```

```
        System.out.println("Ім'я: " + name);
    }
```

Цей цикл автоматично проходить по всіх елементах масиву names, і на кожній ітерації присвоює черговий елемент змінній name.

Практична робота №1

Тема: Робота з базовими конструкціями мови

Мета: Налаштувати робоче середовище, створити першу програму та отримати навички роботи з основними синтаксичними конструкціями Java для вирішення простих задач.

Завдання

1. Налаштувати робоче середовище (JDK, IntelliJ IDEA). Рекомендована версія для лабораторних робіт: Java 21 (LTS)
2. Створити та запустити програму "Hello, World!".
3. Написати програму-калькулятор для простих арифметичних операцій (+, -, *, /) з двома числами.
4. Створити програму, яка знаходить максимальний елемент у заданому масиві чисел.

Лабораторна робота №1

Тема: Алгоритмічні задачі з використанням циклів та масивів

Мета: Закріпити навички використання циклічних конструкцій та масивів для вирішення класичних алгоритмічних задач

Завдання

1. Написати програму для сортування масиву чисел методом "бульбашки" (Bubble Sort).
2. Реалізувати програму, яка перевіряє, чи є введений рядок паліндромом.
3. Написати функцію, що обчислює факторіал числа з використанням циклу.
4. Вивести на консоль прості числа в діапазоні.

Контрольні запитання

1. Розкрийте суть принципу "Write Once, Run Anywhere". Який компонент архітектури Java є ключовим для його реалізації і чому?
2. Поясніть, у чому полягає різниця між JDK, JRE та JVM. Який з цих компонентів потрібен для розробки програми, а який — лише для її запуску?
3. Що таке байт-код? Чим він відрізняється від машинного коду?

4. Опишіть повний цикл від написання коду до його виконання на прикладі простої програми. Яка роль команд javac та java у цьому процесі?
5. Поясніть призначення та структуру методу main(). Чи може Java-програма виконатися без нього?
6. Що таке версія з довготривалою підтримкою (LTS)? Чому великі комерційні проєкти зазвичай орієнтуються саме на LTS-версії?
7. Пояснити ключові нововведення Java 8.
8. У яких ситуаціях доцільніше використовувати конструкцію switch-case замість довгого ланцюжка if-else if?
9. Яке призначення оператора break всередині блоку switch? Що станеться, якщо його пропустити?
10. Як звернутися до першого та останнього елементів у масиві з назвою dataArray?

Лекція 2. Об'єктно-орієнтоване програмування в Java

Основні поняття: клас, об'єкт, екземпляр

Після ознайомлення з базовим синтаксисом ми переходимо до вивчення парадигми, що лежить в основі Java — об'єктно-орієнтованого програмування (ООП). На відміну від процедурного підходу, де програма є послідовністю команд, ООП моделює світ як сукупність взаємодіючих об'єктів. Кожен об'єкт має власний стан (дані) та поведінку (дії, які він може виконувати). Java є високорівневою об’єктно-орієнтованою мовою програмування на основі класів, що робить розуміння її основних концепцій, таких як клас, об'єкт та екземпляр, фундаментальним для будь-якого розробника.

Клас в Java — це шаблон або прототип, що описує структуру (дані) та поведінку (методи) майбутніх об'єктів. Він є абстрактним описом сутності. Наприклад, якщо ми хочемо представити у програмі поняття "Пацієнт", ми можемо створити клас Patient. Цей клас буде описувати, які властивості має кожен пацієнт (наприклад, ім'я, вік, зріст) та що він може робити (наприклад, записуватися на прийом).

Оголошується клас за допомогою ключового слова class, за яким слідує його назва. За загальноприйнятою конвенцією, назви класів у Java пишуться у стилі PascalCase (кожна слово з великої літери). Тіло класу, що знаходиться у фігурних дужках, може містити поля (змінні, що зберігають дані об'єкта) та методи (функції, що визначають його поведінку).

```
class Patient {
    // Поля (зберігають стан об'єкта)
    String name;
    int age;
    float height;
```

```
}
```

Вихідний код класу зазвичай розміщується у файлі з розширенням .java, назва якого збігається з назвою публічного класу всередині нього.

Якщо ми розміщуємо два чи більше класів в одному .java файлі, лише один із них може бути оголошений як публічний, а ім’я файлу .java має збігатися з публічним класом у .java файлі.

Якщо клас — це креслення, то об'єкт — це реальний будинок, побудований за цим кресленням. **Об'єкт** є конкретним **екземпляром** (представником) певного класу. На основі одного класу (креслення) можна створити безліч об'єктів (будинків), і кожен з них буде існувати незалежно, маючи власний унікальний стан.

Процес створення об'єкта називається інстанціюванням. В Java для цього використовується ключове слово new, за яким слідує назва класу та круглі дужки.

```
// Створюємо два різні об'єкти (екземпляри) класу Patient
Patient john = new Patient();
Patient alice = new Patient();
```

Після створення об'єктів ми можемо працювати з їхніми полями, надаючи їм конкретні значення. Стан одного об'єкта не впливає на стан іншого.

```
john.name = "John";
john.age = 30;
alice.name = "Alice";
alice.age = 22;
System.out.println(john.name); // Виведе "John"
System.out.println(alice.name); // Виведе "Alice"
```

При створенні нового об'єкта, якщо полям не присвоєно явних значень, вони ініціалізуються значеннями за замовчуванням: числові типи — нулем, boolean — false, а всі посилальні типи — null.

Важливою особливістю роботи з об'єктами в Java є те, що вони належать до **типів посилань** (reference types). Це кардинально відрізняє їх від примітивних типів (int, double, boolean тощо). Змінна примітивного типу безпосередньо зберігає своє значення. На противагу цьому, змінна посилального типу (наприклад, Patient john) зберігає не сам об'єкт, а посилання — адресу в пам'яті, за якою цей об'єкт розташований. Уявіть, що змінна примітивного типу — це аркуш паперу, на якому написано число 30. А змінна посилального типу — це аркуш паперу, на якому написана адреса будинку. Щоб дізнатися, що всередині, потрібно піти за цією адресою.

Це має кілька важливих наслідків.

1. Якщо змінна посилального типу не вказує на жоден об'єкт, її значенням є null.

2. Оператор == для посилальних типів порівнює не вміст об'єктів, а їхні адреси в пам'яті. Тобто, він перевіряє, чи вказують дві змінні на один і той самий об'єкт. Для порівняння вмісту об'єктів слід використовувати спеціальний метод .equals().
3. Коли ви присвоюєте одну посилальну змінну іншій (Patient patient2 = john;), ви копіюєте не об'єкт, а лише посилання на нього. Після цього обидві змінні (john і patient2) будуть вказувати на один і той самий об'єкт у пам'яті. Зміна стану об'єкта через одну змінну буде видимою через іншу.

Методи: Поведінка об'єктів.

Якщо поля класу представляють його стан або властивості (умовно, "іменники"), то **методи** визначають його поведінку (умовно, "дієслова"). Метод — це іменований блок коду, який виконує певну операцію. Методи є ключовим інструментом для інкапсуляції, оскільки вони дозволяють приховати складну логіку та надати простий спосіб взаємодії з об'єктом. Саме через методи об'єкти маніпулюють своїми даними (полями) та комунікують з іншим світом.

Оголошення методу в Java має чітку структуру, яка включає кілька компонентів:

```
модифікатор_доступу тип_повернення назва_методу(список_параметрів) { ... тіло методу ... }
```

Розглянемо кожен елемент на прикладі:

```
public int calculateSum(int number1, int number2)
```

Модифікатор доступу (public): Визначає, звідки можна викликати цей метод. Ми розглянемо їх детальніше в наступному розділі.

Тип повернення (int): Вказує, який тип даних метод поверне після завершення своєї роботи. Якщо метод не повинен нічого повертати, використовується ключове слово void.

Назва методу (calculateSum): Унікальне ім'я методу в межах класу. За конвенцією, назви методів пишуться в стилі camelCase.

Список параметрів ((int number1, int number2)): Перелік змінних (вхідних даних), які метод приймає для виконання своєї роботи. Список може бути порожнім, якщо метод не потребує вхідних даних.

Тіло методу ({ ... }): Блок коду у фігурних дужках, що містить логіку, яку виконує метод. Якщо тип повернення не void, тіло методу повинно містити оператор return, який повертає значення відповідного типу.

Усі методи в Java можна розділити на дві основні категорії: екземплярні (instance) та статичні (static).

Екземплярний метод — це "стандартний" тип методу, який належить конкретному об'єкту (екземпляру) класу і працює з його унікальним станом. Такі

методи мають прямий доступ до полів свого об'єкта і можуть їх читати та змінювати. Щоб викликати екземплярний метод, необхідно спочатку створити об'єкт цього класу. Розглянемо приклад:

```
class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }
    // Екземплярний метод
    public void introduce() {
        // Має доступ до поля 'name' конкретного об'єкта
        System.out.println("Hello, my name is " + name);
    }
}
// ... в іншому місці програми
Person person1 = new Person("Alice");
Person person2 = new Person("Bob");

person1.introduce(); // Виклик екземплярного методу для об'єкта person1. Виведе "Hello, my name is Alice"
person2.introduce(); // Виклик того ж методу, але для іншого об'єкта. Виведе "Hello, my name is Bob"
```

Як видно з прикладу, хоча метод introduce() один, його результат залежить від стану конкретного об'єкта, для якого він викликаний.

На відміну від екземплярних, **статичний** метод належить не окремому об'єкту, а класу в цілому. Його оголошують за допомогою ключового слова static. Оскільки статичний метод не прив'язаний до конкретного екземпляра, його можна викликати безпосередньо через ім'я класу, не створюючи об'єкт.

Ключовою особливістю статичних методів є те, що вони не мають прямого доступу до полів та екземплярних методів класу. Це логічно, адже якщо не існує конкретного об'єкта, то не існує і його полів (name, age тощо). Статичні методи можуть працювати лише з іншими статичними полями та викликати інші статичні методи свого класу.

Статичні методи підходять для створення допоміжних, утилітарних функцій, які концептуально пов'язані з класом, але не залежать від стану його об'єктів. Класичним прикладом є клас Math у стандартній бібліотеці Java, всі методи якого (Math.max(), Math.sqrt() тощо) є статичними.

```
public class MathUtils {
    // Статичний метод
    public static int add(int a, int b) {
        return a + b;
```

```
    }
}
// ... в іншому місці програми
// Виклик статичного методу через ім'я класу, без створення об'єкта
int sum = MathUtils.add(5, 3);
System.out.println(sum); // Виведе 8
```

Отже, вибір між екземплярним та статичним методом залежить від задачі. Якщо метод повинен працювати з унікальними даними конкретного об'єкта (наприклад, person.getName()), він має бути екземплярним. Якщо ж метод виконує загальну операцію, що не потребує доступу до стану об'єкта (наприклад, MathUtils.add(a, b)), його слід робити статичним.

Інкапсуляція: модифікатори доступу (public, private, protected), гетери та сетери

Одним з чотирьох стовпів об'єктно-орієнтованого програмування, поряд з успадкуванням, поліморфізмом та абстракцією, є інкапсуляція. Щоб зрозуміти її суть, уявіть собі автомобіль. Водій взаємодіє з ним через простий та зрозумілий інтерфейс: кермо, педалі, коробка передач. Йому не потрібно знати, як саме працює двигун внутрішнього згоряння, які процеси відбуваються в трансмісії або як влаштована електронна система управління. Вся ця складна внутрішня "кухня" прихована. Водій має лише контрольований доступ до функціоналу.

Інкапсуляція в програмуванні працює за схожим принципом. Це механізм, який об'єднує дані (поля) та методи, що працюють з цими даними, в єдиний блок (клас) і, що найважливіше, приховує внутрішню реалізацію об'єкта від зовнішнього світу. Прямий доступ до стану об'єкта обмежується, а для взаємодії з ним надається публічний інтерфейс. Цей підхід переслідує дві основні цілі: чистоту коду та безпеку даних.

Основним інструментом для реалізації інкапсуляції в Java є **модифікатори доступу**. Це спеціальні ключові слова, які ставляться перед оголошенням класу, поля або методу і визначають, звідки до них можна отримати доступ. У Java існує чотири рівні доступу.

private (приватний) – це найсуворіший рівень доступу. Члени класу (поля або методи), оголошені з модифікатором private, доступні виключно всередині того самого класу, в якому вони визначені. Жоден інший клас, навіть якщо він знаходиться в тому ж пакеті, не може безпосередньо звернутися до private-члена. Зробити поля класу приватними — це перший і найважливіший крок до правильної інкапсуляції, оскільки це захищає стан об'єкта від неконтрольованих зовнішніх змін.

package-private (або **default**) – якщо перед членом класу не вказано жодного модифікатора доступу, Java за замовчуванням застосовує рівень

доступу package-private. Такі члени є видимими для будь-якого класу, що знаходиться в тому ж самому пакеті. Однак для класів з інших пакетів вони залишаються недоступними. Цей рівень доступу корисний для створення допоміжних класів або методів, які мають використовуватися лише в межах одного логічного модуля (пакета).

protected (захищений) – цей модифікатор тісно пов'язаний з механізмом успадкування. Члени класу, оголошені як protected, доступні всередині свого пакета (так само, як package-private), а також видимі для всіх підкласів (наслідників), навіть якщо ці підкласи знаходяться в інших пакетах. Це дозволяє створювати гнучкі ієрархії класів, де базовий клас надає своїм нащадкам доступ до внутрішньої реалізації, але приховує її від решти світу.

public (публічний) – це найменш суворий модифікатор, що надає найширший доступ. Члени класу, оголошені як public, є видимими з будь-якого місця програми — з будь-якого класу в будь-якому пакеті. Саме public-методи формують публічний інтерфейс класу, через який з ним взаємодіють інші об'єкти.

Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

Рисунок 2.1 - Порівняння рівня доступу для різних модифікаторів

Після того, як ми зробили поля класу приватними, виникає питання: як же з ними працювати ззовні? Прямий доступ заборонено, але нам потрібно якось читати їх значення та, можливо, змінювати їх. Для цього існують спеціальні публічні методи, які називаються гетери (getters) та сеттери (setters).

Гетери (методи доступу) призначені для читання значення приватного поля. За конвенцією, їхня назва починається з префікса get, за яким слідує назва поля з великої літери (наприклад, getName(), getAge()). Для полів типу boolean, замість get прийнято використовувати префікс is (наприклад, isMarried()).

Сеттери (методи-мутатори) призначені для зміни значення приватного поля. Їхня назва починається з префікса set, за яким слідує назва поля з великої літери (наприклад, setName(String newName)).

Використання гетерів та сеттерів надає класу повний контроль над своїми даними та дає кілька важливих переваг.

Усередині сеттера можна додати логіку перевірки. Наприклад, сеттер для віку може перевіряти, чи не є передане значення від'ємним, і у випадку помилки не змінювати поле або кидати виняток. Якщо для приватного поля створити лише геттер, але не створювати сеттер, його значення можна буде прочитати, але неможливо буде змінити ззовні. Клас може змінити внутрішній спосіб зберігання даних (наприклад, зберігати дату не в одному полі, а в трьох), але поки його публічні гетери та сеттери повертають дані у старому форматі, жоден зовнішній код не буде "зламаний".

Конструктори: їх призначення, перевантаження конструкторів

Коли ми створюємо об'єкт за допомогою ключового слова new, ми фактично запускаємо спеціальний механізм, відповідальний за його початкову ініціалізацію. Цей механізм називається конструктором.

Конструктор — це спеціальний метод, який викликається автоматично при створенні екземпляра класу для ініціалізії поля нового об'єкта, тобто присвоїти їм початкові значення.Конструктор легко відрізнити від звичайного методу за двома строгими правилами: його назва завжди повністю збігається з назвою класу та він не має типу повернення, навіть void.

Розглянемо приклад класу Patient з конструктором, який приймає параметри для ініціалізації всіх полів одразу при створенні об'єкта:

```
class Patient {
    String name;
    int age;
    float height;
    // Конструктор з параметрами
    public Patient(String name, int age, float height) {
// 'this.name' - це поле класу, а 'name' - це параметр конструктора
        this.name = name;
        this.age = age;
        this.height = height;
    } }
// Створення об'єкта з використанням конструктора
Patient patient1 = new Patient("Heinrich", 40, 182.0f);
```

У цьому прикладі, створюючи об'єкт patient1, ми одразу передаємо необхідні дані. Всередині конструктора ключове слово this використовується для посилання на поточний об'єкт, що створюється. Це дозволяє розрізнити поля класу (напр., this.name) від параметрів конструктора, які можуть мати такі ж імена (напр., name).

Що станеться, якщо ми не напишемо жодного конструктора у нашому класі? У такому випадку компілятор Java зробить це за нас. Він автоматично надасть так званий **конструктор за замовчуванням (default constructor)**. Це

буде публічний конструктор без параметрів, який просто ініціалізує всі поля класу їхніми стандартними значеннями: 0 для числових типів, false для boolean та null для всіх посилальних типів (об'єктів).

Однак тут є важливе правило: якщо ви визначаєте у класі хоча б один власний конструктор (наприклад, з параметрами), то конструктор за замовчуванням автоматично створюватися вже не буде. Це одна з поширених помилок для початківців: після додавання конструктора з параметрами, спроба створити об'єкт без параметрів (new Patient()) призведе до помилки компіляції.

Якщо вам потрібна можливість створювати об'єкти без параметрів, ви можете (і часто це необхідно) визначити конструктор без аргументів самостійно. Це дозволить вам задати більш осмислені початкові значення, ніж системні null або 0.

```
class Patient {
    String name;
    int age;
    // Явно визначений конструктор без аргументів
    public Patient() {
        this.name = "Unknown";
    }
    // Конструктор з параметрами
    public Patient(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Іноді буває зручно мати кілька способів створення об'єкта. Наприклад, ми можемо хотіти створити об'єкт Robot, не вказуючи жодних параметрів, або вказавши лише його ім'я, або вказавши і ім'я, і модель. Така ситуація, коли клас має декілька конструкторів з різними списками параметрів, називається **перевантаженням конструктора** (constructor overloading). Головна вимога — сигнатури (кількість та/або типи параметрів) конструкторів повинні відрізнятися.

```
public class Robot {
    String name;
    String model;
    // Конструктор 1 (без аргументів)
    public Robot() {
        this.name = "Anonymous";
        this.model = "Unknown";
    }  // Конструктор 2 (з двома аргументами)
    public Robot(String name, String model) {
        this.name = name;
```

```
        this.model = model;
    } }
// Тепер ми можемо створювати об'єкти двома способами
Robot anonymous = new Robot(); // name="Anonymous", model="Unknown"
Robot andrew = new Robot("Andrew", "NDR-114"); // name="Andrew", model="NDR-114"
```

При перевантаженні конструкторів часто виникає дублювання коду. Наприклад, один конструктор може просто викликати інший, більш розширений, передаючи йому якісь значення за замовчуванням. Щоб уникнути цього, в Java існує механізм виклику одного конструктора з іншого в межах того ж класу за допомогою ключового слова this().

Важливе правило: виклик this() має бути першим оператором у тілі конструктора.

```
public Robot() {
 // Викликаємо конструктор 3 з параметрами за замовчуванням
    this("Anonymous", "Unknown", 20);
}
// Конструктор 2
public Robot(String name, String model) {
// Викликаємо конструктор 3, передаючи йому отримані параметри
    // та значення за замовчуванням для lifetime
    this(name, model, 20);
}
// Конструктор 3 (основний)
public Robot(String name, String model, int lifetime) {
    this.name = name;
    this.model = model;
    this.lifetime = lifetime; } }
```

Такий підхід, що називається ланцюжком конструкторів (constructor chaining), робить код чистішим, усуває дублювання та централізує логіку ініціалізації в одному, найбільш повному конструкторі.

Ключове слово this. Поняття null

Усередині будь-якого екземплярного методу або конструктора існує спеціальне ключове слово — this. this — це посилання, яке завжди вказує на поточний об'єкт, тобто на той екземпляр класу, для якого був викликаний метод або конструктор. Це потужний інструмент, який має декілька важливих застосувань, найпоширенішим з яких є розрізнення полів класу та параметрів методу.

Дуже часто назви параметрів у конструкторах або сеттерах збігаються з назвами полів класу. Це робить код більш читабельним та інтуїтивно зрозумілим. Однак такий збіг імен створює неоднозначність: коли всередині методу ми

використовуємо ім'я, наприклад name, Java не може зрозуміти, чи маємо ми на увазі поле класу name, чи параметр методу name. У таких випадках локальна змінна (параметр) "затінює" поле класу.

Саме для вирішення цієї проблеми і використовується ключове слово this. Звертаючись до змінної через this.name, ми явно вказуємо компілятору, що маємо на увазі саме поле, яке належить поточному об'єкту, а не локальний параметр.

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

Іншим важливим застосуванням є виклик одного конструктора з іншого в межах одного класу, що ми розглядали раніше. Конструкція this() дозволяє уникнути дублювання коду ініціалізації. Крім того, ключове слово this може використовуватися для передачі посилання на поточний об'єкт як аргумент в інший метод, що часто зустрічається при роботі зі слухачами подій (event listeners).

Працюючи з посилальними типами, ми неминуче стикаємося з особливим літералом — null. null у Java означає "ніщо" — відсутність посилання на будь-який об'єкт у пам'яті. Це є значенням за замовчуванням для будь-якої неініціалізованої змінної посилального типу.

Важливо розуміти, що null — це не 0, не порожній рядок "" і не об'єкт. Це просто спеціальний маркер, який вказує, що посилальна змінна існує, але вона "порожня" і ні на що не вказує. Уявіть, що посилальна змінна — це повідець для собаки. Якщо повідець прикріплений до собаки (об'єкта), ви можете давати команди (повідець.гавкати()). Якщо ж ви тримаєте в руці порожній повідець, не прикріплений до жодної собаки, то він вказує на null.

Спроба виконати будь-яку дію зі змінною, що має значення null (наприклад, викликати її метод або звернутися до її поля), призведе до однієї з найпоширеніших помилок під час виконання програми — NullPointerException (скорочено NPE). Це відбувається тому, що програма намагається виконати команду для об'єкта, якого насправді не існує.

Щоб уникнути NullPointerException, перед використанням об'єкта слід завжди перевіряти, чи не є посилання на нього null. Найпростіший спосіб зробити це — за допомогою умовної конструкції if:

Ця проста перевірка є фундаментальним правилом безпечного програмування на Java. У подальших темах ми розглянемо більш сучасні та елегантні підходи до роботи з можливим null, зокрема за допомогою класу Optional, який з'явився у Java 8.

Статичні члени класу (static): поля та методи

Зазвичай, коли ми створюємо клас, його поля та методи є **екземплярними** (instance members). Це означає, що вони належать конкретному об'єкту (екземпляру) класу. Кожен об'єкт має власний набір значень для своїх полів (свій унікальний стан), і виклик екземплярного методу відбувається в контексті конкретного об'єкта. Однак у Java існує механізм для створення членів, які належать не окремому об'єкту, а класу в цілому. Такі члени оголошуються за допомогою ключового слова static і називаються статичними.

Статичне поле (static field), також відоме як змінна класу, — це поле, оголошене з ключовим словом static. Його фундаментальна відмінність від екземплярного поля полягає в тому, що статичне поле існує в єдиному екземплярі для всього класу, незалежно від того, скільки об'єктів цього класу було створено (чи не було створено взагалі). Усі екземпляри класу спільно використовують цю одну копію статичної змінної.

Статичні поля доцільно використовувати, коли певне значення має бути спільним для всіх об'єктів класу, виконуючи роль глобальної змінної в межах цього класу. Класичним прикладом є лічильник створених об'єктів:

```
    public static int carCount = 0;

// Доступ до статичного поля здійснюється через ім'я класу
System.out.println("Створено: " + Car.carCount); // Виведе 2
```

Як видно з прикладу, доступ до статичних полів зазвичай здійснюється безпосередньо через ім'я класу (Car.carCount), а не через змінну об'єкта, що підкреслює їхню приналежність до класу в цілому.

Якщо поєднати ключові слова static та final, ми отримаємо константу класу. Слово static означає, що вона існує в єдиному екземплярі і належить класу, а final — що її значення не можна змінити після ініціалізації. За загальноприйнятою конвенцією, імена констант пишуться у верхньому регістрі, а слова розділяються символом підкреслення (_).

Найкращим прикладом є стандартний клас java.lang.Math, який надає фундаментальні математичні константи:

```
public final class Math {
    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
    // ...
}
```

Доступ до цих констант здійснюється глобально через ім'я класу: Math.PI.

За аналогією зі статичними полями, статичний метод — це метод, що належить класу, а не окремому об'єкту. Він оголошується за допомогою

ключового слова static і може бути викликаний безпосередньо через ім'я класу, без необхідності створювати екземпляр цього класу.

Найважливіше правило, що стосується статичних методів: вони не мають прямого доступу до екземплярних (нестатичних) полів та методів. Це логічно, оскільки статичний метод не пов'язаний з жодним конкретним об'єктом, а отже, він не має доступу до посилання this і не знає, стан якого саме об'єкта йому потрібно прочитати чи змінити. Статичні методи можуть працювати лише з іншими статичними полями та викликати інші статичні методи.

Статичні методи ідеально підходять для створення утилітарних функцій — допоміжних операцій, які концептуально пов'язані з класом, але для своєї роботи не потребують доступу до стану конкретного об'єкта.

Отже, при проєктуванні класу слід чітко розмежовувати екземплярні члени (поля та методи без static), що використовуються для опису унікального стану та поведінки кожного окремого об'єкта, та статичні члени (поля та методи з static) використовуються для зберігання спільних даних та реалізації загальних функцій, що стосуються класу в цілому.

Анотації та їх роль (Annotations)

У міру того, як програми стають складнішими, виникає потреба додавати до коду додаткову інформацію, яка не є частиною його основної логіки, але слугує для інших цілей. В Java для цього існує потужний механізм, що називається анотаціями.

Анотація — це форма метаданих, тобто "даних про дані", яку можна додати до елементів коду Java (класів, методів, полів, параметрів). Уявіть анотацію як спеціальну позначку, ярлик або стікер, який ви прикріплюєте до частини вашого коду. Цей стікер не змінює того, що робить сам код, але надає корисну інформацію для компілятора, середовища розробки (IDE), інших інструментів або для самої програми під час її виконання. Синтаксично анотації легко впізнати: вони завжди починаються із символу @, за яким слідує назва анотації, наприклад, @Override.

Роль анотацій можна розділити на кілька основних категорій.

1. Інформація для компілятора

Деякі анотації допомагають компілятору виявляти потенційні помилки та надавати попередження. Це один з найпростіших та найпоширеніших способів використання анотацій.

@Override: Це, найчастіше, перша анотація, з якою стикається кожен Java-розробник. Вона розміщується над методом і вказує компілятору, що цей метод призначений для перевизначення методу з батьківського класу. Якщо ви припуститеся помилки в назві методу або його параметрах, і він насправді не

буде нічого перевизначати, компілятор повідомить про помилку. Без цієї анотації помилку було б легко пропустити, що могло б призвести до неочікуваної поведінки програми.

@Deprecated: Ця анотація позначає метод, клас або поле як застаріле. Вона сигналізує іншим розробникам, що цей елемент коду не слід використовувати в нових розробках, оскільки він може бути видалений у майбутніх версіях або замінений на більш ефективний аналог. Компілятор видасть попередження при кожній спробі використання застарілого коду.

2. Інформація для інструментів розробки та генерації коду

Анотації активно використовуються різноманітними інструментами та бібліотеками для автоматизації рутинних завдань. Наприклад, вони можуть аналізуватися для генерації додаткового коду, документації або для статичного аналізу якості коду. Сучасні фреймворки, такі як Spring (який ми будемо вивчати пізніше), активно використовують анотації (@Component, @Autowired, @GetMapping тощо) для налаштування та конфігурації програми, що дозволяє значно зменшити кількість шаблонного коду.

3. Інформація для обробки під час виконання програми

Деякі анотації зберігаються у скомпільованих .class файлах і можуть бути зчитані самою програмою під час її роботи за допомогою механізму, що називається рефлексією. Це дозволяє створювати дуже гнучкі системи, поведінка яких може змінюватися залежно від наявності тих чи інших анотацій у коді.

Чудовим прикладом анотації, що надає інформацію для компілятора, є @FunctionalInterface. Як ми розглянемо далі, функціональний інтерфейс — це будь-який інтерфейс, що містить рівно один абстрактний метод. Саме ця властивість дозволяє використовувати його разом з лямбда-виразами.

Анотація @FunctionalInterface не є обов'язковою, але її використання є хорошою практикою. Вона працює за тим же принципом, що й @Override. Коли ви позначаєте нею інтерфейс, ви даєте компілятору чітку вказівку: "Перевір, будь ласка, чи цей інтерфейс дійсно є функціональним". Якщо ви випадково додасте до такого інтерфейсу другий абстрактний метод, компілятор одразу повідомить про помилку, не давши вам скомпілювати код. Це захищає від ненавмисних змін, які можуть "зламати" контракт інтерфейсу і зробити неможливим його використання з лямбда-виразами. Варто зазначити, що наявність default методів не впливає на цей підрахунок, оскільки вони не є абстрактними.

```
@FunctionalInterface // Ця анотація захищає контракт інтерфейсу
public interface Calculable {
    int calculate(int x, int y); // Єдиний абстрактний метод

    // default метод не враховується
    default void printResult(int result) {
```

```
            System.out.println("Result is: " + result);
        }
    }
```

Таким чином, анотації — це потужний інструмент, який, не змінюючи логіку програми, додає до неї важливий шар метаданих. Вони роблять код більш безпечним, читабельним, зменшують кількість шаблонних операцій та є основою для роботи багатьох сучасних фреймворків та бібліотек.

Функціональні інтерфейси (@FunctionalInterface)

З появою у Java 8 елементів функціонального програмування, таких як лямбда-вирази, ключову роль в екосистемі мови почали відігравати функціональні інтерфейси. Розуміння їхньої природи та призначення є необхідним для написання сучасного, лаконічного та виразного коду.

Функціональний інтерфейс – це будь-який інтерфейс, який визначає рівно один абстрактний метод. Ця умова є його головною і єдиною вимогою. Такий інтерфейс може містити будь-яку кількість default або static методів, оскільки вони мають реалізацію і не є абстрактними, але абстрактний метод має бути лише один. Саме цей простий "контракт одного методу" дозволяє компілятору розглядати екземпляри такого інтерфейсу як окремі функції, які можна передавати, зберігати та виконувати.

Для того, щоб явно позначити інтерфейс як функціональний і доручити компілятору стежити за дотриманням цього правила, використовується анотація @FunctionalInterface. Вона не є обов'язковою, але її використання є хорошою практикою, оскільки вона захищає інтерфейс від випадкових змін. Якщо хтось спробує додати другий абстрактний метод до інтерфейсу, позначеного цією анотацією, компілятор видасть помилку.

Призначення та переваги

Головне призначення функціональних інтерфейсів — слугувати типом для лямбда-виразів та посилань на методи. Вони є тим мостом, який дозволяє передавати поведінку (тобто, фрагмент коду) як аргумент в інший метод. Це відкриває широкі можливості для написання більш гнучкого та модульного коду.

Використання функціональних інтерфейсів дає кілька значних переваг. До Java 8 для передачі поведінки використовувалися анонімні внутрішні класи, що робило код багатослівним. Функціональні інтерфейси разом з лямбда-виразами дозволяють досягти того ж результату значно лаконічніше та зрозуміліше. Можливість передавати окремі блоки логіки у вигляді параметрів дозволяє краще структурувати код, відокремлюючи загальні алгоритми від конкретних дій. Це особливо помітно при роботі з колекціями, наприклад, при фільтрації, сортуванні або трансформації даних.

Java надає великий набір **стандартних функціональних інтерфейсів** у пакеті java.util.function. У більшості випадків немає потреби створювати власні інтерфейси, оскільки можна скористатися одним із готових. Розглянемо найважливіші з них:

Predicate<T> представляє функцію, що приймає один аргумент типу T і повертає логічне значення boolean. Ідеально підходить для будь-яких перевірок та фільтрації. Основний метод — test().

Function<T, R> описує функцію, яка приймає аргумент типу T і перетворює його на результат типу R. Це основний інтерфейс для операцій трансформації даних (маппінгу). Основний метод — apply().

Consumer<T> представляє операцію, яка приймає один аргумент типу T і нічого не повертає (void). Використовується для виконання певної дії над об'єктом, наприклад, виведення в консоль або збереження в базу даних. Основний метод — accept().

Supplier<T> є "постачальником" даних. Цей інтерфейс не приймає жодних аргументів, але повертає результат типу T. Використовується для створення або генерації нових об'єктів. Основний метод — get().

UnaryOperator<T> та BinaryOperator<T>: Це специфічні випадки Function. UnaryOperator<T> приймає об'єкт типу T і повертає результат того ж типу T (наприклад, операція інкременту). BinaryOperator<T> приймає два об'єкти типу T і повертає результат того ж типу T (наприклад, додавання двох чисел).

Крім того, існують "Bi"-версії цих інтерфейсів для роботи з двома аргументами (напр., BiFunction<T, U, R>, BiPredicate<T, U>), а також спеціалізовані версії для роботи з примітивними типами (напр., IntPredicate, DoubleConsumer), що дозволяє уникнути зайвих операцій "упаковки" примітивів в об'єкти-обгортки.

Функціональні інтерфейси є невід'ємною частиною сучасної розробки на Java. Вони забезпечують елегантний спосіб роботи з логікою як з даними, що значно спрощує розробку, підвищує продуктивність та дозволяє писати більш гнучкий та модульний код. Вони є основою для роботи з потужними API, такими як Streams API, які ми розглянемо пізніше.

Лямбда-вирази, синтаксис та використання

Одним з найважливіших нововведень Java 8, що кардинально змінило стиль написання коду, стали лямбда-вирази. **Лямбда-вираз** — це анонімна (безіменна) функція, яку можна розглядати як блок коду, що може бути збережений у змінну, переданий як аргумент в інший метод або повернутий з методу. Вони є основним інструментом для роботи у функціональному стилі в Java.

Головне призначення лямбда-виразу — надати коротку та лаконічну реалізацію для функціонального інтерфейсу, тобто інтерфейсу з одним абстрактним методом. Лямбда-вираз не існує сам по собі; він завжди асоціюється з певним функціональним інтерфейсом, реалізуючи його єдиний метод.

Основою синтаксису є лямбда-оператор -> (стрілка), який розділяє вираз на дві частини: ліворуч — список параметрів, праворуч — тіло виразу, що містить логіку.

Загальна структура: (параметри) -> { тіло; }

Цей синтаксис є дуже гнучким і може бути спрощений у багатьох випадках. Компілятор Java зазвичай може сам визначити (вивести) типи параметрів з контексту (тобто з опису методу в функціональному інтерфейсі). Тому явно вказувати їх не обов'язково.

Якщо параметр лише один, круглі дужки навколо нього можна опустити. Якщо параметрів немає, необхідно вказати порожні круглі дужки ().

Якщо тіло складається лише з однієї інструкції, фігурні дужки {} можна опустити. Якщо ця інструкція повертає значення, ключове слово return також опускається.

Якщо тіло містить декілька інструкцій, його необхідно взяти у фігурні дужки. У такому випадку, якщо метод повинен повертати значення, необхідно явно використовувати оператор return.

Розглянемо на прикладі функціонального інтерфейсу Operationable:

```
@FunctionalInterface
interface Operationable {
    int calculate(int x, int y);
} // ...
// Повна форма лямбда-виразу
Operationable operation1 = (int x, int y) -> { return x + y; };
// Спрощена форма: типи параметрів виведені компілятором,
// фігурні дужки та return опущені, оскільки тіло складається з одного виразу
Operationable operation2 = (x, y) -> x + y;
int result = operation2.calculate(10, 20); // result буде 30
```

Найпотужнішою можливістю лямбда-виразів є здатність передавати поведінку як параметр методу. Це дозволяє писати більш гнучкі та універсальні методи. Наприклад, ми можемо написати метод, що сортує список, і передати йому логіку порівняння у вигляді лямбда-виразу.

```
List<String> names = new ArrayList<>();
names.add("Вікторія");
names.add("Аліса");
names.add("Богдан");
```

```
// Передача лямбда-виразу як другого аргумента в метод sort
// Лямбда (a, b) -> a.compareTo(b) реалізує функціональний інтерфейс Comparator
Collections.sort(names, (a, b) -> a.compareTo(b));
System.out.println(names); // Виведе: [Аліса, Богдан, Вікторія]
```

Лямбда-вирази мають доступ до змінних із зовнішнього контексту, але з певними обмеженнями. Вони можуть вільно читати та змінювати поля класу (як статичні, так і екземплярні). Однак, якщо лямбда-вираз використовує локальну змінну методу, в якому він визначений, ця змінна повинна бути final або effectively final. Effectively final означає, що значення змінної не змінюється після її ініціалізації. Це обмеження необхідне для забезпечення безпеки при роботі з багатопотоковістю.

Часто лямбда-вираз просто викликає вже існуючий метод. У таких випадках для ще більшої лаконічності можна використовувати **посилання на метод (method reference)**. Це спеціальний синтаксис з подвійною двокрапкою ::, який дозволяє посилатися на метод без його прямого виклику.

Існує кілька видів посилань на методи.

1. На статичний метод: ClassName::staticMethodName
2. На екземплярний метод конкретного об'єкта: object::instanceMethodName
3. На екземплярний метод довільного об'єкта певного типу: ClassName::instanceMethodName
4. На конструктор: ClassName::new

За можливості, рекомендується надавати перевагу посиланням на методи, а не лямбда-виразам, оскільки вони часто є більш ефективними та надають компілятору кращу інформацію про типи.

Практична робота №2

Тема: Створення та використання класів

Мета: Набути практичних навичок у створенні класів, інкапсуляції даних, використанні конструкторів та статичних полів

Завдання

1. Створити клас Student з полями: name, age, major.

2. Інкапсулювати поля класу Student, додавши приватні модифікатори та публічні гетери/сетери.

3. Додати в клас Student конструктор, що ініціалізує всі поля.

4. Створити декілька екземплярів класу Student у main методі та вивести інформацію про них.

5. Додати в клас статичне поле studentCount для підрахунку кількості створених об'єктів.

Лабораторна робота №2

Тема: Взаємодія між об'єктами

Мета: Навчитися проєктувати взаємодію між різними об'єктами та використовувати функціональні інтерфейси для гнучкої фільтрації даних

Завдання

1. Створити клас Book з полями title, author, year.
2. Створити клас Library, який містить масив об'єктів Book.
3. Реалізувати в класі Library метод для додавання нової книги.
4. Використати функціональний інтерфейс: Реалізувати в Library універсальний метод findBooks(Predicate<Book> filter), який приймає умову у вигляді лямбда-виразу і повертає список книг, що задовольняють цій умові.

Контрольні запитання

1. У чому полягає фундаментальна різниця між класом та об'єктом в об'єктно-орієнтованому програмуванні? Наведіть аналогію для пояснення.
2. Що таке інкапсуляція і яку роль у її реалізації відіграють модифікатори доступу? Поясніть різницю у видимості для членів класу, оголошених як private та public.
3. Для чого використовуються гетери та сеттери? Назвіть щонайменше дві переваги їх використання порівняно з прямим доступом до публічних полів.
4. Що таке конструктор за замовчуванням і в якому випадку компілятор Java не створює його автоматично для класу?
5. Поясніть два основних призначення ключового слова this.
6. У чому полягає ключова відмінність між статичним та екземплярним (нестатичним) методом? Чому статичний метод не може безпосередньо звернутися до поля екземпляра?
7. Яка загальна роль анотацій у Java? Наведіть приклад анотації, що надає інформацію для компілятора (наприклад @Override або @FunctionalInterface), і поясніть, що саме вона робить.
8. Що робить інтерфейс "функціональним"? Назвіть два стандартних функціональних інтерфейси з пакета java.util.function та опишіть їх призначення (наприклад, Predicate, Consumer, Function).
9. З яких основних частин складається синтаксис лямбда-виразу? В яких випадках можна опустити фігурні дужки {} та ключове слово return при написанні лямбди?

10. Що таке посилання на метод (method reference) і в яких ситуаціях його доцільно використовувати замість лямбда-виразу?

ТЕМА 2. ПРИНЦИПИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ДИЗАЙНУ (SOLID)

Лекція 3. Успадкування та Поліморфізм. Принцип LSP

Успадкування, ключове слово extends, ієрархія класів

Переходячи до більш глибоких концепцій об'єктно-орієнтованого програмування, ми стикаємося з одним із його наріжних каменів — успадкуванням. Цей потужний механізм дозволяє створювати нові класи на основі вже існуючих, що не тільки сприяє повторному використанню коду, але й дає змогу будувати логічні та зрозумілі ієрархії сутностей у програмі.

Успадкування — це властивість, що дозволяє описати новий клас на основі вже існуючого, запозичуючи його функціональність частково або повністю. Уявіть це як у реальному світі: дитина успадковує певні риси від своїх батьків, але водночас має і свої унікальні особливості. У програмуванні цей принцип працює аналогічно.

У контексті успадкування використовуються два основні терміни.

Суперклас (Superclass) – це батьківський, або базовий, клас, властивості та поведінку якого ми успадковуємо. Він слугує загальним шаблоном.

Підклас (Subclass) – це дочірній, або похідний, клас, який розширює суперклас. Він успадковує члени (поля та методи) суперкласу і може додавати власні, специфічні для нього.

Фундаментальним для успадкування є відношення "IS-A" ("є"). Це означає, що об'єкт підкласу завжди є також і об'єктом суперкласу. Наприклад, якщо ми маємо клас Employee (Працівник), що успадковує клас Person (Людина), то ми можемо стверджувати, що "Працівник є Людина". Базовий клас представляє загальну концепцію, а підклас — її більш специфічну, конкретну реалізацію.

У мові Java для реалізації механізму успадкування використовується ключове слово extends. Підклас "розширює" суперклас, отримуючи доступ до всіх його public та protected полів і методів. private члени суперкласу не успадковуються безпосередньо і недоступні в підкласі. Синтаксис виглядає наступним чином:

```
class SuperClass {
    // ...
}

class SubClass extends SuperClass {
    // ...
}
```

Успадкування дозволяє створювати складні, багаторівневі ієрархії класів. Наприклад, клас Programmer може успадковувати клас Employee, який, у свою чергу, успадковує клас Person.

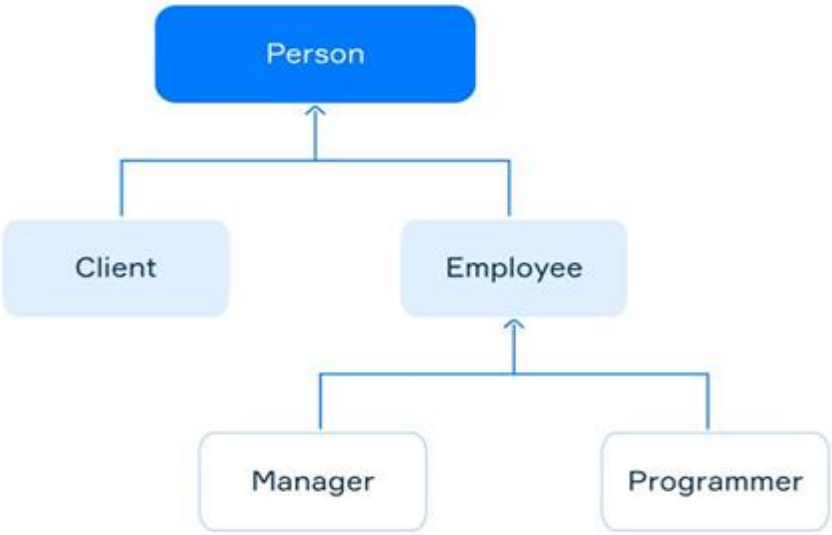

Рисунок 3.1 - Схема успадкування класів

При побудові ієрархій в Java необхідно пам'ятати про кілька важливих правил.

1. На відміну від деяких інших мов, у Java клас може **успадковувати лише один суперклас**. Це зроблено для уникнення складнощів та неоднозначностей, пов'язаних з множинним успадкуванням.
2. Один суперклас може мати необмежену кількість підкласів.
3. Підклас не успадковує конструктори свого батьківського класу. Однак він може (і часто повинен) викликати конструктор суперкласу.

Перевизначення методів (@Override). Ключове слово super

Успадкування дозволяє підкласу запозичувати поведінку свого суперкласу. Однак часто виникає потреба, щоб підклас реагував на той самий виклик методу по-своєму, надаючи більш специфічну або повністю змінену реалізацію. Цей механізм називається перевизначенням методів (method overriding).

Перевизначення — це можливість підкласу надати власну реалізацію для методу, який він успадкував від суперкласу. Наприклад, загальний клас Animal може мати метод makeSound(), який нічого не робить, а його підкласи Dog та Cat можуть перевизначити цей метод, щоб Dog виводив "Гав!", а Cat — "Няв!". Це дозволяє об'єктам по-різному реагувати на однаковий запит, що є основою поліморфізму.

Щоб сигналізувати компілятору та іншим розробникам про намір перевизначити метод, використовується анотація @Override. Хоча вона не є суворо обов'язковою, її використання є стандартом хорошої практики. Якщо ви розмістите @Override над методом, компілятор перевірить, чи дійсно в суперкласі існує метод з такою ж сигнатурою. Якщо ви припуститеся помилки

(наприклад, одрук в назві методу), компілятор повідомить про помилку, запобігаючи створенню нового методу замість перевизначення існуючого.

Щоб метод був коректно перевизначений, він повинен відповідати кільком строгим правилам:

1. Метод у підкласі повинен мати ту ж саму назву та той самий набір параметрів (ту саму сигнатуру), що й метод у суперкласі.
2. Рівень доступу перевизначеного методу не може бути більш суворим, ніж у суперкласі. Наприклад, не можна перевизначити public метод як protected, але можна protected метод перевизначити як public.
3. Тип повернення повинен бути таким самим або підтипом (коваріантним) типу повернення, оголошеного в методі суперкласу.
4. Методи, оголошені як private, не успадковуються, а отже, і не можуть бути перевизначені.
5. Методи, оголошені як static, не можуть бути перевизначені. Вони належать класу, а не екземпляру, тому для них застосовується інший механізм, що називається "приховуванням" (hiding).

Якщо ви хочете заборонити підкласам змінювати реалізацію певного методу, ви можете оголосити його з ключовим словом final. Будь-яка спроба перевизначити final-метод призведе до помилки компіляції.

При перевизначенні методу не завжди потрібно повністю замінювати логіку батьківського класу. Часто необхідно лише доповнити її. Для того, щоб звернутися до членів (полів та методів) суперкласу зсередини підкласу, використовується ключове слово super. Воно надає посилання на батьківський об'єкт. super має два основні застосування:

1. Виклик методів та доступ до полів суперкласу

Якщо ви перевизначили метод, але всередині нього хочете викликати оригінальну реалізацію з батьківського класу, ви можете зробити це за допомогою super.імяМетоду(). Це дозволяє розширювати функціональність, а не повністю її замінювати. Аналогічно, якщо підклас оголошує поле з тим самим ім'ям, що й у суперкласі (це називається "приховуванням поля"), за допомогою super.імяПоля можна звернутися саме до поля батьківського класу.

```
class SuperClass {
    protected int field = 10;
    protected void printValue() {
        System.out.println("SuperClass value: " + field);
    }
}

class SubClass extends SuperClass {
    protected int field = 20; // Це поле приховує поле з SuperClass
```

```
    @Override
    protected void printValue() {
       super.printValue(); // Виклик реалізації методу з SuperClass
        System.out.println("SubClass value: " + this.field);
       System.out.println("Value from SuperClass: " + super.field);
    }
}
```

2. Виклик конструктора суперкласу

Це найважливіше і найчастіше використання super. Як зазначалося раніше, конструктори не успадковуються. Проте, першим завданням будь-якого конструктора підкласу є виклик конструктора свого суперкласу, щоб коректно ініціалізувати успадковану частину об'єкта. Для цього використовується синтаксис super(). При цьому діють два фундаментальні правила.

Виклик конструктора суперкласу super(...) повинен бути першим оператором у тілі конструктора підкласу.

Якщо ви явно не викликаєте super(...), компілятор автоматично додасть на початок конструктора виклик конструктора суперкласу без аргументів (super()). Якщо у суперкласі немає такого конструктора (наприклад, є лише конструктор з параметрами), це призведе до помилки компіляції.

Розглянемо приклад ініціалізації працівника, який успадковує властивості людини:

```
class Person {
    protected String name;
    protected int yearOfBirth;
    public Person(String name, int yearOfBirth) {
        this.name = name;
        this.yearOfBirth = yearOfBirth;
    }
}

class Employee extends Person {
    protected long salary;

    public Employee(String name, int yearOfBirth, long salary) {
    // Першим викликаємо конструктор батьківського класу Person
        // для ініціалізації полів name та yearOfBirth
        super(name, yearOfBirth); [cite: 334]
        // Після цього ініціалізуємо власне поле класу Employee
        this.salary = salary;
    }
}
```

У цьому прикладі конструктор Employee делегує ініціалізацію успадкованих полів конструктору Person за допомогою super(), а сам займається лише ініціалізацією полів, специфічних для Employee. Це забезпечує правильний та логічний ланцюжок створення об'єкта.

Поліморфізм, один інтерфейс, багато реалізацій

Слово "поліморфізм" походить з грецької мови і буквально означає "багато форм". В об'єктно-орієнтованому програмуванні поліморфізм — це здатність об'єкта приймати різні форми або, точніше, здатність об'єктів різних класів по-різному реагувати на один і той самий виклик методу. Це одна з найпотужніших концепцій ООП, яка дозволяє писати гнучкий, розширюваний та легко підтримуваний код. Суть поліморфізму можна описати принципом: "один інтерфейс — багато реалізацій". Це означає, що ми можемо визначити загальну дію (інтерфейс) у базовому класі, а кожен з його нащадків надасть власну, унікальну реалізацію цієї дії.

У Java поліморфізм проявляється у кількох формах, але найважливішою для нас є **поліморфізм підтипу** (Subtype Polymorphism), також відомий як поліморфізм часу виконання. Саме він реалізує ідею "багатьох реалізацій". Його робота базується на двох фундаментальних принципах, що тісно пов'язані з успадкуванням та перевизначенням методів:

1. Посилальна змінна суперкласу може посилатися на об'єкт будь-якого з його підкласів. Це означає, що ми можемо створити змінну типу Animal і присвоїти їй об'єкт класу Dog або Cat.
2. Метод суперкласу може бути перевизначений у підкласі. Підклас може надати власну, специфічну реалізацію для методу, успадкованого від батька.

Якщо ми викликаємо перевизначений метод через посилальну змінну суперкласу, Java використовує механізм, що називається пізнім зв'язуванням (late binding) або динамічною диспетчеризацією методів. Це означає, що рішення про те, яку саме версію методу викликати (з суперкласу чи з одного з підкласів), приймається не на етапі компіляції, а під час виконання програми. JVM аналізує фактичний тип об'єкта, на який посилається змінна в даний момент, і викликає відповідну версію методу.

Розглянемо класичний приклад. У нас є базовий клас MythicalAnimal та два його нащадки: Chimera і Dragon. Кожен з них по-своєму реалізує метод hello().

```
class MythicalAnimal {
    public void hello() {
        System.out.println("Hello, I'm an unknown animal");
    }
}
```

```
class Chimera extends MythicalAnimal {
    @Override
    public void hello() {
        System.out.println("Hello! Hello!");
    }
}
class Dragon extends MythicalAnimal {
    @Override
    public void hello() {
        System.out.println("Rrrr...");
    }
}
```

Тепер, завдяки поліморфізму, ми можемо створити змінні типу MythicalAnimal, але присвоїти їм об'єкти підкласів:

```
// Змінна типу суперкласу посилається на об'єкт підкласу
MythicalAnimal chimera = new Chimera();
MythicalAnimal dragon = new Dragon();
MythicalAnimal animal = new MythicalAnimal();

// Виклик одного й того ж методу hello()
chimera.hello(); // реалізація з класу Chimera -> Виведе: "Hello! Hello!"
dragon.hello();  // реалізація з класу Dragon -> Виведе: "Rrrr..."
animal.hello();   // реалізація з класу MythicalAnimal -> Виведе: "Hello, I'm an unknown animal"
```

У цьому прикладі, хоча всі три змінні (chimera, dragon, animal) мають однаковий тип MythicalAnimal, результат виклику методу hello() залежить від фактичного типу об'єкта, що зберігається в кожній змінній. JVM "дивиться" на реальний об'єкт і викликає його версію методу.

Саме в цьому і полягає принцип "один інтерфейс, багато реалізацій". Загальний "інтерфейс" (у даному випадку, публічний метод hello() з класу MythicalAnimal) залишається незмінним, але його "реалізацій" (конкретних версій методу) може бути багато — по одній на кожен підклас.

Поліморфізм підтипів є фундаментальною концепцією, оскільки він дозволяє писати код, який працює з об'єктами на більш загальному, абстрактному рівні. Наприклад, ми можемо створити метод, який приймає масив об'єктів MythicalAnimal і для кожного викликає метод hello(), не знаючи наперед, чи будуть це дракони, химери чи інші міфічні істоти. Система сама подбає про те, щоб для кожного об'єкта була викликана правильна версія методу. Це робить програми надзвичайно гнучкими, дозволяючи легко додавати нові підкласи з новою поведінкою, не змінюючи при цьому існуючий код, що працює з базовим типом.

Принцип заміщення Лісков, поведінка похідних класів

Успадкування є потужним інструментом, але його неправильне використання може призвести до створення крихких та нелогічних ієрархій класів. Щоб допомогти проєктувати надійні системи, було сформульовано низку принципів, відомих як SOLID. Принцип заміщення Лісков (Liskov Substitution Principle, LSP) є літерою "L" у цій абревіатурі і є одним з найважливіших, хоча й одним з найбільш тонких для розуміння.

Формально, принцип, сформульований Барбарою Лісков, звучить так: "Нехай q(x) є властивістю, доведеною для об'єктів x деякого типу T. Тоді q(y) має залишатися істинним для об'єктів у типу S, де S є підтипом T".

Якщо спростити, суть LSP зводиться до наступного: об'єкти підкласу повинні бути взаємозамінними з об'єктами їхнього суперкласу без зміни коректності роботи програми. Це означає, що будь-який фрагмент коду, який працює з об'єктом базового класу, повинен продовжувати працювати так само коректно та передбачувано, якщо йому передати об'єкт похідного класу. Підклас не повинен "ламати" очікувану поведінку, визначену суперкласом. Іншими словами, успадкування "IS-A" ("є") має бути не лише на рівні властивостей, а й на рівні поведінки.

Класичний приклад порушення LSP: Прямокутник та Квадрат. На перший погляд, створення класу Square (Квадрат), що успадковує клас Rectangle (Прямокутник), здається абсолютно логічним. Адже з точки зору геометрії, квадрат є прямокутником, у якого ширина дорівнює висоті. Давайте спробуємо реалізувати цю ієрархію, як це описано у завданні до лабораторної роботи №3.

Спочатку створимо базовий клас Rectangle:

```
public class Rectangle {
    protected int width;
    protected int height;
    public void setWidth(int width) {
        this.width = width;
    }
    public void setHeight(int height) {
        this.height = height;
    }
    public int getArea() {
        return this.width * this.height;
    }
}
```

Клас Rectangle має неявний, але очевидний "контракт поведінки": зміна ширини не впливає на висоту, і навпаки.

Тепер створимо клас Square, який успадковує Rectangle. Щоб зберегти властивість квадрата (де всі сторони рівні), ми змушені перевизначити сеттери:

```
public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width; // Порушуємо поведінку батька
    }

    @Override
    public void setHeight(int height) {
        this.width = height;
        this.height = height; // Порушуємо поведінку батька
    }
}
```

А тепер уявімо, що у нас є метод, який працює з прямокутниками і перевіряє площу:

```
public class AreaCalculator {
    public static void calculateAndCheck(Rectangle r) {
        r.setWidth(5);
        r.setHeight(10);

        // Ми очікуємо, що площа буде 5 * 10 = 50
        System.out.println("Очікувана площа: 50");
        System.out.println("Фактична площа: " + r.getArea());
    }
}
```

Що станеться, коли ми викличемо цей метод?

```
Rectangle rect = new Rectangle();
Square square = new Square();
System.out.println("Тестуємо прямокутник:");
AreaCalculator.calculateAndCheck(rect);//Все працює, як очікувалось
System.out.println("\nТестуємо квадрат:");
AreaCalculator.calculateAndCheck(square); // Тут виникає проблема
```

Результат виконання буде таким:

```
Тестуємо прямокутник:
Очікувана площа: 50
Фактична площа: 50
Тестуємо квадрат:
Очікувана площа: 50
Фактична площа: 100
```

Для об'єкта Square результат виявився неочікуваним. Метод calculateAndCheck отримав об'єкт, який формально є Rectangle, але поводиться

інакше. Виклик r.setHeight(10) для квадрата несподівано змінив і його ширину, що призвело до невірного результату. Об'єкт Square виявився не взаємозамінним з об'єктом Rectangle без порушення коректності програми. Це і є порушенням Принципу заміщення Лісков.

Проблема виникла через те, що ми базували успадкування на властивостях ("квадрат - це прямокутник"), ігноруючи поведінку. Квадрат порушує інваріант прямокутника, згідно з яким ширина та висота є незалежними.

Правильним рішенням у даному випадку є відмова від успадкування, як це пропонується в лабораторній роботі. Замість цього слід створити більш загальну абстракцію, наприклад, інтерфейс Shape (Фігура), і реалізувати його в обох класах:

```
public interface Shape {
    int getArea();
}
public class Rectangle implements Shape {
    // ... реалізація для прямокутника
}
public class Square implements Shape {
    // ... незалежна реалізація для квадрата
}
```

Такий дизайн є коректним, оскільки він не створює хибних поведінкових очікувань.

Отже, Принцип заміщення Лісков вчить нас, що ієрархія класів має базуватися не лише на спільних властивостях, а й на сумісній поведінці. Підклас повинен лише розширювати функціональність суперкласу, а не змінювати його фундаментальні поведінкові контракти..

Успадкування vs. Композиція: коли і що обирати

При проєктуванні класів однією з головних цілей є повторне використання коду. В об'єктно-орієнтованому програмуванні для досягнення цієї мети існують два фундаментальні підходи: успадкування та композиція. Вибір між ними є одним з найважливіших архітектурних рішень, яке суттєво впливає на гнучкість, надійність та підтримуваність вашої системи. Хоча успадкування часто є першим механізмом, який спадає на думку, досвідчені розробники дотримуються принципу: "Надавайте перевагу композиції над успадкуванням". Розглянемо, чому це так.

Як ми вже знаємо, успадкування моделює відношення "IS-A" ("є"). Ми використовуємо його, коли підклас є справжнім підтипом суперкласу. Наприклад, Dog "є" Animal. Цей підхід забезпечує пряме повторне використання коду, оскільки підклас автоматично отримує всю не-приватну функціональність

батька. Головною перевагою успадкування є підтримка поліморфізму: об'єкт Dog можна використовувати скрізь, де очікується об'єкт Animal.

Однак у спадкування є суттєві недоліки.

Підклас дуже тісно пов'язаний зі своїм суперкласом. Будь-яка зміна в реалізації суперкласу, навіть незначна, може несподівано "зламати" логіку в усіх його нащадках. Ця проблема відома як "крихкість базового класу" (Fragile Base Class Problem).

Як ми бачили на прикладі Rectangle та Square, іноді логічний, на перший погляд, зв'язок "IS-A" виявляється хибним з точки зору поведінки, що призводить до порушення Принципу заміщення Лісков.

У Java клас може успадковувати лише один інший клас. Якщо вам потрібно запозичити функціональність з кількох джерел, успадкування вам не допоможе.

Підклас успадковує всі public та protected методи суперкласу, навіть якщо йому потрібен лише один з них. Це може призвести до того, що підклас матиме занадто широкий та нелогічний публічний інтерфейс.

Композиція — це альтернативний підхід до повторного використання коду, що моделює відношення "HAS-A" ("має"). Замість того, щоб бути чимось, клас має щось. При цьому підході один клас включає в себе екземпляр іншого класу як своє поле і делегує йому виконання певних завдань.

Класичним прикладом є автомобіль та двигун. Car (Автомобіль) не успадковує клас Engine (Двигун), бо автомобіль не "є" двигуном. Натомість, клас Car має поле типу Engine. Коли викликається метод car.start(), він усередині себе викликає метод engine.start().

```
class Engine {
    public void start() {
        System.out.println("Двигун запущено.");
    }
}

// Клас, що використовує композицію
class Car {
    // Car "має" Engine. Це і є композиція.
    private Engine engine;

    public Car() {
        this.engine = new Engine(); // Створюємо екземпляр
    }

    public void start() {
        System.out.println("Автомобіль готується до запуску...");
        // Делегуємо роботу об'єкту engine
```

```
        engine.start();
    }
}
```

Композиція вирішує багато проблем, властивих успадкуванню.

Клас Car залежить лише від публічного інтерфейсу класу Engine (тобто від методу start()). Внутрішня реалізація Engine може повністю змінитися, і це ніяк не вплине на Car, доки метод start() працює, як і раніше.

Клас може містити посилання на об'єкти багатьох різних класів, ефективно "компонуючи" свою функціональність з різних джерел.

Клас-контейнер (Car) сам вирішує, яку функціональність внутрішнього об'єкта (Engine) і в якому вигляді надавати назовні. Він не зобов'язаний "виставляти" всі публічні методи двигуна.

У більш складних сценаріях (з використанням Dependency Injection) можна навіть замінити внутрішній об'єкт engine на інший під час виконання програми, що робить систему надзвичайно гнучкою.

Коли і що обирати? Загальне правило звучить так: завжди починайте з композиції. Лише якщо ви можете з упевненістю сказати, що між класами існує справжнє відношення "IS-A", і підклас гарантовано не порушуватиме поведінковий контракт суперкласу (LSP), розглядайте успадкування.

Використовуйте успадкування, коли вам потрібно скористатися поліморфізмом. Тобто, коли вам важливо, щоб об'єкти підкласу можна було обробляти так само, як і об'єкти суперкласу. Успадкування визначає, чим є ваш об'єкт. Використовуйте композицію у всіх інших випадках для повторного використання коду. Композиція визначає, що ваш об'єкт має або що він може робити.

Практична робота №3

Тема: Реалізація ієрархії класів

Мета: Навчитися будувати ієрархії класів за допомогою успадкування та продемонструвати роботу поліморфізму через перевизначення методів

Завдання

1. Створити базовий клас Employee з полями name, salary та методом calculateBonus().

2. Створити похідні класи Manager та Developer, що успадковуються від Employee.

3. Перевизначити метод calculateBonus() у похідних класах з власною логікою.

4. Створити масив типу Employee та заповнити його об'єктами Manager та Developer.

5. Пройтись по масиву та викликати для кожного об'єкта метод calculateBonus(), демонструючи поліморфізм.

Лабораторна робота №3

Тема: Застосування поліморфізму та принципу LSP

Мета: Дослідити проблему порушення принципу заміщення Лісков (LSP) на практиці та провести рефакторинг коду для усунення цього порушення

Завдання

1. Створити базовий клас Rectangle. Додайте private поля width та height. Публічні методи setWidth, setHeight, getArea який повертає width*height.

2. Створити похідний клас Square. Перевизначте методи setWidth, setHeight так, щоб вони завжди встановлювали однакові значення для ширини та висоти одночасно.

3. Написати код, що продемонструє проблему. Створіть екземпляр Rectangle і передайте його в тестовий метод, описаний нижче. Створіть екземпляр Square, приведіть його до типу Rectangle і також передайте в тестовий метод. Створіть статичний метод calculateAndCheck(Rectangle r), у якому встановлюється висота та ширина, а далі виводиться повідомлення про площу прямокутника.

4. Запустіть код і подивіться на результат.У звіті навести код та коротко пояснити, чому для Square результат виявився неочікуваним і як це порушує основне правило LSP.

5. Рефакторинг: Змініть архітектуру класів, щоб уникнути порушення. Створіть спільний інтерфейс або абстрактний клас Shape з методом getArea(). Rectangle та Square мають бути незалежними класами, що реалізують цей інтерфейс. Продемонструйте, що новий дизайн позбавлений цієї проблеми.

Контрольні запитання

1. Що таке успадкування в ООП та яке відношення воно моделює між класами? За допомогою якого ключового слова в Java реалізується успадкування?
2. Поясніть, що таке перевизначення методу. Яка роль анотації @Override і чому її використання є хорошою практикою?
3. Для чого використовується ключове слово super? Опишіть два основні сценарії його застосування у підкласі.
4. Розкрийте суть поліморфізму підтипу. Що таке "пізнє зв'язування" (dynamic method dispatch) і як воно дозволяє об'єктам різних класів по-різному реагувати на один і той самий виклик методу?

5. Сформулюйте своїми словами Принцип заміщення Лісков (LSP). Чому класична ієрархія, де клас Квадрат успадковує клас Прямокутник, порушує цей принцип?
6. Порівняйте підходи успадкування ("IS-A") та композиції ("HAS-A"). Назвіть одну головну перевагу та один суттєвий недолік успадкування.
7. Чому в сучасному об'єктно-орієнтованому дизайні існує рекомендація "Надавайте перевагу композиції над успадкуванням"?
8. Чи успадковуються конструктори? Поясніть роль виклику super() у конструкторі підкласу та що відбувається, якщо його не написати явно.

Лекція 4. Абстрактні класи та Інтерфейси. Принципи OCP, ISP, DIP

Абстрактні класи та методи, створення часткових реалізацій

В об'єктно-орієнтованому програмуванні абстракція є одним з ключових принципів, що дозволяє керувати складністю. Її суть полягає у приховуванні деталей реалізації та наданні користувачеві лише суттєвої інформації про об'єкт. Ми зосереджуємося на тому, що об'єкт робить, а не на тому, як він це робить. У Java для досягнення абстракції використовуються два основні механізми: абстрактні класи та інтерфейси. У цьому розділі ми детально розглянемо перший з них.

Уявіть, що ви проєктуєте систему для ветеринарної клініки і вам потрібно описати поняття "Домашня тварина" (Pet). Ви знаєте, що кожна тварина має ім'я та вік, але дія "подати голос" (say) для кожної тварини буде унікальною: собака гавкає, кіт нявкає. Ви не можете реалізувати метод say() для загального поняття "Тварина", оскільки воно занадто абстрактне. Саме для таких ситуацій і були створені абстрактні класи та методи.

Абстрактний клас — це клас, оголошений за допомогою ключового слова abstract. Його головна особливість полягає в тому, що неможливо створити екземпляр (об'єкт) абстрактного класу. Він не є повноцінною сутністю, а скоріше слугує шаблоном або базовим класом для інших, більш конкретних класів.

Абстрактний метод — це метод, що також позначається ключовим словом abstract і має лише оголошення (сигнатуру), але не має реалізації, тобто тіла з фігурними дужками. Він просто декларує, що певна дія повинна існувати, але залишає її конкретну реалізацію на розсуд класів-нащадків.

Ключова сила абстрактних класів полягає в їхній здатності поєднувати звичайні (реалізовані) методи та абстрактні. Це дозволяє створювати часткові реалізації:

Спільна поведінка реалізується у звичайних методах абстрактного класу. Цей код успадковується всіма нащадками, що сприяє повторному використанню коду.

Специфічна поведінка оголошується через абстрактні методи. Абстрактний клас таким чином встановлює "контракт": будь-який неабстрактний клас, що його успадковує, зобов'язаний надати власну реалізацію для всіх абстрактних методів.

Абстрактний клас, як і звичайний, може містити поля та конструктори. Хоча ви не можете створити об'єкт абстрактного класу напряму, його конструктор викликається з конструкторів підкласів (за допомогою super()) для ініціалізації спільних полів.

Розглянемо це на нашому прикладі з домашніми тваринами:

```
// Абстрактний клас, що описує загальну концепцію "Домашня тварина"
public abstract class Pet {
    protected String name;
    protected int age;
    // Конструктор для ініціалізації спільних полів
    public Pet(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Абстрактний метод - "контракт" для нащадків
    public abstract void say();
}
```

Тепер створимо конкретні класи, які розширюють Pet. Вони зобов'язані реалізувати метод say().

```
// Конкретний клас Cat
public class Cat extends Pet {
    public Cat(String name, int age) {
        // Виклик конструктора батьківського класу Pet
        super(name, age);
    }
    // Обов'язкова реалізація абстрактного методу
    @Override
    public void say() {
        System.out.println("Meow!");
    }
}
// Конкретний клас Dog
public class Dog extends Pet {
    public Dog(String name, int age) {
        super(name, age);
    }
    @Override
    public void say() {
        System.out.println("Woof!");
```

```
    }
}
```

Тепер ми можемо створювати об'єкти конкретних класів та використовувати їх поліморфно:

```
// Створити об'єкт Dog не можна, оскільки це абстрактний клас
// Pet myPet = new Pet("Unknown", 3); // Помилка компіляції!
// Але можна створювати об'єкти конкретних нащадків
Dog dog = new Dog("Boss", 5);
Cat cat = new Cat("Tiger", 2);
dog.say(); // Виведе: "Woof!"
cat.say(); // Виведе: "Meow!"
```

Таким чином, абстрактні класи є ідеальним інструментом для побудови ієрархій успадкування, коли:

Ви хочете створити загальний базовий клас, що містить спільні поля та реалізовані методи для всіх нащадків.

Водночас ви хочете змусити всі конкретні підкласи надати власну реалізацію для деяких методів, визначивши для них спільний "інтерфейс".

Інтерфейси як повна абстракція. Множинна реалізація інтерфейсів

Якщо абстрактні класи є механізмом для створення часткових реалізацій, що поєднують спільний код та абстрактні "контракти", то інтерфейси є інструментом для досягнення повної (або чистої) абстракції. Інтерфейс — це, по суті, чистий контракт. Він описує, що клас повинен вміти робити, абсолютно не турбуючись про те, як він це буде робити. Він визначає набір публічних методів, які будь-який клас, що бажає відповідати цьому контракту, зобов'язаний реалізувати.

У Java інтерфейс є посилальним типом, схожим на клас, але з ключовими відмінностями. Він оголошується за допомогою ключового слова interface. Інтерфейс не може містити конструкторів або полів екземпляра. Його вміст, в класичному розумінні, обмежується константами та абстрактними методами. Будь-яке поле, оголошене в інтерфейсі, автоматично є public static final, тобто константою. Будь-який метод без реалізації автоматично є public abstract. Саме вони і формують "контракт" поведінки.

Клас може "підписати" контракт, запропонований інтерфейсом, використовуючи ключове слово implements. Коли клас реалізує інтерфейс, він бере на себе зобов'язання надати конкретну реалізацію для всіх абстрактних методів, оголошених в цьому інтерфейсі.

Розглянемо приклад з інструментами для малювання, як у матеріалах лекції. Ми можемо визначити загальну функціональність "вміти малювати" за допомогою інтерфейсу DrawingTool:

```
// Інтерфейс визначає контракт: будь-який інструмент для малювання повинен мати метод draw()
interface DrawingTool {
    void draw(String curve); // Метод є public abstract за замовчуванням
}
```

Тепер будь-який клас може заявити, що він є інструментом для малювання, реалізувавши цей інтерфейс. Кожен з них надасть власну унікальну реалізацію малювання:

```
class Pencil implements DrawingTool {
    @Override
    public void draw(String curve) {
        System.out.println("Малюємо '" + curve + "' олівцем...");
    }
}
class Brush implements DrawingTool {
    @Override
    public void draw(String curve) {
        System.out.println("Малюємо '" + curve + "' пензлем...");
    }
}
```

Таким чином, інтерфейс DrawingTool виступає єдиним контрактом, який може мати безліч різних реалізацій (Pencil, Brush тощо). Це і є втіленням принципу "один інтерфейс — багато реалізацій".

Найважливіша перевага такого підходу полягає в тому, що інтерфейс, так само як і клас, може бути використаний як тип даних. Це дозволяє писати гнучкий код, що працює з абстракцією (DrawingTool), а не з конкретними класами (Pencil чи Brush). Ми можемо створити метод, який прийме будь-який об'єкт, що реалізує DrawingTool, і викличе у нього метод draw, не знаючи наперед, чи це буде олівець, пензель або щось інше. Це сприяє слабкій зв'язаності (loose coupling) компонентів системи.

Ось ми й підійшли до ключової переваги інтерфейсів над абстрактними класами. Як ми пам'ятаємо, у Java діє правило одиничного успадкування: клас може розширити (extends) лише один інший клас. Однак, один клас може реалізувати (implements) будь-яку кількість інтерфейсів. Це є способом Java досягти "множинного успадкування поведінки". Клас може "взяти на себе" контракти з абсолютно різних, не пов'язаних між собою джерел. Наприклад, клас Smartphone може одночасно бути пристроєм, що може дзвонити, і пристроєм, що

може переглядати веб-сторінки. Ми можемо описати ці дві функціональності за допомогою окремих інтерфейсів:

```
interface Callable {
    void makeCall(String number);
}
interface Browsable {
    void browseInternet(String url);
}
// Клас Smartphone реалізує ОБИДВА інтерфейси
class Smartphone implements Callable, Browsable {
    @Override
    public void makeCall(String number) {
        System.out.println("Дзвонимо за номером " + number);
    }
    @Override
    public void browseInternet(String url) {
        System.out.println("Відкриваємо веб-сторінку: " + url);
    }
}
```

Таким чином, об'єкт класу Smartphone можна використовувати скрізь, де очікується Callable, і скрізь, де очікується Browsable, що робить систему надзвичайно гнучкою. Крім того, самі інтерфейси також можуть успадковувати (розширювати) декілька інших інтерфейсів, об'єднуючи їхні контракти в один.

Default-методи інтерфейсів: введення, застосування, обмеження та ризики

Довгий час інтерфейси в Java були втіленням чистої абстракції: вони могли містити лише оголошення методів без їх реалізації. Це створювало проблему, відому як "крихкість інтерфейсів". Уявіть, що ви є автором популярної бібліотеки, яка надає інтерфейс Listable з методом toList(). Тисячі розробників по всьому світу використовують ваш інтерфейс і реалізують його у своїх класах. Що станеться, якщо ви захочете додати до цього інтерфейсу новий корисний метод, наприклад, toJson()? Після оновлення бібліотеки код усіх цих розробників перестане компілюватися, оскільки їхні класи більше не будуть реалізовувати всі абстрактні методи нового інтерфейсу. Це змушувало або відмовлятися від розвитку інтерфейсів, або створювати складні ієрархії, що було незручно.

Саме для вирішення цієї проблеми у Java 8 було введено методи за замовчуванням (default methods).

Default-метод — це метод, оголошений в інтерфейсі з ключовим словом default, який має тіло, тобто містить реалізацію. Це дозволяє додавати нові методи до існуючих інтерфейсів, не "ламаючи" класи, які їх уже реалізують.

Класи, що імплементують такий інтерфейс, автоматично успадковують цю реалізацію за замовчуванням.

Синтаксично default-метод виглядає як звичайний метод, але з ключовим словом default на початку.

```
public interface Feature {
    // Звичайний абстрактний метод (контракт)
    void performAction();
    // Default-метод з реалізацією
    default void logAction() {
        System.out.println("Action is being performed.");
    } }
```

Клас, що реалізує цей інтерфейс, зобов'язаний надати реалізацію для performAction(), але може, за бажанням, або використовувати готову реалізацію logAction(), або перевизначити її.

```
public class MyFeature implements Feature {
    @Override
    public void performAction() {
        // ... якась унікальна логіка ...
        // Ми можемо викликати default-метод
        logAction();
    }
    // Ми також можемо перевизначити default-метод, якщо потрібно
    @Override
    public void logAction() {
       System.out.println("Logging action in MyFeature class...");  } }
```

Головне застосування default-методів — це еволюція API без порушення зворотної сумісності. Повертаючись до нашого початкового прикладу, розробник бібліотеки тепер може безпечно додати новий метод до існуючого інтерфейсу:

Весь існуючий код, який використовує Listable, продовжить працювати без змін. А розробники, які захочуть скористатися новою функціональністю, зможуть це зробити, або ж надати більш ефективну реалізацію методу toJson() у своїх класах.

Хоча default-методи є надзвичайно корисними, вони принесли в Java певну форму множинного успадкування реалізації, а разом з нею і класичну проблему, відому як **"проблема діаманта"** (The Diamond Problem).

Проблема виникає, коли клас реалізує два інтерфейси, які мають default-метод з однаковою сигнатурою. Компілятор не може визначити, яку з двох успадкованих реалізацій слід використовувати.

```
interface InterfaceA {
    default void doSomething() {
        System.out.println("Doing something in A");
```

```
    }
}
interface InterfaceB {
    default void doSomething() {
        System.out.println("Doing something in B");
    }
}
// Помилка компіляції! Клас успадковує конфліктуючі реалізації.
class MyClass implements InterfaceA, InterfaceB {
    // … }
```

Реалізація в класі завжди має пріоритет. Якщо клас MyClass або один з його суперкласів надасть власну реалізацію методу doSomething(), то реалізації з інтерфейсів будуть проігноровані.

Реалізація у більш специфічному під-інтерфейсі має пріоритет. Якщо InterfaceB успадковує InterfaceA, то реалізація з InterfaceB буде використана.

Клас зобов'язаний вирішити конфлікт. Якщо жодне з перших двох правил не застосовується (як у нашому прикладі), клас MyClass зобов'язаний самостійно перевизначити конфліктний метод. Усередині цього методу розробник може або написати абсолютно нову логіку, або явно вибрати, яку з батьківських реалізацій викликати, використовуючи спеціальний синтаксис InterfaceName.super.methodName()

Принцип Відкритості/Закритості (OCP)

Принцип Відкритості/Закритості (The Open/Closed Principle, OCP) є літерою "O" в акронімі SOLID і є одним з найважливіших принципів об'єктно-орієнтованого дизайну. Вперше сформульований Бертраном Мейєром, він звучить так: "Програмні сутності (класи, модулі, функції) повинні бути відкритими для розширення, але закритими для модифікації."

На перший погляд, це твердження може здатися суперечливим. Як можна одночасно розширювати функціональність системи, не змінюючи її існуючий код? Ключ до розуміння цього принципу лежить у правильному використанні абстракцій.

"Відкриті для розширення" означає, що поведінку системи можна змінювати або доповнювати новою функціональністю відповідно до нових вимог.

"Закриті для модифікації" означає, що після того, як модуль розроблений, протестований і працює, його вихідний код не повинен змінюватися для додавання нових можливостей. Будь-яка модифікація існуючого, стабільного коду несе в собі ризик внесення помилок (регресій) у вже працюючу систему.

Щоб зрозуміти важливість цього принципу, розглянемо приклад, що його порушує. Уявімо, що ми розробляємо систему обробки платежів. Наш початковий клас PaymentProcessor вміє працювати з двома типами платежів: кредитними картками та PayPal.

```
// Початковий клас, що порушує OCP
public class CreditCardPayment {
    public void process() { System.out.println("Обробка платежу з кредитної картки..."); }
}
public class PayPalPayment {
    public void process() { System.out.println("Обробка платежу через PayPal..."); }
}
public class PaymentProcessor {
    public void processPayment(Object paymentType) {
        if (paymentType instanceof CreditCardPayment) {
            ((CreditCardPayment) paymentType).process();
        } else if (paymentType instanceof PayPalPayment) {
            ((PayPalPayment) paymentType).process();
        }
    }
}
```

Цей код працює, але він дуже "крихкий". Уявіть, що бізнес вирішив додати новий спосіб оплати, наприклад, через Bitcoin. Щоб реалізувати цю вимогу, ми змушені будемо модифікувати існуючий клас PaymentProcessor, додавши до нього ще один блок else if:

```
// ... всередині класу PaymentProcessor
else if (paymentType instanceof BitcoinPayment) {
    ((BitcoinPayment) paymentType).process();
}
```

Кожна така зміна вимагає повторного тестування всього класу PaymentProcessor, щоб переконатися, що ми не зламали обробку старих типів платежів. Чим більше типів платежів, тим складнішим і більшим стає цей клас, і тим вищий ризик помилки при кожній модифікації. Такий клас є відкритим для модифікації, а отже, він порушує OCP.

Ключем до дотримання Принципу Відкритості/Закритості є залежність від абстракцій (інтерфейсів або абстрактних класів), а не від конкретних реалізацій. Давайте проведемо рефакторинг нашого прикладу, слідуючи крокам, описаним у лабораторній роботі №4.

Крок 1: Створити абстракцію. Визначимо спільний контракт для всіх типів платежів за допомогою інтерфейсу Payable.

```
public interface Payable {
```

```
    void pay();
}
```

Крок 2: Реалізувати абстракцію. Тепер змусимо наші конкретні класи реалізувати цей інтерфейс.

```
public class CreditCardPayment implements Payable {
    @Override
    public void pay() {
      System.out.println("Обробка платежу з кредитної картки...");
    }
}
public class PayPalPayment implements Payable {
    @Override
    public void pay() {
        System.out.println("Обробка платежу через PayPal...");
    }
}
```

Крок 3: Зробити основний клас залежним від абстракції. Перепишемо наш PaymentProcessor, щоб він працював не з конкретними класами, а з інтерфейсом Payable.

```
public class PaymentProcessor {
    public void processPayment(Payable payable) {
        // Тепер процесору неважливо, який саме тип платежу.
        // Він знає лише те, що об'єкт "вміє платити".
        payable.pay();
    }
}
```

Тепер наш клас PaymentProcessor є закритим для модифікації. Його логіка проста, стабільна, і нам більше ніколи не доведеться її змінювати. Але як же щодо розширення? Уявімо, що нам знову потрібно додати оплату через Bitcoin. Тепер для цього достатньо створити новий клас, що реалізує інтерфейс Payable:

```
public class BitcoinPayment implements Payable {
    @Override
    public void pay() {
        System.out.println("Обробка платежу через Bitcoin...");
    }
}
```

Ми розширили функціональність системи, не внісши жодної зміни в існуючий, протестований код класу PaymentProcessor. Наша система виявилася відкритою для розширення.

Принципи Розділення Інтерфейсу (ISP) та Інверсії Залежностей (DIP)

Принцип Розділення Інтерфейсу (Interface Segregation Principle, ISP) є літерою "I" в акронімі SOLID. Його основна ідея, сформульована Робертом Мартіном, полягає в наступному: "Клієнти не повинні бути змушені залежати від методів, які вони не використовують."

Простими словами, цей принцип радить уникати створення великих, "товстих" інтерфейсів, які описують безліч різних операцій. Натомість, краще створювати багато маленьких, вузькоспеціалізованих інтерфейсів, кожен з яких відповідає за конкретну групу поведінки. Це дозволяє класам-клієнтам реалізовувати лише ті методи, які їм справді потрібні.

Уявімо, що ми проєктуємо систему для управління працівниками і створюємо єдиний, універсальний інтерфейс Worker:

```
// "Товстий" інтерфейс, що порушує ISP
interface Worker {
    void work();
    void eatLunch();
}
```

Тепер створимо два класи, що реалізують цей інтерфейс: HumanWorker (людина-працівник) та RobotWorker (робот-працівник).

```
class HumanWorker implements Worker {
    public void work() { System.out.println("Людина працює..."); }
  public void eatLunch() { System.out.println("Людина обідає...");}
}
class RobotWorker implements Worker {
    public void work() { System.out.println("Робот працює..."); }
    // Робот не обідає! Ми змушені реалізувати цей метод.
    public void eatLunch() {
        // Залишаємо порожнім або кидаємо виняток
    }
}
```

Клас RobotWorker потрапляє в незручну ситуацію. Він змушений реалізовувати метод eatLunch(), який для нього не має жодного сенсу. Це змушує нас або залишати метод порожнім, або кидати виняток UnsupportedOperationException, що є поганою практикою. Клас RobotWorker залежить від інтерфейсу (eatLunch), який він не використовує, що є прямим порушенням ISP.

Рішення полягає в тому, щоб розділити "товстий" інтерфейс на кілька менших та більш логічних:

```
// Маленькі, сфокусовані інтерфейси
interface Workable {
    void work();
}
```

```
interface Eatable {
    void eatLunch();
}
```

Тепер кожен клас може реалізовувати лише ті інтерфейси, які йому потрібні:

```
class HumanWorker implements Workable, Eatable {
    public void work() { /* ... */ }
    public void eatLunch() { /* ... */ }
}
// RobotWorker реалізує лише те, що йому потрібно
class RobotWorker implements Workable {
    public void work() { /* ... */ }
}
```

Такий підхід робить систему більш гнучкою, а контракти класів — більш чіткими та зрозумілими.

Принцип Інверсії Залежностей (Dependency Inversion Principle, DIP) є літерою "D" в SOLID і є одним з ключових для побудови слабкозв'язаних систем. Він складається з двох тверджень:

1. Модулі високого рівня не повинні залежати від модулів низького рівня. Обидва повинні залежати від абстракцій.
2. Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.

"Інверсія" в назві принципу означає зміну напрямку залежностей, який здається "природним" на перший погляд. Зазвичай, високоривневі модулі (що містять складну бізнес-логіку) викликають і залежать від низькоривневих модулів (що виконують базові, детальні операції). DIP пропонує "інвертувати" цей зв'язок: змусити обидва рівні залежати від спільної абстракції (інтерфейсу).

Повернемося до нашого прикладу з PaymentProcessor, який ми розглядали при вивченні OCP. Початковий дизайн порушував не лише OCP, а й DIP.

```
// PaymentProcessor - модуль високого рівня
// CreditCardPayment, PayPalPayment - модулі низького рівня
public class PaymentProcessor {
    // Пряма залежність від конкретних класів
    private CreditCardPayment creditCardPayment;
    private PayPalPayment payPalPayment;
    public PaymentProcessor() {
        this.creditCardPayment = new CreditCardPayment();
        this.payPalPayment = new PayPalPayment();
    }
    // ... логіка з if-else ...
}
```

Тут PaymentProcessor (високий рівень) безпосередньо знає про існування та залежить від CreditCardPayment (низький рівень). Це створює жорсткий зв'язок.

Рішення, яке ми застосували для OCP, є також і реалізацією DIP, як це вказано в лабораторній роботі №4. Ми ввели абстракцію — інтерфейс Payable.

Деталі залежать від абстракції: Класи CreditCardPayment та PayPalPayment почали реалізовувати (implements) інтерфейс Payable.

Високий рівень залежить від абстракції: Клас PaymentProcessor почав працювати з інтерфейсом Payable, не знаючи нічого про конкретні реалізації.

Нова структура залежностей виглядає так:

```
PaymentProcessor → Payable ← CreditCardPayment
```

Напрямок залежності було інвертовано. Тепер не високий рівень залежить від низького, а обидва рівні залежать від спільної абстракції.

Після інверсії залежностей виникає питання: якщо PaymentProcessor більше не створює об'єкти CreditCardPayment сам, то хто їх створює і передає процесору? Відповіддю на це питання є патерн **Dependency Injection** (Впровадження залежностей, DI).

DI — це техніка, за якою залежності об'єкта (тобто інші об'єкти, які йому потрібні для роботи) надаються йому ззовні, а не створюються ним самим. Найпоширеніший спосіб — через конструктор.

```
public class PaymentProcessor {
    private final Payable paymentMethod;
    // Залежність "впроваджується" через конструктор
    public PaymentProcessor(Payable paymentMethod) {
        this.paymentMethod = paymentMethod;
    }
    public void processPayment() {
        // Використання залежності
        paymentMethod.pay();
    }
}
// ... десь в іншому місці програми (наприклад, у main)
Payable creditCard = new CreditCardPayment();
// Створюємо процесор і "впроваджуємо" в нього залежність
PaymentProcessor processor = new PaymentProcessor(creditCard);
processor.processPayment();
```

Такий підхід повністю розриває жорсткий зв'язок. Клас PaymentProcessor стає повністю незалежним, легко тестованим та гнучким, що є кінцевою метою Принципу Інверсії Залежностей.

Введення в концепцію Dependency Injection як реалізацію DIP

Ми встановили, що Принцип Інверсії Залежностей (DIP) вимагає, щоб високоривневі модулі не залежали від низькоривневих, а обидва залежали від абстракцій. Це дозволяє створювати слабкозв'язані системи. Однак цей принцип залишає відкритим практичне питання: якщо клас PaymentProcessor більше не створює об'єкти CreditCardPayment сам, то хто, де і коли повинен їх створювати і надавати PaymentProcessor-у?

Відповіддю на це питання є потужний патерн проєктування під назвою Впровадження Залежностей (Dependency Injection, DI).

Dependency Injection — це патерн, за якого об'єкт не створює залежності (тобто інші об'єкти, які йому потрібні для роботи) самостійно, а отримує їх ззовні. Відповідальність за створення та надання залежностей перекладається на зовнішній механізм.

Уявіть, що ви хочете приготувати каву.

Без DI: Ви самі вирощуєте кавові зерна, обсмажуєте їх, мелете, добуваєте воду, кип'ятите її, а потім варите напій. Ви повністю контролюєте процес, але тісно пов'язані з кожним його етапом.

З DI: Ви приходите в кав'ярню. Бариста (зовнішній механізм) надає вам готову залежність — чашку кави. Ваше завдання — лише її випити.

Так само і в програмуванні: клас, що потребує залежності, не створює її за допомогою new, а просто оголошує, що вона йому потрібна, і очікує, що хтось її "впровадить".

Існує три основні способи, якими залежність може бути "впроваджена" в об'єкт.

Впровадження через конструктор (**Constructor Injection**) – це найпоширеніший і рекомендований спосіб. Залежності передаються як параметри конструктора класу. Об'єкт отримує все, що йому потрібно для роботи, в момент свого створення.

```
public class PaymentProcessor {
    // Залежність оголошена через інтерфейс
    private final Payable paymentMethod;
    // Залежність "впроваджується" в конструкторі
    public PaymentProcessor(Payable paymentMethod) {
        this.paymentMethod = paymentMethod;
    }
    public void processPayment() {
        paymentMethod.pay();
    }
}
// ... десь у зовнішньому коді, який збирає систему:
Payable creditCard = new CreditCardPayment();
```

```
PaymentProcessor processor = new PaymentProcessor(creditCard); // Впровадження залежності
processor.processPayment();
```

Об'єкт не може бути створений без своїх обов'язкових залежностей. Це гарантує, що він завжди буде в коректному, готовому до роботи стані. Залежності можна зробити final, що підвищує надійність.

Впровадження через сеттер (**Setter Injection**). При цьому підході клас має публічний метод-сеттер, через який йому можна передати залежність вже після створення об'єкта.

```
public class PaymentProcessor {
    private Payable paymentMethod;
    // Порожній конструктор
    public PaymentProcessor() {}
    // Сеттер для впровадження залежності
    public void setPaymentMethod(Payable paymentMethod) {
        this.paymentMethod = paymentMethod;
    }
    // ...
}
```

Дозволяє змінювати залежності "на льоту" під час роботи програми. Також підходить для необов'язкових залежностей. Але об'єкт може існувати в некоректному стані (без залежності), що може призвести до NullPointerException, якщо залежність не була впроваджена перед використанням.

Впровадження через поле (**Field Injection**) – цей спосіб передбачає впровадження залежності безпосередньо в поле класу, оминаючи конструктори та сеттери. Зазвичай це реалізується за допомогою анотацій та рефлексії, і є дуже популярним у фреймворках, таких як Spring.

```
public class PaymentProcessor {
    @Autowired // Анотація для впровадження залежності
    private Payable paymentMethod;
    // ...
}
```

При ручному керуванні залежностями цей підхід не рекомендується, оскільки він приховує залежності класу та ускладнює тестування.

У великих додатках ручне створення та "впровадження" десятків залежностей стає складним завданням. Для його автоматизації існують спеціальні інструменти, які називаються **DI-контейнерами (або IoC-контейнерами, Inversion of Control).**

DI-контейнер — це фреймворк, який бере на себе відповідальність за життєвий цикл об'єктів: він створює екземпляри класів, автоматично знаходить та впроваджує в них необхідні залежності відповідно до конфігурації, і надає

готові до роботи об'єкти на вимогу. Spring Framework є найпопулярнішим прикладом такого контейнера в світі Java.

Практична робота №4

Тема: Робота з абстрактними класами та інтерфейсами

Мета: Навчитися використовувати інтерфейси для створення слабкозв'язаних компонентів та продемонструвати залежність від абстракцій, а не від конкретних реалізацій

Завдання

1. Створити інтерфейс Shape з методами getArea() та getPerimeter().
2. Створити класи Circle та Rectangle, що реалізують інтерфейс Shape.
3. Створити клас-сервіс ShapeService, який приймає в конструктор об'єкт типу Shape.
4. Реалізувати в ShapeService метод, що виводить площу та периметр фігури.
5. Продемонструвати, що ShapeService залежить від абстракції (Shape), а не від конкретних класів.

Лабораторна робота №4

Тема: Рефакторинг з використанням принципів дизайну

Мета: Провести рефакторинг коду для реалізації принципів DIP та OCP, замінивши жорсткі залежності на абстракції (інтерфейси)

Завдання

1. Створити клас PaymentProcessor, який жорстко залежить від класів CreditCardPayment та PayPalPayment.
2. Створити інтерфейс Payable з методом pay().
3. Змусити класи CreditCardPayment та PayPalPayment реалізувати цей інтерфейс.
4. Провести рефакторинг класу PaymentProcessor, щоб він залежав від інтерфейсу Payable (реалізація DIP).
5. Додати новий спосіб оплати (напр., BitcoinPayment), не змінюючи код PaymentProcessor (демонстрація OCP).

Контрольні запитання

1. У чому полягає ключова різниця між абстрактним класом та інтерфейсом? Наведіть сценарій, коли доцільніше використати абстрактний клас, і сценарій, коли краще обрати інтерфейс.

2. Що таке абстрактний метод і яка його роль? Що станеться, якщо звичайний (неабстрактний) клас-нащадок не реалізує абстрактний метод свого батьківського класу?
3. Яку проблему вирішує можливість реалізації кількох інтерфейсів одним класом (implements)? Чому аналогічна можливість відсутня для успадкування від класів (extends) у Java?
4. Яку проблему вирішують default-методи в інтерфейсах? Опишіть "проблему діаманта" і як Java змушує розробника вирішувати цей конфлікт.
5. Сформулюйте Принцип Відкритості/Закритості (OCP). Поясніть на прикладі, як використання абстракцій (інтерфейсів) дозволяє розширювати функціональність системи, не змінюючи існуючий код.
6. Що таке "товстий" інтерфейс і чому він є порушенням Принципу Розділення Інтерфейсу (ISP)? Як слід рефакторити такий інтерфейс для дотримання принципу?
7. Розкрийте суть Принципу Інверсії Залежностей (DIP). Що означає фраза "модулі високого рівня не повинні залежати від модулів низького рівня"? Від чого вони повинні залежати натомість?
8. Що таке Впровадження Залежностей (Dependency Injection) і як цей патерн є практичною реалізацією DIP? Опишіть найпоширеніший спосіб DI — через конструктор.

ТЕМА 3. КЛЮЧОВІ БІБЛІОТЕКИ ТА ПІДГОТОВКА ДО ENTERPRISE-РОЗРОБКИ

Лекція 5. Java Collections Framework та обробка винятків

Огляд ієрархії колекцій: Collection, List, Set, Queue, Map

Після освоєння основ об'єктно-орієнтованого програмування ми переходимо до вивчення одного з найважливіших компонентів стандартної бібліотеки Java — Java Collections Framework (JCF). Практично будь-яка програма потребує зберігання та обробки груп об'єктів, чи то список користувачів, набір унікальних товарів, чи словник налаштувань. JCF надає потужну, уніфіковану та високоефективну архітектуру для роботи з такими групами даних, які називаються колекціями. Цей фреймворк є частиною пакета java.util і складається з набору інтерфейсів, що визначають абстрактні типи колекцій, та їх конкретних реалізацій.

В основі архітектури Java Collections Framework лежать два фундаментальні інтерфейси, які розділяють усі колекції на дві великі категорії залежно від способу зберігання даних: Collection та Map.

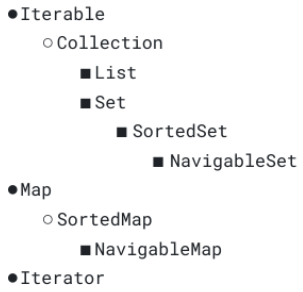

Рисунок 5.1 - Ієрархія інтерфейсів в Java Collections Framework

Інтерфейс Collection представляє найпростішу форму колекції — це просто група окремих об'єктів, що розглядаються як єдине ціле. Уявіть її як мішок з предметами. Цей інтерфейс, у свою чергу, є базовим для трьох основних, більш спеціалізованих типів колекцій: List, Set та Queue.

List (Список) це впорядкована колекція, яка також відома як послідовність. Ключовими характеристиками List є те, що елементи в ньому зберігаються у визначеному порядку вставки, і він допускає зберігання дублікатів. Оскільки список впорядкований, до його елементів можна звертатися за їхнім числовим індексом (позицією), так само, як у масивах. List ідеально підходить для ситуацій, коли важливий порядок елементів.

Set (Множина) моделює математичне поняття множини. Його головна характеристика — Set не може містити елементів, що дублюються. Спроба додати в множину елемент, який там уже є, буде просто проігнорована. Більшість

реалізацій Set не гарантують збереження порядку елементів. Set є ідеальним вибором, коли вам потрібно зберігати колекцію унікальних об'єктів.

Queue (Черга) інтерфейс призначений для зберігання елементів у порядку, що підходить для їх подальшої обробки. Зазвичай черги працюють за принципом FIFO (First-In, First-Out), тобто "перший увійшов — перший вийшов", що нагадує звичайну чергу людей. Елементи додаються в кінець черги, а витягуються з її початку.

Важливо зазначити, що інтерфейс Map не є нащадком інтерфейсу Collection і стоїть окремо в ієрархії. Якщо Collection зберігає окремі об'єкти, то Map призначений для зберігання пар "ключ-значення". Кожен елемент у Map — це асоціація, де унікальний ключ пов'язаний з певним значенням. Це схоже на словник, де кожне слово (ключ) має своє визначення (значення). Ключі в Map повинні бути унікальними, тоді як значення можуть повторюватися. Map є ідеальним інструментом, коли потрібен швидкий доступ до даних за унікальним ідентифікатором.

Варто відрізняти інтерфейс java.util.Collection, що є основою ієрархії, від класу java.util.Collections, який є допоміжним (утилітарним) класом і містить набір статичних методів для виконання різних операцій над колекціями, таких як сортування, пошук, перемішування тощо.

Вибір конкретного типу колекції — List, Set, Queue чи Map — є важливим архітектурним рішенням і повністю залежить від вимог задачі: чи потрібен вам порядок елементів, чи важлива їх унікальність, чи необхідний доступ за ключем. У наступному розділі ми розглянемо конкретні реалізації цих інтерфейсів.

Вибір правильної колекції для задачі: ArrayList vs LinkedList, HashSet vs TreeSet

Після того, як ми визначилися з абстрактним типом колекції (List, Set тощо), який найкраще відповідає нашій задачі, наступним кроком є вибір його конкретної реалізації. Цей вибір не є суто стилістичним; він має прямий та суттєвий вплив на продуктивність вашої програми, зокрема на швидкість виконання операцій та обсяг використовуваної пам'яті. Правильний вибір залежить від аналізу того, які операції з колекцією будуть виконуватися найчастіше: читання, вставка, видалення, пошук чи сортування. Розглянемо найпопулярніші пари реалізацій.

Обидва класи реалізують інтерфейс List, тобто обидва є впорядкованими колекціями, що дозволяють дублікати. Однак їхня внутрішня структура кардинально відрізняється, що робить їх ефективними для абсолютно різних завдань.

ArrayList. Як випливає з назви, ArrayList всередині реалізований на основі динамічного масиву. Це означає, що всі його елементи зберігаються в пам'яті послідовно, один за одним.

Головною перевагою такої структури є надзвичайно швидкий доступ до елементів за індексом (операція get(index)). Оскільки елементи лежать в суцільному блоці пам'яті, комп'ютер може миттєво обчислити адресу потрібного елемента, тому ця операція виконується за константний час O(1).

Але слабкою стороною виступає те, що вставка або видалення елемента з середини списку є дуже повільною операцією (O(n)). Щоб, наприклад, видалити елемент, потрібно фізично "зсунути" всі наступні елементи на одну позицію вліво, щоб заповнити прогалину. Аналогічно, при вставці потрібно "розсунути" частину масиву.

ArrayList є вибором за замовчуванням для більшості завдань. Його слід використовувати, коли найчастішими операціями є читання даних за індексом або ітерація по всій колекції, а кількість вставок та видалень у середину списку є мінімальною.

LinkedList. На відміну від ArrayList, LinkedList базується на структурі двонаправленого зв'язного списку. Кожен елемент (вузол) у такому списку зберігає не лише свої дані, а й посилання на попередній та наступний елементи.

Завдяки такій структурі, вставка та видалення елементів, особливо на початку або в кінці списку, є надзвичайно швидкими операціями (O(1)). Для цього потрібно лише змінити кілька посилань у сусідніх елементів, не рухаючи весь масив даних.

Платою за швидкі модифікації є дуже повільний доступ до елемента за індексом (O(n)). Щоб знайти, наприклад, 500-й елемент, програма змушена послідовно пройти від початку списку через усі 499 попередніх елементів.

LinkedList є ідеальним вибором, коли ваша програма виконує велику кількість операцій вставки та видалення, особливо на початку чи в кінці колекції. Завдяки цьому він також часто використовується для реалізації таких структур даних, як стек та черга.

HashSet vs TreeSet?. Обидва класи реалізують інтерфейс Set, а отже, зберігають лише унікальні елементи. Вибір між ними залежить головним чином від того, чи потрібен вам певний порядок елементів.

HashSet реалізація використовує для зберігання елементів хеш-таблицю (всередині вона базується на HashMap). При додаванні об'єкта HashSet обчислює його хеш-код (за допомогою методу hashCode()) і використовує його для визначення місця зберігання.

Завдяки хешуванню, основні операції — додавання (add), видалення (remove) та перевірка наявності (contains) — виконуються в середньому за

константний час O(1). Це робить HashSet найшвидшою реалізацією Set. Але HashSet не гарантує жодного порядку елементів. Порядок, в якому ви будете отримувати елементи при ітерації, може здаватися випадковим і навіть змінюватися з часом при додаванні нових елементів.

Використовуйте HashSet, коли вам потрібна максимальна продуктивність для зберігання унікальних елементів, і при цьому їхній порядок не має жодного значення.

TreeSet, на відміну від HashSet, базується на складній деревоподібній структурі даних (червоно-чорне дерево).

Головна перевага TreeSet — він автоматично зберігає елементи у відсортованому порядку. Сортування відбувається або за "природним порядком" елементів (якщо їхній клас реалізує інтерфейс Comparable), або за допомогою спеціального об'єкта Comparator, переданого при створенні TreeSet.

Через необхідність підтримувати відсортовану структуру, операції додавання, видалення та пошуку працюють трохи повільніше, ніж у HashSet, і займають час O(log n). Крім того, TreeSet не дозволяє зберігати null, оскільки null неможливо порівняти з іншими елементами.

TreeSet є незамінним, коли вам потрібно не просто зберігати унікальні елементи, а й завжди мати до них доступ у відсортованому вигляді.

Таким чином, вибір правильної реалізації колекції — це завжди пошук компромісу між швидкістю виконання різних операцій та вимогами до структури даних, такими як впорядкованість або унікальність.

Реалізація методів equals() та hashCode(): контракти, типові помилки та вплив на роботу колекцій

Кожен об'єкт у Java успадковує від базового класу Object два надзвичайно важливі методи: equals() та hashCode(). На перший погляд, їх призначення може здатися не очевидним, але правильна реалізація цих методів є критично важливою для коректної роботи з колекціями, особливо з тими, що базуються на хешуванні, як-от HashSet та HashMap. Ці два методи утворюють єдиний "контракт", порушення якого призводить до неочікуваної та важковідтворюваної поведінки програми.

Спочатку необхідно зрозуміти, що в Java існує два поняття рівності для об'єктів – ідентичність та еквівалентність.

Ідентичність (Reference Equality) перевіряється за допомогою оператора ==. Він відповідає на питання: "Чи є ці дві змінні посиланнями на один і той самий об'єкт у пам'яті?".

Еквівалентність (Logical Equality) перевіряється за допомогою методу equals(). Він повинен відповідати на питання: "Чи є ці два об'єкти логічно рівноцінними за своїм вмістом, навіть якщо це різні екземпляри в пам'яті?".

Уявіть, що у вас є дві окремі, але абсолютно ідентичні книги "Кобзар". Оператор == сказав би, що вони не рівні, бо це два різні фізичні об'єкти. Але з логічної точки зору, вони еквівалентні. Метод equals() і призначений для визначення такої логічної еквівалентності.

За замовчуванням, реалізація методу equals() у класі Object просто використовує ==, тобто перевіряє ідентичність. Тому, якщо ви хочете, щоб ваші об'єкти могли порівнюватися за вмістом, ви зобов'язані перевизначити метод equals().

При перевизначенні equals(Object obj) необхідно дотримуватися строгого контракту:

1. Рефлексивність: Для будь-якого ненульового посилання x, виклик x.equals(x) повинен повертати true.
2. Симетричність: Для будь-яких x та y, виклик x.equals(y) повинен повертати true тоді і тільки тоді, коли y.equals(x) повертає true.
3. Транзитивність: Якщо x.equals(y) повертає true і y.equals(z) повертає true, то і x.equals(z) повинен повертати true.
4. Консистентність: Повторні виклики x.equals(y) повинні повертати однакове значення, доки поля об'єктів, що беруть участь у порівнянні, не були змінені.
5. Порівняння з null: Для будь-якого ненульового посилання x, виклик x.equals(null) повинен повертати false.

Метод hashCode() повертає ціле число (хеш-код), яке є числовим представленням об'єкта. Цей метод не призначений для точного порівняння, а використовується для ефективної організації об'єктів у хеш-таблицях (HashSet, HashMap). Він допомагає швидко знайти "кошик" (bucket), в якому потенційно може знаходитись об'єкт.

Головне правило, що пов'язує equals() та hashCode(), звучить так: якщо два об'єкти є рівними за методом equals(), вони зобов'язані мати однаковий hashCode().

Зворотне твердження не є обов'язковим: якщо два об'єкти мають однаковий hashCode(), вони не зобов'язані бути рівними за методом equals(). Така ситуація називається колізією.

Порушення контракту між equals() та hashCode() призводить до катастрофічних наслідків при роботі з хеш-колекціями. HashSet та HashMap для всіх своїх ключових операцій (add, contains, remove) використовують двохетапний алгоритм:

1. Знайти "кошик". Викликається метод hashCode() об'єкта, щоб миттєво визначити індекс "кошика", де цей об'єкт має зберігатися.
2. Перевірити на рівність. Усередині цього кошика відбувається послідовний перебір елементів і порівняння шуканого об'єкта з кожним з них за допомогою методу equals().

Найпоширеніша помилка: перевизначити equals(), але не перевизначити hashCode(). У цьому випадку ви порушуєте головне правило контракту.

```
User user1 = new User(1, "test@example.com");
User user2 = new User(1, "test@example.com");
// user1.equals(user2) поверне true (якщо перевизначено правильно)
Set<User> users = new HashSet<>();
users.add(user1);
// А тепер найцікавіше:
System.out.println(users.contains(user2)); // Поверне FALSE!
```

Чому так сталося? Оскільки hashCode() не був перевизначений, user1 та user2 (хоч і рівні за equals) згенерували різні хеш-коди. Коли HashSet шукав user2, він подивився в інший "кошик" і, не знайшовши там нічого, одразу повернув false, навіть не дійшовши до виклику equals().

Запам'ятайте правило: завжди перевизначайте hashCode(), коли перевизначаєте equals()!

Сучасні IDE можуть автоматично генерувати коректні реалізації цих методів. Однак важливо розуміти, як вони влаштовані.

```
    @Override
    public boolean equals(Object o) {
        // 1. Перевірка на ідентичність
        if (this == o) return true;
        // 2. Перевірка на null та відповідність класів
        if (o == null || getClass() != o.getClass()) return false;
        // 3. Приведення типів
        User user = (User) o;
        // 4. Порівняння значущих полів
        return id == user.id && Objects.equals(email, user.email);
    }
    @Override
    public int hashCode() {
        // Використання допоміжного методу для генерації хеш-коду
        // на основі тих самих полів, що й у equals()
        return Objects.hash(id, email);
    }
}
```

У цьому прикладі для порівняння полів-об'єктів (email) використовується Objects.equals(), що коректно обробляє null. Для генерації хеш-коду

використовується допоміжний метод Objects.hash(), якому передаються ті ж самі поля, що й у методі equals(). Це гарантує дотримання контракту і, як наслідок, коректну роботу об'єктів User у хеш-колекціях.

Stream API, основні операції (filter, map, collect), робота з колекціями у функціональному стилі

До появи Java 8 основним способом обробки колекцій були цикли (for, for-each). Цей підхід, відомий як імперативний стиль, вимагає від програміста детального опису того, як саме потрібно виконати завдання: як ініціалізувати цикл, як отримати кожен елемент, як його перевірити, і як додати до нової колекції.

Наприклад, якщо нам потрібно відфільтрувати зі списку цілих чисел лише парні, помножити кожне з них на два і зібрати результат у новий список, імперативний код виглядатиме так:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Integer> result = new ArrayList<>();
for (Integer number : numbers) {
    if (number % 2 == 0) { // Як фільтрувати
        int transformed = number * 2; // Як трансформувати
        result.add(transformed); // Як збирати результат
    }
}
```

Такий код є багатослівним і змішує логіку з механікою ітерації. З появою Java 8 був представлений Stream API, який пропонує абсолютно новий, декларативний підхід до обробки даних у функціональному стилі.

Stream (потік) — це не структура даних, а послідовність елементів із джерела, яка підтримує сукупні операції. Уявіть його як конвеєрну стрічку. На початку стрічки розміщуються елементи з джерела (наприклад, з колекції). Потім вони проходять через низку верстатів (операцій), кожен з яких виконує певну дію: відфільтровує, трансформує тощо. В кінці стрічки результат збирається у фінальну форму.

Розглянемо ключові властивості Stream API.

1. Stream не є сховищем даних; він лише обробляє елементи з джерела.
2. Операції над потоком виражаються через функціональні інтерфейси та лямбда-вирази, що дозволяє писати код у декларативному стилі (ми описуємо, що хочемо отримати, а не як це зробити).
3. Проміжні операції не виконуються негайно. Вони лише вибудовують план обробки. Всі обчислення запускаються лише тоді, коли викликається фінальна, термінальна операція.

4. Після виклику термінальної операції потік вважається "спожитим", і для повторної обробки потрібно створювати новий потік з того ж джерела.
5. Робота зі Stream API зазвичай виглядає як ланцюжок викликів методів, який називається конвеєром (Stream Pipeline) і складається з трьох етапів:
6. Отримання потоку з джерела даних. Найчастіше джерелом є колекція (наприклад, List або Set), і потік створюється викликом методу .stream().
7. Нуль або більше операцій, які перетворюють потік на інший потік. До них належать filter, map, sorted та інші. Вони є "лінивими".
8. Одна операція в кінці конвеєра, яка запускає обробку і створює кінцевий результат (наприклад, нову колекцію, число) або виконує дію (наприклад, виведення в консоль).

Основні операції:

1. Проміжна операція filter(Predicate<T>)

Метод filter призначений для фільтрації елементів потоку. Він приймає як аргумент Predicate — функціональний інтерфейс, що представляє умову. filter пропускає далі по конвеєру лише ті елементи, для яких умова повертає true.

2. Проміжна операція map(Function<T, R>)

Метод map призначений для трансформації кожного елемента потоку. Він приймає Function, яка застосовується до кожного елемента, перетворюючи його на новий елемент (можливо, іншого типу). Наприклад, перетворити String на Integer або об'єкт User на його ім'я.

3. Термінальна операція collect(Collector)

Метод collect є однією з найпотужніших термінальних операцій. Він використовується для збору елементів потоку в якусь структуру даних, найчастіше — в нову колекцію. Для визначення того, як саме збирати елементи, використовується об'єкт Collector, який зручно отримувати з утилітарного класу java.util.stream.Collectors.

Тепер давайте перепишемо наш початковий приклад, використовуючи Stream API:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

List<Integer> result = numbers.stream() // 1. Отримуємо потік
    .filter(n -> n % 2 == 0)          // 2. Фільтруємо парні числа
    .map(n -> n * 2)            // 3. Трансформуємо (множимо на 2)
.collect(Collectors.toList());//4.Збираємо результат у новий список
```

Цей код робить те ж саме, що і приклад з циклом for, але він значно більш читабельний та лаконічний. Ми не описуємо механіку ітерації, а декларативно

вказуємо, що ми хочемо зробити з даними. Кожен крок конвеєра відповідає за одну чітку операцію.

Окрім collect, існують й інші корисні термінальні операції, такі як forEach() (виконати дію для кожного елемента), count() (порахувати кількість елементів), findFirst() (знайти перший елемент), anyMatch() (перевірити, чи відповідає хоча б один елемент умові) та багато інших.

Клас Optional, усунення проблеми null, приклади правильного використання

Однією з найчастіших помилок, з якою стикаються Java-розробники, є NullPointerException (NPE). Вона виникає при спробі звернутися до методів або полів змінної, яка насправді не посилається на жоден об'єкт, а має значення null. Творець концепції null, Тоні Хоар, назвав своє творіння "помилкою на мільярд доларів" через незліченну кількість годин, витрачених на відлагодження помилок, пов'язаних з NPE. Проблема null полягає в тому, що він руйнує контракт методу: коли метод повертає об'єкт, незрозуміло, чи він завжди повертає об'єкт, чи може повернути null. Це змушує програмістів писати безліч захисних перевірок if (value != null), які засмічують код.

Для вирішення цієї проблеми у Java 8 було введено клас java.util.Optional<T>. Важливо розуміти, що Optional — це не заміна null. Це клас-контейнер (або обгортка), який може містити або один не-null об'єкт, або бути порожнім.

Уявіть Optional як коробку. Ця коробка завжди існує. Ви можете її передавати, перевіряти, але щоб дістати вміст, вам потрібно її відкрити. Коли ви її відкриваєте, всередині або є предмет (значення), або його немає. Optional робить неявний контракт (можливість повернення null) явним. Якщо метод повертає Optional<String>, він чітко "говорить" клієнтському коду: "Я спробую знайти для вас рядок, але його може й не бути, і ви зобов'язані обробити обидва випадки".

Існує кілька статичних методів для створення екземплярів Optional:

Optional.of(value) створює Optional, що містить передане значення. Використовуйте цей метод, коли ви на 100% впевнені, що значення не є null. Якщо передати null в Optional.of(), буде викинуто NullPointerException.

Optional.ofNullable(value) найбільш безпечний та поширений спосіб. Якщо передане значення не є null, створюється Optional з цим значенням. Якщо ж значення null, створюється порожній Optional.

```
Optional.empty() явно створює порожній Optional.
String name = "John";
```

```
Optional<String> optName = Optional.ofNullable(name); // Створить Optional з "John"
String nullName = null;
Optional<String> optNullName = Optional.ofNullable(nullName); // Створить порожній Optional
```

Головна ідея Optional — уникнути явних null-перевірок і використовувати більш виразні методи для обробки значення.

Найпростіший, але найменш ідіоматичний спосіб роботи з Optional виглядає так:

```
// Цей підхід не рекомендується
if (optName.isPresent()) {
    String value = optName.get();
    System.out.println("Name: " + value);
}
```

Метод isPresent() перевіряє, чи є значення всередині, а get() — дістає його. Якщо викликати get() для порожнього Optional, ви отримаєте NoSuchElementException. Хоча цей код працює, він є просто більш багатослівною версією старої перевірки if (name != null), і не використовує переваг функціонального стилю.

Optional надає низку методів, що дозволяють обробляти значення у функціональному, декларативному стилі.

ifPresent(Consumer<T>) виконує передану дію (лямбда-вираз) лише в тому випадку, якщо значення присутнє.

```
optName.ifPresent(nameValue -> System.out.println("Name: " + nameValue));
```

orElse(T other): Повертає значення, якщо воно є, або вказане значення за замовчуванням, якщо Optional порожній.

```
String nameOrDefault = optNullName.orElse("Guest"); // nameOrDefault буде "Guest"
```

orElseGet(Supplier<T>): Схожий на orElse, але значення за замовчуванням генерується за допомогою Supplier (лямбда-виразу) і викликається тільки якщо Optional порожній. Це ефективно, якщо створення значення за замовчуванням є ресурсозатратною операцією.

```
String nameFromDb = optNullName.orElseGet(() -> fetchDefaultNameFromDatabase());
```

orElseThrow(Supplier<Exception>): Найкращий спосіб обробити ситуацію, коли відсутність значення є помилкою. Повертає значення, якщо воно є, інакше кидає виняток, створений переданим Supplier.

```
String nameRequired = optNullName.orElseThrow(() -> new IllegalStateException("Name is required"));
```

map(Function<T, R>): Дозволяє безпечно трансформувати значення всередині Optional. Якщо значення є, до нього застосовується передана функція, і map повертає новий Optional з результатом. Якщо Optional порожній, map повертає порожній Optional. Це дозволяє створювати ланцюжки безпечних операцій.

```
optName.map(String::toUpperCase) // Поверне Optional<String> з "JOHN"
.ifPresent(upperName -> System.out.println("Upper case: " + upperName));
```

Optional — це потужний інструмент, але його слід використовувати за призначенням. Основне призначення: як тип повернення для методів, які можуть не знайти результат. Не використовуйте Optional для полів класу, параметрів методів або як елементи колекцій. Це безпідставно ускладнює код, додаючи зайвий шар обгортки без суттєвих переваг.

Основи вводу/виводу (I/O API)

Будь-яка програма так чи інакше взаємодіє із зовнішнім світом: читає дані з файлів, отримує інформацію з мережі, виводить результати на консоль або зберігає їх на диск. Цей процес взаємодії називається вводом/виводом (Input/Output, I/O). У Java для роботи з I/O існує потужний та гнучкий набір класів, відомий як I/O API. Основою цього API є концепція потоку (Stream) — абстракції, що представляє послідовний потік даних, який або читається з джерела (ввід), або записується до приймача (вивід).

Java розділяє всі потоки на два основні типи, залежно від природи даних, з якими вони працюють.

1. **Байтові потоки**: InputStream та OutputStream

Байтові потоки призначені для роботи з необробленими двійковими даними у вигляді байтів. Вони є універсальними і можуть використовуватися для читання та запису будь-яких типів файлів, будь то текстові файли, зображення, аудіо, відео або виконувані файли.

InputStream — це абстрактний клас, що є базовим для всіх потоків вводу байтів. Його нащадки, такі як FileInputStream, дозволяють читати дані з файлів.

OutputStream — це абстрактний клас, що є базовим для всіх потоків виводу байтів. Його нащадки, такі як FileOutputStream, дозволяють записувати дані у файли.

2. **Символьні потоки**: Reader та Writer

На відміну від байтових, символьні потоки призначені спеціально для роботи з текстовими даними у вигляді символів. Вони автоматично обробляють кодування, перетворюючи послідовності байтів у символи відповідно до заданої

кодової таблиці (наприклад, UTF-8) і навпаки. Це робить їх значно зручнішими та безпечнішими для роботи саме з текстом.

Reader — це абстрактний клас для потоків вводу символів, представлений, наприклад, класом FileReader.

Writer — це абстрактний клас для потоків виводу символів, представлений, наприклад, класом FileWriter.

Архітектура Java I/O широко використовує патерн "Декоратор", дозволяючи "обгортати" одні потоки іншими для додавання нової функціональності.

Буферизовані потоки (BufferedReader, BufferedWriter): Обгортання базового потоку (наприклад, FileReader) у BufferedReader значно підвищує продуктивність. Замість того, щоб читати дані з файлу по одному символу, буферизований потік читає їх великими блоками в буфер (внутрішню область пам'яті), а програма вже отримує дані з цього швидкого буфера. Це різко зменшує кількість повільних звернень до фізичного диска. Крім того, BufferedReader надає зручний метод readLine(), що дозволяє читати текст рядок за рядком.

Мости між потоками (InputStreamReader, OutputStreamWriter): Ці класи є "мостами" між байтовими та символьними потоками. Вони дозволяють, наприклад, читати текстовий файл з певним кодуванням, обгорнувши FileInputStream (байтовий) в InputStreamReader (символьний) і вказавши потрібне кодування.

Будь-який потік є системним ресурсом, який необхідно обов'язково закривати після завершення роботи з ним. Раніше для цього використовувалися громіздкі блоки try-catch-finally. Починаючи з Java 7, найкращою практикою є використання конструкції try-with-resources, яка автоматично закриває всі ресурси, оголошені в її круглих дужках.

```
// Приклад читання текстового файлу рядок за рядком
try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Окремо від ієрархії потоків стоїть клас RandomAccessFile. На відміну від потоків, які є суто послідовними, цей клас дозволяє читати та записувати дані у довільному місці файлу. Він використовує внутрішній "вказівник" (file pointer), який можна переміщувати за допомогою методу seek(). Це робить його

ідеальним для роботи з файлами, що мають фіксовану структуру, наприклад, з базами даних. При створенні RandomAccessFile необхідно вказати режим доступу, наприклад, "r" (тільки читання) або "rw" (читання та запис).

Починаючи з Java 7, для високоефективних та інтенсивних операцій I/O був представлений новий API — **Java NIO (New I/O)**. Основна відмінність полягає в парадигмі:

Класичний I/O є потоко-орієнтованим, тобто дані читаються/пишуться послідовно, байт за байтом.

NIO є буфер-орієнтованим – дані спочатку завантажуються в буфер (блок пам'яті), а вже потім обробляються. Це дозволяє гнучко маніпулювати даними в буфері, переміщаючись по них у будь-якому напрямку.

Основними компонентами NIO є Канали (Channels), що представляють джерело/приймач даних, та Буфери (Buffers), що є тимчасовими сховищами для них.

Разом з NIO з'явився і сучасний API для роботи з файловою системою (java.nio.file), який є значно зручнішим за старий клас java.io.File. Класи Paths та Files надають простий та потужний інтерфейс для маніпуляцій з файлами та каталогами. Наприклад, копіювання файлу тепер можна виконати в один рядок:

```
Path source = Paths.get("input.txt");
Path destination = Paths.get("output.txt");
Files.copy(source,destination,StandardCopyOption.REPLACE_EXISTING);
```

Загалом, для простих послідовних операцій з текстом класичний I/O з буферизацією є достатнім та простим. Для більш складних завдань, маніпуляцій з файловою системою та високопродуктивних операцій перевагу слід надавати сучасному NIO API.

Обробка винятків try-catch-finally, throws

Під час виконання програми можуть виникати непередбачувані або помилкові ситуації, що порушують її нормальний хід — наприклад, спроба поділити на нуль, прочитати неіснуючий файл або звернутися до об'єкта за посиланням null. Такі події в Java називаються винятками (Exceptions). Механізм обробки винятків дозволяє програмі не аварійно завершувати роботу, а коректно реагувати на ці ситуації.

Усі винятки в Java поділяються на дві основні категорії.

Checked Exceptions (Перевірювані винятки). Це винятки, які компілятор змушує обробляти. Зазвичай вони виникають через зовнішні фактори, які не залежать від коду, наприклад IOException (помилка вводу/виводу). Ви повинні або обробити такий виняток у блоці try-catch, або вказати, що ваш метод може його "викинути", за допомогою ключового слова throws.

Unchecked Exceptions (Неперевірювані винятки). До них належать RuntimeException та його нащадки (NullPointerException, ArrayIndexOutOfBoundsException тощо), а також помилки (Error). Зазвичай вони свідчать про логічні помилки в програмі. Компілятор не вимагає їх обов'язкової обробки, але за бажанням їх також можна перехопити.

Для обробки винятків використовується конструкція try-catch-finally.

У try блок поміщається "небезпечний" код, який потенційно може згенерувати виняток.

Якщо в блоці try виникає виняток певного типу, виконання try негайно припиняється, і управління передається відповідному блоку catch. Блок catch містить логіку для обробки помилки. Можна визначити кілька блоків catch для різних типів винятків.

Блок finally є необов'язковим. Код усередині finally гарантовано виконається після завершення try та catch, незалежно від того, чи виник виняток, і чи був він оброблений. Зазвичай він використовується для закриття ресурсів (файлів, з'єднань з базою даних), щоб уникнути їх "витоку".

```
try {
    // Код, що може згенерувати IOException
    FileReader reader = new FileReader("someFile.txt");
} catch (FileNotFoundException e) {
    // Обробка конкретної помилки: файл не знайдено
    System.err.println("Файл не знайдено: " + e.getMessage());
} finally {
  System.out.println("Цей блок виконається в будь-якому випадку.");
}
```

Якщо метод не призначений для самостійної обробки винятку, він може "прокинути" його на вищий рівень — тому коду, який його викликав. Для цього в сигнатурі методу використовується ключове слово throws, за яким слідує список типів винятків.

```
//Цей метод не обробляє IOException, а делегує це відповідальність коду, який буде викликати readContent.
public String readContent(String filePath) throws IOException {
    // ... логіка читання файлу ...
    return content;
}
```

Для моделювання специфічних помилок вашої бізнес-логіки можна створювати власні класи винятків. Для цього достатньо успадкувати свій клас від Exception (для створення перевірюваного винятку) або від RuntimeException (для неперевірюваного), як це пропонується в лабораторній роботі для InvalidEmailException та WeakPasswordException.

Практична робота №5

Тема: Робота з колекціями

Мета: Отримати практичний досвід роботи з основними типами колекцій Java (List, Set, Map) та зрозуміти різницю в їх продуктивності та використанні.

Завдання

1. Створити ArrayList рядків, додати кілька елементів, видалити один та вивести результат.

2. Створити HashSet та додати до нього кілька однакових елементів, переконатись, що дублікати не зберігаються.

3. Створити HashMap для зберігання пар "студент-оцінка" та вивести всіх студентів з оцінкою вище 80.

4. Написати функцію, яка приймає List<Integer> і повертає новий список без дублікатів.

5. Продемонструвати різницю у продуктивності додавання елементів в ArrayList та LinkedList.

Лабораторна робота №5

Тема: Розробка системи з обробкою винятків

Мета: Навчитися створювати власні типи винятків та реалізовувати механізми їх обробки для підвищення надійності програмного забезпечення.

Завдання

1. Створити клас UserRegistrationService.

2. Реалізувати в ньому метод registerUser(String email, String password).

3. Створити власні винятки: InvalidEmailException та WeakPasswordException.

4. Метод registerUser має перевіряти email та пароль і кидати відповідні винятки.

5. Написати клієнтський код, що викликає цей метод та коректно обробляє можливі винятки в блоці try-catch.

Контрольні запитання

1. Яка фундаментальна різниця між інтерфейсами Collection та Map? Назвіть три основні інтерфейси, що є нащадками Collection, та коротко опишіть їх призначення.
2. У якому випадку для реалізації списку доцільніше обрати LinkedList замість ArrayList? А в якому випадку HashSet буде кращим вибором, ніж TreeSet?

3. Сформулюйте головне правило ("контракт"), що пов'язує методи equals() та hashCode(). Що станеться, якщо перевизначити equals(), але не перевизначити hashCode() при роботі з HashSet?
4. Що таке Stream API? Опишіть три етапи конвеєра операцій (stream pipeline) та поясніть призначення проміжних операцій filter() і map().
5. Яку проблему вирішує клас Optional? Чому використання ланцюжка if (opt.isPresent()) { opt.get() } вважається антипатерном, і які методи Optional є кращою альтернативою для обробки значення за замовчуванням?
6. У чому полягає різниця між байтовими (InputStream/OutputStream) та символьними (Reader/Writer) потоками в Java I/O? Для чого використовуються буферизовані потоки (наприклад, BufferedReader)?
7. Опишіть призначення кожного з блоків у конструкції try-catch-finally. Який з цих блоків гарантовано виконається, незалежно від наявності винятку?
8. Яка різниця між checked та unchecked винятками в Java? Яку роль виконує ключове слово throws у сигнатурі методу?

Лекція 6. Багатопотоковість та введення в асинхронність

Поняття процесу та потоку. Проблеми паралельного доступу до даних

Сучасні комп'ютерні системи мають багатоядерні процесори, здатні виконувати кілька задач одночасно. Щоб ефективно використовувати ці ресурси, Java надає потужну підтримку для багатопотоковості (multithreading) — можливості виконувати кілька частин програми паралельно. Це дозволяє створювати більш продуктивні та чутливі до користувача додатки.

Процес — це екземпляр програми, що виконується операційною системою. Кожен процес має власний, ізольований простір пам'яті. Наприклад, ваш веб-браузер та текстовий редактор є двома окремими процесами.

Потік (Thread) — це найменша одиниця виконання всередині процесу. Кожен процес має щонайменше один головний потік, але може створювати й додаткові. Ключовою відмінністю є те, що всі потоки в межах одного процесу використовують спільний простір пам'яті. Це дозволяє їм легко обмінюватися даними, але водночас створює серйозні проблеми.

Коли кілька потоків одночасно намагаються читати та змінювати спільні дані, виникають проблеми паралельного доступу. Найпоширенішою з них є стан гонитви (Race Condition). Це ситуація, коли кінцевий результат залежить від непередбачуваної послідовності виконання операцій різними потоками.

Класичний приклад — два потоки, що намагаються інкрементувати спільний лічильник. Потік А може прочитати значення (напр., 5), але перш ніж він запише нове значення (6), Потік Б також прочитає старе значення (5). У результаті обидва потоки запишуть 6, і один інкремент буде втрачено. Для

вирішення таких проблем необхідні механізми синхронізації, які ми розглянемо пізніше.

Створення потоків: успадкування від Thread та реалізація Runnable

У Java є два основні способи створення нового потоку виконання.

1. Успадкування від класу Thread

Цей спосіб полягає у створенні класу, що розширює java.lang.Thread, та перевизначенні його методу run(), в якому розміщується логіка, що має виконуватися в новому потоці.

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Новий потік виконується.");
    }
}
// ...
MyThread myThread = new MyThread();
myThread.start(); // Запускаємо потік
```

Важливо викликати саме метод start(), який готує та запускає новий потік. Прямий виклик myThread.run() просто виконає код у поточному потоці, а не в новому.

2. Реалізація інтерфейсу Runnable

Цей спосіб передбачає створення класу, що реалізує інтерфейс java.lang.Runnable, який має єдиний метод run().

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
      System.out.println("Завдання виконується в новому потоці.");
    }
}
// ...
MyRunnable task = new MyRunnable();
Thread thread = new Thread(task); // Передаємо завдання в конструктор Thread
thread.start(); // Запускаємо потік
```

Незважаючи на те, що обидва підходи працюють, настійно рекомендується надавати перевагу реалізації інтерфейсу Runnable. Java не підтримує множинне успадкування класів. Якщо ваш клас успадковує Thread, він більше не може успадковувати жоден інший клас. Реалізація Runnable залишає цю можливість відкритою.

Підхід з розділення відповідальності є кращим з точки зору дизайну. Він розділяє завдання (Runnable), яке потрібно виконати, від виконавця (Thread), який це завдання виконує.

Один і той самий об'єкт Runnable може бути переданий кільком об'єктам Thread або, що більш важливо, сучасним інструментам для керування потоками, таким як ExecutorService.

Синхронізація, ключове слово synchronized, монітори

Як ми з'ясували, одночасний доступ кількох потоків до спільних змінних може призвести до помилок, таких як стан гонитви. Щоб запобігти цьому, Java надає механізм синхронізації, який гарантує, що критична ділянка коду (та, що працює зі спільними ресурсами) в один момент часу виконується лише одним потоком.

Основним вбудованим інструментом для синхронізації в Java є ключове слово synchronized. В його основі лежить концепція монітора (також відомого як внутрішнє блокування або intrinsic lock). Кожен об'єкт у Java має асоційований з ним унікальний монітор.

Принцип роботи synchronized полягає в захопленні цього монітора.

- Коли потік намагається увійти в блок коду, позначений як synchronized, він спершу повинен "захопити" монітор відповідного об'єкта.
- Якщо монітор вільний, потік захоплює його, входить у критичну секцію і виконує код.
- Якщо монітор вже захоплений іншим потоком, поточний потік блокується і чекає, доки монітор не буде звільнено.
- Коли потік завершує виконання коду в synchronized блоці, він автоматично звільняє монітор, дозволяючи іншим потокам, що очікують, спробувати його захопити.
- Цей механізм гарантує взаємне виключення (mutual exclusion), забезпечуючи потокобезпечність.

Існує два способи застосування цього ключового слова:

1. Синхронізовані методи

Найпростіший спосіб — додати ключове слово synchronized до оголошення методу. Це перетворює весь метод на критичну секцію.

Для екземплярного (нестатичного) методу потік захоплює монітор об'єкта this (того екземпляра, для якого викликано метод).

Для статичного методу потік захоплює монітор об'єкта .class, що представляє сам клас.

Розв'яжемо проблему з лічильником з попереднього розділу:

```
class Counter {
```

```
    private int count = 0;
    // Тільки один потік може виконувати цей метод одночасно
    public synchronized void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

2. Синхронізовані блоки

Цей спосіб забезпечує більш гнучкий контроль, дозволяючи синхронізувати не весь метод, а лише його частину. Це може покращити продуктивність, оскільки блокування діє на менший проміжок часу.

```
public void myMethod() {
    // ... несинхронізований код ...
  synchronized (this) { // Блокування на моніторі поточного об'єкта
        // Критична секція: тільки один потік може бути тут
        // ... робота зі спільними даними ...
    }
    // ... інший несинхронізований код … }
```

У дужках після synchronized вказується об'єкт, монітор якого буде використовуватися для блокування. Це може бути this, або будь-який інший спільний для потоків об'єкт.

Хоча synchronized є потужним інструментом, його надмірне або неправильне використання може призвести до проблем продуктивності та "проблем живучості" (liveness issues), найвідомішою з яких є взаємне блокування (deadlock) — ситуація, коли два або більше потоків нескінченно чекають один на одного, щоб звільнити ресурси.

ExecutorService, сучасний підхід до управління пулом потоків

Створення нового об'єкта Thread для кожного завдання (new Thread(runnable).start()) є простим, але неефективним підходом, особливо у великих додатках. Створення та знищення потоків — це ресурсозатратна операція. Постійне виконання цих дій для великої кількості короткочасних завдань може суттєво знизити продуктивність системи.

Для вирішення цієї проблеми в Java був представлений ExecutorService — інтерфейс з пакета java.util.concurrent, що є сучасним та значно більш ефективним підходом до управління асинхронними задачами.

ExecutorService працює з концепцією пулу потоків. **Пул потоків** — це керована колекція заздалегідь створених потоків, які готові до виконання завдань. Замість того, щоб створювати новий потік для кожного завдання, ви

просто передаєте завдання в ExecutorService, а він призначає його вільному потоку з пулу. Після виконання завдання потік не знищується, а повертається в пул, готовий до виконання наступного завдання.

Повторне використання потоків усуває накладні витрати на їх постійне створення та знищення. Пул обмежує максимальну кількість одночасно працюючих потоків, запобігаючи вичерпанню системних ресурсів.

ExecutorService надає простий API, що відокремлює логіку "що виконати" (Runnable) від механіки "як виконати" (управління потоками).

Найпростіший спосіб створити ExecutorService — використати статичні методи-фабрики з класу Executors. Розглянемо приклади методів, що використовуються найчастіше.

Executors.newFixedThreadPool(int n) створює пул з фіксованою кількістю потоків. Якщо всі потоки зайняті, нові завдання очікують у черзі. Це ідеальний варіант для контролю над максимальним навантаженням.

Executors.newCachedThreadPool() створює гнучкий пул, який може створювати нові потоки за потребою або пере-використовувати існуючі.

Executors.newSingleThreadExecutor() створює сервіс з одним потоком, що гарантує послідовне виконання всіх завдань.

Після створення пулу, завдання передаються на виконання за допомогою методу execute():

```
// Створюємо пул з 3 потоків, як у лабораторній роботі
ExecutorService executor = Executors.newFixedThreadPool(3);
// Створюємо 10 завдань
for (int i = 0; i < 10; i++) {
    Runnable task = () -> {
        System.out.println("Виконується завдання в потоці: " +
Thread.currentThread().getName());
        try {
            Thread.sleep(1000); // Імітація довгої роботи
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    };
    executor.execute(task); // Передаємо завдання на виконання
}
```

При запуску цього коду ви побачите, що завдання виконуються паралельно, але не більше ніж 3 одночасно, що відповідає розміру пулу.

Дуже важливим кроком є коректне завершення роботи ExecutorService. Потоки в пулі не є "демонами", тому програма (JVM) не завершить свою роботу, поки пул активний.

shutdown() ініціює "м'яке" завершення. Пул перестає приймати нові завдання, але виконує всі, що вже знаходяться в черзі.

shutdownNow() намагається негайно зупинити всі активні завдання та повертає список завдань, що очікували в черзі.

Правильний життєвий цикл роботи з ExecutorService виглядає так:

```
ExecutorService executor = Executors.newFixedThreadPool(3);
// ... передача завдань на виконання ...
executor.shutdown();
```

ExecutorService є стандартним та рекомендованим способом для управління потоками в сучасних Java-додатках, забезпечуючи значно кращу продуктивність та контроль над ресурсами порівняно з ручним створенням об'єктів Thread.

Введення в асинхронну розробку, Future та CompletableFuture

Традиційно, виклик методу в програмі є синхронним — основний потік коду блокується і чекає, поки метод завершить свою роботу і поверне результат. Це схоже на телефонний дзвінок: ви чекаєте на лінії, доки співрозмовник не відповість. Такий підхід є простим, але неефективним для довготривалих операцій (наприклад, мережевих запитів або складних обчислень), оскільки вся програма "зависає" в очікуванні.

Асинхронна розробка пропонує інший підхід: ви запускаєте довготривале завдання у фоновому потоці, і основний потік негайно отримує управління назад, продовжуючи виконувати іншу роботу. Це схоже на відправку SMS: ви відправили повідомлення і можете займатися своїми справами, а результат (відповідь) надійде пізніше.

Для роботи з результатами асинхронних операцій у Java існує інтерфейс Future<V>. Коли ви передаєте завдання (Callable) в ExecutorService за допомогою методу submit(), він негайно повертає об'єкт Future. Цей об'єкт є "обіцянкою" або заповнювачем для результату, який буде доступний у майбутньому.

Основні методи Future:

- get() блокуючий метод. Він зупиняє поточний потік і чекає, доки асинхронне завдання не завершиться, після чого повертає його результат.
- isDone() перевіряє, чи завершилося завдання.
- cancel() намагається скасувати виконання завдання.

Хоча Future дозволяє отримати результат, він має суттєві обмеження. Щоб виконати якусь дію з результатом, ви змушені викликати блокуючий метод get(), що нівелює переваги асинхронності. Створювати ланцюжки залежних асинхронних операцій з Future складно і громіздко.

У Java 8 з'явився значно більш гнучкий та потужний клас CompletableFuture<T>. Він не лише представляє майбутній результат, а й надає багатий API для побудови декларативних конвеєрів асинхронних операцій без блокувань.

Замість того, щоб чекати на результат за допомогою get(), CompletableFuture дозволяє реєструвати колбеки — дії, які будуть автоматично виконані після завершення завдання. Легко створюється за допомогою статичних методів, наприклад supplyAsync(), який приймає завдання та ExecutorService.

Методи thenApply(), thenAccept(), thenRun() дозволяють створювати ланцюжки залежних дій. thenApply() приймає результат попереднього етапу, трансформує його і передає далі.

Методи thenCombine(), allOf(), anyOf() дозволяють комбінувати результати кількох асинхронних завдань.

Метод exceptionally() дозволяє визначити логіку на випадок, якщо під час асинхронного виконання виникне виняток.

Практична робота №6

Тема: Створення та запуск потоків

Мета: Ознайомитися з основами багатопотоковості в Java, продемонструвати проблему "race condition" та вирішити її за допомогою синхронізації.

Завдання

1. Створити та запустити два потоки, один через успадкування від Thread, інший – через Runnable.

2. Кожен потік має виводити своє ім'я та лічильник від 1 до 10.

3. Написати програму з загальним лічильником, який інкрементують кілька потоків.

4. Продемонструвати проблему race condition.

5. Використати synchronized для вирішення проблеми з доступом до спільного лічильника.

Лабораторна робота №6

Тема: Робота з пулом потоків

Мета: Навчитися використовувати ExecutorService для ефективного управління потоками та виконання паралельних завдань.

Завдання

1. Створити ExecutorService з фіксованим пулом потоків (напр., 3 потоки).

2. Створити клас-завдання (Runnable), який імітує довгу операцію (напр., засинає на 2 секунди).

3. Відправити на виконання в пул 10 таких завдань.

4. Переконатися, що завдання виконуються паралельно, але не більше ніж 3 одночасно.

5. Коректно завершити роботу ExecutorService після виконання всіх завдань.

Контрольні запитання

1. У чому полягає ключова відмінність між процесом і потоком? Яка головна проблема виникає через те, що потоки використовують спільний простір пам'яті?
2. Назвіть два основні способи створення потоків у Java. Якому з них і чому рекомендується надавати перевагу в сучасному програмуванні?
3. Що таке "стан гонитви" (Race Condition)? Поясніть, як ключове слово synchronized та концепція монітора допомагають вирішити цю проблему.
4. Які існують два способи застосування ключового слова synchronized? У якому випадку доцільно використовувати синхронізований блок замість синхронізованого методу?
5. Чому використання ExecutorService та пулу потоків є більш ефективним підходом, ніж ручне створення new Thread() для кожного завдання?
6. Чому важливо викликати метод shutdown() після завершення роботи з ExecutorService?
7. Поясніть різницю між синхронним та асинхронним виконанням.
8. Чим CompletableFuture (Java 8) є більш потужним інструментом для асинхронної розробки порівняно з оригінальним інтерфейсом Future?

ТЕМА 4. ЯКІСТЬ ТА АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Лекція 7. Основи патернів проєктування

Що таке патерни проєктування, їх користь, класифікація (твірні, структурні, поведінкові)

При розробці програмного забезпечення програмісти часто стикаються з однотипними, повторюваними проблемами архітектури та дизайну. Замість того, щоб щоразу "винаходити велосипед", досвідчені розробники використовують перевірені часом рішення. Ці формалізовані, найкращі практики відомі як патерни проєктування.

Патерн проєктування (Design Pattern) — це універсальне, повторно використовуване архітектурне рішення для часто виникаючої проблеми в рамках певного контексту. Важливо розуміти, що патерн — це не готовий фрагмент коду чи бібліотека, а скоріше шаблон, ідея або концепція. Він описує взаємодію об'єктів та класів для вирішення задачі, а конкретна реалізація може відрізнятися залежно від мови програмування та специфіки проєкту.

Патерни є результатом багаторічного досвіду спільноти розробників. Вони допомагають уникнути поширених помилок і побудувати надійну та стабільну архітектуру.

Патерни надають розробникам спільну мову для комунікації. Замість того, щоб довго пояснювати складну структуру, можна просто сказати: "Тут використаємо патерн 'Фабрика'". Це значно прискорює обговорення та проєктування.Застосування патернів сприяє дотриманню SOLID-принципів, роблячи код більш гнучким, слабкозв'язаним та легким для підтримки та розширення.

Найбільш відома класифікація, запропонована "Бандою чотирьох" (Gang of Four, GoF), поділяє патерни на три основні категорії залежно від їх призначення.

Твірні патерни (Creational Patterns) – займаються процесом створення об'єктів. Вони дозволяють зробити систему незалежною від того, як саме створюються, компонуються та представляються її об'єкти. Замість прямого виклику new, використовуються спеціальні методи, що приховують логіку створення.

Структурні патерни (Structural Patterns) – описують, як формувати більші структури з окремих класів та об'єктів. Вони допомагають проєктувати ієрархії та композиції, які є гнучкими та ефективними.

Поведінкові патерни (Behavioral Patterns) – стосуються алгоритмів та розподілу відповідальності між об'єктами. Вони визначають ефективні та безпечні способи взаємодії та комунікації між об'єктами в системі.

Вивчення патернів проєктування є ключовим етапом у становленні професійного розробника, оскільки вони дозволяють мислити на більш високому рівні абстракції та приймати грамотні архітектурні рішення.

Factory Method: делегування створення об'єктів підкласам

Одним з найпоширеніших та найважливіших твірних патернів є Фабричний метод (Factory Method). Його основна ідея полягає у визначенні загального інтерфейсу для створення об'єктів, але при цьому делегуванні рішення про те, екземпляр якого саме класу створювати, своїм підкласам. Цей патерн є класичною реалізацією Принципу Відкритості/Закритості (OCP) та Принципу Інверсії Залежностей (DIP).

Уявіть, що ви розробляєте логістичну програму. На початковому етапі ваша програма має організовувати доставку лише вантажівками. Ви можете написати код, який безпосередньо створює об'єкти Truck:

```
// Пряме створення об'єкта
public class LogisticsApp {
    public void planDelivery() {
        Truck truck = new Truck(); // Жорстка прив'язка до конкретного класу
        truck.deliver();
    }
}
```

Такий підхід працює, доки вимоги не змінюються. Але що, якщо бізнес розширюється, і тепер потрібно доставляти вантажі ще й кораблями? Вам доведеться модифікувати клас LogisticsApp, додавши в нього логіку вибору: if...else, щоб створювати або Truck, або Ship. Кожного разу, коли з'являтиметься новий вид транспорту (літак, потяг), ви будете змушені модифікувати цей центральний клас, що робить його складним, крихким та порушує OCP. Проблема полягає в тому, що високоривнева логіка (planDelivery) тісно пов'язана з низькоривневими деталями створення конкретних об'єктів.

Патерн "Фабричний метод" пропонує вирішити цю проблему, винісши логіку створення об'єктів в окремий метод — фабричний метод. Замість того, щоб викликати new напряму, клієнтський код викликає цей метод.

Структура патерну складається з наступних учасників:

Продукт (Product) загальний інтерфейс або абстрактний клас для об'єктів, які ми хочемо створювати (наприклад, Transport). Він визначає спільні методи, які будуть у всіх продуктів (наприклад, deliver()).

Конкретний Продукт (Concrete Product) конкретні класи, що реалізують інтерфейс Продукту (Truck, Ship).

Творець (Creator) абстрактний клас, який оголошує абстрактний фабричний метод, що повертає об'єкт типу Продукт. Творець може містити загальну бізнес-логіку, яка працює з продуктами, створеними фабричним методом.

Конкретний Творець (Concrete Creator) підкласи, що розширюють Творця і перевизначають фабричний метод, щоб він повертав екземпляр конкретного продукту.

Давайте застосуємо цей патерн до нашого прикладу.

Крок 1: Створити інтерфейс Продукту

```
// Інтерфейс для всіх видів транспорту
interface Transport {
    void deliver();
}
```

Крок 2: Створити Конкретні Продукти

```
class Truck implements Transport {
   public void deliver() { System.out.println("Доставка вантажівкою по дорозі."); }
}
class Ship implements Transport {
    public void deliver() { System.out.println("Доставка кораблем по морю."); }
}
```

Крок 3: Створити абстрактного Творця з фабричним методом

```
// Абстрактний клас, що містить бізнес-логіку
abstract class Logistics {
    // Загальна логіка, що не залежить від конкретного транспорту
    public void planDelivery() {
        // Викликаємо фабричний метод для створення об'єкта
        Transport t = createTransport();
        // Використовуємо створений продукт
        t.deliver();
    }
//Абстрактний фабричний метод-його реалізація делегується підкласам
    protected abstract Transport createTransport();
}
```

Зверніть увагу: метод planDelivery тепер працює з абстракцією Transport і нічого не знає про те, як саме створюється цей об'єкт.

Крок 4: Створити Конкретних Творців

```
// Конкретний творець, що створює вантажівки
class RoadLogistics extends Logistics {
    @Override
    protected Transport createTransport() {
        return new Truck();   } }
```

```
// Конкретний творець, що створює кораблі
class SeaLogistics extends Logistics {
    @Override
    protected Transport createTransport() {
        return new Ship();
    }
}
```

Тепер, якщо нам знадобиться додати доставку літаками, ми просто створимо новий клас AirLogistics та Airplane, не змінюючи жодного рядка в існуючих класах. Система є закритою для модифікації, але відкритою для розширення.

Патерн "Фабричний метод" є потужним інструментом для послаблення зв'язків у системі. Він дозволяє делегувати відповідальність за створення об'єктів підкласам, що робить код більш гнучким, розширюваним та відповідним до принципів SOLID. Використовуйте цей патерн, коли у вас є клас, який не може заздалегідь знати, об'єкти яких саме підкласів йому потрібно буде створювати.

Builder: покрокове створення складних об'єктів

Ще одним важливим твірним патерном є Будівельник (Builder). Він призначений для вирішення проблеми створення складних об'єктів — об'єктів, що мають велику кількість полів та конфігураційних параметрів, багато з яких є необов'язковими.

Уявіть, що вам потрібно створити об'єкт HttpClient, який може мати безліч налаштувань: URL, HTTP-метод, заголовки, тіло запиту, таймаут, налаштування проксі тощо. Як це зробити?

Можна створити безліч конструкторів з різною кількістю параметрів — один для обов'язкових полів, другий для них же плюс один необов'язковий, третій — плюс два, і так далі. Це призводить до появи *"телескопічного" конструктора*, коли один конструктор викликає інший. Такий код дуже важко читати та підтримувати, а при великій кількості параметрів легко припуститися помилки, переплутавши їх місцями.

```
// Поганий підхід: телескопічний конструктор
HttpClient client = new HttpClient("http://...", "GET", null, null, 30000, null);
```

JavaBeans підхід (сеттери). Можна створити об'єкт за допомогою конструктора без параметрів, а потім налаштувати його поля через публічні сеттери. Цей підхід є більш читабельним, але має серйозний недолік: об'єкт може існувати в неконсистентному (незавершеному) стані протягом кількох кроків його налаштування. Крім того, цей підхід не дозволяє зробити поля об'єкта final, тобто зробити його незмінним (immutable) після створення.

Патерн "Будівельник" пропонує елегантне рішення: він виносить логіку конструювання складного об'єкта в окремий клас — **Будівельник** (Builder). Замість того, щоб створювати об'єкт напряму, ви спочатку створюєте об'єкт Будівельника, "налаштовуєте" його за допомогою ланцюжка викликів методів, а в кінці викликаєте фінальний метод, який і створює кінцевий об'єкт.

Структура патерну зазвичай виглядає наступним чином.

Продукт (Product) складний об'єкт, який ми хочеться створити (HttpClient). Його конструктор робиться private, щоб унеможливити створення об'єкта напряму.

Будівельник (Builder) статичний вкладений клас всередині класу Продукту. Він має поля, що дублюють поля Продукту.

Методи налаштування. Будівельник має методи для налаштування кожного параметра (наприклад, method(), headers(), timeout()). Кожен такий метод повертає посилання на самого себе (return this;), що дозволяє вибудовувати ланцюжки викликів (method chaining).

Метод build() фінальний метод, який створює екземпляр Продукту, передаючи йому в конструктор налаштовані в Будівельнику значення, і повертає готовий, незмінний об'єкт.

Приклад реалізації для HttpClient:

```
public class HttpClient {
    // Поля кінцевого об'єкта
    private final String url;
    private final String method;
    private final String body;
    private final int timeout;
    // 1. Приватний конструктор, який приймає Будівельника
    private HttpClient(Builder builder) {
        this.url = builder.url;
        this.method = builder.method;
        this.body = builder.body;
        this.timeout = builder.timeout;
    }
    // 2. Статичний вкладений клас Будівельника
    public static class Builder {
        // Поля Будівельника
        private String url;
        private String method = "GET"; // значення за замовчуванням
        private String body;
        private int timeout = 30000;
        // Конструктор Будівельника з обов'язковими параметрами
        public Builder(String url) {
            this.url = url;
```

```
            }
            // 3. Методи налаштування, що повертають 'this'
            public Builder method(String method) {
                this.method = method;
                return this;
            }
            public Builder body(String body) {
                this.body = body;
                return this;
            }
            public Builder timeout(int timeout) {
                this.timeout = timeout;
                return this;
            }
            // 4. Фінальний метод build()
            public HttpClient build() {
          // Створюємо екземпляр HttpClient, передаючи йому самого себе
                return new HttpClient(this);
            }
        }
    }
```

Тепер створення складного об'єкта стає простим, читабельним та безпечним:

```
    // Створення клієнта за допомогою Будівельника
HttpClient client = new HttpClient.Builder("http://example.com/api")
        .method("POST")
        .body("{\"key\":\"value\"}")
        .timeout(10000)
        .build();
```

Переваги патерну "Будівельник":

- Код стає значно більш зрозумілим, оскільки кожен виклик методу явно вказує, який саме параметр налаштовується.
- Дозволяє створювати об'єкти з будь-якою комбінацією необов'язкових параметрів.
- Дозволяє зробити кінцевий об'єкт незмінним, оскільки всі його поля final і ініціалізуються один раз у приватному конструкторі.
- Об'єкт створюється в один крок (викликом build()) і одразу знаходиться в коректному, завершеному стані.

Патерн "Будівельник" є ідеальним рішенням, коли вам потрібно конструювати складні об'єкти з великою кількістю параметрів, забезпечуючи при цьому чистоту коду та надійність.

Decorator: динамічне додавання нових обов'язків об'єкту

Декоратор (Decorator), також відомий як "Обгортка" (Wrapper), — це структурний патерн проєктування, який дозволяє динамічно додавати нову функціональність або обов'язки існуючим об'єктам, не змінюючи їхній код. Цей патерн є гнучкою альтернативою успадкуванню для розширення функціональності.

Класичний спосіб додати нову поведінку до класу — створити його нащадка. Однак цей підхід має суттєві обмеження. Функціональність додається на етапі компіляції. Неможливо додати або прибрати обов'язки з об'єкта під час виконання програми. Якщо потрібно додати кілька незалежних функцій, кількість можливих комбінацій може призвести до створення величезної кількості підкласів, що називають “вибухом класів”. Наприклад, для вікна, до якого можна додати рамку та/або смугу прокрутки, знадобилися б класи WindowWithBorder, WindowWithScrollbar та WindowWithBorderAndScrollbar.

Патерн "Декоратор" вирішує цю проблему за допомогою композиції. Ідея полягає в тому, щоб "обгорнути" вихідний об'єкт в інший об'єкт-декоратор. Цей декоратор має такий самий інтерфейс, як і вихідний об'єкт, що дозволяє використовувати його на місці оригінала. Коли клієнт викликає метод у декоратора, той виконує власну додаткову логіку, а потім делегує виклик об'єкту, який він "обгортає". Декоратори можна вкладати один в одного, створюючи ланцюжки з новою функціональністю.

Структура патерну включає наступних учасників:

- Компонент (Component) загальний інтерфейс як для об'єктів, що декоруються, так і для самих декораторів (наприклад, Report).
- Конкретний Компонент (Concrete Component) базовий клас з початковою функціональністю, яку ми хочемо розширити (наприклад, SimpleReport).
- Декоратор (Decorator) абстрактний клас, що реалізує інтерфейс Компонента. Він містить посилання на об'єкт Компонента (wrapped object) і делегує йому виклики.
- Конкретний Декоратор (Concrete Decorator) класи, що розширюють Декоратор і додають власну поведінку до або після виклику методу обгорнутого об'єкта (наприклад, HeaderDecorator, FooterDecorator).

Приклад реалізації для Report:

```
// 1. Component Interface
interface Report {
    String generate();
}
// 2. Concrete Component
class SimpleReport implements Report {
```

```
        public String generate() {
            return "Base report data";
        }
    }
    // 3. Abstract Decorator
    abstract class ReportDecorator implements Report {
        protected Report wrappedReport;

        public ReportDecorator(Report report) {
            this.wrappedReport = report;
        }
        public String generate() {
            return wrappedReport.generate(); // Делегування
        }
    }
    // 4. Concrete Decorator
    class HeaderDecorator extends ReportDecorator {
        public HeaderDecorator(Report report) {
            super(report);
        }
        @Override
        public String generate() {
            String originalReport = super.generate();
            return "Report Header\n---\n" + originalReport;
        }
    }
```

Тепер ми можемо динамічно "прикрашати" наш звіт:

```
    // Створюємо базовий звіт
    Report simple = new SimpleReport();
    // "Обгортаємо" його в декоратор для додавання заголовка
    Report withHeader = new HeaderDecorator(simple);
    // Можна обгортати декоратори один в одного
    Report finalReport = new FooterDecorator(new
HeaderDecorator(simple));

    System.out.println(withHeader.generate());
    // Виведе:
    // Report Header
    // ---
    // Base report data
```

Отже, патерн "Декоратор" дозволяє додавати функціональність до об'єктів динамічно, під час виконання програми. Система відкрита для розширення (можна створювати нові декоратори), але закрита для модифікації (не потрібно

змінювати код існуючих компонентів). Замість створення підкласів для кожної комбінації функцій, можна комбінувати прості декоратори.

Патерн "Декоратор" є чудовим вибором, коли вам потрібно розширювати поведінку об'єктів, не вдаючись до жорсткого механізму успадкування.

Strategy: інкапсуляція сімейства алгоритмів та забезпечення їх взаємозамінності

Стратегія (Strategy) — це поведінковий патерн проєктування, який дозволяє визначити сімейство схожих алгоритмів, помістити кожен з них у свій власний клас, та зробити їх об'єкти взаємозамінними. Цей патерн дозволяє змінювати алгоритми незалежно від клієнтського коду, який їх використовує.

Уявіть, що ви розробляєте навігатор. Основне завдання — побудувати маршрут з точки А в точку Б. Однак, сам алгоритм побудови маршруту може сильно відрізнятися залежно від обраного способу пересування: автомобілем, громадським транспортом чи пішки.

Наївним рішенням було б реалізувати всю логіку вибору всередині одного методу класу Navigator за допомогою if-else або switch:

```
public class Navigator {
   public void buildRoute(String transportType, Point a, Point b) {
        if (transportType.equals("car")) {
            // ... складна логіка для автомобільного маршруту ...
        } else if (transportType.equals("walk")) {
            // ... складна логіка для пішого маршруту ...
        } // ... і так далі
    }
}
```

Такий підхід є крихким і порушує Принцип Відкритості/Закритості. При додаванні нового способу пересування (наприклад, велосипедом), вам доведеться модифікувати клас Navigator, що робить його все більш складним, громіздким та важким для тестування.

Патерн "Стратегія" пропонує винести кожен з цих алгоритмів в окремий клас, який реалізує спільний інтерфейс. Клас Navigator (який називається Контекстом) більше не містить логіки алгоритмів. Замість цього він має посилання на об'єкт-стратегію і делегує йому роботу з побудови маршруту.

Структура патерну складається з:

- Стратегія (Strategy) загальний інтерфейс для всіх алгоритмів. Він оголошує єдиний метод, який буде викликати Контекст (наприклад, buildRoute()).

- Конкретна Стратегія (Concrete Strategy) класи, що реалізують інтерфейс Стратегії. Кожен такий клас інкапсулює один конкретний алгоритм (CarStrategy, WalkingStrategy).
- Контекст (Context) клас, що використовує стратегію (Navigator). Він зберігає посилання на об'єкт-стратегію і може змінювати його під час виконання. Контекст не знає про деталі реалізації конкретних стратегій.

Приклад реалізації для Navigator:

```
// 1. Інтерфейс Стратегії
interface RouteStrategy {
    void buildRoute(Point a, Point b);
}
// 2. Конкретні Стратегії
class CarStrategy implements RouteStrategy {
    public void buildRoute(Point a, Point b) {
        System.out.println("Побудова автомобільного маршруту...");
    }
}
class WalkingStrategy implements RouteStrategy {
    public void buildRoute(Point a, Point b) {
        System.out.println("Побудова пішого маршруту...");
    }
}
// 3. Контекст
class Navigator {
    private RouteStrategy routeStrategy;
    // Метод для зміни стратегії "на льоту"
    public void setRouteStrategy(RouteStrategy routeStrategy) {
        this.routeStrategy = routeStrategy;
    }
    public void buildRoute(Point a, Point b) {
        // Делегування роботи поточній стратегії
        routeStrategy.buildRoute(a, b);
    }
}
// Клієнтський код
Navigator navigator = new Navigator();
navigator.setRouteStrategy(new CarStrategy()); // Встановлюємо
автомобільну стратегію
navigator.buildRoute(pointA, pointB);

navigator.setRouteStrategy(new WalkingStrategy()); // Легко змінюємо
на пішу
navigator.buildRoute(pointA, pointB);
```

Переваги патерну "Стратегія":

- Алгоритми можна легко змінювати під час виконання програми.
- Для додавання нового алгоритму достатньо створити новий клас-стратегію, не змінюючи код Контексту.
- Складна логіка алгоритмів виноситься з основного класу, роблячи його простішим.
- Патерн дозволяє позбутися громіздких конструкцій if-else або switch.

Патерн "Стратегія" є чудовим рішенням, коли у вас є об'єкт, поведінка якого може змінюватися залежно від ситуації або налаштувань. Він дозволяє інкапсулювати ці варіанти поведінки в окремі, взаємозамінні об'єкти.

Практична робота №7

Тема: Реалізація патерну "Декоратор"

Мета: На практиці реалізувати патерн "Декоратор" для динамічного розширення функціональності об'єктів без зміни їхнього коду.

Завдання

1.Створити інтерфейс Report з методом generate(), що повертає String.

2. Створити базовий клас SimpleReport, що реалізує інтерфейс і повертає простий текстовий звіт (наприклад, "Base report data").

3. Створити абстрактний клас-декоратор ReportDecorator, що також реалізує Report і зберігає посилання на "обгорнутий" об'єкт Report.

4. Створити конкретні декоратори: HeaderDecorator: додає до звіту заголовок (наприклад, "Report Header\n---\n"). FooterDecorator: додає до звіту підвал (наприклад, "\n---\nReport Footer").

5. У main методі створити об'єкт SimpleReport і послідовно "обгорнути" його декораторами HeaderDecorator та FooterDecorator. Вивести результат на консоль.

Лабораторна робота №7

Тема: Реалізація патерну "Будівельник"

Мета: Навчитися застосовувати патерн "Будівельник" для створення складних, конфігурованих об'єктів з великою кількістю параметрів.

Завдання

1. Створити клас HttpClient* з великою кількістю конфігураційних параметрів (url, method, headers, body, timeout тощо).

2. Реалізувати для цього класу внутрішній статичний клас Builder.

3. Зробити конструктор HttpClient приватним.

4. Реалізувати в Builder методи для налаштування кожного параметра та метод build(), що повертає готовий об'єкт HttpClient.

5. Створити кілька клієнтів з різними конфігураціями за допомогою будівельника.

*Примітка: У цій лабораторній роботі HttpClient не виконує реальних HTTP-запитів. Його мета — лише демонстрація застосування патерну Builder для створення складних об’єктів.

Контрольні запитання

1. Що таке патерн проєктування? Назвіть дві основні переваги їх використання у розробці програмного забезпечення.
2. Назвіть три основні категорії патернів проєктування за класифікацією GoF та коротко опишіть, які задачі вирішує кожна категорія.
3. Яку проблему вирішує патерн "Фабричний метод" (Factory Method)? Поясніть, як він делегує створення об'єктів підкласам.
4. У якій ситуації доцільно використовувати патерн "Будівельник" (Builder)? Опишіть його переваги порівняно з використанням великої кількості конструкторів.
5. Поясніть, як патерн "Декоратор" (Decorator) дозволяє динамічно додавати функціональність до об'єктів. Чим цей підхід гнучкіший за успадкування?
6. Яку проблему вирішує патерн "Стратегія" (Strategy)? Як він дозволяє інкапсулювати та робити взаємозамінними різні алгоритми?

Лекція 8. Unit-тестування та керування залежностями (JUnit, Mockito, Maven)

Поняття unit-тестів, піраміда тестування

Написання коду — це лише частина процесу розробки. Не менш важливим етапом є перевірка того, що цей код працює коректно, відповідає вимогам і не містить помилок. Цей процес називається тестуванням.

Unit-тест (модульний тест) — це автоматизований тест, який перевіряє коректність роботи найменшої ізольованої частини програмного коду, що називається "юнітом" або "модулем". В об'єктно-орієнтованому програмуванні юнітом найчастіше є окремий метод або цілий клас.

Ключовою характеристикою unit-тесту є ізоляція. Його мета — перевірити логіку конкретного юніта, від'єднавши його від усіх зовнішніх залежностей, таких як бази даних, мережеві сервіси, файлова система чи навіть інші класи. Для досягнення ізоляції ці залежності замінюються на "заглушки" або "моки" (mocks), про які ми поговоримо пізніше.

Існує багато видів автоматизованих тестів, і важливо правильно збалансувати їх кількість. Піраміда тестування — це модель, що ілюструє оптимальне співвідношення різних рівнів тестів у проєкті.

Unit-тести (Основа піраміди) це найширший шар. Таких тестів має бути найбільше. Вони швидкі, надійні, легко пишуться та підтримуються, оскільки перевіряють логіку в ізоляції.

Інтеграційних тестів (Середній шар) має бути менше. Вони перевіряють взаємодію кількох компонентів системи між собою (наприклад, чи коректно сервісний шар працює з базою даних). Вони повільніші та складніші за unit-тести, оскільки потребують налаштування зовнішнього середовища.

UI / End-to-End тести (Верхівка піраміди) є найвужчим шаром. Таких тестів має бути найменше. Вони перевіряють роботу всього додатку від початку до кінця з точки зору користувача, часто імітуючи дії в графічному інтерфейсі. Вони дуже повільні, крихкі (часто ламаються через незначні зміни в UI) та дорогі в підтримці.

Основна ідея піраміди полягає в тому, щоб більшість помилок "ловити" на найнижчому, найшвидшому та найдешевшому рівні — рівні unit-тестів.

JUnit 5: структура тесту (given-when-then), фікстури (@BeforeEach, @AfterEach)

JUnit — це стандартний фреймворк для написання та запуску автоматизованих unit-тестів у Java. JUnit 5, його сучасна версія, надає розробникам потужний набір анотацій та інструментів для перевірки коректності коду.

В основі JUnit лежить тестовий метод — звичайний Java-метод, позначений анотацією @Test. Ця анотація сигналізує JUnit, що даний метод є тестовим сценарієм, який потрібно виконати.

Для структурування логіки всередині тестового методу широко використовується підхід Given-When-Then (Дано-Коли-Тоді), що є популярною практикою з Behavior-Driven Development (BDD).

На етапі Given (Дано) готується початковий стан. Ви створюєте об'єкти, задаєте вхідні дані та налаштовуєте все необхідне для тестування.

When (Коли) етапі виконується дія, що тестується — зазвичай це виклик одного методу об'єкта.

Then (Тоді) етапі перевіряється результат. Ви стверджуєте (assert), що результат дії відповідає вашим очікуванням.

Для перевірки результатів JUnit надає клас Assertions з набором статичних методів (асершенів), найпоширеніші з яких:

assertEquals(expected, actual) перевіряє, чи дорівнює фактичний результат очікуваному.

assertTrue(condition) / assertFalse(condition) перевіряє, чи є умова істинною/хибною.

assertNotNull(object) перевіряє, що об'єкт не є null.

Приклад тесту для класу Calculator:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class CalculatorTest {
    @Test
    void givenTwoNumbers_whenAdd_thenReturnsSum() {
        // Given (Дано)
        Calculator calculator = new Calculator();
        int a = 5;
        int b = 3;

        // When (Коли)
        int result = calculator.add(a, b);

        // Then (Тоді)
        assertEquals(8, result);
    }
}
```

Часто для виконання кількох тестів у класі потрібен однаковий початковий стан (наприклад, створений екземпляр об'єкта, що тестується). Щоб уникнути дублювання коду, JUnit пропонує використовувати фікстури — методи, позначені спеціальними анотаціями для налаштування та очищення тестового середовища.

Метод, позначений @BeforeEach анотацією, буде виконуватися перед кожним тестовим методом (@Test) у класі. Він ідеально підходить для ініціалізації об'єктів та скидання їх стану до початкового перед кожним тестом, забезпечуючи їх незалежність.

Метод з @AfterEach анотацією виконується після кожного тестового методу. Зазвичай використовується для очищення ресурсів (наприклад, закриття файлів).

@BeforeAll та @AfterAll анотації позначають статичні методи, які виконуються лише один раз на весь тестовий клас — перед першим тестом (@BeforeAll) та після останнього (@AfterAll). Вони корисні для ресурсозатратних операцій, які не потрібно повторювати для кожного тесту (наприклад, підключення до бази даних).

Приклад використання @BeforeEach для рефакторингу CalculatorTest:

```
import org.junit.jupiter.api.BeforeEach;

class CalculatorTest {
    private Calculator calculator;

    @BeforeEach
    void setUp() {
        // Цей код виконається перед кожним тестом
        calculator = new Calculator();
    }

    @Test
    void testAddition() {
        // Given - калькулятор вже створено в setUp()
        // When
        int result = calculator.add(5, 3);
        // Then
        assertEquals(8, result);
    }

    @Test
    void testSubtraction() {
   // Given - новий, чистий екземпляр калькулятора створено знову
        // When
        int result = calculator.subtract(5, 3);
        // Then
        assertEquals(2, result);
    }
}
```

Використання цих анотацій та структури Given-When-Then дозволяє писати чисті, читабельні та легко підтримувані тести, що є основою надійної розробки.

Mockito: створення моків, @Mock, @InjectMocks

Одним з ключових принципів unit-тестування є ізоляція: ми повинні тестувати логіку одного класу, не залежачи від реальної роботи його залежностей (наприклад, бази даних чи мережевого сервісу). Для досягнення цієї ізоляції використовуються мок-об'єкти (mocks).

Mockito — це найпопулярніша бібліотека в Java для створення моків. Мок — це "фальшивий" об'єкт, який імітує поведінку справжнього об'єкта, але знаходиться під повним контролем тесту. Це дозволяє нам:

- Ізолювати клас, що тестується (System Under Test, SUT): Наш UserService буде працювати з "фальшивим" UserRepository, а не зі справжньою базою даних.
- Налаштовувати поведінку залежностей: Ми можемо наказати моку повертати певні дані у відповідь на виклики його методів.
- Перевіряти взаємодію: Ми можемо перевірити, чи викликав наш UserService потрібні методи у UserRepository з правильними параметрами.

Основні анотації Mockito:

@Mock анотація розміщується над полем у тестовому класі та наказує Mockito створити мок-об'єкт для цього поля.

@InjectMocks анотація розміщується над полем, що представляє клас, який ми тестуємо (SUT). Mockito автоматично спробує "впровадити" (inject) всі поля, позначені @Mock, у відповідні поля об'єкта SUT (зазвичай через конструктор або сеттери).

Для активації цих анотацій тестовий клас потрібно позначити анотацією @ExtendWith(MockitoExtension.class).

Приклад тестування UserService:

```
// Інтерфейс залежності
public interface UserRepository {
    User findById(String id);
}

// Клас, який ми тестуємо (System Under Test)
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public String getUserName(String id) {
        User user = userRepository.findById(id);
        return user.getName();
    }
}

// Тестовий клас
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.when;
```

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {
    @Mock
    private UserRepository userRepositoryMock; // 1. Створюємо мок
    @InjectMocks
private UserService userService; // 2. Створюємо SUT і впроваджуємо
в нього мок
    @Test
    void testGetUserName() {
        // Given (Дано) - Налаштовуємо поведінку мока
        User testUser = new User("John Doe");

when(userRepositoryMock.findById("1")).thenReturn(testUser);
        // When (Коли)
        String userName = userService.getUserName("1");
        // Then (Тоді)
        assertEquals("John Doe", userName);
    }
}
```

Системи збірки Maven для управління залежностями та життєвим циклом проєкту

Maven — це інструмент для автоматизації збірки проєктів, який є стандартом у світі Java. Він вирішує два основні завдання: описує, з чого складається проєкт, і керує процесом його збірки.

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.9.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>4.8.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Рисунок 8.1 - Компоненти JDK

Керування залежностями це найважливіша функція Maven. Замість того, щоб вручну завантажувати .jar-файли бібліотек (як JUnit чи Mockito), шукати їхні

залежності і додавати в проєкт, ви просто оголошуєте їх у центральному конфігураційному файлі проєкту — pom.xml.

Maven автоматично завантажить ці бібліотеки та всі їхні транзитивні залежності (бібліотеки, від яких залежать вони самі) з центрального репозиторію (Maven Central) і підключить до вашого проєкту.

Maven визначає стандартний життєвий цикл, що складається з послідовності фаз. Коли ви викликаєте певну фазу, Maven послідовно виконує всі попередні. Основні фази:

validate: перевірка коректності проєкту.

compile: компіляція вихідного коду (src/main/java).

test: запуск тестів (src/test/java). Ця фаза виконується командою mvn test.

package: пакування скомпільованого коду у фінальний артефакт (зазвичай .jar або .war файл).

install: встановлення артефакту у ваш локальний Maven-репозиторій.

deploy: копіювання артефакту у віддалений репозиторій.

Таким чином, Maven стандартизує структуру проєкту, спрощує управління залежностями та автоматизує весь процес збірки, компіляції, тестування та пакування, роблячи розробку більш надійною та передбачуваною.

Практична робота №8

Тема: JUnit та структура тестів

Мета: Отримати базові навички написання Unit-тестів за допомогою JUnit 5 та організовувати їх за структурою given-when-then.

Завдання

1. Створити Maven-проєкт (типу maven-archetype-quickstart).

2. Додати залежність junit-jupiter у pom.xml.

3. Створити простий клас Calculator з методами add, subtract, multiply, divide.

4. Написати тести для цього класу з використанням JUnit 5. Використати структуру given-when-then у назвах методів. Продемонструвати використання @BeforeEach для підготовки об’єкта.

Лабораторна робота №8

Тема: Unit-тестування з Mockito та Maven

Мета: Навчитися використовувати Mockito для тестування класів в ізоляції від їхніх залежностей та запускати тести за допомогою Maven.

Завдання

1. Додати залежність mockito-core у pom.xml.

2. Створити інтерфейс UserRepository з методом findById(String id).

3. Створити клас UserService, який у конструкторі приймає UserRepository і має метод getUserName(String id).

4. Написати unit-тести для UserService, замокавши UserRepository за допомогою Mockito. Використати анотації @Mock, @InjectMocks. Продемонструвати поведінкове тестування: коли викликається getUserName, метод findById має бути викликаний рівно 1 раз.

5. Запустити тести через Maven (mvn test)".

Контрольні запитання

1. Що таке unit-тест і яка його головна мета? Опишіть рівні піраміди тестування та поясніть, чому unit-тестів має бути найбільше.
2. Яка анотація використовується в JUnit 5 для позначення тестового методу? Опишіть структуру тесту за принципом "Given-When-Then".
3. Для чого призначені анотації @BeforeEach та @BeforeAll у JUnit 5 та яка між ними різниця?
4. Чому при написанні unit-тестів важливо ізолювати клас, що тестується, від його залежностей? Яку роль у цьому процесі відіграє бібліотека Mockito?
5. Поясніть призначення анотацій @Mock та @InjectMocks у Mockito. Як налаштувати поведінку мок-об'єкта для повернення певного значення?
6. Яке основне завдання вирішує система збірки Maven? Опишіть, як відбувається керування залежностями за допомогою файлу pom.xml.
7. Що таке життєвий цикл збірки в Maven? Які основні фази він включає, і що робить команда mvn test?

Лекція 9. Асинхронна взаємодія

Введення в асинхронну комунікацію. Синхронна vs. Асинхронна комунікація

У попередніх лекціях ми розглядали, як організувати паралельну роботу всередині одного додатку за допомогою потоків. Тепер ми перейдемо на вищий рівень і розглянемо, як різні додатки або їх окремі компоненти (наприклад, мікросервіси) взаємодіють між собою. Ця взаємодія, або комунікація, може бути організована двома фундаментально різними способами: синхронно та асинхронно.

Синхронна комунікація працює за принципом "запит-відповідь" і є аналогічною до телефонного дзвінка. Клієнт відправляє запит до сервісу і блокується, очікуючи на відповідь. Він не може продовжувати свою роботу, доки не отримає відповідь від сервера, чи то успішну, чи то повідомлення про помилку.

Класичний приклад — це виклик REST API через HTTP. Коли ваш браузер робить GET-запит на веб-сервер, він чекає, доки сервер не поверне HTML-сторінку. Клієнт та сервер тісно пов'язані в часі. Якщо сервер недоступний або повільно відповідає, клієнт "зависає" або отримує помилку. Збій у роботі сервера безпосередньо впливає на клієнта.

Асинхронна комунікація працює за принципом "відправив і забув" і є аналогічною до відправки електронного листа або SMS. Відправник (producer) надсилає повідомлення і не чекає на відповідь. Він одразу продовжує виконувати свою роботу. Отримувач (consumer) обробить це повідомлення пізніше, коли буде готовий.

Приклад: Відправка email-підтвердження після реєстрації користувача. Основний сервіс реєстрації не повинен чекати, доки email-сервіс реально відправить листа. Він просто передає йому завдання і завершує свою роботу.

Відправник та отримувач не залежать один від одного. Якщо отримувач тимчасово недоступний, повідомлення не буде втрачено; воно буде оброблено пізніше. Має високу відмовостійкість та масштабованість. Навантаження на сервіси-отримувачі можна легко масштабувати, додаючи нових отримувачів для обробки черги повідомлень.

Але при цьому складніша архітектура, тому зазвичай вимагає наявності проміжної ланки — брокера повідомлень (message broker), який відповідає за зберігання та доставку повідомлень.

Синхронний підхід підходить для швидких операцій, де клієнту потрібна негайна відповідь (наприклад, запит на отримання даних користувача).

Асинхронний підхід є ідеальним для довготривалих операцій, фонових завдань, сповіщень та побудови гнучких, відмовостійких мікросервісних архітектур.

Патерн "Видавець-Підписник" (Publisher-Subscriber)

Патерн "Видавець-Підписник" (також відомий як Pub-Sub або "Спостерігач" або Observer) — це поведінковий патерн проєктування, що створює механізм підписки, який дозволяє одним об'єктам ("підписникам") автоматично отримувати сповіщення про зміни стану іншого об'єкта ("видавця").

Основна мета патерну — організувати слабкий зв'язок (loose coupling) між компонентами системи. Об'єкт, що генерує події (видавець), не має жодної інформації про те, хто і як на них реагує. Він лише сповіщає всіх зацікавлених, що подія відбулася. Це дозволяє динамічно додавати та видаляти "слухачів" подій, не вносячи змін у код видавця, що повністю відповідає Принципу Відкритості/Закритості.

Структура патерну складається з наступних учасників:

1. Видавець (Publisher):
    a. Є джерелом подій (наприклад, EventManager).
    b. Зберігає список посилань на своїх підписників.
    c. Надає методи для керування підпискою (subscribe() та unsubscribe()).
    d. Має метод notify(), який при виникненні події послідовно викликає метод оновлення у кожного підписника зі свого списку.
2. Підписник (Subscriber):
    a. Об'єкт, зацікавлений у подіях видавця (наприклад, EmailNotificationListener, LogListener).
    b. Реалізує спільний для всіх підписників інтерфейс.
    c. Цей інтерфейс визначає єдиний метод (наприклад, update()), який видавець буде викликати для надсилання сповіщень.

Клієнтський код створює об'єкти-підписники та реєструє їх у видавця за допомогою методу subscribe(). Коли у видавця відбувається важлива подія (наприклад, створюється нове замовлення), він викликає свій метод notify(). Метод notify() проходить по списку підписників і для кожного з них викликає метод update(), передаючи необхідну інформацію про подію. Кожен підписник самостійно реагує на отримане сповіщення відповідно до своєї логіки.

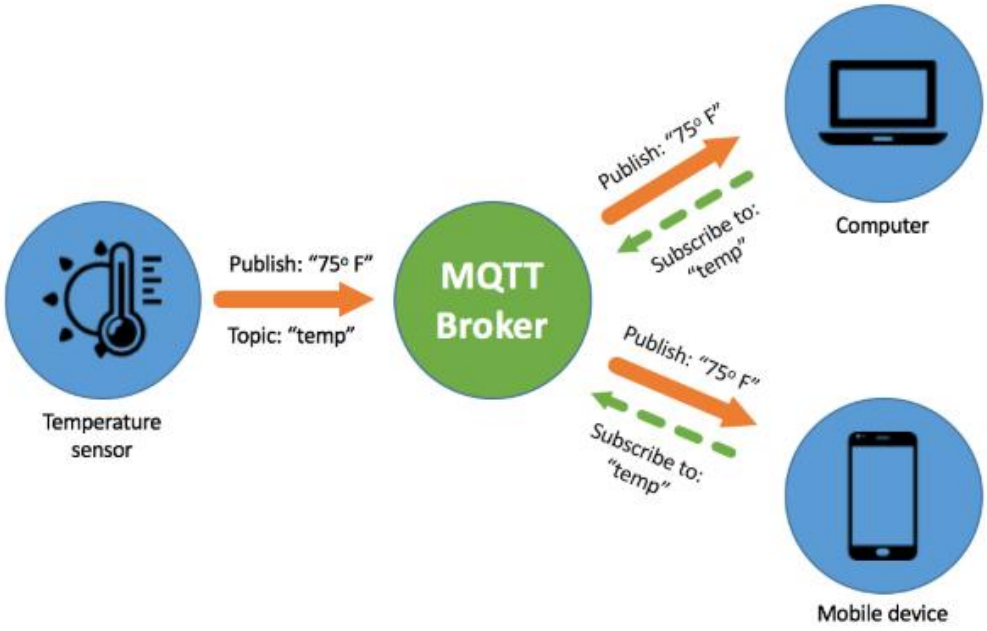

Рисунок 8.1 - Протокол MQTT для розумного дому є прикладом Pub-Sub патерну

У більш складних розподілених системах між видавцем та підписниками часто існує проміжна ланка — канал або тема (Topic). Видавець відправляє повідомлення у визначений канал, а підписники "слухають" цей канал. Такий

підхід ще більше роз'єднує компоненти, оскільки вони навіть не знають про існування один одного, а лише про канал, через який відбувається комунікація.

Видавець нічого не знає про конкретні класи підписників, лише про їхній спільний інтерфейс. Підписників можна додавати та видаляти під час виконання програми. Систему легко розширювати, додаючи нові класи підписників, не змінюючи при цьому код видавця.

Патерн "Видавець-Підписник" є фундаментальним для побудови подійно-орієнтованих (event-driven) архітектур і лежить в основі багатьох сучасних асинхронних систем, зокрема тих, що використовують брокери повідомлень.

Брокери повідомлень (Message Brokers): Огляд RabbitMQ та Apache Kafka

У простих реалізаціях патерну "Видавець-Підписник", видавець безпосередньо керує списком своїх підписників. Однак у складних, розподілених системах (наприклад, мікросервісних) такий підхід є неефективним. Для організації надійної асинхронної комунікації між сервісами використовуються спеціалізовані проміжні програми — брокери повідомлень.

Брокер повідомлень — це спеціалізоване програмне забезпечення, яке виступає посередником між відправниками (producers/publishers) та отримувачами (consumers/subscribers) повідомлень. Відправник надсилає повідомлення брокеру, а брокер гарантує, що воно буде збережено та доставлено одному чи кільком отримувачам. Це створює надійну, відмовостійку та слабкозв'язану систему.

Розглянемо два найпопулярніші брокери повідомлень у світі Java: RabbitMQ та Apache Kafka.

RabbitMQ — це один з найвідоміших брокерів повідомлень з відкритим вихідним кодом. Він реалізує протокол AMQP (Advanced Message Queuing Protocol) і є чудовим вибором для складних сценаріїв маршрутизації повідомлень та організації черг завдань.

Основні концепції:

- Producer (Виробник) додаток, що відправляє повідомлення.
- Consumer (Споживач) додаток, що отримує повідомлення.
- Exchange (Обмінник) отримує повідомлення від виробників і маршрутизує їх в одну або кілька черг. Тип обмінника (direct, topic, fanout) визначає логіку маршрутизації.
- Queue (Черга) буфер, що зберігає повідомлення до моменту їх отримання споживачем.
- Binding (Зв'язок) правило, яке пов'язує обмінник з чергою.

Приклад коду (з використанням бібліотеки amqp-client). Виробник (Producer), що відправляє повідомлення:

```
// Потрібна залежність: com.rabbitmq:amqp-client
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
try (Connection connection = factory.newConnection();
     Channel channel = connection.createChannel()) {

    String queueName = "hello-queue";
    // Оголошуємо чергу (якщо її немає, вона буде створена)
    channel.queueDeclare(queueName, false, false, false, null);
    String message = "Hello, RabbitMQ!";

    // Відправляємо повідомлення напряму в чергу
    channel.basicPublish("", queueName, null, message.getBytes());
    System.out.println(" [x] Sent '" + message + "'");
}
```

Споживач (Consumer), що отримує повідомлення:

```
// ... налаштування з'єднання аналогічне ...
Channel channel = connection.createChannel();
String queueName = "hello-queue";
channel.queueDeclare(queueName, false, false, false, null);

DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] Received '" + message + "'");
};

// Починаємо слухати чергу
channel.basicConsume(queueName, true, deliverCallback, consumerTag -
> { });
```

Apache Kafka — we розподілена платформа для потокової передачі подій (event streaming). Вона була розроблена для роботи з величезними потоками даних у режимі реального часу, забезпечуючи високу пропускну здатність та відмовостійкість, тобто це більше, ніж просто брокер повідомлень.

В Kafka Topic (Тема) це категорія або канал, куди публікуються повідомлення (події). На відміну від черги RabbitMQ, повідомлення в Kafka не видаляються після прочитання, а зберігаються протягом певного часу.

Partition (Розділ) кожна тема розділена на один або більше розділів. Це дозволяє розпаралелювати обробку: кілька споживачів з однієї групи можуть одночасно читати дані з різних розділів однієї теми.

Offset (Зсув) - унікальний ідентифікатор кожного повідомлення всередині розділу. Споживачі відстежують свій прогрес, зберігаючи offset останнього прочитаного повідомлення.

В основі Kafka лежить ідея незмінного, впорядкованого логу подій Distributed Log. Це дозволяє "перемотувати" та перечитувати повідомлення, що неможливо в класичних брокерах.

Приклад коду (з використанням бібліотеки kafka-clients). Виробник (Producer):

```
// Потрібна залежність: org.apache.kafka:kafka-clients
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
Producer<String, String> producer = new KafkaProducer<>(props);
String topic = "my-topic";
String key = "1";
String value = "Hello, Kafka!";
producer.send(new ProducerRecord<>(topic, key, value));
producer.close();
```

Споживач (Consumer):

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "my-group");
props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("my-topic"));
while (true) {
  ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset = %d, key = %s, value = %s%n",
record.offset(), record.key(), record.value());
    }
}
```

Отже, RabbitMQ є ідеальним вибором для традиційних завдань, що вимагають складних правил маршрутизації, гарантованої доставки та обробки завдань у черзі (task queuing).

В свою чергу Apache Kafka є стандартом для побудови систем, що працюють з великими потоками даних у реальному часі (event streaming), лог-агрегації, аналітики та побудови подійно-орієнтованих мікросервісів.

Практична робота №9

Тема: Імплементація патерну "Видавець-Підписник"

Мета: Реалізувати патерн "Видавець-Підписник" на чистому Java для розуміння механізму сповіщення об'єктів про події.

Завдання

1. Створити інтерфейс EventListener з одним методом update(String message).

2. Створити кілька конкретних реалізацій-підписників:

EmailNotificationListener: виводить у консоль "Sending email with message: [message]".

LogListener: виводить у консоль "Logging message: [message]".

3. Створити клас EventManager (видавець), який буде керувати підписниками. Клас повинен мати методи subscribe(EventListener listener) та unsubscribe(EventListener listener). Клас повинен мати метод notify(String message), який проходить по всіх підписниках і викликає їх метод update.

4. У main методі: Створити екземпляр EventManager. Створити та підписати кілька слухачів. Викликати метод notify кілька разів, щоб продемонструвати, як усі підписники отримують сповіщення.

Лабораторна робота №9

Тема: Взаємодія сервісів через RabbitMQ

Мета: Навчитися налаштовувати асинхронну взаємодію між двома окремими сервісами за допомогою брокера повідомлень RabbitMQ.

Завдання

1. Налаштування середовища: Запустити офіційний образ RabbitMQ в Docker-контейнері.

2. Створити проєкт "Видавець" (OrderService): Додати залежність для RabbitMQ клієнта. Написати код, який підключається до RabbitMQ, створює повідомлення (напр., JSON з інформацією про замовлення) і відправляє його в exchange з назвою orders.

3. Створити проєкт "Підписник" (NotificationService): Також додати залежність RabbitMQ клієнта. Написати код, який підключається до RabbitMQ. Створює queue (чергу), "прив'язує" її до exchange orders. Запускає слухача, який чекає на повідомлення з черги і, при отриманні, виводить його вміст у консоль.

4. Демонстрація: Спочатку запустити "Підписника". Потім запустити "Видавця". Переконатись, що повідомлення, відправлене видавцем, було отримане та оброблене підписником.

Контрольні запитання

1. Поясніть ключову різницю між синхронною та асинхронною комунікацією між сервісами. Який підхід забезпечує кращу відмовостійкість і чому?
2. Яку основну проблему вирішує патерн "Видавець-Підписник"? Назвіть основних учасників цього патерну та опишіть їхні ролі.
3. Яку роль виконує брокер повідомлень в асинхронній архітектурі? Чому його використання сприяє слабкій зв'язаності (loose coupling) компонентів?
4. Назвіть та коротко опишіть основні компоненти архітектури RabbitMQ (Exchange, Queue, Binding).
5. Що таке "Topic" та "Partition" в Apache Kafka? Чим підхід Kafka до зберігання повідомлень (як до розподіленого логу) відрізняється від класичної черги в RabbitMQ?
6. Наведіть приклад задачі, для якої краще підійшов би RabbitMQ, і приклад задачі, де перевагу слід віддати Apache Kafka.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Bloch J. Effective Java / Joshua Bloch. – [S. l.] : Pearson Education, Limited, 2018. ISBN-10: 0-13-468599-7
2. Кормен, Т. Х. Вступ до алгоритмів / Томас Г. Кормен [та ін.] ; пер. з англ. – К. : К.І.С., 2019. – 1288 с. – ISBN 978-617-684-237-7.
3. Мартін, Р. К. Чистий код : Створення та рефакторинг програмного коду / Роберт К. Мартін ; пер. з англ. – Харків : Фабула, 2021. – 448 с. – ISBN 978-617-09-5192-1.
4. Тарнавський, Ю. А. Java-програмування: комп'ютерний практикум [Електронний ресурс] : навч. посіб. / КПІ ім. Ігоря Сікорського ; уклад.: Ю. А. Тарнавський. – Київ, 2021. – 95 с.
5. Фрімен, Е. Патерни проєктування Head First / Ерік Фрімен, Елізабет Робсон, Берт Бейтс, Кеті Сьєрра ; пер. з англ. – Харків : Фабула, 2021. – 672 с. – ISBN 978-617-09-6853-0.
6. The Java™ Tutorials. [Електронний ресурс] // Oracle and/or its affiliates. – Режим доступу: https://docs.oracle.com/javase/tutorial/
7. Baeldung. [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/
8. Martin Fowler's Blog. [Електронний ресурс]. – Режим доступу: https://martinfowler.com/
9. The Destination for Java Developers. Dev.java. URL: https://dev.java/
10. Bloch, J. Effective Java. 3rd ed. — Addison-Wesley, 2018. — 416 p.
11. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A. The Java Language Specification. Java SE 21 Edition. — Oracle, 2023. — 1024 p.
12. Horstmann, C. Core Java. Volume I–II: Fundamentals and Advanced Features. 12th ed. — Pearson, 2023. — 1500 p.
13. Sierra, K., Bates, B. Head First Java. 3rd ed. — O’Reilly Media, 2022. — 720 p.
14. Freeman, E., Freeman, E. Head First Design Patterns. 2nd ed. — O’Reilly Media, 2021. — 694 p.
15. Goetz, B. Java Concurrency in Practice. — Addison-Wesley, 2006. — 424 p.
16. Fowler, M. Refactoring: Improving the Design of Existing Code. 2nd ed. — Addison-Wesley, 2018. — 448 p.
17. Syer, D., Long, J. Spring Boot Reference Guide. — Pivotal Software, 2023. — [Електронний ресурс].
18. Kousen, K. Modern Java Recipes: Simple Solutions to Difficult Problems in Java 8 and 9. — O’Reilly Media, 2017. — 322 p.

19. Lippman, S. Modern Java in Action: Lambdas, Streams, Functional and Reactive Programming. — Manning Publications, 2019. — 600 p.
20. Beck, K. Test Driven Development: By Example. — Addison-Wesley, 2022. — 240 p.
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley, 1994. — 395 p.
22. Martin, R.C. Clean Code: A Handbook of Agile Software Craftsmanship. — Prentice Hall, 2008. — 464 p.
23. Martin, R.C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. — Prentice Hall, 2017. — 432 p.
24. Sonmez, J. The Complete Software Developer’s Career Guide. — Simple Programmer, 2017. — 796 p.
25. Bass, L., Clements, P., Kazman, R. Software Architecture in Practice. 4th ed. — Addison-Wesley, 2021. — 704 p.
26. Richardson, C. Microservices Patterns: With Examples in Java. — Manning Publications, 2018. — 520 p.
27. Walls, C. Spring in Action. 6th ed. — Manning Publications, 2022. — 720 p.
28. Oracle Corporation. Java Platform, Standard Edition Documentation. — [Електронний ресурс]. URL: https://docs.oracle.com/javase/