Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

АНАЛІЗ ПОКРАЩЕНЬ В JAVA 19 У ПОРІВНЯННІ З JAVA 17

Текстова частина до курсової роботи

за спеціальністю „Комп'ютерні науки" 6.050101

Керівник курсової роботи

ас. Яремко С. А.

_____

(підпис)

"____" _____ 2023 р.

Виконав студент

Новак В. І.

"____" _____ 2023 р.

Київ 2023

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
проф., д.ф.-м.н.
_____М. М. Глибовець
(підпис)
„____”_____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Новаку В. І. факультету інформатики 3-го курсу
ТЕМА Аналіз покращень у Java 19 у порівнянні з Java 17

Вихідні дані:

- Текстовий опис оновлень в Java 19, порівняних з Java 17
- Програма з реалізацією оновлень в Java 19

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ
1 Огляд версій Java 17 та Java 19
2 Порівняння покращень версії Java 19 з Java 17
Висновки
Список використаної літератури та електронних ресурсів

Дата видачі „___” _____ 2023 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Аналіз покращень у Java 19 у порівнянні з Java 17

Календарний план виконання роботи:

№ п/п	Назва етапу виконання курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання.	27.12.2022	
2.	Огляд технічної літератури за темою роботи.	03.02.2023	
3.	Аналіз оновлень в Java 17.	04.03.2023	
4.	Аналіз оновлень в Java 19.	26.03.2023	
5.	Написання текстової частини роботи про оновлення в Java 17 та Java 19.	15.04.2023	
6.	Створення програми з реалізацією оновлень в Java 19 та Java 17.	03.05.2023	
7.	Написання текстової частини роботи про порівняння оновлень в Java 17 та Java 19.	10.05.2023	
8.	Створення презентації та написання доповіді.	14.05.2023	
9.	Захист проекту.	23.05.2023	

ЗМІСТ

Анотація

Дана робота розкриває детальний аналіз оновлень в версіях мови програмування Java 17 та Java 19, включаючи їх визначення, приклади коду та застосування в програмах або в різних сферах розробки програмного забезпечення. Особливу увагу було надано таким оновленням в Java 19, як віртуальні потоки, структурована паралельність та використання паттерн-матчингу в різних випадках.

У наступному розділі ці оновлення було реалізовано в примітивній програмі, яка імітує роботу онлайн-магазину. До їх реалізацій запропоновано можливі варіанти їх реалізацій в версії Java 17. Також ці реалізації було порівняно між собою, щоби визначити, як були покращені певні особливості мови Java.

Для аналізу оновлень в версіях Java було використано як джерело офіційні примітки з сайту компанії Oracle та інформацію з JEP 0, який зберігає інформацію про всі покращення в Java.

ВСТУП

Вже десятки років об’єктно-орієнтовна мова програмування Java користується великим попитом для розробки комерційного програмного забезпечення на світовому ринку. Щороку вона посідає найвищі місця в рейтингах найпопулярніших мов програмування. Java – це універсальна мова, яка часто використовується для розробки ігор, роботи з великими даними, хмарних обчислень, тощо.

Необхідно зазначити, що можливості мови програмування Java постійно розширюються завдяки оновленням, що виходять в релізах її нових версій. Керуючись змінами тенденцій в програмуванні, розробники з Oracle та інших великих корпорацій щоразу реалізовують новий функціонал, завдяки чому Java рухається нога в ногу з новими мовами. Якщо раніше нові версії Java виходили з періодичністю 2-5 років, то, починаючи з вересня 2017, коли вийшла версія Java 9, нові версії даної мови тепер виходять раз на півроку. Завдяки цьому мова все швидше і швидше оновлюється та оптимізується.

Актуальність даної роботи є значною, оскільки в ній розглядаються версії мови Java, які стали доступними ще рік-два тому. Незважаючи на те, що Java 17 була випущена нещодавно відносно Java 19, її наступниця містить ряд важливих змін, які зробили мову Java потужнішою та ефективнішою для розробки сучасного програмного забезпечення.

Об’єктом дослідження в курсовій роботі є оновлення в версіях Java 17 та Java 19. Мета дослідження полягає в порівнянні оновлень в цих версіях, та в аналізі позитивних змін в версії Java 19 порівняно з саме Java 17. Як завдання дослідження, можна виділити такі пункти:

- Зібрати та проаналізувати інформацію про нововведення та покращення, які були впроваджені у Java 19.

- Виконати порівняльний аналіз між Java 19 та Java 17, визначити ключові різниці та переваги нової версії.
- Провести практичне дослідження, щоб підтвердити переваги нових функціональностей та оптимізацій в Java 19.
- Зробити висновки щодо використання Java 19 у порівнянні з Java 17 та визначити можливості й рекомендації для подальшого вдосконалення програмного забезпечення.

Основними методами дослідження в курсовій роботі можна назвати аналіз та синтез інформації з різних онлайн-ресурсів, а також моделювання, застосоване при роботі з практичним додатком та імплементації оновлень. Джерела дослідження, що застосовувались при роботі – офіційні документації та оглядові статті з Інтеренету про оновлення в різних версіях Java.

Наукова новизна даної роботи полягає в конкретному порівнянні певних версій, оскільки його результат дає осягнути значимість та користь покращень в новішій версії. Досі більшість Java-розробників використовують старіші LTS-версії мови, і перехід на новіші версії Java дозволив би значно оптимізувати їх роботу. Результати цього дослідження можуть бути корисними для розробників програмного забезпечення, та сприяти оптимізації мови програмування Java.

РОЗДІЛ 1. ОГЛЯД ВЕРСІЙ JAVA 17 ТА JAVA 19

1.1 Оновлення в версії Java 17

Версія Java 17 вийшла в світ 14 вересня 2021 року як LTS-версія (long-term support, версія, що буде оновлюватись більш тривалий час, ніж інші). Згідно з дослідженням сайту datastorageasean.com, який публікує новини в розвитку АСЕАН в Big Data, проведеним на початку 2023 року, ця версія починає наздоганяти Java 11 за популярністю. Понад 9% застосунків зараз використовують Java 17 у виробництві (порівняно з майже 2% у 2022 році), що становить ріст її популярності в 430%. [4][5]

Всього в реліз Java 17 потрапило 14 JEP-ів (JDK Enhancement Proposal, документ, який пропонує вдосконалення певної основної технології Java). Розглянемо найголовніші з них детальніше нижче. [3]

1.1.1 Restore Always-Strict Floating-Point Semantics (JEP 306)

У попередніх версіях Java існували ситуації, коли певні операції з плаваючою комою були визначені як strictfp, точніше, не відповідали стандарту IEEE 754, який є галузевим стандартом для арифметики з плаваючою комою.

Це могло призвести до невідповідностей у способі виконання операцій з плаваючою комою, що призводило до малопомітних помилок у додатках, які покладалися на точні обчислення з плаваючою комою. JEP 306 пропонує привести всі операції з плаваючою комою в Java у відповідність до стандарту IEEE 754, тобто визначити їх як strict, незалежно від рівня оптимізації або середовища виконання.

Завдяки відновленню strict семантики операцій з плаваючою комою, Java-додатки можуть бути більш надійними та коректними, що допоможе запобігти

помилкам, які важко діагностувати і виправити. Як вказали розробники, це потрібно для того, щоби відновити оригінальну семантику операцій з плаваючою комою у мові та віртуальній машині, яка відповідала семантиці до введення режимів строгої та стандартної операцій з плаваючою комою у Java SE 1.2. [1]

1.1.2 Enhanced Pseudo-Random Number Generators (JEP 356)

У Java 17 було оновлено клас java.util.Random, в якому були реалізовані нові інтерфейси та класи реалізацій генератора псевдовипадкових чисел. Насправді генератор псевдовипадкових чисел (PRNG) - це алгоритм, який використовує математичні формули для створення послідовностей випадкових чисел. PRNG генерує послідовність чисел, що наближається до властивостей випадкових чисел, тому вони і є псевдовипадковими. [6]

За документацією, java.util.Random містить реалізацію інтерфейсу RandomGenerator, який містить такі методи, як ints, doubles, longs, nextInt, nextDouble, тощо, які повертають потік випадкових чисельних значень. Також було створено ще чотири інтерфейси, що є дочірніми від RandomGenerator:

- SplittableRandomGenerator (розширює RandomGenerator, а також надає методи split та splits для створення нових екземплярів SplittableRandom, які, як правило, видаватимуть статистично незалежні результати)
- JumpableRandomGenerator (розширює RandomGenerator, а також надає методи jump та jumps для того, щоб змінити стан псевдогенератора так, щоб він «стрибнув вперед» на велику фіксовану відстань (зазвичай на 2^{64} або більше) протягом циклу його станів)
- LeapableRandomGenerator (розширює RandomGenerator, а також надає методи leap та leaps для того, щоб змінити стан псевдогенератора так, щоб він «стрибнув вперед» на велику фіксовану відстань (зазвичай на 2^{96} або більше) протягом циклу його станів)

- ArbitrarilyJumpableRandomGenerator (розширює LeapableRandomGenerator, а також надає додаткові варіації для методів jump та jumps)

Також розробники переробили класи Random, ThreadLocalRandom та SplittableRandom таким чином, щоб зробити код в вищевказаних інтерфейсах придатним для повторного використання іншими алгоритмами.

В цілому, нові інтерфейси для генерації псевдовипадкових чисел полегшили генерування випадкових чисел у Java та налаштування реалізації генератора випадкових чисел. Це також забезпечить кращу продуктивність і безпеку, що особливо важливо для додатків, які потребують високоякісної генерації випадкових чисел, таких як криптографічні програми. [1]

1.1.3 Pattern Matching for switch (Preview) (JEP 406)

Дане оновлення дозволяє використовувати pattern matching в операторах switch в мові програмування Java.

До появи цієї можливості оператори switch в Java можна було використовувати лише з примітивними типами, перелічуваними типами або класом String. Однак, pattern matching для switch дозволяє розробникам зіставляти більш складні структури даних, такі як об'єкти, масиви або колекції. Ось приклад такого switch з використанням pattern matching:

```
static void checkFigure (Object o) {
    switch (o) {
        case Triangle tr -> System.out.println("It's a triangle");
        case Circle cir -> System.out.println("It's a circle");
        case Square sq -> System.out.println("It's a square");
        case null -> System.out.println("It's an undefined object");
        default -> System.out.println("It's an unknown figure");
    }

}
```

Рис. 1.1.3.1. Скріншот прикладу оператора switch з використанням паттерн-матчингу

Як бачимо, у даному коді також використовуються pattern variables (вони з'явились ще у версії Java 16). Вони дозволяють позбутись явних приведень класів до інших типів. Якби ми використовували приведення типів, то код був би доволі громіздкий. Розглянемо це на прикладі коду з використанням оператора instanceof (де було вперше реалізовано pattern matching, ще в версії Java 16) [8]:

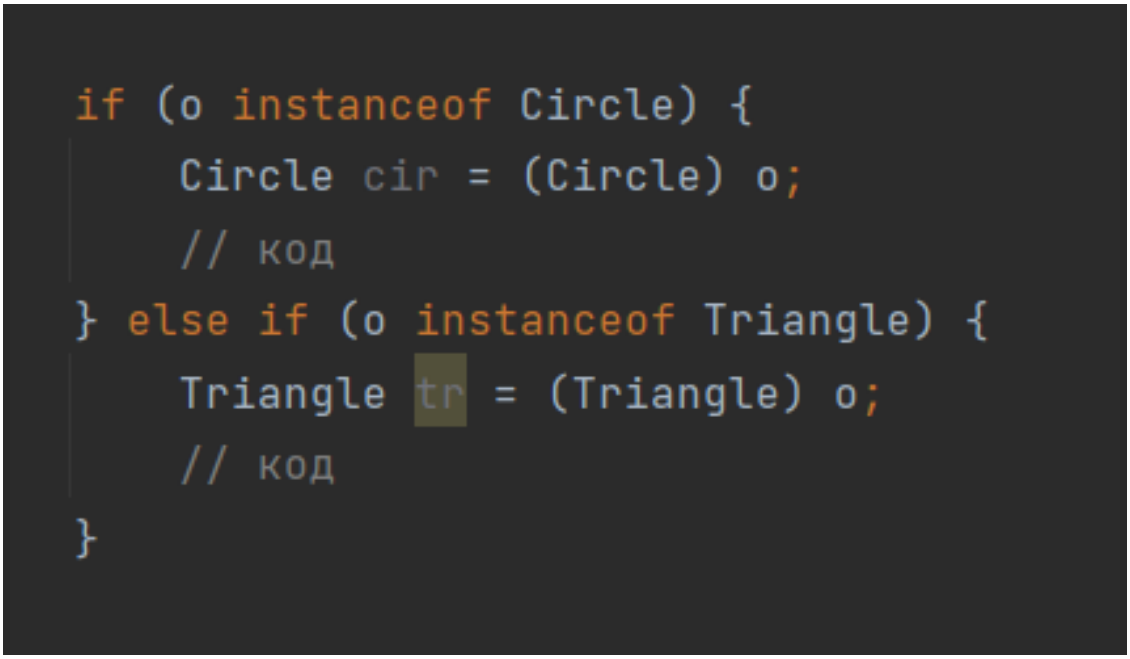

Рис. 1.1.3.2. Скріншот прикладу оператора instanceof без використання паттерн-матчингу

Постійне використання явних приведень в коді використовує більше пам’яті, та робить код менш читабельним для програміста. Натомість можна використовувати pattern variables, відредагуємо код вище таким чином:

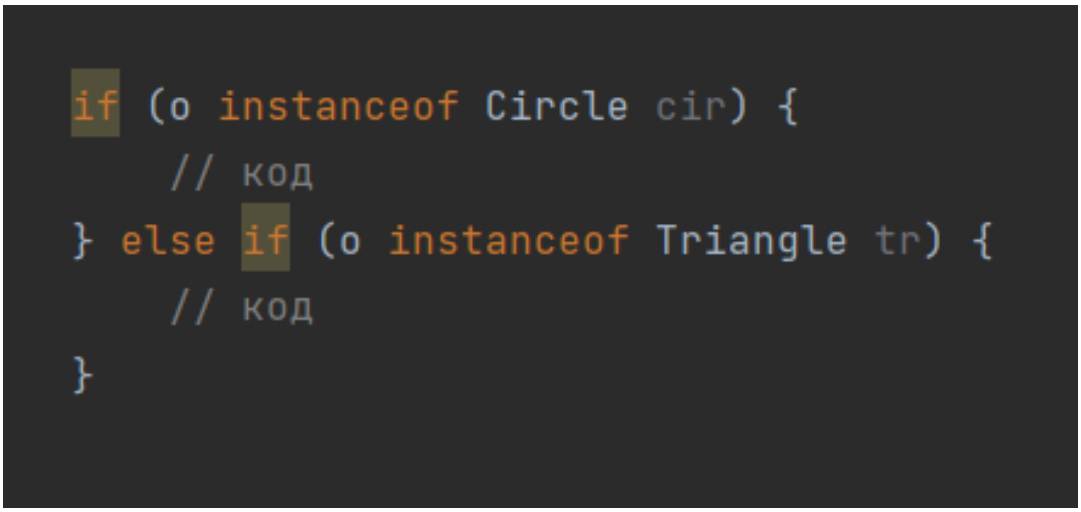

Рис. 1.1.3.3. Скріншот прикладу оператора instanceof з використанням паттерн-матчингу

У Java 17 також було введено можливість створювати case null в операторі switch, щоби запобігати неконтрольованих NullPointerException під час виконання коду. Також у даній версії було додано можливість поєднувати case null з іншими випадками оператора switch, як-от: *case null, Integer num -> action()*. До того ж, оператор if тепер можна прописувати не в блоці коду певного case, а як guarded pattern до нього, наприклад: *case Integer num && (num > 1) -> action()*.

Важливо відмітити, що в цій версії pattern matching для оператора switch відмічений як preview, тобто як такий, що вже повністю реалізований, але його реалізації будуть змінюватись в майбутніх версіях згідно з фідбеком девелоперів (для використання preview features потрібно дозволити в налаштуваннях їх використання (--enable-preview)), але вже в певній версії це оновлення не буде визначене як preview. [1] [2]

1.1.4 Sealed Classes (JEP 409)

Запечатаний (sealed) клас в Java – це клас, в якому можна визначити список дозволених нащадків. Окрім класів, інтерфейси також можуть бути запечатаними, і, відповідно, можуть мати список класів та інтерфейсів, які можуть його реалізовувати. Sealed класи та інтерфейси були реалізовані ще в версії Java 15 як preview, проте вже в Java 17 вони були доопрацьовані.

Щоби визначити клас/інтерфейс як запечатаний, необхідно визначити в коді його як sealed. Потім після його оголошення, щоби зазначити список класів та інтерфейсів, які можуть його унаслідовувати/реалізовувати, після слова permits потрібно вказати всі ці класи та інтерфейси через кому. Ось приклад оголошення sealed class в Java:

```
public sealed class Mammal
        permits Squirrel, Hare, Fox, Monkey, Dog  {
    // код
}
```

Рис. 1.1.4.1. Скріншот прикладу оголошення запечатаного класу

```
1 usage
public final class Fox extends Mammal {
    // код
}
```

Рис. 1.1.4.2. Скріншот прикладу оголошення класу, дочірнього від запечатаного

Як правило, класи, які є дочірніми від запечатаних класів, визначаються як final, і тоді вони не можуть бути батьківськими класами для якихось інших. Натомість, щоби вони могли мати свої дочірні класи, їх потрібно визначити як non-sealed або sealed. Non-sealed класи відрізняються від sealed тим, що їх дочірні класи можуть без проблем мати свої дочірні класи, як і звичайні класи. Доповнимо попередній приклад відповідними прикладами:

```
public sealed class Dog extends Mammal
        permits Bulldog {
    // код
}
```

Рис. 1.1.4.3. Скріншот прикладу оголошення запечатаного класу, дочірнього від запечатаного

```
public non-sealed class Monkey extends Mammal {
    // код
}
```

Рис. 1.1.4.4. Скріншот прикладу оголошення незапечатаного класу, дочірнього від запечатаного

```
public class Chimpanzee extends Monkey {
    // код
}
```

Рис. 1.1.4.5. Скріншот прикладу оголошення класу, дочірнього від незапечатаного

До речі, використання pattern matching в операторі switch дозволяє легше працювати з запечатаними класами. Компілятор перевіряє, чи всі дочірні класи

певного sealed класу використовуються в case-блоках в switch, тому в такому випадку використання *default* не потрібне. [2]

1.1.5 Deprecate the Applet API for Removal (JEP 398)

Якщо раніше Applet API (в версії Java 9) для створення Java апплетів було визначено як deprecated, то тепер вони підлягали видаленню з мови, хоча насправді всі веб-браузери або в той час видалили підтримку плагінів для Java-браузерів, або вже тоді збирались це зробити. Applet API був популярним ще на початку розвитку Інтернету, коли веб-браузери були основним засобом доступу до онлайн-контенту. Однак з розвитком сучасних веб-технологій на основі HTML5, CSS та JavaScript, потреба в Applet API значно зменшилася.

Було видалено з мови такі класи та інтерфейси, як java.applet.Applet, javax.swing.JApplet, java.beans.Beans, тощо. [1]

1.1.6 New macOS Rendering Pipeline (JEP 382)

Більшість графічних Java-додатків написані за допомогою інструментарію Swing UI, який виконує рендеринг за допомогою Java 2D API. Усередині Java 2D може використовувати програмний рендеринг та відображення на екрані або ж використовувати специфічний для платформи API, наприклад, X11/Xrender на Linux, Direct3D на Windows або OpenGL на macOS. Запропонований новий конвеєр рендерингу, який називався Metal pipeline, базується на графічній технології Apple Metal API. Metal pipeline забезпечує кращу продуктивність, візуалізацію графіки та зменшене енергоспоживання для Java-додатків, що працюють на macOS.

Metal pipeline також забезпечує кращу підтримку дисплеїв з високою роздільною здатністю. Це дозволить Java-додаткам відображати текст і графіку з вищою роздільною здатністю і кращою чіткістю на екрані.

Загалом, дане оновлення графічного конвеєра мало на меті покращити продуктивність та можливості рендерингу графіки Java-додатків на macOS шляхом впровадження нового конвеєра рендерингу, заснованого на сучасних графічних технологіях. Metal pipeline забезпечує кращу продуктивність, покращену якість рендерингу графіки та зменшене енергоспоживання, що принесло користь як розробникам, так і користувачам Java-додатків на системах macOS. [1] [7]

1.1.7 Context-Specific Deserialization Filters (JEP 415)

Вочевидь, десеріалізація невідомих даних може бути небезпечною. У багатьох випадках набір байтів в потоці отримується від невідомого або неперевіреного клієнта. Ретельно конструюючи потік, код може бути викликаний з певними зловмисними намірами. Якщо конструювання об'єктів має побічні ефекти, які змінюють стан або викликають інші дії, ці дії можуть поставити під загрозу цілісність об'єктів програми і виконання самої програми на Java.

У версії Java 9 було реалізовано фільтри десеріалізації для того, щоб код додатків і бібліотек міг перевіряти вхідні потоки даних перед їхньою десеріалізацією. Для перевірки потрібно використати об'єкт класу java.io.ObjectInputFilter при створенні потоку десеріалізації як об'єкт класу java.io.ObjectInputStream.

У версії Java 17 реалізовано конфігуровану фабрику фільтрів, або filter factory. Щоразу, коли створюється ObjectInputStream, його потоковий фільтр ініціалізується значенням, яке повертається при виклику статичної filter factory. Ці фільтри є динамічними та залежать від контексту, на відміну від єдиного статичного фільтра десеріалізації в їх попередній версії.

Filter factory використовується для кожної операції десеріалізації під час виконання Java, незалежно від того, чи це код програми, чи код бібліотеки, чи код самого JDK. Вона є специфічною для кожної програми і викликається з конструктора ObjectInputStream, а також з ObjectInputStream.setObjectInputFilter.

В цілому, це оновлення гарантує більшу безпеку Java-додатків шляхом впровадження контекстно-залежних фільтрів десеріалізації як filter factory, які надають розробникам більший контроль над процесом десеріалізації та допомагають запобігти проблемам, пов'язаним зі створенням об'єктів та виконанням коду. [1]

1.2 Оновлення в версії Java 19

Версія Java 19 вийшла у світ 20 вересня 2022, але не як LTS-версія, тобто, її Premier Support (підтримка останніх оновлень та її технічна підтримка) завершилась вже з виходом найновішої версії Java 20 21 березня 2023 року. [9] [10] Всього в реліз Java 19 потрапило 7 JEP-ів, розглянемо їх детальніше. [11]

1.2.1 Record Patterns (Preview) (JEP 405)

Записи (records) були остаточно реалізовані в мові програмування Java ще в версії Java 16. Записи дозволяють лаконічно та компактно визначити нові класи, основною метою яких є зберігання певних даних та доступ до них, а не реалізація їх складної поведінки чи інкапсуляції. В певній мірі, це звичайні класи в Java, але зі скороченим шаблонним кодом для частого використання. Ось приклад оголошення запису:

```
public record Point(int x, int y, int z) { };
```

Рис. 1.2.1.1. Скріншот прикладу оголошення запису

Тобто, нам не потрібно тепер оголошувати новий клас, визначати його поля, конструктор та селектори, натомість можна використовувати для цього записи. Ба більше, вони вже мають згенеровані реалізації методів equals(), hashCode() та toString(). Також записи за замовчуванням є final та immutable, і не можуть бути суперкласами та підкласами, це було зроблено для того, щоби забечпечити моделювання об'єктів даних, які повинні бути immutable, як-от об'єкти передачі даних (DTO, Data Transfer Object).

В версії Java 19 цей JEP ще визначений як preview.

Певне значення v буде відповідати record pattern (візьмемо Point(int x, int y, int z) p), якщо воно належить класу паттерну (в даному випадку Point). Якщо так, то змінні з record pattern тоді ініціалізуються відповідними значеннями з v, і тоді p весь ініціалізується, при цьому набуваючи переведення до класу Point.

Також record patterns можуть бути використані в операторі switch. Що важливо, блок switch повинен містити усі можливі випадки значень полів запису.

Важливо зазначити, що значення null не може відповідати record patterns. Також в паттернах можна замінити тип поля словом var (наприклад, *Point(var x, var y, var x) p*). [1]

1.2.2 Linux/RISC-V Port (JEP 422)

Даний JEP пропонує повністю перенести OpenJDK на платформу Linux/RISC-V. RISC-V – це архітектура набору інструкцій (ISA, Instruction Set Architecture) RISC з відкритим вихідним кодом та безоплатною ліцензією для обчислювальних платформ, і зі збільшенням популярності та доступності апаратного забезпечення RISC-V, наявність порту JDK для цієї платформи була б доволі корисною.

Перенесення відбулось орієнтовано на RV64GV-конфігурацію RISC-V, яка є 64-розрядною архітектурою набору інструкцій (Instruction Set Architecture)

загального призначення. Початкова увага зосереджена на інтеграції переносу до основного репозиторію JDK, а не на самому перенесенні, оскільки більша частина роботи на той момент вже практично завершилась.

Що цікаво, розробники зазначили, що компанія Huawei Technologies зобов'язалась повністю підтримувати код (включаючи оновлення, покращення та тестування), імплементований в цьому JEP-і. [1] [12]

Важливо зазначити, що цей JEP та JEP-и під номерами 424 та 426 були реалізовані як частина Java-проекту під назвою Panama, який дозволяє полегшити взаємодію між Java-кодом та іншими APIs, розроблених на мовах C, C++, тощо. [13]

1.2.3 Foreign Function & Memory API (Preview) (JEP 424)

У мові програмування Java давно існує можливість використовувати бібліотеки, розроблені на інших мовах. Проте, постійно існувала проблема в отриманні доступу до ресурсів за межами Java Runtime Environment.

Даний API дозволяє програмам на Java взаємодіяти з кодом та даними поза межами JRE (Java Runtime Environment, середовище виконання Java). Він працює, викликаючи зовнішні функції за межами середовища виконання ефективним шляхом та отриманням доступу до пам'яті, до якої сама JVM не має доступу.

Foreign Function & Memory API визначає різні класи та інтерфейси для різних функцій, наприклад:

- Виділення зовнішньої пам'яті (MemorySegment, SegmentAllocator);
- Керування виділенням та вивільненням зовнішньої пам'яті (MemorySession);
- Керування доступом до структурованої зовнішньої пам'яті (MemoryLayout, VarHandle);

- Виклик зовнішніх функцій (Linker, FunctionDescriptor).

Цей API знаходиться в пакеті java.lang.foreign в модулі java.base. [1] [13]

1.2.4 Virtual Threads (Preview) (JEP 425)

У версії Java 19 реалізували віртуальні потоки в режимі preview. Віртуальний потік є об’єктом класу java.lang.Thread, який виконує код програми у базовому потоці операційної системи, але не захоплює цей потік на весь час виконання коду.

Код програми, який працює з потоками в режимі thread-per-request, може виконуватись у віртуальному потоці протягом усього часу виконання запиту, проте цей віртуальний потік використовує потік операційної системи лише під час виконання обчислень у процесорі. Коли код, що виконується у віртуальному потоці, викликає блокуючу операцію введення/виведення, середовище виконання виконує її та автоматично призупиняє віртуальний потік, поки він не зможе відновити роботу пізніше. У мові програмування Java віртуальні потоки - це просто потоки, які вимагають мало ресурсів для створення, та їх кількість може сягати надзвичайно велике число.

Віртуальні потоки є дешевими в плані використання ресурсів та доступними, тому їх ніколи не слід об'єднувати, адже для кожної задачі програми слід створювати новий віртуальний потік. Таким чином, більшість віртуальних потоків будуть недовговічними і матимуть невеликий стек викликів, виконуючи лише один виклик HTTP-клієнта. Звичайні потоки в Java, навпаки, важкі і дорогі, тому їх часто доводиться об'єднувати в пули. Вони, як правило, довговічні і розподіляються між багатьма завданнями.

Віртуальні потоки в Java виконуються потоками платформи. Платформний потік може одночасно виконувати лише один віртуальний потік. Коли віртуальний потік виконується потоком платформи - віртуальний потік

вважається «змонтованим» до цього потоку. Нові віртуальні потоки знаходяється у черзі, доки платформний потік не буде готовий їх виконати. Коли платформний потік буде готовий, він захоплює віртуальний потік і починає виконувати його. Віртуальний потік, який виконує певну блокуючу операцію введення/виведення, буде від'єднано від платформного потоку в очікуванні відповіді. Тим часом потік платформи може виконувати інший віртуальний потік.

Між віртуальними потоками не відбувається розподілу часу. Іншими словами, потік платформи не перемикається між виконанням декількох віртуальних потоків - за винятком випадків блокування мережевих викликів. Поки віртуальний потік виконує код і не є заблокованим в очікуванні відповіді від мережі - потік платформи продовжуватиме виконувати той самий віртуальний потік. [14]

Використання віртуальних потоків не вимагає розуміння нових концепцій, пов'язаних з класичними потоками в Java, хоча може вимагати відмовитись від певних правил, існуючих для того, щоби працювати з високою вартістю потоків.

Віртуальні потоки допоможуть розробити масштабний проект в Java, в якому буде здійснюватись велика кількість викликів. [1]

1.2.5 Vector API (Fourth Incubator) (JEP 426)

Vector API був запропонований та реалізований ще в версії Java 16, і, включаючи цю версію, даний JEP визначений як Incubator (це API або інструмент, що ще недостатньо протестований, та підлягає подальшому виправленню в наступних версіях).

У даному API вектор реалізовано як абстрактний клас Vector<E>, де E – це цілочисельний тип, або тип числа з плаваючою комою. Вектор також має форму, яка визначає розмір вектора у бітах.

Vector API допомагає покращити продуктивність додатків у таких галузях, як машинне навчання, криптографія тощо, при цьому виконуючи векторні обчислення. У цій версії Vector API набув таких змін: [15]

- Новий функціонал для збереження та отримання векторів з використанням MemorySegments, описаного в Foreign Function & Memory API (нагадаємо, що цей API теж належить проекту Panama для роботи з функціоналом з інших мов, окрім Java);
- Нові операції з міжсмуговими векторами compress та expand;
- Додаткові порозрядні посмугові операції підрахунку кількості одиничних бітів, передніх нульових бітів, задніх нульових бітів, зміни порядку бітів та байтів, стиснення та розширення бітів;

Як і в попередніх версіях, в Java 20 JEP з Vector API досі визначений як Incubator, та і в Java 21, яка вийде восени 2023, він буде Incubator. [1]

1.2.6 Pattern Matching for switch (Third Preview) (JEP 427)

Як було згадано вище, pattern matching для оператора switch було реалізовано ще в версії Java 17. З оновлень у цьому JEP-і подвійний амперсанд було замінено на зарезервоване слово when для детальнішого розуміння, як працює guarded pattern. [1] Ось приклад використання такого switch:

```
switch (o) {
    case Integer num when (num > 1) -> System.out.println("Natural Int");
    case Boolean bool when bool -> System.out.println("True Bool");
    default -> System.out.println("Default");
}
```

Рис. 1.2.6.1. Скріншот прикладу оператора switch з оновленим паттерн-матчингом

1.2.7 Structured Concurrency (Incubator) (JEP 428)

Структурований паралелізм (Structured Concurrency) – це окремий підхід до багатопоточного програмування, де задача, поділена на паралельні підзадачі, реалізована в коді, через що вона стає легшою та простішою для роботи з потоками. Таким чином, забезпечуються чітко визначені точки входу і виходу для виконання коду і сувору вкладеність операцій, як і у структурованому програмуванні для однопотокового коду. При структурованому паралелізмі підзадачами керує їхня батьківська задача, яка чекає на їхні результати і відстежує збої. Час життя підзадач обмежується блоком батьківської задачі, що дозволяє керувати підзадачами як єдиним цілим.

Структурований паралелізм добре працює з віртуальними потоками, які теж було реалізовано в Java 19, як ми згадували вище. Віртуальних потоків може бути багато і вони можуть доволі ефективно реалізувати паралельну поведінку, навіть включаючи блокуючі операції вводу/виводу. У серверних додатках структурований паралелізм у поєднанні з віртуальними потоками дозволяє обробляти декілька запитів одночасно, виділяючи віртуальний потік для кожної задачі та її підзадачі. Зв'язок між задачами та підзадачами представляється у вигляді дерева, подібно до стеку викликів в одному потоці.

Таким чином, віртуальні потоки забезпечують створення і роботу великої кількості потоків, і в той час структурований паралелізм гарантує, що вони правильно і надійно координуються, і дозволяє відображати роботу потоків в коді так, як їх зможе зрозуміти розробник. Наявність даного API для реалізації структурованого паралелізму у JDK може значно покращити роботу серверних додатків. [1]

РОЗДІЛ 2. ПОРІВНЯННЯ ПОКРАЩЕНЬ ВЕРСІЇ JAVA 19 З JAVA 17

2.1 Реалізація прикладної програми

У даному розділі детально описані порівняння оновлень в версії Java 19, та як вони покращили роботу з мовою Java в порівнянні з останньою на даний момент LTS-версією мови Java 17.

Для того, щоби краще продемонструвати, як працюють нові реалізовані JEP-и в Java 19, ми створили невеликий прикладний додаток, який імітує роботу онлайн-магазину. У даному коді ми імплементували ці оновлення, та за можливості порівнюємо їх з можливими реалізаціями в старішій версії мови Java 17.

У коді є один головний клас OnlineShop, в якому реалізовано логіку програми. Він містить класи для управління продуктами, клієнтами та замовленнями. Код дозволяє покупцям реєструватися, входити в систему, переглядати каталог товарів і здійснювати покупки. Розглянемо код та його ключові компоненти більш детально.

Клас OnlineShop організовує функціональність нашої “системи інтернет-магазину”. Він містить методи для обробки реєстрації, входу, перегляду товарів і здійснення покупок. У ньому зберігаються три списки: catalog, customers і orders. У класі OnlineShop визначено три record-класи: Product, Customer і Order. Ці класи є певними моделями даних для представлення товарів, клієнтів та замовлень в “системі інтернет-магазину”. Відповідно, кожен запис (record) складається з полів, які зберігають відповідну інформацію, наприклад, клас Product має змінні name та price.

Розглянемо трохи детальніше методи в класі OnlineShop. Метод registerCustomer дозволяє новим клієнтам “зареєструватися” в інтернет-магазині,

надавши параметри username, password та email. Він створює новий об'єкт Customer і додає його до списку клієнтів.

Метод login дозволяє зареєстрованим клієнтам увійти в систему, вказавши свій username та password. Він звіряє облікові дані з існуючими клієнтами у списку клієнтів. Якщо облікові дані присутні в списку customers, то метод повертає відповідний об'єкт Customer.

Метод shopProducts симулює процес купівлі товару. Він приймає об'єкт Customer та id товару як параметри, отримує вибраний товар зі списку за допомогою методу getProductById і створює об'єкт Order, і потім він додається до списку orders. Якщо вибір товару є невірним, виводиться відповідне повідомлення. Також для зручності була реалізована функція performPurchases, в якій можна здійснити покупку одразу декількох товарів, і вона використовує метод shopProducts.

Метод start розпочинає виконання самої програми. Він виводить консольне меню і викликає відповідні методи для виконання потрібних користувачу дій. Він дозволяє клієнтам зареєструватися, увійти або вийти з системи інтернет-магазину.

2.2 Імплементація віртуальних потоків та порівняння їх роботи з платформними

У даній програмі в методі main є реалізація багатопоточності. Припустимо таку ситуацію, що багато клієнтів одночасно роблять покупку в онлайн-магазині, і для цього кожен вхідний запит повинен оброблятися в окремому потоці, що дозволяє декільком користувачам переглядати і взаємодіяти з програмою одночасно. У коді багатопоточність реалізована таким чином, що кожен раз, коли хтось здійснює покупку (для цього викликається функція shopProducts), для цього створюється новий потік, і ця функція викликається.

Ось як це було реалізовано з використанням віртуальних потоків:

```
startTime = System.nanoTime();

executor = Executors.newVirtualThreadPerTaskExecutor();
for (int i = 0; i < 100000; i++) {
    executor.submit(task);
}
executor.close();

endTime = System.nanoTime();
elapsedTime = endTime - startTime;

double elapsedTimeMsVir = elapsedTime / 1_000_000.0;
```

Рис. 2.2.1. Скріншот з коду прикладної програми, де було реалізовано використання віртуальних потоків, разом з використанням функції nanoTime() для виміру пройденого часу

```
Runnable task = () -> {
    performPurchases(shop, numPurchases: 2);
};
```

Рис. 2.2.2. Скріншот з коду прикладної програми, де було реалізовано об'єкт Runnable, який передаватиметься в потоки як код для виконання

Отже, спочатку ми створюємо об'єкт інтерфейсу Runnable - task, в якому визначено частину коду, яка повинна бути виконана в певному потоці, а саме метод performPurchase. Далі створюється об'єкт класу ExecutorService за допомогою методу Executors.newVirtualThreadPerTaskExecutor(). Він буде створювати «виконавця», який призначає кожну задачу окремому віртуальному потоку. Далі, в циклі ми передаємо наш task на виконання, і вже після закінчення

циклу завершується робота «виконавця» та звільняються усі його ресурси з пам’яті.

Порівняємо цю реалізацію з іншою в програмі, де створюються не віртуальні потоки, а платформні. Ця реалізація є актуальною для версії Java 17, оскільки в ній не було реалізовано віртуальні потоки.

```
long startTime = System.nanoTime();

ExecutorService executor = Executors.newCachedThreadPool();

for (int i = 0; i < 100000; i++) {
    executor.submit(task);
}
executor.close();

long endTime = System.nanoTime();
long elapsedTime = endTime - startTime;
double elapsedTimeMs = elapsedTime / 1_000_000.0;
```

Рис. 2.2.3. Скріншот з коду прикладної програми, де було реалізовано використання платформних потоків, разом з використанням функції nanoTime() для виміру пройденого часу

На відміну від методу Executors.newVirtualThreadPerTaskExecutor(), Executors.newCachedThreadPool() створює «виконавця», який керує пулом потоків і динамічно регулює його розмір залежно від програмного навантаження. Якщо надходить нове завдання і в пулі є ще вільні потоки, воно призначається одному з них, інакше для обробки завдання створюється новий потік.

Віртуальні потоки займають менше пам'яті та мають менші накладні витрати порівняно з платформними потоками. Вони можуть бути більш

масштабованими і краще працювати паралельно між собою. Віртуальні потоки забезпечують певний рівень ізоляції між завданнями. Якщо один віртуальний потік стикається з блокувальною операцією, він не блокує інші віртуальні потоки, дозволяючи їм продовжувати виконання.

Перевіримо власноруч в коді, які потоки дозволяють швидше працювати: віртуальні чи платформні.

Для перевірки часу роботи певної частини коду з початку і кінця ми отримуємо поточний час за допомогою функції System.nanoTime(), віднімаємо кінцевий від початкового та переводимо в мілісекунди. Ось що було виведено в консоль при запуску коду:

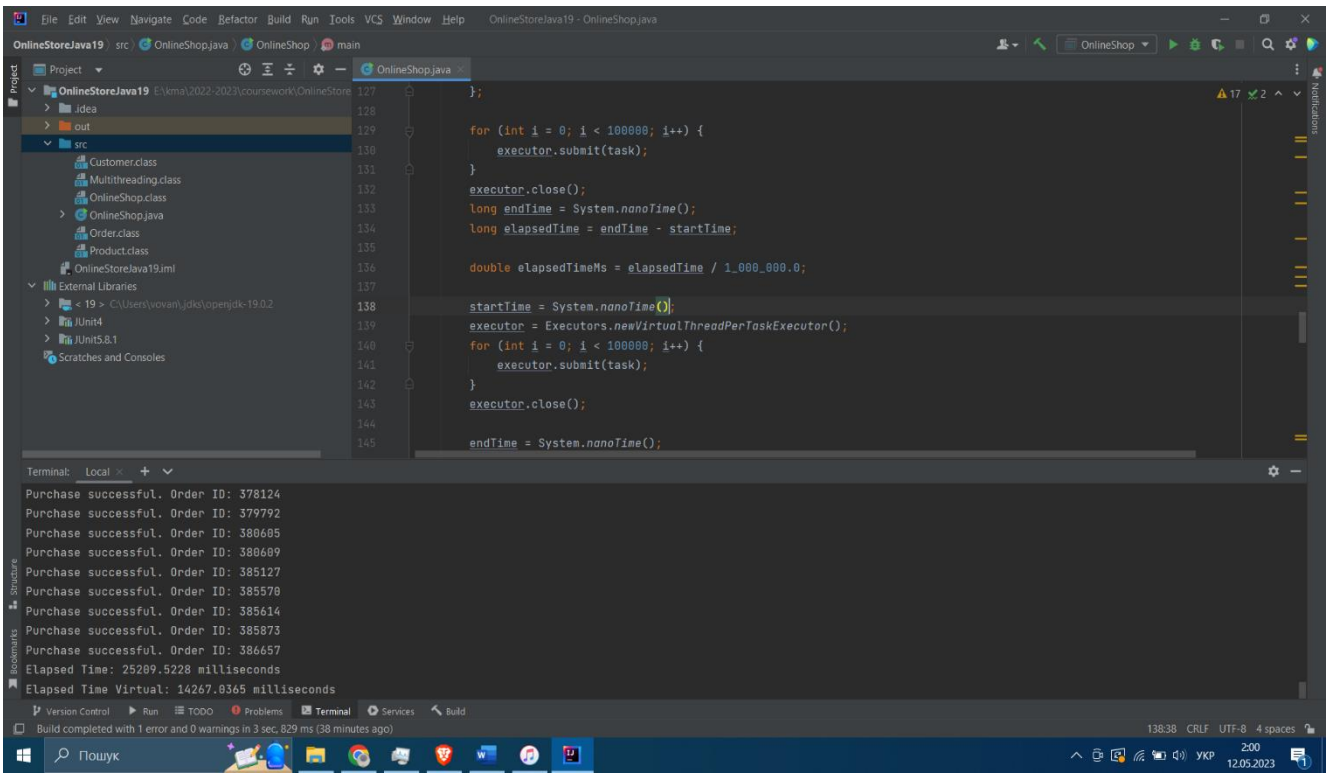

Рис. 2.2.4. Скріншот Intellij IDEA при завершенні виконання програми

Як бачимо, час роботи програми з використанням віртуальних потоків майже в два рази менший, ніж коду з використання платформних.

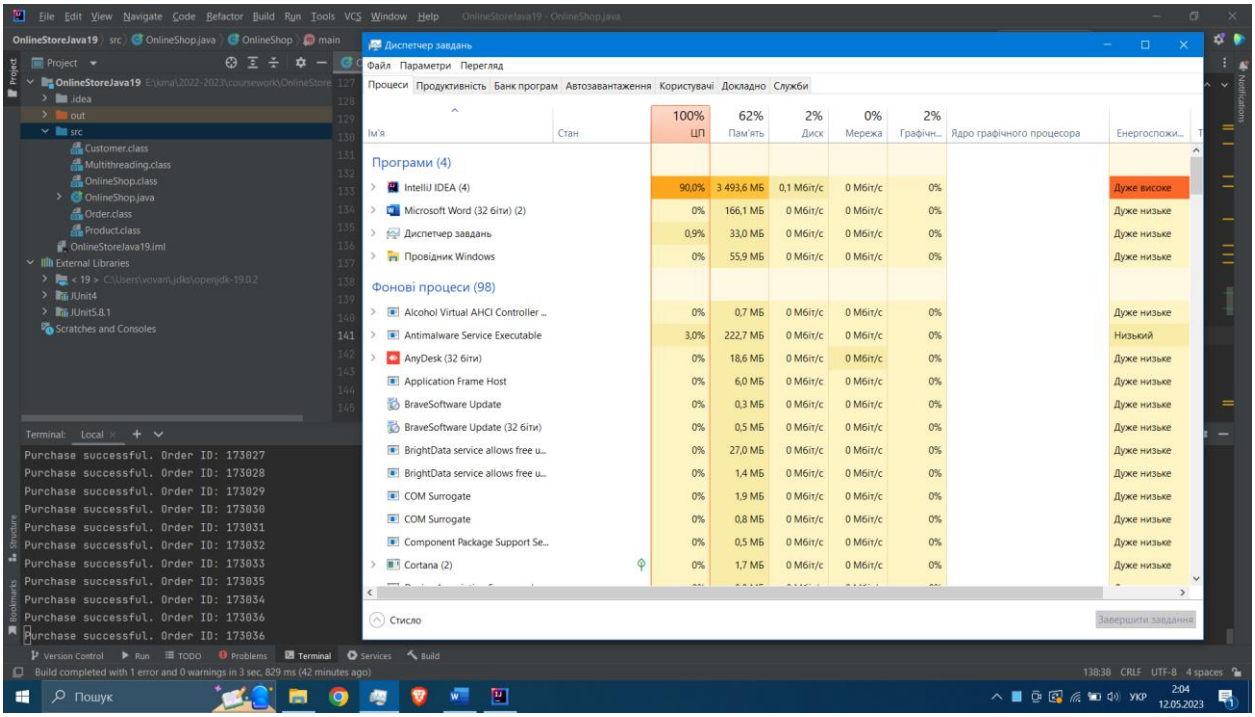

Рис. 2.2.5. Скріншот диспетчера завдань при роботі з платформними потоками

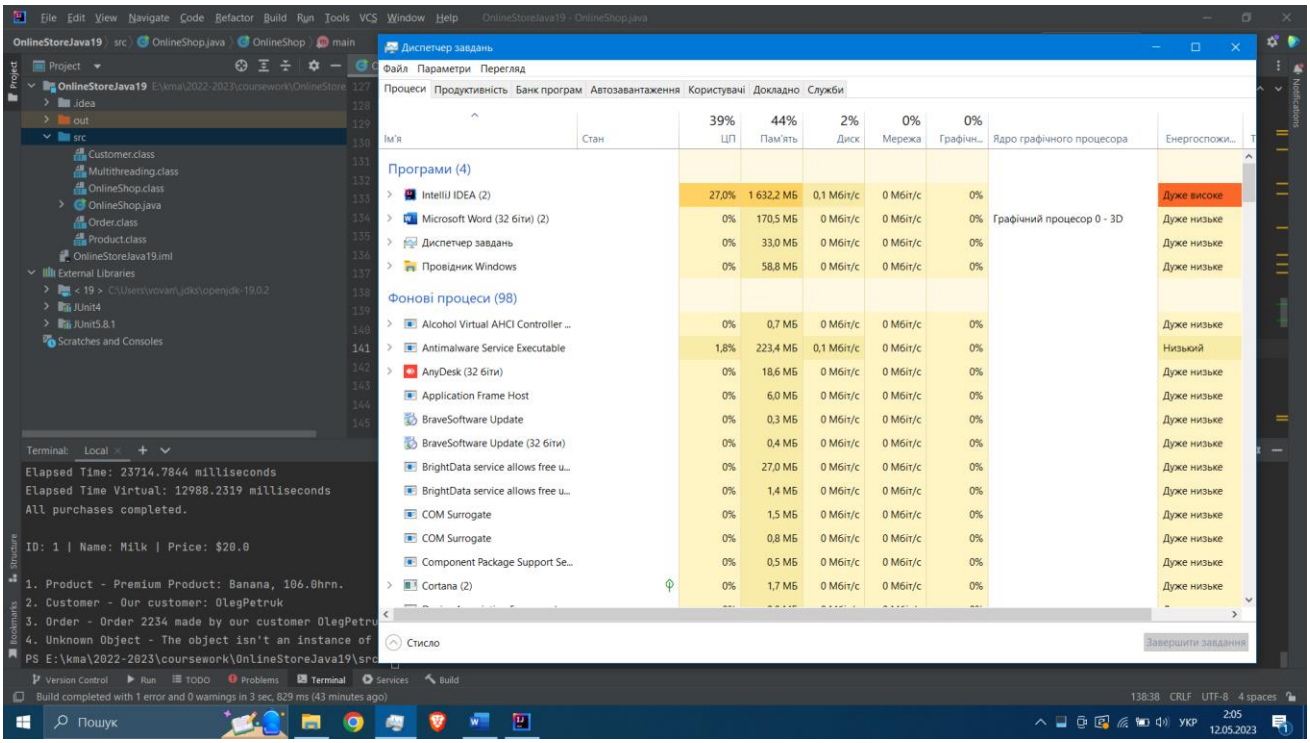

Рис. 2.2.6. Скріншот диспетчера завдань при роботі з віртуальними потоками

Також можемо бачити, що віртуальні потоки використовують менше ресурсів, ніж платформні, що ще раз свідчить про оптимізацію роботи з багатопоточністю в версії Java 19 в порівнянні з версією Java 17.

2.3 Імплементація структурованого паралелізму

У методі main для реалізації прикладу роботи структурованого паралелізму в версії Java 19 використовується об’єкт класу StructuredTaskScope. Цей клас дозволяє структурувати задачі у вигляді їх паралельних підзадач і координувати їх як єдине ціле. Підзадачі виконуються окремо у власних потоках, а потім об'єднуються з використанням StructuredTaskScope в єдине. Успішні результати підзадач або exceptions агрегуються і обробляються батьківською задачею. StructuredTaskScope обмежує час життя підзадач чіткою областю, в якій відбувається вся взаємодія задачі з її підзадачами - розгалуження, об'єднання, скасування, обробка exceptions і складання результатів.

Ось як ми реалізували структурований паралелізм у нашій програмі:

```
int id = 1;
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<String> productName = scope.fork(() -> shop.getProductName(id));
    Future<Double> productPrice = scope.fork(() -> shop.getProductPrice(id));

    scope.join();
    scope.throwIfFailed();

    System.out.println("ID: " + id + " | Name: " +
            productName.resultNow() + " | Price: $" +
            productPrice.resultNow());
} catch (ExecutionException | InterruptedException e) {
    throw new RuntimeException(e);
}
```

Рис. 2.3.1. Скріншот з коду прикладної програми, де було реалізовано структурований паралелізм

Оскільки в версії Java 17 немає реалізації API для роботи зі структурованим паралелізмом, для цього можна використовувати вищезгаданий java.util.concurrent.ExecutorService API. Наприклад, реалізуємо цей код в методі handle() з використанням даного API. Він обробляє вхідний запит, передаючи дві підзадачі об’єкту класу ExecutorService. Одна підзадача виконує метод findUser(), а інша підзадача виконує метод fetchOrder(). ExecutorService тоді повертає Future для кожної підзадачі, і виконує кожну у власному потоці. Метод handle() очікує результати підзадач, блокуючи виклики методів get() їхніх Futures, отже, в даному коді задача об'єднує свої підзадачі.

```
Response handle() throws ExecutionException, InterruptedException {
    Future<String> productName = esvc.submit(() -> shop.getProductName(id));
    Future<Double> productPrice = esvc.submit(() -> shop.getProductPrice(id));
    String name  = productName.get();
    Double price = productPrice.get();
    return new Response(name, price);
}
```

Рис. 2.3.2. Скріншот прикладу коду, де було реалізовано структурований паралелізм з використанням засобів Java 17

Оскільки підзадачі виконуються паралельно, кожна з них може успішно завершитися або призвести до exception. Метод на зразок handle() завершує своє виконання, якщо будь-яка з його підзадач завершує роботу з помилкою. Розуміння часу життя потоків може бути напрочуд складним, коли відбувається збій, наприклад, якщо productName() викине exception, то сам handle() це зробить при виклику productName.get(), але productPrice() продовжить виконання у власному потоці. Це витік потоку, який, у кращому випадку, призведе до марної трати ресурсів, а у гіршому - потік productPrice() буде заважати іншим задачам.

Отже, реалізація Structured Concurrency API в версії Java 19 стала значним покращенням в роботі з паралельністю потоків, оскільки вона вже не містить проблем, які мали інші засоби для створення паралельних потоків. [1]

2.4 Використання record patterns

У коді було реалізовано додатковий метод checkInfo, який використовує record patterns, що значно спрощує роботу з записами. Також було використано JEP 427 “Pattern Matching for switch”, де подвійний амперсанд для guarded patterns було замінено на слово when. Розглянемо цей метод:

```
private static void checkInfo(Object o) {
    switch (o) {
        case Product(String name, double price) when price > 100 ->
                System.out.println("Premium Product: " + name + ", " + price + "hrn.");
        case Product(String name, double price) ->
                System.out.println("Product: " + name + ", " + price + "hrn.");
        case Customer(String username, String password, String email) ->
                System.out.println("Our customer: " + username);
        case Order(int orderId, Customer customer,
                   List<Product> products, double totalAmount, String status) ->
                System.out.println("Order " + orderId
                        + " made by our customer " + customer.username()
                        + ", amount: " + totalAmount);
        default -> System.out.println("The object isn't an instance of OnlineShop.");
    }
}
```

Рис. 2.4.1. Скріншот з коду прикладної програми, де було реалізовано record patterns з оновленим паттерн-матчингом

Саме тому реалізація цього JEP-у пішла на користь для спрощеної роботи з записами.

2.5 Інші оновлення

Окрім реалізованих оновлень, інші, що згадані у першому розділі, теж сприяли покращенню версії Java 19 в порівнянні з Java 17, а саме:

- Foreign Function & Memory API (для оптимізованої роботи з ресурсами з інших мов програмування, як-от C, C++, тощо);
- Vector API (Fourth Incubator) (додано нові корисні функції для роботи з великими об’єктами Vector).

ВИСНОВКИ

У цій курсовій роботі ми провели аналіз покращень, реалізованих у версії Java 19 порівняно з Java 17, разом з детальним аналізом кожної з версій. Мова програмування Java постійно розвивається, щоб відповідати трендам в розробці програмного забезпечення. Ці покращення включають в себе новий функціонал, вдосконалення вже існуючих функцій та бібліотек, що дозволяє краще працювати з мовою.

Одне з помітних удосконалень, впроваджених у Java 19 - це реалізація віртуальних потоків. Вони допомагають прискорити роботу з багатопоточністю, що підвищує продуктивність роботи складних додатків. Ще одним значним покращенням у Java 19 є реалізація API для структурованої паралельності, який вирішує проблему з витіками потоків у паралельному програмуванні, і представляє потужне рішення, що теж покращує роботу з багатопоточністю у складних застосунках.

Порівнюючи ці вдосконалення з Java 17, очевидно, що Java продовжує розвиватися. У той час як Java 17 представила такі оновлення, як запечатані класи, зіставлення шаблонів і вдосконалені оператори перемикання, Java 19 спирається на них і вводить нові бібліотеки та функції, які ще більше розширюють можливості розробників.

Перспективами розвитку у даній темі є покращення таких JEP-ів, як віртуальні потоки, структурована паралельність, pattern matching, тощо, та створення нового функціоналу, який буде підтримувати Java, що дозволить їй триматись на одному рівні з іншими популярними та широко застосовуваними мовами програмування.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ ТА ЕЛЕКТРОННИХ РЕСУРСІВ

1. JEP 0 – JEP Index, список усіх існуючих JEP-ів. [Електронний ресурс]. – Режим доступу: https://openjdk.org/jeps/0
2. Стаття «Нові можливості в Java 17». [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/java-17-new-features
3. Примітки до випуску Java 17. [Електронний ресурс]. – Режим доступу: https://builds.shipilev.net/backports-monitor/release-notes-17.html
4. Стаття «Звіт New Relic про стан екосистеми Java у 2023 році свідчить про подальше домінування Java». [Електронний ресурс]. – Режим доступу: https://datastorageasean.com/news-press-releases/2023-state-java-ecosystem-report-new-relic-reveals-javas-continued-dominance
5. Стаття «Стан Java у 2022». [Електронний ресурс]. – Режим доступу: https://newrelic.com/resources/report/2022-state-of-java-ecosystem
6. Генератор псевдовипадкових чисел (ГПВЧ). [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/pseudo-random-number-generator-prng/
7. Конвеєр рендерингу OpenGL | Огляд. [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/opengl-rendering-pipeline-overview/
8. Pattern Matching для Java. [Електронний ресурс]. – Режим доступу: https://openjdk.org/projects/amber/design-notes/patterns/pattern-matching-for-java
9. Особливості Java 19. [Електронний ресурс]. – Режим доступу: https://www.jrebel.com/blog/java-19-features
10. Особливості Java 20. [Електронний ресурс]. – Режим доступу: https://www.happycoders.eu/java/java-20-features/
11. Примітки до випуску Java 19. [Електронний ресурс]. – Режим доступу: https://builds.shipilev.net/backports-monitor/release-notes-19.html

12. Реліз Java 19: 7 нових JEP та ще багато покращень. [Електронний ресурс]. – Режим доступу: https://dou.ua/forums/topic/40030/
13. Гайд по проекту Panama. [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/java-project-panama
14. Віртуальні потоки в Java. [Електронний ресурс]. – Режим доступу: https://jenkov.com/tutorials/java-concurrency/java-virtual-threads.html
15. Java 19 - нові функції та покращення. [Електронний ресурс]. – Режим доступу: https://bell-sw.com/announcements/2022/06/22/java-19-new-features-and-enhancements/#jep-426-vector-api-fourth-incubator
16. Структурований паралелізм в Java 19. [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/java-structured-concurrency
17. Примітки до випуску JDK 19. [Електронний ресурс]. – Режим доступу: https://www.oracle.com/java/technologies/javase/19-relnote-issues.html