МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

М. І. Шаповалова, О. О. Водка

МОВА ПРОГРАМУВАННЯ JAVA
Частина 1

Навчальний посібник
для студентів першого (бакалаврського)
рівня підготовки спеціальностей
F1(113) «Прикладна математика» та
F3(122) «Комп’ютерні науки»

Затверджено
редакційно-видавничою
радою університету,
протокол № 2 від 26.06.2025 р.

Харків
НТУ «ХПІ»
2025

УДК 004.432(075)
Ш 24

Рецензенти:
Д. М. Крицький, канд. техн. наук, доц., Національний аерокосмічний університет «Харківський авіаційний інститут»;
О. В. Каратанов, канд. техн. наук, доц., Національний аерокосмічний університет «Харківський авіаційний інститут»

Шаповалова М. І.

Ш 24 Мова програмування Java. Частина 1 : навчальний посібник для студентів першого (бакалаврського) рівня підготовки спеціальностей F1(113) «Прикладна математика» та F3(122) «Комп'ютерні науки» / М. І. Шаповалова, О. О. Водка. – Харків : НТУ «ХПІ», 2025. – 125 с.

ISBN 978-617-05-0585-9

У навчальному посібнику розглянуто основи програмування мовою Java, її синтаксис та об'єктно-орієнтовану модель. Наведено основні принципи роботи з колекціями, рядками, потоками та структурою Java-програм. Усі розділи супроводжуються прикладами, контрольними запитаннями та завданнями для самостійної роботи, що сприяє ефективному засвоєнню матеріалу.

Призначено для студентів бакалаврського рівня підготовки спеціальностей F1(113) «Прикладна математика», F3(122) «Комп'ютерні науки» та інших технічних спеціальностей денної та заочної форм навчання.

Іл. 14. Бібліогр. 20 назв.

УДК 004.432(075)

ISBN 978-617-05-0585-9

ЗМІСТ

ВСТУП

У сучасну цифрову епоху, коли інформаційні технології проникають у всі сфери життя та діяльності людини, вивчення мов програмування стає ключовою складовою професійної підготовки спеціалістів у галузі комп'ютерних наук, прикладної математики та суміжних технічних дисциплін. Мова програмування Java, завдяки своїм особливостям і широкій сфері застосування, посідає одне з провідних місць серед мов, які використовуються у сфері розробки програмного забезпечення, мобільних додатків, веб-сервісів, корпоративних інформаційних систем, вбудованих рішень тощо.

Метою даного навчального посібника є формування у студентів цілісного уявлення про основи програмування на Java, розуміння філософії мови, принципів її функціонування, ключових концепцій, а також здобуття практичних навичок розробки програм різного рівня складності. Посібник структуровано відповідно до логіки послідовного оволодіння знаннями – від історичного огляду та базових синтаксичних конструкцій до глибокого занурення в об'єктно-орієнтовану парадигму, роботу з колекціями, рядками, потоками та іншими складовими сучасного програмування.

Перший розділ посібника присвячено історичним передумовам створення мови Java, аналізу її переваг і відмінностей від інших мов програмування. Зокрема, розглядається еволюція від мови Oak до Java, обґрунтовується принцип Write Once, Run Anywhere, аналізуються типи Java (SE, EE, ME, Card), а також розкриваються особливості Java Virtual Machine (JVM), Java Runtime Environment (JRE) і Java Development Kit (JDK). Увага приділяється основним етапам компіляції та виконання Java-програм, що закладає фундамент для подальшого поглибленого вивчення.

Другий розділ зосереджено на лексичній структурі мови. У ньому розглядаються правила формування ідентифікаторів, використання літералів, операторів, роздільників, а також правила коментування коду. Особлива увага приділяється правилам типізації, ініціалізації змінних, роботі з масивами, оголошенню та виклику методів, а також структуруванню коду за допомогою блоків. Це дозволяє сформувати необхідні технічні знання для створення функціональних програм на початковому рівні.

Третій розділ присвячено об'єктно-орієнтованому програмуванню, що є базовою концепцією Java. Розглядаються основні принципи ООП: інкапсуляція, спадкування, поліморфізм. Аналізується побудова класів, створення методів і конструкторів, використання модифікаторів доступу та абстрактних класів, реалізація інтерфейсів. Окремо розглядається обробка виключних ситуацій, що дозволяє забезпечити надійність і безпеку програмного коду.

У четвертому розділі увагу зосереджено на фреймворку Java Collections – потужному інструментарії для роботи з динамічними структурами даних. Розглядаються інтерфейси List, Set, Map, Queue, а також класи HashSet, TreeSet, ArrayList, LinkedList, PriorityQueue, HashMap, TreeMap

тощо. Теоретичний матеріал супроводжується прикладами коду, що дозволяє закріпити знання на практиці та навчитися ефективно оперувати з наборами об’єктів.

П’ятий розділ висвітлює аспекти роботи з рядковими даними. Окрім аналізу класу String, докладно розглядаються класи StringBuilder і StringBuffer, що забезпечують ефективне управління пам’яттю та дозволяють уникати надлишкових витрат ресурсів при обробці великих обсягів текстової інформації. Порушуються питання імімутабельності об’єктів, порівняння рядків, перетворення типів та регулярних виразів.

Шостий розділ вводить студентів у концепції багатопотокового програмування – критично важливого для створення високопродуктивного ПЗ. Вивчаються основи створення потоків за допомогою класів Thread і Runnable, методи керування потоками, синхронізація, міжпотокова взаємодія та модель пам’яті Java. Детально аналізується механізм моніторів та блокування, розглядаються типові помилки при роботі з потоками та способи їх уникнення.

Кожен розділ завершується завданнями для самостійного виконання та контрольними запитаннями, що сприяє закріпленню теоретичного матеріалу, розвитку логіко-аналітичного мислення та вмінню застосовувати знання на практиці. Завдання мають різний рівень складності – від базових до розширених, що дозволяє адаптувати навчальний процес до індивідуального рівня підготовки студентів.

Таким чином, навчальний посібник створений із урахуванням вимог до сучасної вищої освіти в галузі інформаційних технологій, поєднуючи теоретичну глибину та практичну спрямованість. Він стане корисним не лише студентам, а й усім, хто прагне оволодіти мовою Java як одним із найпотужніших інструментів сучасного програмування. Успішне освоєння матеріалу дозволить читачеві впевнено застосовувати Java для створення реальних програмних рішень і продовжити навчання на просунутому рівні, включаючи вивчення фреймворків, бібліотек та архітектурних підходів, що використовуються в сучасній індустрії розробки програмного забезпечення.

Розділ 1. ІСТОРІЯ JAVA

1.1. Історичні передумови виникнення мови Java

Наприкінці 1980-х початку 1990-х років, основним засобом об'єктно орієнтованого програмування вважалась мова C++. Для неї були характерні наступні особливості: пізня стандартизація; сильна залежність від реалізації і платформи; відсутність стандарту на бінарне представлення; досить висока складність розробки програм; відсутність стандартної бібліотеки, що містила би засоби мережевої та між процесної взаємодії і побудови графічного інтерфейсу; а також, характерна класична модель побудови систем: компіляція, зв'язування та побудова образу процесу [1].

У той період, інженери компанії Sun Microsystems – Патрік Ноутон та Джеймс Гослінг, працювали над проєктом «Green». Мета проєкту полягала у розробці мову, яка підходила б для програмування побутових електронних пристроїв. Яких саме? Найрізноманітніших. Наприклад, це могли бути контролери для перемикання каналів кабельного телебачення, регулятори температури у приміщенні, різноманітні логічні перемикачі та реле.

Чому саме на це звернули уваги? Тому що побутові пристрої:

1. мало споживають енергії (звідси і кодова назва проєкту – «Green»);
2. побутові електронні пристрої мають невеликі мікросхеми та пам'ять, а, відповідно, і програми, написані під них, повинні були бути невеликими.

Крім цього, перед розробниками стояло завдання, щоб одного разу написаний код працював на будь-якій машині. Ідея полягала у розробці універсального коду, інструменту, що б дозволив не писався під кожен пристрій окремо код «з нуля». Мова програмування C ++ для таких цілей підходила не повністю, тому було прийняте рішення розробили під потреби проєкту нову мову (Oak), що у подальшому отримала назву – Java.

Мова програмування Oak (1991) James Gosling, Patric Naughton, Chris Warth, Ed Frank, Mike Sheridan, Sun Microsystems, Inc:

- розробка тривала 18 місяців, перший компілятор було протестовано восени 1991р.;
- для розробленої мови характерна: платформна незалежність, переносимість, створена віртуальна машина;
- початковий акцент був на «побутову електроніку» та програмування мікроконтролерів;
- час виходу мови збігається з бурхливим розвитком мережі Internet, тому у 1993р. активно долучаєть до використання в мережі.

1.2. А чому назва – Java

Варто згадати, що не завжди ця мова мала звичну нам назву – Java.

Спочатку новій мові було дано ім’я Oak (від англ. «Дуб»). Справа в тому, що під вікном офісу Джеймса Гослінга ріс дуб, яким він дуже захоплювався, тому на момент створення мови, розробники вирішили обрати саме цю назву.

Коли ж компанія Sun Microsystems побачила, що мова добре себе зарекомендувала і її можна зробити загальнодоступною, було прийнято рішення змінити назву Oak. Необхідно було більш співзвучне та більш комерційне ім’я. З цього моменту мову стали називати – Java.

1.3. Види Java

Існує деякі види мови, які характеризуються особливостями застосування у тій чи іншій сфері [14]. Можна виділити наступні:

- Java Enterprise Edition або Java 2 Enterprise Edition (скорочено Java EE або J2EE) – використовується при розробці додатків для великих підприємств та корпорацій. Наприклад, при розробці додатків для банків, страхових компаній, роздрібних мереж і тому подібе.
- Java 2 Standard Edition (скорочено J2SE) – застосовується для розробки простих Java додатків. Використовуючи дану редакцію Java, можна створювати консольні додатки, аплети (applet), додатки з графічним інтерфейсом користувача, та ін.
- Java Micro Edition або Java 2 Micro Edition (скорочено Java ME або J2ME) – використовується під час створення додатків для мобільних пристроїв, кишенькових персональних комп’ютерів та інших малопотужних обчислювальних систем.
- Java Card – використовується для смарт-карт. Наприклад, банківські платіжні картки, SIM-картки у мобільному зв’язку, та ін.
- Java EE – це найпопулярніший вид Java. Характеризуються високими вимогами до вмінь та навичок програмістів та розробників. С таким видо мови працюють із середніми та великими підприємствами (банки, страхові компанії, великі роздрібні компанії). Щоб написати програми для таких підприємств, необхідно не тільки гарно володіти навичками написання програм мовою Java, а й кожного разу, працюючи на нове підприємство, вивчати всю його специфіку та особливості.

1.4. Принципи Java

Для мови Java характерний принцип **простоти**, **незалежності** від **архітектури** та **переносимості**. Один і той же написаний Java-код буде працювати однаково добре для різних платформ (Windows, Linux, MacOs). Така особливість мови, що «написано один раз, працює скрізь» – називається кросплатформенностю.

Як забезпечується **багатоплатформність** в Java?

Програми написані на Java зберігаються окремими файлами, що мають розширення .java. Наприклад, Program.java. Такий код є людиночетабельним. Його легко можна прочитати, відредагувати чи видалити.

Після запуску компілятора (javac), код програми перетворюється в байт-код (комбінацію 0 і 1). Код стає виключно машинозчитуваним. З'явиться ще один файл, який завжди буде мати розширення .class. У нашому випадку – Program.class. Далі Java Virtual Machine (JVM) – віртуальна машина Java – виконує байт-код.

Ще одним принципом Java є **дружній синтаксис**. Розробники не стали винаходити нову мову з нуля, а взяли все найкраще від мови програмування С і її прямої спадкоємці мови програмування С ++. Це стало у нагоді та сприяло швидкому зростанню популярності мови серед програмістів. Розробникам було набагато легше переходити на нову мову, адже не треба було абсолютно все вчити заново. Багато конструкцій запозичено і лишилось без змін.

Принцип **об'єктно-орієнтованості** мови [17]. ООП – це програмування за допомогою класів і об'єктів. Все навколо нас є об'єктом. Наприклад, машина – це об'єкт, людина – це об'єкт, кішка – це об'єкт, собака – це об'єкт, стіл – це об'єкт та інше.

У кожного об'єкта є властивості. Наприклад, властивості машини: модель, колір, розмір.

У кожного об'єкта є методи (тобто дії, які може робити об'єкт). Наприклад, методи машини: загальмувати, натиснути на газ та ін. Тоді сам клас – можна вважати готовим шаблоном.

Безпека. В мові програмування Java існує система винятків або ситуацій, коли програма зустрічається з неочікуваними труднощами, наприклад: операції над елементом масиву поза його межами або над порожнім елементом; читання з недоступного каталогу або неправильної адреси URL; введення недопустимих даних користувачем, та ін.

Одна з особливостей віртуальної машини полягає в тому, що помилки (виключення) не призводять до повного краху системи. Крім того, існують інструменти, які «приєднуються» до середовища періоду виконання. Кожного разу, коли сталося певне виключення, відбувається запис інформації про нього з пам'яті для налагодження програми. Ці інструменти автоматизованої обробки виключень надають основну інформацію щодо виключень у Java-програмах.

Принцип **автоматичного керування пам'яттю** [1]. У мові використовується автоматичний збирач сміття (GC – Garbage Collector) для керування пам'яттю під час життєвого циклу об'єкта. Розробник вирішує, коли створювати об'єкти, а віртуальна машина відповідальна за звільнення пам'яті після того, як об'єкт стає непотрібним. Коли до певного об'єкта вже не залишається посилань, збирач сміття може автоматично прибирати його із пам'яті. Проте, витік пам'яті все ж може статися, якщо код, написаний програмістом, має посилання на вже непотрібні об'єкти, наприклад на об'єкти,

що зберігаються у діючих контейнерах.

1.5. Цикл розробки та компіляції

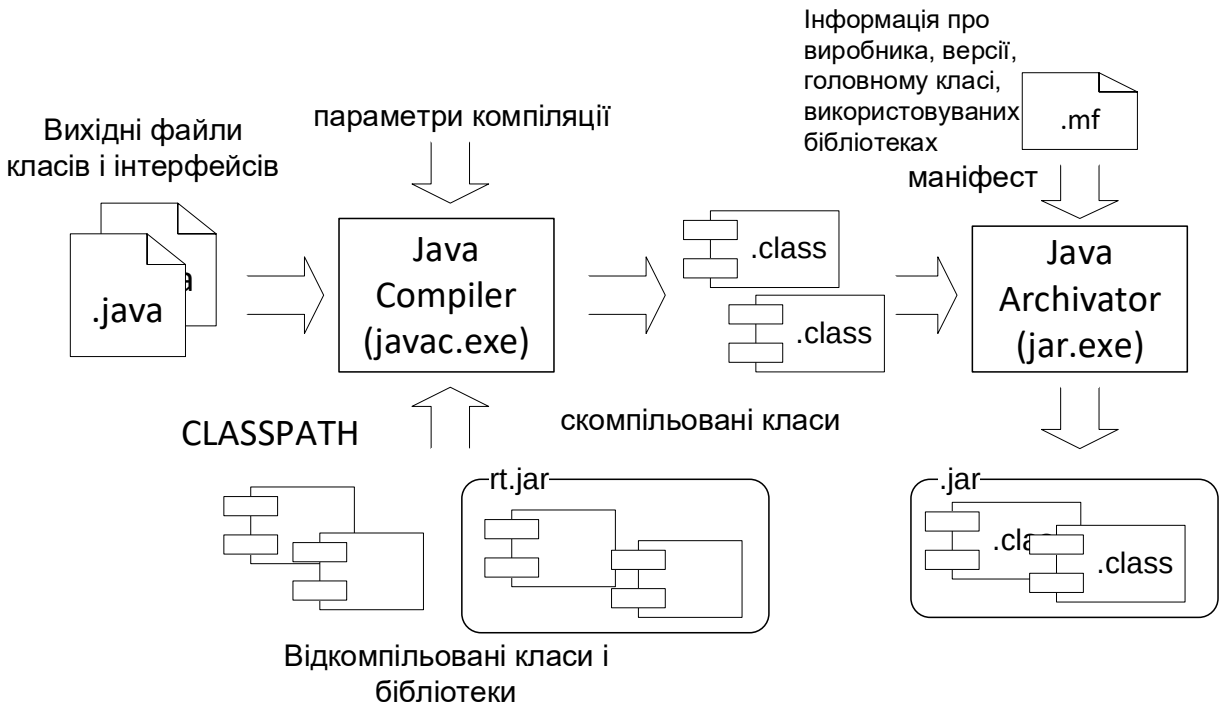

Рисунок 1.1 – Цикл розробки та компіляції програми Java

Програми Java зазвичай проходять п’ять стадій обробки, перш, ніж вони будуть виконані: редагування, компіляція, завантаження, перевірка байт-коду та виконання (Рис. 1.1):

1) Редагування файлу виконується в звичайному редакторі, при збереженні на диску файлу задають розширення *.java.

2) Компіляція програми здійснюється за допомогою команди javac [опції]; компілятор Java виконує трансляцію програми в байт-код (мова інтерпретатора Java). Якщо програма успішно компілюється, то компілятор створює файл з ім’ям *.class Всередині файлу з розширенням *.class міститься текст у байт-коді.

3) Завантаження програми в пам’ять, виконується завантажувачем класів, який зчитує файл *.class; завантажувач класів може завантажувати файли *.class двох типів: додатків та аплетів. **Аплети** – програми, з якими користувач з’єднується за допомогою браузера. **Додатки** – виконуються на локальному комп’ютері. Вони завантажуються в пам’ять і виконуються інтерпретатором Java. Завантажувач класів також викликається, коли браузер завантажує аплет Java. Кожен браузер має вбудований інтерпретатор, який виконує аплет.

4) Перш, ніж інтерпретатор Java, вбудований в браузер приступить до виконання аплету, байт-код перевіряється верифікатором. Ця перевірка гарантує, що байт-коди не будуть містити помилок і відсутні порушення вимог Java до безпеки.

5) Інтерпретація програми – це послідовне виконання байт-кодів, виконання дій, закладених в програму.

У разі виникнення помилки, необхідно повернутися до етапу редагування і повторити виконання всіх етапів.

1.6. Як працює Java компілятор

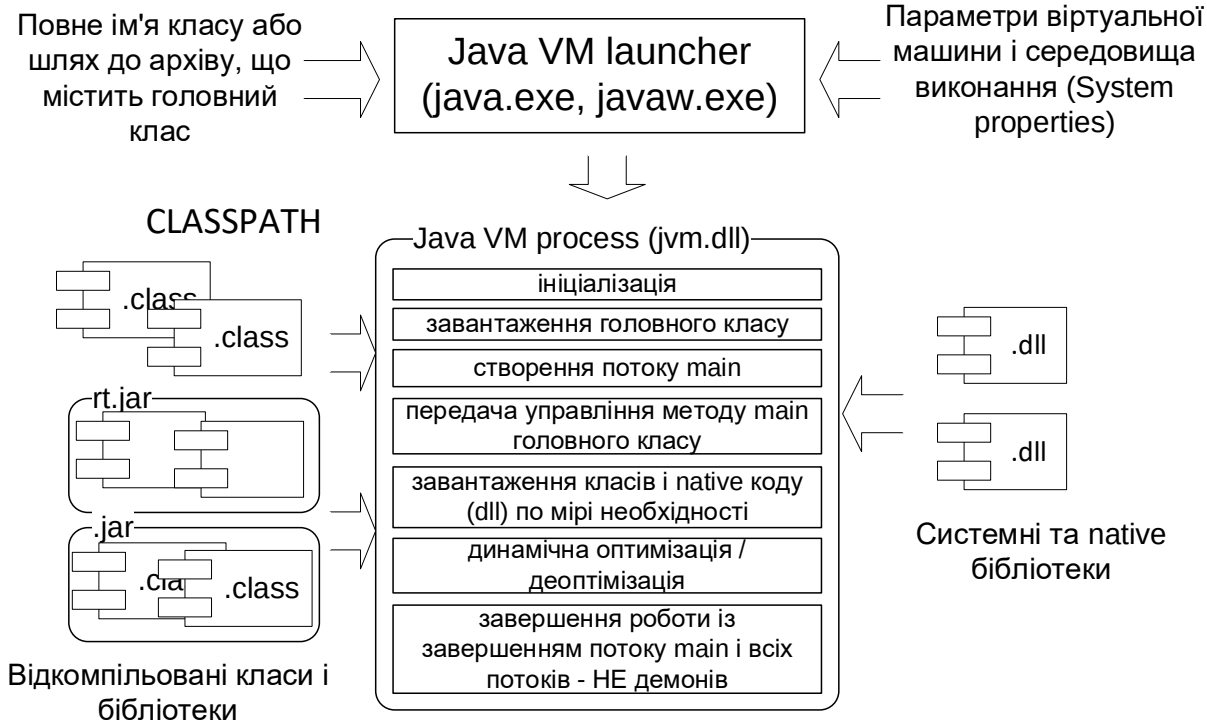

Рисунок 1.2 – Запуск та виконання програми Java

Текст програми – це вихідний код програми на мові Java. Додатки – це класи, які необхідно враховувати під час складання (бібліотеки). У підсумку отримуємо набір файлів з розширенням .class. Тобто, якщо використовуємо сторонні бібліотеки – необхідно вказати їх при складанні. Це можуть бути скомпільовані класи або зібрані підсистеми.

Не завжди для компіляції необхідно вказувати додаткові бібліотеки (наприклад, якщо програма складається з одного файлу). Але якщо все ж таки потрібно, то для цього компілятор java необхідно запустити з аргументом «-cp» (скорочення від -classpath). Після цього аргументу йде список бібліотек (jar файлів або файлів class).

У Java немає розмежування між зібраною бібліотекою, виконуваним додатком або ж підсистемою. Якщо необхідно зробити самостійну сутність – в єдиному файлі створюється jar файл. Наприклад, при створенні бібліотеки – це буде jar файл з набором класів, який може бути використаний іншими розробниками. Для підсистеми – це буде частина функціоналу, набір класів, винесених за рамки основного модуля, але таких, що використовуються в ньому (приватна бібліотека), та ін.

Виконання java-програми. Виконання класів працює схожим чином з компіляцією (використовуються навіть ті ж аргументи). Якщо після компіляції вийшло 10 класів, то виконуються тільки класи, що містять функцію main, інші класи повинні бути представлені як бібліотеки [17].

За замовчуванням, всі класи в поточному каталозі включені в шлях. Це означає, що якщо програма скомпільована, і було отримано безліч класів в одній папці, тоді можна запустити тільки головний клас, інші класи java спробує знайти сама в поточному каталозі (навіть якщо вони знаходяться у вкладених папках, java і туди загляне).

Такий підхід допустимий, коли класів незначна кількість, але при великих системах перерахування всіх класів неможливо (їх кількість може перевищувати тисячі). Тому краще використовувати не клас, а спеціально зібраний jar-файл.

Jar-файл – це ZIP архів (його можна розпакувати). Jar-файл повинен в собі містити набір класів та файл META, в якому описані характеристики цього jar-файлу.

1.7. Java Runtime Environment

Java Runtime Environment (JRE) – середовище виконання для Java, що є мінімальною реалізацією віртуальної машини, яка необхідна для роботи Java-додатків, без компілятора й інших засобів розробки. Складається з віртуальної машини JVM, та бібліотеки класів [1].

Набір інструментів розробки на Java (JDK), віртуальна машина (JVM) і середовище виконання (JRE) – утворюють разом потужну трійку компонентів платформи для розробки і запуску Java-додатків. Фактично, середовище виконання – це частина програмного забезпечення, призначена для запуску іншого програмного забезпечення. Воно містить бібліотеки класів, завантажувач класів і віртуальної машини Java. У цій системі: завантажувач класів відповідає за правильність завантаження класів і їх зв’язок з основними бібліотеками. JVM – відповідає за забезпечення Java-додатків ресурсами, необхідними для їх запуску та ефективної роботи на пристрої або в хмарному середовищі. JRE – в основному є контейнером для цих компонентів і відповідає за організацію їх діяльності.

Програмне забезпечення можна розглядати як набір шарів, розташованих поверх системного обладнання. Кожен шар надає служби, які будуть використовуватися (запитуватися) шарами над ним. Середовище виконання Java – це рівень програмного забезпечення, що працює поверх операційної системи, та надає додаткові служби, специфічні для Java.

Середовище виконання згладжує різноманітність операційних систем, гарантуючи, що програми Java можуть працювати практично на будь-який з них без змін. Також автоматичне керування пам’яттю – одна з найважливіших функцій JRE, яка гарантує, що розробникам не доведеться вручну управляти розподілом пам’яті.

1.8. Як JRE працює з JVM

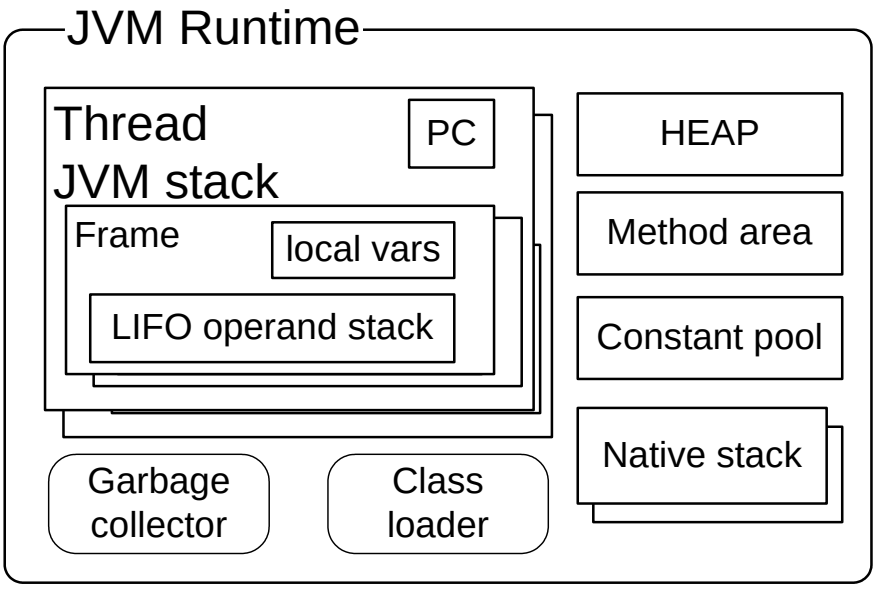

Рисунок 1.3 – Віртуальна машина Java: runtime

Віртуальна машина – програмне забезпечення, яке відповідає за виконання Java-програм. JRE – це програма, яка бере Java-код, об'єднує його з необхідними бібліотеками і запускає JVM для його виконання (Рис. 1.3). Середовище виконання містить програмне забезпечення та бібліотеки, які потрібні для роботи вашої програми. Наприклад, завантажувач класів Java є частиною JRE. Ця важлива частина програмного забезпечення завантажує скомпільований Java-код в пам'ять і з'єднує з відповідними бібліотеками. У такому багаторівневому поданні JVM створюється середовищем виконання Java.

1.9. Особливості мови Java

Виділимо ряд особливостей мови:

- заснована на синтаксисі C;
- розвинена система типів («абстракція», «інкапсуляція», «типізація»);
- притаманне одиночне спадкування класів і множинне спадкування інтерфейсів («ієрархія»):
- розвинена система пакетів («модульність»);
- потужна система обробки винятків;
- автоматичне прибирання сміття;
- забезпечення конкурентного доступу до даних при багатопоточності («паралелізм»);
- є доступ до метаінформації (reflection api);
- відсутність низькорівневого управління пам'яттю;
- розвинена бібліотека (з підтримкою «зберігання»).

1.10. Специфікація мов

Java була складена настільки вдало, що практично без змін використовується і до цього дня. Звичайно, було внесено велику кількість уточнень, більш докладних описів, були додані і деякі нові можливості (наприклад, оголошення внутрішніх класів), проте основні концепції залишаються незмінними [14].

Перша версія містила всього 8 стандартних бібліотек:

java.lang – базові класи, необхідні для роботи будь-якої програми (назва – скорочення від language);

java.util – містить багато корисних допоміжних класів, утилітів;

java.applet – класи для створення аплетів;

java.awt, java.awt.peer – бібліотека для створення графічного інтерфейсу користувача (GUI);

java.awt.image – додаткові класи для роботи із зображеннями;

java.io – робота з потоками даних (streams) і з файлами;

java.net – робота з мережею.

Таким чином, всі бібліотеки, які починаються з java, є стандартними. Всі інші (що починаються з com, org та інше), можуть змінюватися в будь-якій версії мови без підтримки сумісності версій.

1.11. Основні етапи розвитку Java

- Ранні версії (JDK Alpha, Beta, 1.0, 1.1): Цей період характеризується закладенням фундаменту мови, розробкою основних бібліотек і поступовим розширенням функціональності [17].
- Перехід до Java 2 (J2SE 1.2, 1.3, 1.4): Значне розширення можливостей мови, поява нових API, таких як Swing для створення графічних інтерфейсів, і суттєве підвищення продуктивності.
- Java 5 (J2SE 5.0): Революційна версія, яка принесла в Java такі важливі нововведення, як generic types, annotations, autoboxing, varargs та інші.
- Java SE 6, 7, 8: Подальший розвиток мови, удосконалення колекцій, паралельне програмування, лямбда-вирази та інші сучасні можливості.
- Модульна Java (Java 9 і новіші): Перехід до модульної системи, що дозволила покращити масштабованість і модульність великих проектів. Регулярні випуски нових версій з новими можливостями та покращеннями.

Розглянемо особливості різних версій мови, у хронологічній послідовності.

- **Java 1.0**, випущена у 1996 році, стала першою офіційною версією мови програмування Java. Вона була розроблена з акцентом на простоту, безпеку та кросплатформеність. Основні особливості Java 1.0:

Кросплатформеність (Write Once, Run Anywhere): Головна концепція Java – програми можна написати один раз і запускати на будь-якій платформі, що підтримує Java Virtual Machine (JVM). Це стало можливим завдяки байт-коду, який виконується на JVM.

Автоматичне управління пам'яттю: Java 1.0 впровадила збирач сміття (Garbage Collector), який автоматично звільняє невикористану пам'ять, запобігаючи витокам і спрощуючи роботу з пам'яттю для розробників.

Безпека: У Java 1.0 був впроваджений «пісочниця» (sandbox), що забезпечувала безпеку виконання коду, особливо при роботі з аплетами. Це дозволяло виконувати код у веб-браузерах із певними обмеженнями для запобігання загрозам.

Об'єктно-орієнтованість: Java 1.0 була повністю об'єктно-орієнтованою мовою, де все (від змінних до класів) трактувалося як об'єкти. Це дозволяло структурувати код у вигляді об'єктів і методів, що підвищувало гнучкість і повторне використання коду.

Бібліотеки та API: Java 1.0 включала базовий набір бібліотек для розробки додатків, включаючи API для роботи з графікою, мережею (java.net), потоками вводу/виводу (java.io) і багатопоточністю (java.lang.Thread).

Аплети: Однією з головних інновацій Java 1.0 були аплети – невеликі програми, які можна вбудовувати у веб-сторінки та виконувати прямо в браузері. Це забезпечувало інтерактивність, хоча з часом аплети втратили популярність через проблеми з безпекою та продуктивністю.

Відсутність багатьох сучасних концепцій: У Java 1.0 не було деяких особливостей, які з'явилися пізніше, таких як внутрішні класи, рефлексія та покращена модель обробки подій. Це була відносно проста мова порівняно з пізнішими версіями. Java 1.0 стала основою для подальших версій і швидко здобула популярність завдяки своїй простоті й універсальності.

- **Java 1.1**, випущена в 1997 році, принесла кілька важливих нововведень і вдосконалень у порівнянні з початковою версією Java 1.0. Основні особливості Java 1.1 включають:

Покращена модель обробки подій: Було запроваджено нову модель обробки подій (Event Delegation Model), що зробило роботу з графічним інтерфейсом користувача (GUI) більш ефективною і менш громіздкою.

Внутрішні класи (Inner Classes): Введено підтримку внутрішніх класів, що дозволяє визначати класи всередині інших класів. Це підвищує інкапсуляцію та полегшує організацію коду.

Потоки вводу-виводу (Java I/O): Додано нові класи для роботи з потоками вводу-виводу, такі як Reader і Writer, що спростило роботу з текстовими файлами та символами.

Рефлексія (Reflection API): З'явилася підтримка рефлексії, що дозволяє програмам динамічно досліджувати та взаємодіяти з класами, інтерфейсами та методами під час виконання.

Серійність об'єктів (Object Serialization): Введено можливість

серіалізації об’єктів, тобто збереження їх у потік для подальшого відновлення, що стало важливим для мережевих додатків і зберігання даних.

JDBC (Java Database Connectivity): Java 1.1 включала першу версію JDBC API, що дозволило розробникам працювати з базами даних через Java-програми.

Міжнародна підтримка (Internationalization): Додано нові можливості для роботи з різними мовами та локалями, що зробило Java більш універсальною для розробки програм для міжнародного ринку.

- **Java 1.2**, випущена в 1998 році, була значним оновленням мови та платформи Java, настільки, що її іноді називають «Java 2». Це оновлення принесло велику кількість нових функцій і поліпшень. Основні особливості Java 1.2:

Java Foundation Classes (JFC):

Swing: впровадження нового графічного інтерфейсу (GUI) під назвою Swing, який став альтернативою AWT (Abstract Window Toolkit). Swing дозволив створювати більш складні та кастомізовані інтерфейси користувача з покращеною кросплатформенною підтримкою.

Pluggable Look and Feel: Swing дозволяє змінювати вигляд і поведінку GUI, що дозволяє створювати інтерфейси з різними стилями.

Collections Framework: було додано колекційний фреймворк (Collections Framework), який включав нові інтерфейси та класи для роботи з наборами даних, такими як списки, множини та мапи. Основні інтерфейси включали List, Set, Map, а серед основних класів були ArrayList, HashSet, HashMap.

Віртуальна машина Java (JVM) і покращення продуктивності: з’явилася Just-In-Time (JIT) компіляція, яка значно покращила продуктивність виконання програм, компілюючи байт-код у машинний код під час виконання.

Безпека: у Java 1.2 було суттєво покращено безпекову модель. Зокрема, з’явилися політики безпеки та система управління дозволами, що дозволяло більш точно контролювати доступ програм до ресурсів системи.

Покращення потоків (Thread Improvements): удосконалено механізми управління потоками, що підвищило стабільність і продуктивність багатопотокових програм.

Java 2D API: Java 1.2 представила Java 2D API, що дозволило працювати з графікою більш високої якості, включаючи обробку шрифтів, текстур, градієнтів і векторної графіки.

Корпоративні технології: ведення Java Enterprise Edition (Java EE) відбулося на базі цієї версії, хоча воно продовжувалося і в наступних оновленнях. Це закріпило позиції Java як важливої технології для створення серверних і корпоративних додатків.

Поліпшена модель обробки подій: модель делегування подій, яка була введена у Java 1.1, продовжила вдосконалюватися і в Java 1.2, забезпечуючи більш ефективне управління подіями в графічних додатках.

Розширення API для роботи з файлами: вдосконалено API для роботи з

файлами і потоками, що спростило процеси вводу-виводу в додатках.

Java 1.2 була величезним кроком уперед для платформи Java, забезпечивши як нові функціональні можливості, так і значне поліпшення продуктивності та зручності розробки. Вона стала основою для всіх подальших версій Java.

- Особливості **Java 1.3.**

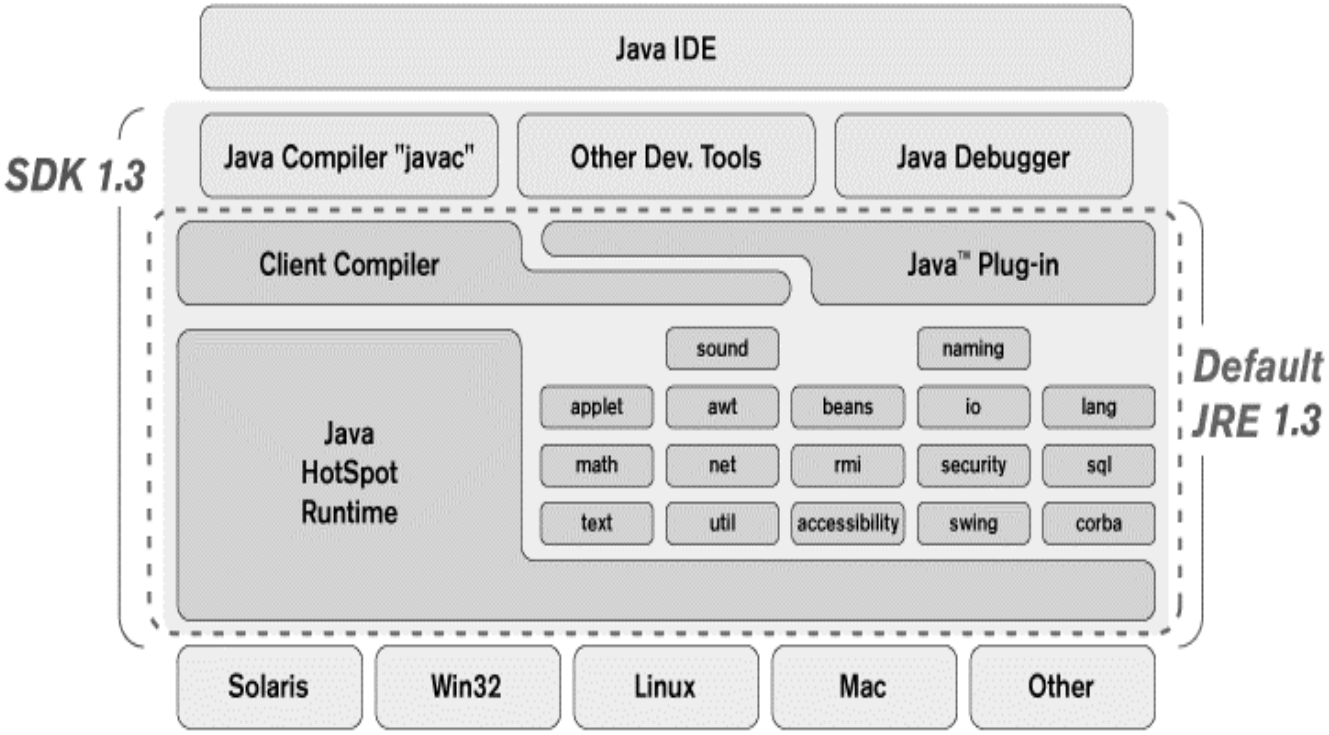

Рисунок 1.4 – Особливості версії Java 1.3

Java 1.3, випущена в 2000 році, була менш масштабним оновленням порівняно з Java 1.2, але все ж включала важливі покращення продуктивності та стабільності платформи. Основні нововведення Java 1.3 (Рис. 1.4):

Покращення продуктивності: Java 1.3 принесла численні оптимізації для збільшення швидкодії. Було вдосконалено Just-In-Time (JIT) компілятор, що призвело до підвищення швидкості виконання Java-додатків.

Оптимізовано Java 2D API для роботи з графікою, що підвищило швидкість рендерингу зображень і графічних елементів.

HotSpot JVM: вперше офіційно з'явилася HotSpot JVM як стандартна віртуальна машина. HotSpot JVM забезпечила кращу продуктивність за рахунок адаптивної оптимізації, що дозволяє програмам працювати швидше завдяки динамічній оптимізації найчастіше виконуваних частин коду.

Нове API для обробки звуку (Java Sound API): додано Java Sound API, яке надало розробникам можливість працювати з цифровими аудіо-даними. Це включало можливість програвання аудіофайлів, роботу з MIDI і обробку звукових ефектів.

Покращення у роботі з багатопоточністю: оптимізовано управління

потоками, що дозволило знизити навантаження на систему при роботі з багатопоточними додатками [17].

RMI (Remote Method Invocation) поліпшення: додано підтримку RMI-IIOP (Internet Inter-Orb Protocol), що дозволило Java-програмам легше взаємодіяти з CORBA-додатками, забезпечуючи кращу інтеграцію з корпоративними системами.

Java Naming and Directory Interface (JNDI) у стандартній бібліотеці: JNDI стало частиною стандартного комплекту API. Це дозволило програмам працювати з різними службами каталогів, такими як LDAP, більш ефективно.

Покращення в Swing: оптимізовано роботу з графічним інтерфейсом користувача (Swing), зменшено використання ресурсів системи та покращено загальну продуктивність GUI.

Оптимізація збирання сміття (Garbage Collection): введено нові стратегії збирання сміття, що покращило управління пам'яттю і зменшило затримки при виконанні великих додатків.

Усунення багів і поліпшення стабільності: виправлено значну кількість багів із попередніх версій, що зробило платформу більш надійною для використання в критично важливих додатках.

Java 1.3 стала важливою еволюцією платформи, зосередивши увагу на підвищенні продуктивності та стабільності, а також полегшенні інтеграції з іншими системами через розширене API.

- **Java 1.4**, випущена в 2002 році, стала одним із найважливіших оновлень платформи Java, додавши нові функціональні можливості та покращення, що значно підвищили продуктивність і безпеку. Основні нововведення Java 1.4:

Java New I/O (NIO): введено новий пакет java.nio, який включав канали (Channels), буфери (Buffers) та селектори (Selectors) для більш ефективної роботи з введенням-виведенням (I/O). NIO забезпечує неблокуюче введення-виведення, що покращило продуктивність серверних програм і додатків із високим навантаженням.

Logging API: додано новий API для ведення журналів (логування) у пакеті java.util.logging. Це спростило процес налагодження та моніторингу роботи програм завдяки можливості вести журнали подій і помилок різних рівнів важливості (INFO, WARNING, SEVERE тощо).

Regular Expressions (регулярні вирази): додала підтримку регулярних виразів через пакет java.util.regex, що дозволило ефективно обробляти текст, здійснювати пошук та обробку рядків за складними шаблонами.

Exception Chaining (ланцюгування винятків): введене нове поняття Exception Chaining дозволило вказувати первинну причину винятку при його обробці. Це зробило діагностику помилок точнішою, оскільки тепер можна було передати інформацію про початковий виняток разом із поточним.

Image I/O API: інтерфейс програмування додатків для спрощення роботи з графічними файлами. Це дозволило легко читати і записувати зображення у

форматах, таких як PNG, JPEG та інші.

Security Enhancements (покращення безпеки): було вдосконалено систему безпеки, включаючи підтримку 128-бітного SSL/TLS шифрування та поліпшене управління ключами та сертифікатами. Це зробило Java ще більш придатною для розробки захищених веб- і корпоративних додатків.

Додано нові криптографічні алгоритми та покращено роботу з цифровими підписами та шифруванням.

XML Processing (обробка XML): додано вбудовану підтримку XML через пакети JAXP (Java API for XML Processing). Це дозволило ефективно працювати з XML-документами, парсити їх і здійснювати перетворення XSLT без сторонніх бібліотек.

Assertions (твердження): введено механізм тверджень, що дозволяє розробникам вставляти перевірки в код для тестування правильності виконання програм. Це стало потужним інструментом для налагодження та виявлення логічних помилок під час розробки.

Покращення роботи з багатопоточністю: удосконалено управління потоками (threads), що включало оптимізацію продуктивності багатопоточних додатків і покращене управління синхронізацією потоків.

Покращення для міжнародної підтримки (Internationalization): Java 1.4 додала нові можливості для роботи з локалізацією та міжнародними налаштуваннями, зокрема підтримку різних мов та форматів даних.

Java 1.4 стала суттєвим кроком уперед, додавши багато ключових можливостей для розробки як настільних, так і серверних програм, підвищивши при цьому продуктивність, безпеку та зручність для розробників.

- **Java 1.5**, випущена в 2004 році (також відома як Java 5 або Tiger), принесла важливі оновлення, що зробили мову більш гнучкою та зручною. Основні нововведення:

- Дженерики (Generics): узагальнені типи забезпечили безпеку типів і зменшили кількість помилок при компіляції.
- Цикл «for-each»: полегшена ітерація по колекціях без використання індексів.
- Автоупаковка/Автораспаковка: автоматична конвертація примітивів у об’єкти і навпаки.
- Перечислення (Enums): новий тип даних для створення наборів констант.
- Змінна кількість аргументів (Varargs): дозволяє передавати різну кількість аргументів у методи.
- Статичний імпорт: використання статичних методів без зазначення класу.
- java.util.concurrent: нові класи для полегшення багатопотокової роботи.
- Анотації (Annotations): додали метадані для класів і методів.
- Покращена безпека: нові алгоритми шифрування.

Java 1.5 зробила розробку простішою, зберігаючи сумісність із попередніми версіями.

- **Java 1.6**, також відома як Java SE 6, була випущена в 2006 році й зосередилася на покращенні продуктивності та зручності розробки. Основні зміни:

- Покращена продуктивність: оптимізована JVM, що прискорило виконання програм.
- Scripting API: додана підтримка скриптових мов, зокрема JavaScript.
- Compiler API: можливість динамічної компіляції Java-коду всередині додатків.
- Покращення JDBC 4.0: спрощене підключення до баз даних.
- Web Services API: підтримка створення веб-сервісів через JAX-WS і JAXB.
- Поліпшення GUI: нові компоненти в Swing та AWT, System Tray API.
- Garbage Collection: оптимізовано роботу з пам’яттю для зменшення затримок.

Java 1.6 зміцнила платформу завдяки покращенню продуктивності та розширеній підтримці сучасних технологій.

- **Java 1.7**: Ключові нововведення.

Випущена у 2011 році, Java 1.7 (або Java SE 7) принесла значні покращення, спрямовані на спрощення розробки та підвищення продуктивності. Основні нововведення:

- Проект Coin: строки в switch: можливість використовувати рядки як значення в операторі switch.
- Try-with-resources: автоматичне закриття ресурсів після використання.
- Multi-catch: перехоплення кількох типів виключень в одному блоці.
- Інтеграція з динамічними мовами: підтримка мов, таких як Ruby та Python.
- NIO.2: нова бібліотека для вводу-виводу з покращеною роботою з файлами.
- Паралельне програмування: ефективніше використання багатоядерних процесорів.
- Діаманти: скорочення запису для створення колекцій.
- Двійкові літерали: запис чисел у двійковій системі.
- Покращення інструментів: нові можливості для розробників.

- **Java 8**: Новий виток розвитку.

Java 8 стала знаковим релізом, що суттєво змінив підхід до розробки на

Java. Ключові нововведення:

- Лямбда-вирази: функціональне програмування в Java, компактніший код для роботи з колекціями.
- Потоки (Streams API): простий і ефективний спосіб обробки даних, включаючи паралельні обчислення.
- Методи за замовчуванням та статичні методи в інтерфейсах: більша гнучкість у роботі з інтерфейсами.
- Новий API для роботи з датою та часом.
- Опціональні типи: безпечніша робота з null-значеннями.
- Виразніший код: завдяки лямбда-виразам та потокам.
- Вища продуктивність: особливо при використанні багатьох ядер процесора.
- Краща читабельність коду.
- Широка підтримка в інструментах розробки.

➢ Станом на початок 2024 року, найновішою версією Java є **Java 21**. Ця версія була випущена компанією Oracle і містить в собі безліч нових функцій та покращень. Кожна нова версія Java приносить з собою нові можливості, підвищення продуктивності та покращення безпеки. Ключові особливості Java 21:

- Vector API: ця нова функція дозволяє писати високопродуктивний код, який може бути ефективно компілюватися в машинний код.
- Virtual Threads: це експериментальна функція, яка дозволяє створювати велику кількість легковажких потоків, що може значно підвищити пропускну здатність додатків.
- Pattern Matching for switch: розширення можливостей патерн-матчингу для оператора switch.
- Foreign Function & Memory API: цей API дозволяє викликати зовнішні функції та працювати з зовнішньою пам’яттю без використання JNI.

Java 21 містить багато інших змін, таких як покращення збирача сміття, нові API для роботи з мережею та файловою системою, а також підвищення безпеки. Надає розробникам нові інструменти та можливості для написання більш ефективного та безпечного коду.

- Покращена продуктивність: багато змін у Java 21 спрямовані на підвищення швидкості виконання коду.
- Підтримка безпеки: регулярні оновлення Java виправляють вразливості, що робить ваші програми більш захищеними.
- Сумісність: Java 21, як правило, сумісна з попередніми версіями, тому перехід на нову версію зазвичай не вимагає значних змін у коді.

Завдання для самостійного виконання:

1). Робота у консолі. Отримати від користувача числа та вивести на екран

результат (орати **одне** із варіантів, приклад класу дивитись нижче):

- вивести на екран подвоєний добуток двох чисел збільшених на 1;
- перше число збільшити на 1, друге число зменшити на 1. Вивести на екран результат у вигляді «було-стало» для обох чисел;
- вивести на екран суму першого числа поділеного на трете, та другого числа помноженого на третє;
- вивести на екран частку від суму двох чисел поділену на їх різницю;
- вивести на екран квадратний корінь першого числа помножений на різницю першого і другого числа;
- вивести на екран різницю першого і другого числа підведену у степінь, що дорівнює третьому введеному числу;
- вивести на екран суму чисел першого, підведеного у степінь, що дорівнює третьому введеному числу, плюс друге число, поділене на трете введене число;
- вивести на екран суму двох чисел мінус їх різницю, підведену у степінь 3;
- вивести на екран корінь квадратний суми двох чисел мінус їх різницю;
- вивести на екран перше число помножене на різницю другого і третього, та вивести через знак тире друге число, як добуток першого і третього введеного користувачем числа.

Приклад роботи з двома класами та вводом-виводом інформації за допомогою Scanner. Створити новий проект (LabaFirstExpression). У лівому меню з’явився проект. У ньому створити 2 клас «Expression» і «Reader».

```
public class Expression {
 public static void main(String[] args) {
 Reader r = new Reader();
 r.Scan();
 r.i = count(r.i);
 r.k = count(r.k);
 System.out.println(r.i);
 System.out.println(r.k);
 }
 private static int count(int x) {
 x = x + 1;
 return x;
 }
}
```

Другий клас:

```
import java.util.*;

public class Reader {
 int i;
 int k;
 public void Scan() {
 System.out.println(«Введіть перше число:»);
 Scanner scn1 = new Scanner(System.in);
 i = scn1.nextInt();
 System.out.println(«Введіть друге число:»);
 Scanner scn2 = new Scanner(System.in);
 k = scn2.nextInt();
 }
}
```

2). Вивести на екран таблицю множення, в залежності від основи, яку ввів користувач (наприклад, таблицю множення на «3», якщо користувач ввів число «3»). Обмежити можливість введеня лише цифр від 1 до 9 за допомогою умовного оператора if – else.

3). Створити програму, яка виводитиме на екран менше за модулем з трьох введених користувачем дійсних чисел.

Контрольні запитання

1. Які основні проблеми мови C++ спонукали розробників створити нову мову програмування під час проєкту “Green”?
2. Що таке кросплатформеність у Java і яким чином її реалізовано за допомогою байт-коду та JVM?
3. Опишіть п’ять етапів життєвого циклу Java-програми від написання до виконання.
4. Яка роль JRE, JVM і JDK у виконанні Java-програм і чим ці компоненти відрізняються між собою?
5. Які основні типи Java (SE, EE, ME, Card) існують і для яких сфер застосувань вони призначені?
6. У чому полягають головні переваги Java 1.2, що дозволили називати її «Java 2»? Зазначте ключові нововведення.
7. Поясніть, як механізм серіалізації об’єктів вплинув на розробку мережевих програм у Java.
8. Що таке колекційний фреймворк (Collections Framework) і які основні інтерфейси/класи він включає?

Розділ 2. ЛЕКСИЧНА СТРУКТУРА JAVA

2.1. Лексична структура мови програмування Java

Рисунок 2.1 – Лексична структура Java

Як і будь-яка мова, мова програмування Java визначається:

- граматичними правилами (grammar rules), які вказують як будувати синтаксично вірні конструкції з елементів мови;
- семантичним визначенням (semantic definition), яке вказує на значення синтаксично вірних конструкцій.

Оригінальний текст програм складається із сукупності відступів, знаків закінчення рядка, табуляцій, ідентифікаторів, літералів, коментарів, операторів, роздільників і ключових слів (Рис. 2.1). Умовно код програми можна розділити на три частини: незначні символи, коментарі, лексеми.

Незначущі символи – символи, що не служать для генерації будь-якого коду або даних (відступи, табуляція, обмежувачі рядків, LF – символ нового рядка, CR – повернення каретки, CR + LF).

Коментарі – служать для пояснення тексту програми або генерації документації, не генерують жодного виконуваного коду.

Лексеми – це елементарні конструкції мови, з яких транслятор javac генерує виконуваний байт-код, що буде виконаний віртуальною машиною. Вони виступають будівельними блоками для більш комплексних конструкцій. Ідентифікатори, числа, оператори і спеціальні символи є прикладами лексем, які можуть бути використані для побудови високорівневих конструкцій таких як вираз, інструкції (statements), методи та класи [18-20].

Ідентифікатор. Ім’я в програмі називається ідентифікатором.

Ідентифікатори використовуються для позначення класів, методів, змінних та міток. В Java ідентифікатори є послідовностями символів в яких кожен символ є або буквою (A-Z і a-z) або цифрою (0-9). Однак перший символ в ідентифікатор завжди повинен бути буквою (не цифрою). Java програми пишуться в Unicode, тому в іменах ідентифікаторів допустимі символи з цього символьного набору. Також допустимі connecting punctuation character (такий як символ підкреслення _) і символи валюти (такі як $, ¢, ¥, або £) [4].

2.2. Особливості ідентифікаторів

- ідентифікатори чутливі до регістру (case sensitive), тому price, Price та PRICE – це різні ідентифікатори.

приклади вірних ідентифікаторів:

number, Number, sum_ $, bingo, $$ _ 100, _007, mål, grüß

приклади невірних ідентифікаторів:

48chevy, all @ hands, grand-sum

- два ідентифікатора збігаються тільки якщо збігаються Unicode коди всіх символів з яких вони складаються.

- символи в ідентифікатор можуть мати однакове накреслення, але відрізнятися за кодами:

LATIN CAPITAL LETTER A (A, \u0041) та GREEK CAPITAL LETTER ALPHA (A, u0391)

LATIN SMALL LETTER A (a, \u0061), and CYRILLIC SMALL LETTER A (a, \u0430)

- неможна використовувати **ключові слова.**

Ключові слова є зарезервованими і не можуть бути використані для позначення інших сутностей. Всі ключові слова в Java знаходяться в нижньому регістрі символів (їх 50 + посилання null і булеві літерали true і false). Зарезервовані слова не можуть бути використані в якості ідентифікаторів [5].

2.3. Escape послідовності Unicode

Сполучення символів, що складаються з зворотної косої риски (\), за якою слідує буква або набір цифр, називаються escape-послідовностями. Для представлення знаку нового рядка, одиночної лапки (‘), або деяких інших символів у символьній константі, необхідно використовувати escape-послідовності. Escape-послідовність розглядається як один символ і, отже, є допустимою символьною константою.

Escape-послідовністю зазвичай використовуються для вказівки дій, наприклад повернення каретки або табуляції, на терміналах і принтерах. Вони

також використовуються для позначення буквених уявлень недрукованих символів, а також символів, які зазвичай мають спеціальне значення, наприклад: подвійних лапок («).

Приклад:

```
char ch = ‘a’;
// Unicode for uppercase Greek omega character
char uniChar = ‘\u03A9’;
// an array of chars
char[] charArray = { ‘a’, ‘b’, ‘c’, ‘d’, ‘e’ };
```

2.4. Роздільники та оператори

Роздільники (або пунктуатори) – це лексеми що набувають значення в залежності від контексту в якому вони використовуються. Вони допомагають компілятору виконувати синтаксичний і семантичний аналіз програми. Залежно від контексту квадратні дужки, круглі дужки, і оператор точка можуть інтерпретуватися як оператори.

Роздільники – це спеціальні символи, які використовуються в службових цілях мови. В Java існують наступні 12 роздільників:

( ) { } [ ] … @ :: ; , .

Оператори – використовуються в різних операціях: арифметичних, логічних, бітових, порівнянні та присвоюванні. Нижче, на рисунку 2.2, представлені всі основні оператори Java.

Оператори відношення

Оператор	Назва	Приклад
==	дорівнює	a==b
!=	Не дорівнює	a!=b
>	Більше	a>b
<	Менше	a<b
>=	Більше чи дорівнює	a>=b
<=	Менше чи не дорівнює	a<=b

Логічні оператори

Оператор	Назва	Приклад
&&	«і» (and)	a&&b
||	«чи» (or)	a||b
^	«виключне або» (xor)	a^b
!	«не» (not)	!a

Рисунок 2.2 – Оператори в Java

2.5. Типи літералів

Літерали дозволяють задавати безпосередньо в коді програми значення для числових, символьних, логічних і рядкових виразів.

Типи літералів:

- цілий (**integer**) – за замовчуванням має тип даних **int**.

55 (десяткове), 555_555 (десяткове 555555, цей формат введений в Java 7).
7000000000L (десяткове тип long, ASCII символ L у будь-якому регістрі).
07 (вісімкове починається з нуля, де цифри 8 і – 9 заборонені).
0xFF, 0XFF або 0xff (шістнадцяткове, ASCII символ X в будь-якому регістрі, шістнадцяткові значення A, B, C, D, E і F, так само можуть записувати в будь-якому регістрі).
0b0101 або 0B0101 (двійкове, ASCII символ B у будь-якому регістрі).

- дійсне (**floating-point**) – за замовчуванням має тип даних **double**.

5.5 (тип double), 5.5f або 5.5F (тип float, ASCII символ F у будь-якому регістрі).
5. (5.0 тип double), .24f (0.24 тип float), 0.125e4 або 0.125E4 (тип double, ASCII символ E у будь-якому регістрі), 1600E-2 (тип double).

- логічний (**boolean**) – приклад: true або false.

- символьний (**charaster**) – може містити один символ з набору Unicode, укладений в одинарні лапки, спеціальна послідовність (керуючі символи) починається зі знака косої риски – \

- Керуючі символи:

\b – backspace BS – повернення на одну позицію.
\t – horizontal tab HT – табуляція.
\n – line feed LF – кінець рядка.
\f – form feed FF – кінець сторінки.
\r – carriage return CR – повернення каретки.
\»- лапки.
\’- одинарна лапка.
\\ **–** backslash \ – зворотна коса риска.
\Uxxxx – символ Unicode, де xxxx цифровий код символу Unicode.
\Xxx – символ кодової таблиці Latin-1, де xxx восьмеричний код символу.

Наприклад: ‘A’, \u0950, \b, \t, \333 та ін.

- строковий (**string**) – завжди має тип String і посилається на екземпляр класу String. Складається з 0 або більшої кількості символів, кожен

символ може бути представлений Unicode-послідовністю, так само може містити строковий літерал.
Наприклад: ««, «string», «symbol \u0950», «test \t test»

- **null**-літерал (null-literal) – це літерний типу посилання, причому це посилання **null** ні куди не веде. Приклад: null.

2.6. Коментарі у мові Java

Як і для більшості інших мов програмування, коментарі у мові Java ігноруються під час виконання програми. Таким чином, до програми можна додавати стільки коментарів, скільки потрібно, не побоюючись збільшити її обсяг. У мові Java є три способи виділення коментарів у тексті. Найчастіше використовуються дві косі риси //, причому коментар починається відразу за символами // і триває остаточно рядки.

Якщо потрібні більш довгі коментарі, кожен рядок можна починати символами. Хоча зручніше обмежувати блоки коментарів роздільниками /* і */

/** багатостроковий документувальний коментар java */
/* Традиційний багатостроковий коментар в стилі C */
// традиційний коментар в стилі C ++

- Коментарі не можуть бути вкладеними.
- символи /* і */ не мають спеціального значення в коментарях, що починаються з символів //.
- символи // не мають спеціального значення в коментарях, що починаються з символів /* або /**

/ Даний коментар / * // / ** закінчується тут: */*

2.7.. Оголошення і ініціалізація змінних

Ініціалізація змінних означає, що змінна запущена в роботу, їй присвоєно початкове значення, вона ініціалізована. Без присвоєння початкового значення змінна просто оголошена, а з початковим значенням – вона ще й ініціалізована.

тип_даних ім’я_змінної = значення ;

Приклади ініціалізації:

```
int a;
a = 32;
```

```
int a = 32;

double a = 3.2;

boolean b = (5<2);  // false
b = (5>2); /* true*/

int a, b, sum;
a = 32;
b = 8;
sum = a + b; // 40

String   str = «Hello JAVA!!!»;
char c =  ‘A’;
```

2.8. Масиви у Java

Масив – це структура даних, у якій зберігаються елементи одного типу. Його можна уявити, як набір пронумерованих осередків, у кожну з яких можна помістити якісь дані (один елемент даних в одну комірку). Доступ до конкретного осередку здійснюється через його номер. Номер елементу у масиві також називають індексом [18-20].

Однорідний масив, у всіх його осередках зберігатимуться елементи одного типу. Так, масив цілих чисел містить лише цілі числа (наприклад, типу int), масив рядків – лише рядки. У Java не можна помістити в першу комірку масиву ціле число, в другу String, а в третю – «собаку».

Оголошення масиву. Як і будь-яку змінну, масив Java потрібно оголосити. Зробити це можна одним із двох способів.

Вони рівноправні: int[] myArray; int myArray[];

Створення масиву. Як і будь-який інший об’єкт створити масив Java, тобто зарезервувати під нього місце в пам’яті, можна за допомогою оператора new.

тип_даних **[ ]** змінна_масива**;**
змінна_масива **= new** тип **[**розмір**];**

Приклад створення масивів:

```
int[] mas;
int mas[ ];
mas = new int[3];

int[] mas = new int[3];
```

```
int[] mas = { 3, 4, 10, 2, 5, 6};

int[][] mas = new int[3][4];
```

Треба звернути увагу, що після створення масиву за допомогою new, у його осередках записуються значення за замовчуванням. Для чисельних типів – це буде 0, для boolean – false, для посилальних типів – null.

Довжина масиву – це кількість елементів, під яку розрахований масив. Довжину масиву не можна змінити після створення. Індексація починається із 0. Отримати доступ до довжини масиву можна за допомогою змінної length. Вивести елементи масиву на екран можна декількома способами, наприклад, за допомогою циклу for.

Є одновимірні та багатовимірні масиви. Для роботи з масивами Java є клас java.util.Arrays. Основні операції над масивами: заповнення елементами (ініціалізація), вилучення елемента (за номером), сортування та пошук.

2.9. Типи значень у виразах

Приведення типів у виразах може відбуватись у тих випадках, коли змінній одного типу потрібно присвоїти значення змінної іншого типу. Приведення типів може бути явне та автоматичне (неявне) [5]. При явному приведенні типів сама операція приведення задається явним чином. При автоматичному приведенні типів потрібно, щоб виконувались дві умови:

- обидва типи мають бути сумісними;
- довжина вихідного типу (типу джерела) повинна бути менше довжини цільового типу (типу приймача).

Явне приведення типів дозволяє здійснювати присвоєння несумісних типів. Загальна форма явного приведення типів має вигляд: (цільовий_тип) значення, де цільовий_тип – це тип, в який потрібно привести вказане значення.

При обчисленні виразів аргументи оператора приводяться до одного типу, при цьому більш простий тип приводиться до більш складного шляхом руху:

byte-> short -> int -> long;
float -> double;
int-> float або double

Приклад:

```
int a;
double d;
d = -39.9203;
a = (int)d;   // a = -39
```

Автоматичне просування типів відбувається у виразах. При цьому,

значення, що фігурують у виразах, автоматично просуваються до типів з більшими діапазонами значень. При автоматичному просуванні типів у виразах:

- якщо один з операндів є int, то усі значення byte, short та char просуваються в int;
- якщо один з операндів є long, то весь вираз просувається до типу long;
- якщо один з операндів відноситься до float, то тип усього виразу буде також float;
- якщо один з операндів відноситься до double, то тип усього виразу буде також double.

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

```
double d = 31/5 + 6 * (1/5);
// d == 6.0 !!!!!!
double d = ((double) 31) / 5 + 6 * (1. / 5);
// d == 7.4 Ok!
```

Подібні помилки точності виникають через те, що на низькому рівні для представлення чисел з плаваючою крапкою застосовується двійкова система, однак для числа 0.1 не існує двійкового представлення, також як і для інших дрібних значень. Тому якщо в таких випадках зазвичай застосовується клас BigDecimal, який дозволяє обійти подібні ситуації.

2.10. Втрата інформації при перетвореннях

Звужуючі перетворення відбуваються при необхідності присвоєння змінної вужчого типу значення виразу більш широкого типу:

```
byte b;
b = 100000 / 20; // помилка, тому що результат не поміщається в тип byte
Type mismatch: cannot convert from int to byte
```

Стрілками на рисунку 2.3 показано, які перетворення типів можуть виконуватися автоматично [4]. Пунктирними стрілками показані автоматичні перетворення з втратою точності. Деякі перетворення можуть здійснюватися автоматично між типами даних однакової розрядності або навіть від типу даних з більшою розрядністю до типу з меншою розрядністю. Це такі ланцюжки перетворень як: int -> float, long -> float і long -> double

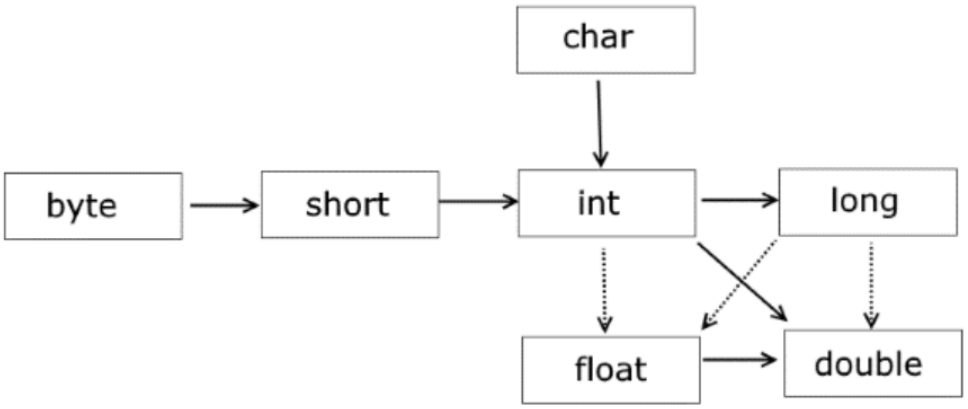

Рисунок 2.3 – Автоматично перетворення типів

Але при перетворенні можна зіткнутися з втратою інформації.

```
// НЕявне перетворення типів
int a = 2147483647;
float b = a; // від типу int до типу float
System.out.println (b); // 2.14748365E9
// явне перетворення типів
long a = 4;
int b = (int) a;
```

У всіх інших перетвореннях примітивних типів явно застосовується операція перетворення типів. Зазвичай це звужує перетворення (narrowing) від типу з більшою розрядністю до типу з меншою розрядністю.

При застосуванні явних перетворень можна зіткнутися з втратою даних, наприклад при перетворенні значень з плаваючою крапкою до цілочислового значення, відбувається урізання дробової частини.

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

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

2.11. Методи Java

Метод Java – це блок коду, що виконує певне завдання. По суті це як маленька програма всередині програми, яка може приймати вхідні дані,

виконувати якісь дії і повертати результат.

Використання методів:

- Структурування коду: розбивають програму на дрібніші, зрозумілі частини, що полегшує читання, розуміння та налагодження.
- Повторне використання коду: один і той самий метод можна викликати з різних місць програми, що дозволяє уникнути дублювання коду.
- Абстракція: приховують деталі реалізації, надаючи простий інтерфейс для використання.

Типи даних, які повертає метод: int, String, boolean або тип користувача.

Ім’я методу: унікальне ім’я, за яким метод викликається.

Параметри: змінні, що передаються методом виклику.

Тіло методу: блок коду, що містить інструкції, які виконуються під час виклику методу.

Види методів:

- статичні методи: належать класу, а не об’єкту, і викликаються на ім’я класу.
- нестатичні методи: належать об’єкту та викликаються через екземпляр класу.
- конструктори: спеціальні методи, які ініціалізують об’єкти.
- абстрактні методи: оголошуються в абстрактних класах та не мають реалізації.
- фінальні методи: не можуть бути перевизначені у підкласах.

Важливі моменти:

Перевантаження методів: можливість створення кількох методів з однаковим ім’ям, але різними параметрами.

Рекурсія: метод може викликати сам себе.

Область видимості: змінні, оголошені всередині методу, доступні лише всередині цього методу.

Методи використовуються практично у всіх програмах Java. Вони дозволяють:

- Розбивати великі завдання на менші підзавдання.
- Підвищувати повторне використання коду.
- Приховувати деталі реалізації.
- Робити код більш модульним та підтримуваним.

Оголошення методу:

public – тип доступу, що дозволяє виклик методу з іншого класу.

private – метод доступний лише всередині класу.

protected – метод доступний всередині класу та його нащадках.

Нехай заданий клас Main:

```
public class Main {
```

```
public static void foo() {…}
  public void methodName() {…} }
```

static означає, що метод статичний, він належить класу Main, а не конкретному екземпляру класу Main. Можемо викликати цей метод з іншого класу так:

```
ClassName.foo();
```

void означає, що це метод не повертає значення. Методи можуть повертати значення і воно має бути визначено при оголошенні методу. Однак, можна використовувати return просто для виходу з методу.

Якщо тип значення, що повертається не void, в тілі методу повинен бути хоча б один оператор return та вираз, де тип виразу повинен збігатися з типом значення, яке повертається. Цей оператор повертає результат обчислення виразу в точку виклику методу. Якщо тип значення – void, повернення з методу виконується або після виконання останнього оператора тіла методу, або в результаті виконання оператора return; (таких операторів у тілі методу може бути кілька).

Виклик методу примірника (об’єкта):

```
ClassName obj = new ClassName();
// створення об’єкта
obj.methodName();
// виклик методу використовуючи посилання
```

У об’єкта можна викликати статичний метод:

```
obj.foo(); //виклик статичного методу
```

Приклад оголошення методу, що повертає значення типу int – суму двох параметрів типу int:

```
int sum(int a, int b){
   int x;
   x = a + b;
   return x;
}
```

Виклик методу, наприклад, sum(5, 3), параметри 5 і 3 передаються в метод, як значення відповідно a і b, і оператор виклику методу sum(5, 3) – замінюється значенням, що повертається методом (8).

На відміну від мови С, в якому тип параметра, що задається при виклику, наводиться до типу параметра в оголошенні функції, тип параметра, що задається в Java, повинен суворо відповідати типу параметра в оголошенні методу, тому виклик методу sum(1.5, 8) приведе до помилки при компіляція програми.

Не статичні методи Java використовуються частіше, ніж статичні методи. Ці методи можуть належати будь-якому об’єкту, екземпляру класу, а не всьому класу. Не статичні методи можуть отримувати доступ та змінювати поля об’єкта.

Перевантажені методи. У мові Java в межах одного класу можна визначити два або більше методів, які спільно використовують те саме ім’я, але мають різну кількість параметрів. Коли це має місце, методи називають перевантаженими, а про процес говорять як про перевантаження методу (method overloading) [18-20].

Коли метод викликається, то за кількістю параметрів та/або їх типів середовище виконання визначає, яку саме версію перевантаженого методу треба викликати (тип значення, що повертається до уваги, не приймається, хоча, в принципі, він теж може відрізнятися у різних версій перевантажених методів).

Крім перевантаження існує заміщення, чи **перевизначення** методів (англ. overriding). Заміщення відбувається, коли клас нащадок (підклас) визначає певний метод, що вже є у батьківському класі (суперкласі), у такий спосіб новий метод замінює метод суперкласу. У нового методу підкласу повинні бути ті ж параметри або сигнатура, тип повертаємого результату, що і у методу батьківського класу.

2.12. Блоки коду

Блоки коду служать для угруповання декількома операторами (statement) і використовуються для задання тілу класів, методів, блоків статичної та динамічної ініціалізації, областей перехоплення виключень, блоків синхронізації, а також для вказання частин складних операторів (розгалуження, вибору, циклу).

Початок і кінець блоку задається роздільниками { }. Блоки можуть містити:

- оголошення і ініціалізатор локальних змінних;
- оголошення локальних типів (класів і інтерфейсів);
- вирази та оператори мови;
- вкладені блоки.

Область видимості локальних змінних обмежена межами блоку.

Завдання для самостійного виконання:

1). Обрати **два** пункти на вибір:

a) Ввести n рядків з консолі, знайти найкоротший рядок. Вивести цей рядок і його довжину, записати результат у файл.

b) Ввести n рядків з консолі. Впорядкувати і вивести рядки в порядку зростання їх довжин, а також вивести значення їх довжин, записати результат у файл.

c) Введіть n рядків з консолі. Вивести у консоль ті рядки, довжина яких менше середньої, а також вивести значення їх довжин, записати результат у файл.

d) У кожному слові тексту (використовувати читання з файлу), кожну k-ту букву замінити заданим користувачем символом. Якщо k більше довжини слова, коригування не виконувати.

e) У тексті кожну літеру замінити її номером у алфавіті (використовувати читання з файлу). В одному рядку друкувати текст з двома пробілами між літерами, у наступному рядку внизу під кожною буквою друкувати її номер.

f) З невеликого тексту (використовувати читання з файлу), видалити всі символи, крім пробілів, які не є літерами. Між усіма літерами залишити один пробіл.

g) З тексту видалити всі слова заданої довжини, що починаються на приголосну букву. Результат записати до файлу.

h) У тексті знайти всі пари слів, з яких одне є відображенням іншого (наприклад: сир-рис). Результати вивести у консоль та записати у файл.

i) Знайти і вивести у консоль, скільки разів повторюється в тексті кожне слово (використовувати читання з файлу).

j) Знайти, яких літер, голосних або приголосних, більше в тексті (використовувати читання з файлу).

2). Заповніть масив випадковими числами та виведіть максимальне, мінімальне та середнє значення. Для генерації випадкового числа використовуйте метод Math.random(), який повертає значення у проміжку [0, 1].

3). Реалізуйте алгоритм сортування бульбашкою для сортування масиву, отриманого у завданні 2.

Контрольні запитання

1. Що таке лексеми в Java і які три категорії синтаксичних елементів можна виділити в коді Java-програми?

2. Які правила слід дотримуватись при створенні ідентифікаторів у Java? Наведіть приклади правильних та неправильних ідентифікаторів.

3. Для чого використовуються escape-послідовності в Java і як представити символ нового рядка чи лапки в символьних та рядкових літералах?

4. Які є основні типи літералів у Java та приклади кожного з них?

5. У чому полягає різниця між оголошенням та ініціалізацією змінної? Наведіть приклади.

6. Що таке масив у Java? Як створити одновимірний і двовимірний масив та як дізнатися його довжину?

7. Які існують правила автоматичного і явного приведення типів у виразах Java? Які типи підтримують просування без втрати даних?

8. Чим відрізняється narrowing (звуження) типів від widening (розширення) і які ризики пов’язані з явним перетворенням типів?

9. Які є основні типи методів у Java і які особливості їх оголошення, виклику та перевантаження?

10. Що таке блоки коду в Java і для яких конструкцій мови вони використовуються? Як визначається область видимості змінних у таких блоках?

Розділ 3. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ JAVA

3.1. Інкапсуляція

Об'єктно-орієнтовані мови програмування складаються із об'єктів – одиниць програмування, які вміщують у собі певну кількість даних та способи їх взаємодії між собою. Крім того, що об'єкти є міні-програмками, вони також можуть взаємодіяти із іншими такими ж об'єктами. Завдяки цьому, при програмуванні, вдалося значно спростити перебіг написання коду – замість сотень стрічок коду, програміст може оперувати лише кількома об'єктами. Щоб створити об'єкт, спочатку створюють його «сутність» —тобто клас, який являє собою образ даних, та їх взаємодія, за допомогою якої об'єкт «втілюється» [3, 15, 16].

Крім того, об'єктно-орієнтоване програмування вирізняється ще трьома рисами: інкапсуляцією, успадкуванням та поліморфізмом.

Інкапсуляція – це можливість об'єкту обмежувати доступ до його складових іншими частинами програми. Завдяки цьому вдається уникнути помилок коду, коли програма змінює сторонні дані, що часто повністю порушує її роботу. Інкапсуляція – означає використання класів – таких типів, в яких крім даних описані підпрограми, що дозволяють працювати з цими даними. Підпрограми, інкапсульовані в клас, називаються методами. Поля даних і методи, задані в класі, називають учасниками класу (class members) [8].

Клас – це опис того, як буде влаштований об'єкт, який є екземпляром класу, а також які методи об'єкт може викликати.

Класи називають об'єктними типами.

Клас – це шаблон об'єкта, а об'єкт це екземпляр класу.

Створюється об'єкт за допомогою виклику спеціальної підпрограми, що задається в класі.

Ім'я конструктора збігається з ім'ям класу, екземпляр якого створюється.

Конструктор повертає посилання на створений об'єкт.

Загальна форма класу:

```
class  ім'я_класу
{
    тип змінної 1;
    тип змінної 2;
    тип змінної N;
    тип ім'я_ методу 1 (список_параметрів)
    {
  // тіло методу
    }
    тип ім'я_методу 2 (список_параметрів)
    {
```

```
   // тіло методу
    }
    тип ім'я_методу N (список_параметрів )
    {
   // тіло методу
    }
}
```

Створення об'єкту екземпляру класу:

об'єктний_тип змінна_класу = new ім'я_класу( );

Ініціалізайія полів даних класу:

змінна_класу.поле_даних_класу = значення;

Приклад створення об'єкту myBox типу Box

```
public class Box {
   double width;
   double height;
   double depth;
}
// цей клас оголошує об'єкт типу Box
public class BoxDemo {
   public static void main(String args[])
   {
      Box myBox = new Box();
      double vol;
      // присвоювання значень змінної екземпляру myBox
      myBox.width =10;
      myBox.height =20;
      myBox.depth =15;
      // розрахунок об'єму паралелепіпеду
      vol = myBox.width * myBox.height * myBox.depth;
      System.out.println(«Об'єм дорівнює « + vol);
   }
}
```

3.2. Методи в Java

Методи в Java – це комплекс виразів, сукупність яких дозволяє виконати певну операцію. Так, наприклад, при виклику методу System.out.println(), система виконує ряд команд для виведення повідомлення у консоль [10].

Якщо бачите в програмі якесь слово, а потім круглі дужки – значить це

метод. Метод – це сукупність команд, що дозволяє виконати деяку операцію в програмі. Іншими словами, метод – це деяка функція – щось, що вміє робити створенний клас. Але в Java у методів є головне завдання – вони повинні виконувати дії над даними об'єкта. Міняти значення даних об'єкта, перетворювати їх, виводити в консоль або робити з ними щось інше. Методи можуть приймати на вхід значення, які називають «параметрами методів».

```
тип_повертаємого_значення ім'я_методу (тип парметру 1, …, тип парамету N)
{
        тип_повертаємого_значення  змінна 1;
        …
        тіло методу
        …
        return змінна 1;
}
```

Приклад роботи методу addition:

```
public class Demo {
  public static void main(String args[])
  {
    double a = 2.3;
    double b = 3.4;
    double sum = addition(a, b)
    System.out.println(«Сума дорівнює « + sum);
  }
} // Сума дорівнює 5.7

double addition(double x, double y){
    double rez = x + y;
    return rez;
}
```

3.3. Додавання методу у клас

Перед використанням методу його необхідно викликати. Існує два способи для виклику методу, з поверненням значень або без [8, 15].

Алгоритм виклику методу досить простий. Коли програма виробляє в Java виклик методу, програмне керування передається викликаному методу. Потім він повертає управління клієнту в двох випадках, якщо:

- виконується оператор повернення return;
- досягнута закриваюча фігурна дужка закінчення методу.

Приклад додавання методу у клас:

```
// цей клас оголошує об’єкт типу Box
public class BoxDemo {
   public static void main(String args[])
   {
      Box myBox = new Box();
      double vol;
 /* присвоювання значень змінної екземпляру myBox */
      myBox.width =10;
      myBox.height =20;
      myBox.depth =15;
 /* розрахунок об’єму паралелепіпеду */
      vol = GetVolume();
      System.out.println(«Об’єм дорівнює « + vol);
   }
} /* Об’єм дорівнює 3000.0*/

public class Box {
  double width;
  double height;
  double depth;

  double GetVolume ()
  {
    double rez=width *
        height * depth;
    return rez;
  }
}
```

3.4. Додавання конструктора у клас

В Java конструктор ініціалізує об’єкт при його створенні. Його ім’я аналогічно імені класу, а синтаксис схожий з синтаксисом методу. Однак, на відміну від останнього, в конструкторі відсутнє значення, яке повертається. Як правило, конструктор може використовуватися для присвоєння початкового значення змінних екземпляра, що визначається класом, або для виконання будь-яких інших процедур запуску, необхідних для створення повністю сформованого об’єкта. Конструктори присутні у всіх класах, незалежно від їх вказівки, з огляду на те, що Java автоматично надає конструктор за замовчуванням, який ініціалізує всі змінні членів класу до нуля. Разом з цим, після того як було визначено власний конструктор, конструктор за замовчуванням більше не буде задіяний [3, 8, 16].

Приклад додавання конструктора у клас:

```
public  class BoxDemo
{
   public static void main(String args[ ])
   {
      Box  myBox = new  Box(10, 20,  15);

      double  vol = myBox .GetVolume();

     System.out.println(«Об’єм дорівнює « + vol);
   }
}

public  class Box
{
      double width;
      double height;
      double depth;

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

      double GetVolume ()
      {
            double rez=width*height*depth;
            return  rez;
      }
}
```

Ключове слово ***this*** – використовується для посилання на поточний клас з урахуванням методу або конструктора екземпляра. Таким чином можна посилатися на екземпляри класу, такі як конструктори, змінні і методи.

Ключове слово ***this*** виступає посиланням на об’єкт, для якого був викликаний метод, та використовується тільки в складі методів або конструкторів примірника:

- для диференціювання між змінними примірника і локальними змінними в разі, якщо у них однакові імена, в складі конструктора або методу;

- виклику конструктора одного типу (параметризовані конструктори або конструктори за замовчуванням) з іншого в складі класу. Даний процес також носить назву явного виклику конструктора.

3.5. Поліморфізм

Поліморфізм (означає «багато форм») – властивість, яка дозволяє використовувати один і той самий інтерфейс для загального класу дій .

Конкретна дія визначається конкретним характером ситуації. Наприклад: існує програма, яка вимагає застосування трьох типів стеків:

- один стек для цілочисельних значень,
- другий стек для значень з плаваючою точкою,
- третій стек для символів.

Алгоритм реалізації кожного з цих стеків залишається незмінним, незважаючи на відмінність даних, що в них зберігаються.

Поліморфізм – один інтерфейс, кілька методів.

Перевантаження методів, є одним із способів підтримки поліморфізму в Java, такий випадок, коли в класі присутні два і більше методи з однаковим ім'ям, але різними параметрами. Даний процес відрізняється від перевизначення методів. При перевизначенні методів, метод характеризується аналогічним ім'ям, типом, числом параметрів, та ін.

Приклад перевантаження методів:

```
public  class  Demo
{
   public static void main(String args[ ])
   {
        MyMath obj  = new   MyMath( );
        double  rezDouble = obj .sum(2.5,  3.2);
        int  rezInt = obj .sum(2,  3);

        System.out.println(« Сума дорівнює « +
                                  rezDouble );
        System.out.println(« Сума дорівнює « + rezInt);
   }
}
/*
    Сума дорівнює 5.7
    Сума дорівнює 5
*/

public  class  MyMath
{
```

```
    ……….
    double sum (double  a,  double b)
    {
        double  rez =  a+b;
        return  rez;
    }

    int  sum (int  a,  int  b)
    {
        int  rez =  a+b;
        return  rez;
    }
     ……
}
```

3.6. Успадкування в Java

Успадкування дозволяє повторно використовувати код одного класу в іншому класі, тобто ви можете успадкувати новий клас від вже існуючого класу. Головний успадкований клас називають батьківським класом, або суперкласом. Наслідуючий клас називають дочірнім класом, чи підкласом. Підклас успадковує всі поля та властивості суперкласу, а також може мати свої поля та властивості, відсутні в класі-батьку (Рис. 3.1) [15].

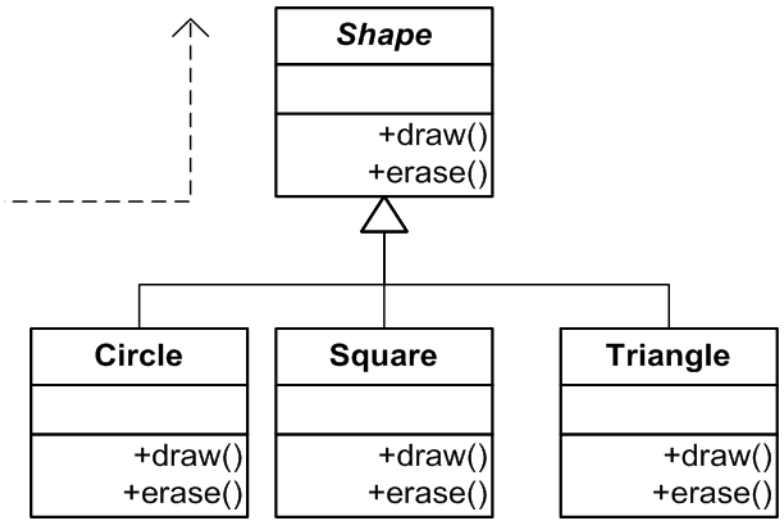

Рисунок 3.1 – Діаграма наслідування

Спадкування класів (inheritance) один із суттєвих атрибутів об'єктно-орієнтованого програмування. Воно дозволяє будувати нові класи з урахуванням існуючих, додаючи у них нові можливості чи перевизначаючи існуючі. Щоб наслідувати клас, досить просто вставити визначення одного

класу в інший з використанням ключового слова extends.

Успадковані поля можуть бути використані безпосередньо, як будь-які інші поля. Можна оголошувати поле в дочірньому класі з тим самим ім'ям, що й у суперкласі, приховуючи його таким чином (але це не рекомендується). Також, можна оголосити нові поля у підкласі, яких немає у суперкласі. Успадковані методи можна використовувати безпосередньо, написавши в підкласі новий метод, який має ту ж саму сигнатуру, що й метод суперкласу. Це буде мати назву – перевизначенням методу. Дозволяється писати новий статичний метод у підкласі, який має ту ж сигнатуру, що і метод суперкласу, таким чином приховуючи його (тобто метод суперкласу буде видно всередині підкласу тільки через ключове слово super). Можна оголосити нові методи у підкласі, яких немає у суперкласі, написавши конструктор підкласу, який викликає конструктор суперкласу, неявно чи за допомогою ключового слова super. Підклас не успадковує закриті члени батьківського класу, наприклад, позначені модифікатором private [3, 5].

Приклад спадкування:

```
public class Box
{
      double width;
      double height;
      double depth;
      Box (double w, double h, double d)
      {
    width = w;
    height = h;
    depth = d;
      }
      double GetVolume ()
      {
            double rez = width* height*depth;
            return rez;
      }
}

// Розширення класу Box
class BoxColor extends Box
{
     String color; // колір паралелепіпеду
    // конструктор BoxColor
    BoxColor(double w, double h, double d, String c)
    {
       width = w;
```

```
        height = h;
        depth = d;
        color= c;
    }
}

public class BoxExtends
{
    public static void main(String args[ ])
    {
        BoxColor myBox = ne BoxColor (10, 20, 15, “red”);

        System.out.println(«Колір фігури « + myBox.color);
        System.out.println(«Об’єм дорівнює « + myBox.GetVolume ());
    }
}
// Колір фігури red
// Об’єм дорівнює 3000.0
```

Правила спадкування:

1. Унаслідується тільки один клас. У Java не підтримуються спадкування декількох класів. Один клас – один батьківський. Не можна успадковувати самого себе!

2. Успадковується все крім приватних змінних і методів. Клас-спадкоємець буде мати доступ до всіх змінним і методам суперкласу. Але усі методи і змінні, помічені модифікатором **private**, не доступні класу-спадкоємцю.

3. Є можливість переробити метод класу-батька. Уявімо, що успадковується клас, але не все що успадкували необхідно. Припустимо що необхідно, аби певний метод працював не так, як в суперкласі. Для того, щоб перевизначити метод класу-батька, використовується конструкція @Override.

4. Виклик методу батька через ключове слово super. Якщо необхідно змінити метод батьківського класу зовсім трохи – буквально дописати пару рядків, тоді у методі викликається батьківський метод за допомогою ключового слова super.

5. Заборонено успадкування. Щоб заборонити успадкування класу, перед ним ставиться модифікатор final.

Використання ключового слова super.

Існують дві форми використання ключового слова super:

- для виклику конструктора суперкласу;
- для доступу до члена суперкласу з підкласу.

Такий доступ потрібен в ситуаціях, коли ім’я члена підкласу співпадає з

іменем члена суперкласу, до якого потрібно доступитись з підкласу. У цьому випадку ім'я члену даних підкласу перекриває ім'я суперкласу.

Приклад використання ключового слова super:

```
   // Використання super для попередження укриття імен
class A
      int i;
}
// Створення підкласу за допомогою розширення класу A
class B extends A {
     int i;     // змінна i скриває змінну i у класі A
     B(int a, int b) {
          super.i = a;  // i у класі A
          i = b;    // i у класі B
     }
void show () {
     System.out.print(«i у суперкласі: « + super.i);
     System.out.print(“;  i у підкласі: « + i);
}
class UseSuper {
   public static void main(String args[]) {
        B subOb  new B(1, 2);
        subOb.show();
}
}   /* i у суперкласі: 1; i у підкласі : 2 */
```

Одна з форм super діє подібно ключовим словом this, за винятком того, що super завжди посилається на суперклас підкласу, в якому super використаний.

Як здійснюється виклик з підкласу конструктора суперкласу?

Конструктор суперкласу (базового класу) може бути викликаний з підкласу (похідного класу). Цей виклик здійснюється за допомогою ключового слова super. Виклик конструктора суперкласу повинен бути здійснений в тілі конструктора підкласу першим [8, 10, 15].

Загальна форма виклику конструктора суперкласу з конструктора підкласу наступна: super(parameters); тут parameters – перелік параметрів, які отримує конструктор. Якщо конструктор немає параметрів, то конструктор суперкласу викликається як super().

Приклад виклику конструктора суперкласу:

```
// для ініціалізації своїх атрибутів об'єкта Box.
```

```
class  BoxWeight extends Box {
       double weight; // вага паралелепіпеду
       // ініціалізація змінних width, height та depth за допомогою super()
       BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // виклик конструктора суперкласу
    weight = m;
        }
}
```

Тут у прикладі, super () з аргументами w, h та d – приводить до виклику конструктора Box (), який ініціалізує width, height та depth.

Тепер клас BoxWeight НЕ ініціалізує ці значення самостійно. Йому потрібно форматувати тільки своє унікальне значення weight.

В результаті ці значення можуть залишатися приватними значеннями класу Box.

Приклад перевизначення методів:

```
class B extends A {
   int k;
   B(int a, int b, int c) {
      super (a, b); //виклик конструтору суперкласу
      k = c;
}
//перевизначає метод show() класу A
   void show () {
      System.out.println(«k: «+ k);
}
class  Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); //метод show() класу B
    }
}  /*k:  3*/

class A
{
    int i,  j;
    A(int a, int b)
    {
       i = a;
       j = b;
     }
```

```
    void show ()
    {
        System.out.print(«і та j: «+ i +» «+j);
    }
}
```

Якщо в ієрархії класів ім’я і сигнатура типу методу підкласу збігається з атрибутами методу суперкласу, кажуть, що метод підкласу перевизначає метод суперкласу. Ще одна форма поліморфізму

3.7. SOLID принципи ООП

SOLID – це абревіатура п’яти основних принципів проектування в об’єктно-орієнтованому програмуванні. Абревіатура SOLID була запропонована Робертом Мартіном. Ці принципи дозволяють будувати на базі ООП масштабовані і супроводжувані програмні продукти зі зрозумілою бізнес-логікою [10].

1). *Принцип єдиною обов’язки* / відповідальності (single responsibility principle / SRP) Принцип декларує, що кожен об’єкт повинен мати один обов’язок і цей обов’язок повинна бути повністю інкапсульовані в клас, а всі його сервіси повинні бути спрямовані виключно на забезпечення цього обов’язку. Дотримання принципу полягає зазвичай в декомпозиції складних класів, які роблять відразу багато речей, на прості, відповідальність яких дуже спеціалізована. Але також і об’єднанні в окремий клас однотипної функціональності, яка може виявитися розподіленої за багатьма класами, може розглядатися як дотримання цього принципу. Проектування класів з спрямованістю на забезпечення єдиного обов’язку спрощує подальші модифікації і супроводження, так як простіше розібратися в одному блоці функціональності, ніж розплутувати складні взаємозв’язки між різними функціональними блоками. Також при модифікації логіки в одному місці додатка знижуються ризики виникнення проблем в інших «несподіваних» його місцях.

Дотримання SRP вельми корисна практика з точки зору повторного використання коду. Складні об’єкти з комплексними залежностями зазвичай дуже складно використовувати повторно, особливо якщо потрібна тільки частина реалізованого в них функціоналу. А невеликі класи з чітко окресленим функціоналом, навпаки, простіше використовувати повторно, так як вони не надмірні і рідко тягнуть за собою істотний обсяг залежностей.

2). *Принцип відкритості / закритості* (open-closed principle / OCP)

Принцип декларує, що програмні сутності (класи, модулі, функції), повинні бути відкриті для розширення, але закриті для зміни. Це означає, що ці сутності можуть міняти свою поведінку без зміни їх вихідного коду.

Дотримання принципу OCP полягає в тому, що програмне забезпечення

змінюється не через зміну існуючого коду, а через додавання нового коду. Тобто створений спочатку код залишається «незайманим» і стабільним, а нова функціональність впроваджується або через успадкування реалізації, або через використання абстрактних інтерфейсів і поліморфізм.

3). *Принцип підстановки Лісков* (Liskov substitution principle / LSP), це принцип в формулюванні Роберта Мартіна декларує, що функції, які використовують базовий тип, повинні мати можливість використовувати підтипи базового типу не знаючи про це. Визначення Барбари Лісков більш формальне і помітно складніше для сприйняття.

Дотримання принципу LSP полягає в тому, що при побудові ієрархій успадкування створювані спадкоємці повинні коректно реалізовувати поведінку базового типу. Тобто якщо базовий тип реалізує певну поведінку, то це поведінка повинна бути коректно реалізовано і для всіх його спадкоємців.

Спадкоємець класу доповнює, але не замінює поведінку базового класу. Тобто в будь-якому місці програми заміна базового класу на клас-спадкоємець не повинна викликати проблем. Якщо з якихось причин так не виходить, то найімовірніше має місце або некоректна реалізація, або невірно обрана абстракція для наслідування.

4). *Принцип поділу інтерфейсу* (interface segregation principle / ISP), в формулюванні Роберта Мартіна: «клієнти не повинні залежати від методів, які вони не використовують». Принцип поділу інтерфейсів говорить про те, що занадто «складні» інтерфейси необхідно розділяти на більш дрібні і специфічні, щоб клієнти маленьких інтерфейсів знали тільки про методи, які необхідні їм у роботі. У підсумку, при зміні методу інтерфейсу не повинні змінюватися клієнти, які цей метод не використовують.

У чомусь принцип поділу інтерфейсу перегукується з принципом єдиної відповідальності – інтерфейси не повинні бути надмірно «товстими», якщо раптом в додатку формується надто об’ємний інтерфейс, то є висока ймовірність, що так відбувається через те, що в цьому інтерфейсі занадто багато різних відповідальностей, а значить найлогічніше провести декомпозицію складного інтерфейсу на кілька простих.

Принцип поділу інтерфейсу знижує складність підтримки і розвитку програми. Чим простіше і мінімалістичний використовуваний інтерфейс, тим менше ресурсномісткою є його реалізація в нових класах, тим менше причин його модифікувати, але і в разі модифікації вона буде значно простіше.

5). Принцип інверсії залежності (dependency inversion principle / DIP), декларує, що модулі верхніх рівнів не повинні залежати від модулів нижніх рівнів, а обидва типи модулів повинні залежати від абстракцій; самі абстракції не повинні залежати від деталей, а ось деталі повинні залежати від абстракцій.

Дотримання принципу інверсії залежності «змушує» реалізовувати високорівневі компоненти без вбудовування залежності від конкретних

низькорівневих класів, що, наприклад, сильно спрощує заміну використовуваних залежностей, як по бізнес-вимогам, так і для цілей тестування. При цьому залежність формується не від конкретної реалізації, а від абстракції – реалізованого залежністю інтерфейсу.

Наприклад, необхідно реалізувати зберігання документів в веб-додатку. На перший погляд, здається логічним додати залежність від модулів роботи з файловою системою безпосередньо в клас, який відповідає за високорівневу роботу з цими документами. Але в перспективі така залежність може створити проблеми – наприклад, якщо буде потрібно зберігати дані не тільки на диску, але і в хмарі. Якщо залежність впроваджена від реалізації, то зтикаємось з необхідністю її переробки. Якщо ж залежність виведена на рівень абстракції (інтерфейсу), то достатньо буде реалізувати функціонал роботи з хмарою, відповідний раніше створеному інтерфейсу роботи з файлами.

3.8. Пакети в Java

Пакети це контейнери класів, які використовуються для збереження ізольованості простору імен класу. Наприклад, пакет дозволяє створити клас з іменем List, який можна зберігати в окремому пакеті, не турбуючись про можливі конфлікти з іншим класом List, що зберігаються, в іншому місці.

Визначення пакету (package) – package MyPackage;

Оператор package визначає простір імен, в якому зберігаються класи. Для зберігання пакетів використовується каталоги файлової системи.

```
package пакет1[.пакет2[.пакет3]];
```

Пакети зберігаються в ієрархічній структурі і явно імпортуються у визначення нових класів і слідують безпосередньо за оператором package:

```
import пакет1[.пакет2].(имякласса1*);
```

Наприклад:

```
import java.util.Date;
import java.io.*;
```

3.9 Керування доступом

Специфікатори доступу Java:

- public (загальнодоступний),
- private (приватний),
- protected (захищений).

Коли член класу змінюється специфікатором доступу public, він стає доступним для будь-якого іншого коду. Коли член класу вказаний як private,

він доступний тільки членам цього ж класу. Якщо потрібно, щоб елемент був видимий тільки класам, які є безпосередніми підкласами даного класу, елемент повинен бути оголошений як protected. При відсутності специфікатору доступу за замовчуванням член класу вважається загальнодоступним всередині свого власного пакета, але недоступним для коду, розташованого поза цим пакетом [10].

Приклад керування доступом:

```
public class Box
{
    private double width;
    public void setWidth(double width)
    {
       this.width = width;
    }
    public double getWidth()
    {
       return width;
    }
}

public class BoxDemo
{
   public static void main(String args[])
   {
     Box objBox= new Box();
     objBox.width =  10;  // нема доступу
     objBox.setWidth(10);
     double w = objBox.width;
     double w = objBox.getWidth(); // нема доступу
   }
}
```

Приклад керування доступом:

```
public class Box {
    protected double width;
    public void setWidth(double width) {
       this.width = width;
    }
    public double getWidth() {
          return width;
    }
```

```
}
public class BoxExt extends Box {
        Box objBox= new Box();
        objBox.width = 10;
        objBox.setWidth(10);
        double w = objBox.width;
        double w = objBox.getWidth();
}

public class BoxDemo
{
        Box objBox= new Box();
        objBox.width = 10; // нема доступу
        objBox.setWidth(10);
        double w = objBox.width; // нема доступу
        double w = objBox.getWidth();
        BoxExt objBoxEx = new BoxExt ();
        objBoxEx.width = 10; // нема доступу
        objBoxEx.setWidth(10);
        double w = objBoxEx.width; // нема доступу
        double w = objBoxEx.getWidth();
}
```

3.10. Модифікатор **static**

Щоб створити член класу, який може використовуватися самостійно, без посилання на конкретний екземпляр, в початок його оголошення потрібно помістити ключове слово static. Коли член класу оголошений як static (статичний), він доступний до створення будь-яких об'єктів його класу без посилання на об'єкт [10, 15].

Статичними можуть бути оголошені як методи, так і змінні.

Найбільш поширений приклад статичного члена – метод Main(). Цей метод оголошують як static, оскільки він повинен бути оголошений до створення будь-яких об'єктів.

На static методи, накладаються обмеження:

- вони можуть викликати тільки інші статичні методи;
- вони повинні здійснювати доступ тільки до статичних змінних;
- вони жодним чином не можуть посилатися на члени типу this або super

3.11.. Модифікатор **final**

Змінна може бути оголошена як final – остаточна або константна. Це

дозволяє запобігти зміні вмісту змінної. Змінна типу final повинна бути ініціалізована під час її оголошення.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
```

Прийнято ідентифікатори всіх змінних типу final записувати прописними (великими) буквами. Воно може застосовуватися до класів, методів, змінних (у тому числі аргументів методів).

Для класу це означає, що клас не зможе мати підкласів, тобто заборонено успадкування. Це корисно при створенні незмінних об'єктів. Слід зазначити, що з абстрактним класам (з ключовим словом abstract), не можна застосувати модифікатор final, так як це взаємовиключні поняття.

Для методу final означає, що він може бути перевизначений у підкласах. Це корисно, коли потрыбно, щоб вихідну реалізацію не можна було перевизначити.

Для змінних примітивного типу final означає, що одного разу надане значення не може бути змінено.

Для змінних посилання final означає, що після присвоєння об'єкта, не можна змінити посилання на цей об'єкт. Це важливо! Посилання змінити не можна, але можна змінити стан об'єкта.

3.12. Модифікатор **abstract**

Клас, який під час оголошення містить в собі ключове слово abstract, відомий як абстрактний клас. Абстрактні класи в Java можуть містити або не містити абстрактні методи, тобто методи без тіла (public void get();). Якщо клас має хоча б один абстрактний метод, то клас повинен бути оголошений абстрактним. Якщо клас оголошений абстрактним, його не можна реалізувати [8, 10, 15].

Щоб використовувати абстрактний клас, потрібно наслідувати його з іншого класу, забезпечити реалізацію абстрактних методів в ньому. Якщо успадковується абстрактний клас, то необхідно забезпечити реалізацію всіх абстрактних методів в ньому.

Абстрактний метод. Якщо потрібно, щоб клас містив конкретний метод, але щоб фактична реалізація цього методу визначалася дочірніми класами, треба оголосити метод в батьківському класі як абстрактний.

Ключове слово abstract використовується для оголошення абстрактного методу. Ключове слово abstract розміщують перед ім'ям методу під час його оголошення. Абстрактний метод містить сигнатуру методу, але не містить тіла методу. Замість фігурних дужок у абстрактного методу буде крапка з комою (;) наприкінці.

Приклад використання модифікатору **abstract**:

```
public abstract class Shape {
      // Малювати геометричну фігуру
     public abstract void draw();
     // реальний метод
     public void showInfo(){
         System.out.print(«Фігура Shape»);
     }
}

public class Triangle extends Shape {
      public void draw() {
          /* Малюємо трикутник... */
          System.out.println(«Triangle.draw()»);
       }
      public void showInfo() {
          System.out.print(«Фігура Triangle»);
     }
}

public class Circl extends Shape {
      public void draw() {
          /* Малюємо коло... */
          System.out.println («Circle.draw()»);
      }
      public void showInfo(){
          System.out.print(«Фігура Circle»);
      }
}

public class Square extends Shape {
       public void draw() {
           /* Малюємо прямокутник… */
           System.out.println («Square.draw()»);
        }
       public void showInfo(){
           System.out.print(«Фігура Square»);
       }
}

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

```
        // Shape sh = new Shape();  // Неможна
        Shape sh; // МОЖНО

        Triangle t = new Triangle();
        // draw() та showInfo() з Triangle
        t.draw();
        t.showInfo();

        Circle c = new Circle();
        // draw() та showInfo() з Circle
        c.draw();
        c.showInfo();

        Square s = ne Square();
         // draw() та showInfo() з Square
         s.draw();
         s.showInfo();
    }
}
```

Приклад використання модифікатору **final** у сполучені із спадкуванням:

final для заборони перевизначення методу

```
class A
{
    final void meth()
    {
        System.out.println(«Цей метод final»);
    }
}
class B extends A
{
    void meth()
    { /* ПОМИЛКА! Цей метод не може бути перевизначений*/
      System.out.println(«Недопускається!»);
    }
}
```

final для запобігання успадкування

```
 final class A
 {
    // ...
```

```
}

// Наступний клас не допустимий
class B extends A
{
    /*  ПОМИЛКА! Клас A не може мати підкласи*/
  // ...
}
```

Одночасне оголошення класу як abstract та final неприпустимо !!!

3.13. Інтерфейс у Java

Інтерфейс – це контрольний тип в Java. Він схожий з класом. Це сукупність абстрактних методів. Клас реалізує інтерфейс наслідуючи абстрактні методи інтерфейсу. Разом з абстрактними методами інтерфейс може містити константи, звичайні методи, статичні методи і вкладені типи. Тіла методів існують тільки для звичайних методів і статичних методів [10].

Схожість класів та інтерфейсів:

- Інтерфейс може містити будь-яку кількість методів.
- Інтерфейс записаний у файлі з розширенням .java, і ім’я інтерфейсу збігається з ім’ям файлу.
- Байт-код інтерфейсу знаходиться в .class файлі.
- Інтерфейси з’являються в пакетах, і їх відповідний файл байт-коду повинен бути в структурі каталогів, яка збігається з ім’ям пакета.

Відмінність класів та інтерфейсів:

- Не можна створити екземпляр інтерфейсу.
- В інтерфейсі не містяться конструктори.
- Всі методи в інтерфейсі абстрактні.
- Інтерфейс не може містити поля примірників. Поля, які можуть з’явитися в інтерфейсі, зобов’язані бути оголошені і статичними, і final.
- Інтерфейс не розширюється класом, він реалізується класом.
- Інтерфейс може розширити безліч інтерфейсів.

Щоб реалізувати інтерфейс, у визначенні класу необхідно включити конструкцію implements, а потім створити методи, визначені інтерфейсом.

```
доступ class им’я_класу [extends суперклас] [implements інтерфейс ] {
      // тіло класу
   }
interface ICallback {
   void callback(int param);
}
```

Приклад реалізації інтерфейсів:

```
class Client implements ICallback {
/* Реалізує інтерфейс Callback. При реалізації методу інтерфейсу він повинен бути оголошений як public */
   public void callback(int p) {
      System.out.println(«Метод callback, викликаний із значенням « + p);
   }
   void nonlntrfaceMeth() {
      System.out.println(«Класи, які реалізують інтерфейси» + «можуть визначати також і інші члени.»);
   }
}
```

3.14. Виключні ситуації

Виключна ситуація – це програмна помилка, яка виникає під час виконання послідовності програмного коду. Програмна помилка може бути логічною помилкою програміста під час розробки програми. Наприклад, виключна ситуація може виникати у випадку:

- спроби ділення на нуль;
- спроби звернення до елементу масиву, номер індексу якого виходить за межі оголошеного;
- спроби взяти корінь з від'ємного числа;
- спроби відкрити файл за іменем, якого немає на диску.

Правильна обробка виключень є важливим елементом написання програм.

Поняття виключення в мові Java.

Як правило, кожна виключна ситуація повинна мати свій код помилки та обробку цього коду (виведення відповідних повідомлень, тощо). У Java розроблено механізм обробки виключних ситуацій.

Виключення – це спеціальний об'єкт, який описує виключну ситуацію, що виникла в деякій частині програмного коду. Об'єкт, що представляє виключення, генерується в момент виникнення виключної ситуації. Після виникнення критичної ситуації виключення перехоплюється та обробляється. Таким чином, виникає поняття *генерування виключення*.

Генерування виключення – процес створення об'єкту, що описує дане виключення.

Способи генерування виключень:

- *автоматично* – у цьому випадку виключення генеруються виконавчою системою Java. До таких виключень входять помилки, що порушують правила мови чи обмеження, що накладуються системою;

- *вручну* – у цьому випадку виключення генеруються в програмному коді, який розробляється. Ручні виключення програмуються для повідомлення викликаючому коду про можливі помилки в методах розроблюваної програми.

Стандартний обробник виключень викликається у випадках, коли програма:

- не використовує блок try…catch для обробки та перехоплення виключної ситуації взагалі;
- містить блок try…catch, однак у цьому блоці даний тип виключення не перехоплюється.

Якщо програма містить власний код try…catch для обробки даної виключної ситуації, тоді стандартний обробник виключень не викликається.

Призначення конструкції **try… catch…finally**. Загальна форма:

```
try {
   // блок коду, в якому відслідковуються помилки
}
catch (Тип_виключень_1 exOb) {
   // обробник виключень типу ExceptionType1
}
catch (Тип_виключень_2 exOЬ) {
   // обробник виключень типу ExceptionType2
}

finally {
   // блок коду, який повинен бути виконаний після завершення блоку try
}
```

У блоці *try* розміщуються оператори програми, які потрібно проконтролювати та у випадку виникнення виключної ситуації згенерувати виключення. Всередині блоку try можуть викликатись різні методи, здатні згенерувати те чи інше виключення. Однак, обробник виключення буде тільки один.

У блоці *catch* розміщується програмний код, який обробляє перехоплене виключення (обробник виключення). Код у блоці catch реалізує виконання відповідних дій, якщо відбулась виключна ситуація у блоці try. Блоків catch може бути декілька. Якщо генерується виключення, механізм обробки виключень шукає перший з обробників, аргумент якого відповідає поточному типу виключення. Після цього він входить у блок catch, і, в результаті цього виключення вважається обробленим. Тільки один відповідний блок catch обробляється.

У блоці *finally* вказується код, який повинен бути обов'язково виконаний після завершення блоку try. Блок операторів finally виконується незалежно від

того, чи буде згенероване виключення чи ні. Якщо виключення згенероване, блок операторів finally виконується навіть при умові, що жоден з операторів catch не співпадає з цим виключенням. Оператори try і catch складають єдине ціле. Оператор finally може бути відсутній.

Приклад обробки виключень **Exception**:

```
// Обробка виключення з продовженням роботи
import java.util.Random;
class HandleError {
   public static void main(String args[]) {
      int a=0, b=0, c=0;
      Random r = new Random();
      for (int i=0; i<32000; i++) {
         try {
              b = r.nextlnt();
              c = r.nextlnt();
              a = 12345 / (b/c);
            }
         catch (ArithmeticException e) {
      System.out.println(«Ділення на нуль.  Виключення: « + e );
      a = 0; // привласнити нуль і продовжити робот
   }
   System.out.println(«a: « + a);
      }
   }
}
```

Винятки поділяються на кілька класів, які мають спільного предка – клас Throwable. Його нащадками є підкласи Exception та Error. Винятки (Exceptions) є результатом проблем у програмі, які в принципі вирішуються та передбачувані. Наприклад, стався поділ на нуль у цілих числах [10].

Помилки (Errors) є серйознішими проблемами, які, згідно специфікації Java, не слід намагатися обробляти у власній програмі, оскільки вони пов’язані з проблемами рівня JVM. Наприклад, винятки такого роду виникають, якщо закінчилася пам’ять, доступна віртуальній машині.

У Java всі винятки поділяються на три типи: контрольовані винятки (checked) та неконтрольовані винятки (unchecked), до яких відносяться помилки (Errors) та виключення часу виконання (RuntimeExceptions, нащадок класу Exception).

Контрольовані винятки являють собою помилки, які можна і потрібно обробляти в програмі, до цього типу відносяться всі нащадки класу Exception (крім RuntimeException).

Завдання для самостійного виконання:

1. За варіантом обрати **одне** завдання, створити класи, в них передбачити різні члени класів і методи для роботи, описати відповідні класи:

1) Базовий клас – учень. Похідні – школяр і студент. Створити клас Конференція, який може містити обидва типи учнів. Передбачити метод підрахунку учасників конференції окремо по школярах і по студентах (використовувати оператор instanceof).

2) Базовий клас – працівник. Похідні – працівник на почасову оплату та на окладі. Створити клас Підприємство, який може містити обидва види працівників. Передбачити метод підрахунку працівників окремо на почасову оплату і на окладі (використовувати оператор instanceof).

3) Базовий клас – комп'ютер. Похідні – ноутбук і смартфон. Створити клас Ремонт-Сервіс, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо ремонтованих ноутбуків і смартфонів (використовувати оператор instanceof).

4) Базовий клас – друковані видання. Похідні – книги та журнали. Створити клас Книжковий Магазин, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо книг, окремо журналів (використовувати оператор instanceof).

5) Базовий клас – приміщення. Похідні – квартира та офіс. Створити клас Будинок, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо квартир , окремо офісів (використовувати оператор instanceof).

6) Базовий клас – файл. Похідні – звуковий файл і відеофайл. Створити клас Каталог, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо звукових та відео-файлів (використовувати оператор instanceof).

7) Базовий клас – літальний апарат. Похідні – літак і вертоліт. Створити клас Авіакомпанія, яка може містити обидва види об'єктів. Передбачити метод підрахунку окремо літаків, окремо вертольотів (використовувати оператор instanceof).

8) Базовий клас – змагання. Похідні – командні змагання та особисті. Створити клас Чемпіонат який може містити обидва види об'єктів. Передбачити метод підрахунку окремо командних змагань і особистих (використовувати оператор instanceof).

9) Базовий клас – меблі. Похідні – диван та шафа. Створити клас Кімната, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо диванів, окремо шаф (використовувати оператор instanceof).

10) Базовий клас – зброя. Похідні – вогнепальна та холодна. Створити клас Збройна Палата, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо вогнепальної і холодної зброї (використовувати

оператор instanceof).

11) Базовий клас – оргтехніка. Похідні – принтер та сканер. Створити клас Офіс, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо принтерів і сканерів (використовувати оператор instanceof).

12) Базовий клас – ЗМІ. Похідні – телеканал і газета. Створити клас Холдинг, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо телеканалів і газет (використовувати оператор instanceof).

2. Створити клас «Продукт» (Product):

Властивості:

- назва продукту;
- категорія (1 – напої, 2 – харчові продукти, 3 – побутова хімія);
- виробник;
- дата виготовлення;
- термін придатності (днів);
- базова ціна.

Методи:

- toString() – повна інформація про продукт;
- getPrice() – обчислення ціни (напої: -5 %, харчові продукти: -10 %, побутова хімія: без знижки).

3. Створити клас «Курс» (Course):

Властивості:

- назва курсу;
- викладач;
- рік запуску;
- тривалість (тижнів);
- тип (1 – онлайн, 2 – офлайн);
- базова вартість.

Методи:

- toString() – відображення повної інформації про курс;
- getPrice() – визначення вартості (онлайн: знижка 20 %, офлайн: без знижки, якщо курс триває більше 12 тижнів – націнка 10 %).

Контрольні запитання

1. Що таке інкапсуляція в Java і яку роль вона відіграє в об'єктно-орієнтованому програмуванні?
2. Яка різниця між класом і об'єктом у Java?
3. Яким чином створюється об'єкт у Java? Наведіть приклад.
4. Що таке метод у Java і як він визначається в класі?

5. Поясніть призначення конструктора в Java. Як він відрізняється від звичайного методу?

6. Яка роль ключового слова this у Java? Наведіть приклад його використання.

7. Що таке поліморфізм у Java і як він досягається?

8. Поясніть поняття перевантаження методів (method overloading) у Java.

9. Що таке успадкування в Java і як воно реалізується?

10. Які основні переваги використання об’єктно-орієнтованого підходу в програмуванні на Java?

Розділ 4. КОЛЕКЦІЇ JAVA.UTIL

4.1. Огляд колекцій

При написанні програми дуже часто виникає потреба зберігати набір будь-яких об’єктів. Це можуть бути числа, рядки, об’єкти призначених для користувача класів та ін. У бібліотеці колекцій Java існує два базових інтерфейсу, реалізації яких і представляють сукупність всіх класів колекцій. На вершині ієрархії Java Collection Framework розташовуються 2 інтерфейси: Collection і Map. Ці інтерфейси поділяють всі колекції, що входять до фреймворку на дві частини за типом зберігання даних: прості послідовні набори елементів і набори пар «ключ – значення» (Рис. 4.1):

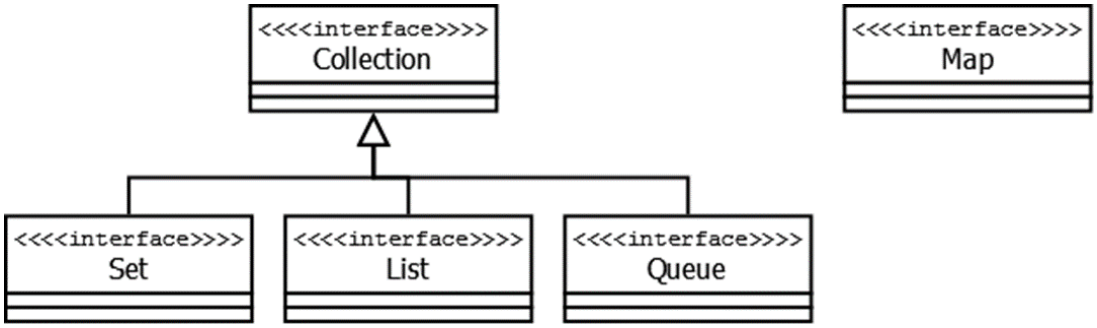

Рисунок 4.1 – Ієрархія інтерфейсів

Collection – колекція містить набір об’єктів (елементів). Тут визначено основні методи для маніпуляції з даними, такі як вставка (add, addAll), видалення (remove, removeAll, clear), пошук (contains).
Map – набори пар «ключ – значення» (словники).

Інтерфейс Collection розширює інтерфейс Iterable, у якого є тільки один метод iterator(). Це означає що будь-яка колекція, яка є спадкоємцем Iterable повинна повертати ітератор. Ітератор – це патерн, що дозволяє отримати доступ до елементів будь-якої колекції не звертаючи на суть її реалізації.

Collection – цей інтерфейс визначає основні методи роботи з простими наборами елементів, які будуть спільними для всіх його реалізацій (наприклад, size(), isEmpty(), add(E e) та ін.). Важливо також відзначити, що ці методи були реалізовані безпосередньо в інтерфейсі як default-методи. Інтерфейс Map також знаходиться у складі JDK з версії 1.2 і надає розробнику базові методи для роботи з даними виду «ключ – значення» [7, 9].

Інтерфейс Collection розширюють інтерфейси List, Set і Queue.

List – являє собою невпорядковану колекцію, в якій допустимі дублюючі значення. Іноді їх називають послідовностями (sequence). Елементи такої колекції пронумеровані, починаючи від нуля, до них можна звернутися за індексом.

Set – описує неупорядковану колекцію, яка містить повторюваних елементів. Це відповідає математичному поняттю множини (set).

Queue – черга. Це колекція, призначена для зберігання елементів в порядку, потрібному для їх обробки. На додаток до базових операцій інтерфейсу Collection, чергу надає додаткові операції вставки, отримання і контролю.

Основні методи, визначені в Collection:

- boolean add(E obj) – додає obj до колекції. Повертає true, якщо obj був доданий до колекції.
- void clear() – видаляє всі елементи колекції.
- boolean isEmpty() – повертає true, якщо колекція, що його викликає – порожня.
- boolean remove(Object obj) – видаляє один екземпляр obj з колекції. Повертає true, якщо елемент був видалений.
- int size() – повертає кількість елементів, що містяться в колекції.
- Object [] toArray() – повертає масив, що містить всі елементи у викликаній колекції.

4.2. Інтерфейс Map

Цей інтерфейс надає розробнику базові методи для роботи з даними виду «ключ – значення» (Рис. 4.2).

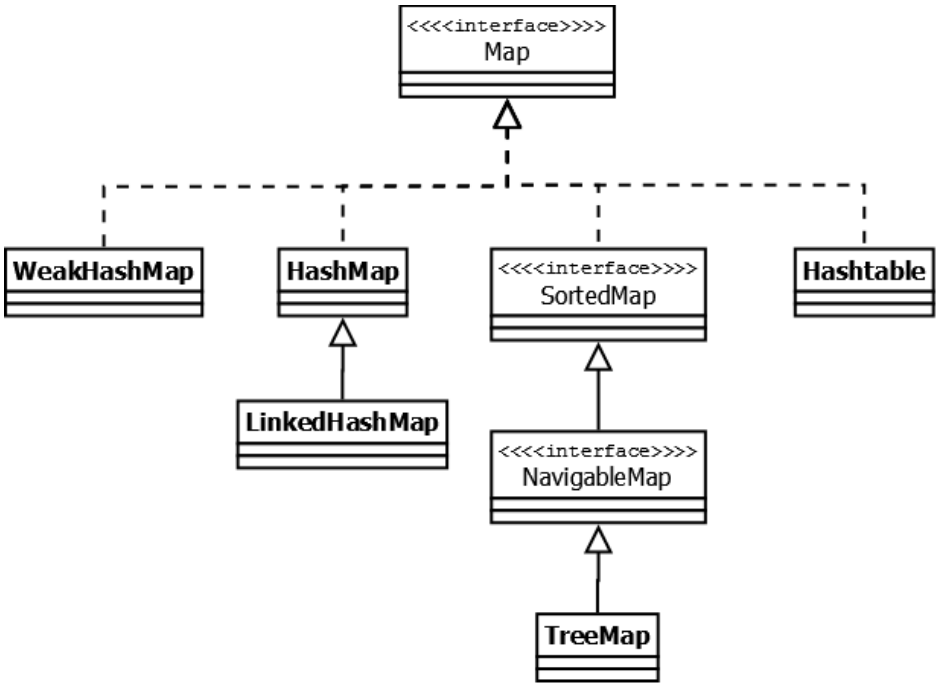

Рисунок 4.2 – Інтерфейс Map

Hashtable – реалізація такої структури даних, як хеш-таблиця [10]. Вона не дозволяє використовувати null як значення або ключ. Ця колекція була реалізована раніше, ніж Java Collection Framework, але згодом була включена до його складу. Hashtable є синхронізованою (майже всі методи позначені як synchronized). Через цю особливість у неї є суттєві проблеми з продуктивністю

і, починаючи з Java 1.2, у більшості випадків рекомендується використовувати інші реалізації інтерфейсу Map через відсутність у них синхронізації.

HashMap – колекція є альтернативою Hashtable. Двома основними відмінностями від Hashtable є те, що HashMap не синхронізована і HashMap дозволяє використовувати null як ключ, так і значення. Так само, як і Hashtable, дана колекція не є впорядкованою: порядок зберігання елементів залежить від хеш-функції. Додавання елемента виконується за константний час O(1), але час видалення отримання залежить від розподілу хеш-функції. В ідеалі є константним, але може бути лінійним O(n).

LinkedHashMap – це впорядкована реалізація хеш-таблиці. Тут, на відміну від HashMap, порядок ітерування дорівнює порядку додавання елементів. Ця особливість досягається завдяки двоспрямованим зв'язкам між елементами (аналогічно LinkedList). Але ця перевага має також і недолік – збільшення пам'яті, яке забере колекція.

TreeMap – реалізація Map заснована на червоно-чорних деревах. Як і LinkedHashMap є впорядкованою. За замовчуванням, колекція сортується за ключами з використанням принципу «natural ordering», але ця поведінка може бути налаштована під конкретне завдання за допомогою об'єкта Comparator, який вказується як параметр під час створення об'єкта TreeMap.

WeakHashMap – реалізація хеш-таблиці, яка організована з використанням weak references. Іншими словами, Garbage Collector автоматично видалить елемент з колекції при наступному складанні сміття, якщо на ключ цього елемента немає жорстких посилань.

4.3. Інтерфейс List

Реалізації інтерфейсу List (список), є упорядковані колекції. Крім того, розробнику надається можливість доступу до елементів колекції за індексом та за значенням (оскільки реалізації дозволяють зберігати дублікати, результатом пошуку за значенням буде перше знайдене входження). Ієрархія зв'язків інтерфейсу списку та черги наведено на рисунку 4.3 [5].

Vector – реалізація динамічного масиву об'єктів. Дозволяє зберігати будь-які дані, включаючи null як елемент. Vector з'явився в JDK версії Java 1.0, але як і Hashtable, цю колекцію не рекомендується використовувати, якщо не потрібне досягнення безпеки. Тому що у Vector, на відміну від інших реалізацій List, всі операції з даними є синхронізованими. В якості альтернативи часто застосовується аналог ArrayList.

Stack – дана колекція є розширенням колекції Vector. Була додана Java 1.0 як реалізація стека LIFO (last-in-first-out). Є частково синхронізованою колекцією (крім методу додавання push()). Після додавання до Java 1.6 інтерфейсу Deque, рекомендується використовувати саме для реалізації цього інтерфейсу, наприклад ArrayDeque.

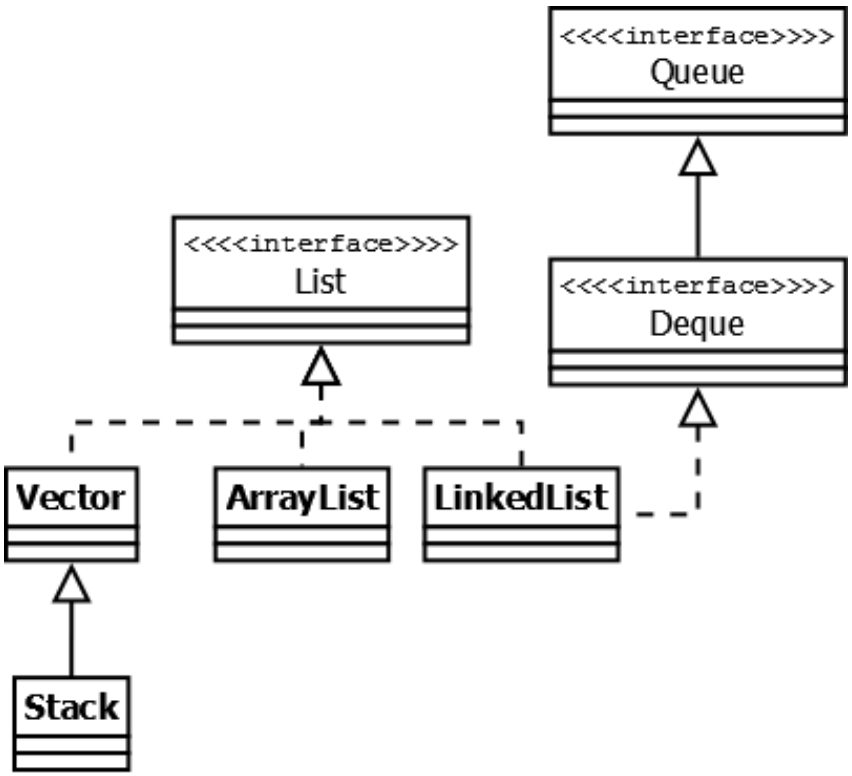

Рисунок 4.3 – Інтерфейс List

ArrayList – як і Vector є реалізацією динамічного масиву об’єктів. Дозволяє зберігати будь-які дані, включаючи null як елемент. Як можна здогадатися з назви, його реалізація ґрунтується на звичайному масиві. Цю реалізацію слід застосовувати, якщо в процесі роботи з колекцією передбачається часте звернення до елементів індексу. Через особливості реалізації по-індексне звернення до елементів виконується за константний час O(1). Але цю колекцію рекомендується уникати, якщо потрібне часте видалення/додавання елементів у середину колекції [18].

LinkedList – ще одна реалізація List. Дозволяє зберігати будь-які дані, включаючи null. Особливістю реалізації цієї колекції і те, що у її основі лежить двонаправлений зв’язковий список (кожен елемент має посилання на попередній і наступний елементи). Завдяки цьому, організована можливість додавання та видалення елементів із середини списку, доступ за індексом чи значенням відбувається за лінійний час O(n), а з початку і кінця списку – за константний час O(1). Так само, зважаючи на реалізацію, цю колекцію можна використовувати як стек або чергу. Для цього в ній реалізовано відповідні методи.

4.4. Інтерфейс Set

Інтерфейс Set (множина), являє собою невпорядковану колекцію, яка не може містити дані, що дублюються. Є програмною моделлю математичного поняття «безліч». Ієрархія інтерфейсу множини представлена на рисунку 4.4.

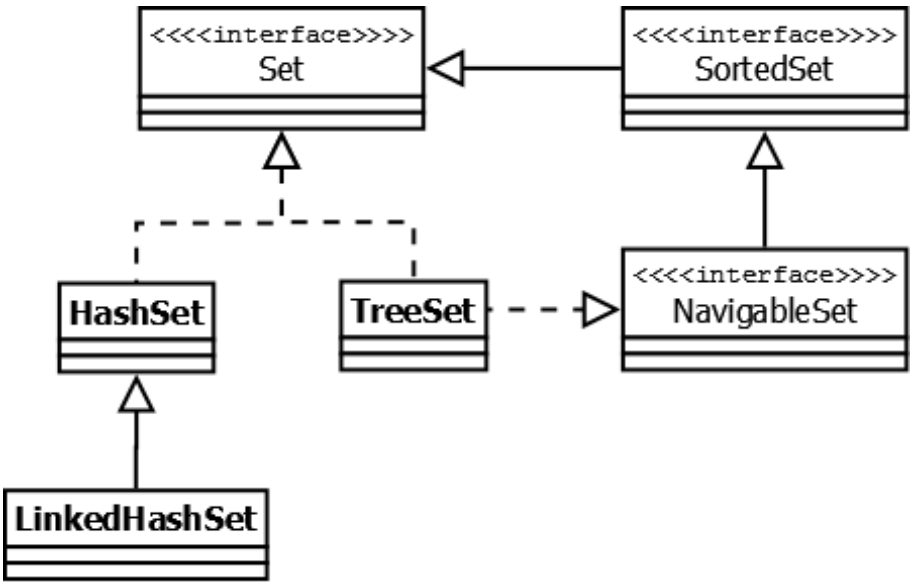

Рисунок 4.4 – Інтерфейс Set

HashSet – реалізація інтерфейсу Set, що базується на HashMap. Всередині використовує об'єкт HashMap для зберігання даних. В якості ключа використовується елемент, що додається, а в якості значення – об'єкт-пустушка (new Object()). Через особливості реалізації порядок елементів не гарантується під час додавання.

LinkedHashSet відрізняється від HashSet тільки тим, що в основі лежить LinkedHashMap замість HashMap. Завдяки цієї відмінності, порядок елементів при обході колекції є ідентичним порядку додавання елементів.

TreeSet – аналогічно іншим класам-реалізаціям інтерфейсу Set містить у собі об'єкт NavigableMap, що і зумовлює його поведінку. Надає можливість керувати порядком елементів у колекції за допомогою об'єкта Comparator або зберігає елементи з використанням «natural ordering».

4.5. Інтерфейс Queue

Інтерфейс Queue (черга), описує колекції з певним способом вставки та вилучення елементів, а саме черги FIFO (first-in-first-out). Крім методів, визначених в інтерфейсі Collection, визначає додаткові методи для вилучення та додавання елементів до черги. Більшість реалізацій цього інтерфейсу перебувають у пакеті java.util.concurrent. Ієрархія зв'язків інтерфейсів черги представлена на рисунку 4.5 [5, 9].

PriorityQueue – є єдиною прямою реалізацією інтерфейсу Queue (була додана, як і інтерфейс Queue, в Java 1.5), крім класу LinkedList, який так само реалізує цей інтерфейс, але був реалізований набагато раніше. Особливістю цієї черги є можливість управління порядком елементів. За замовчуванням, елементи сортуються з використанням «natural ordering», але ця поведінка може бути перевизначена за допомогою об'єкта Comparator, який задається під час створення черги. Ця колекція не підтримує null як елементи.

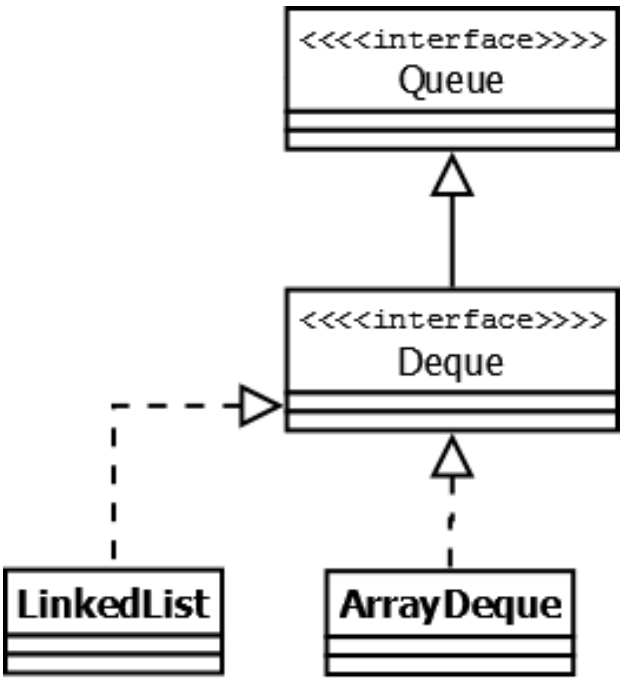

Рисунок 4.5 – Інтерфейс Queue

ArrayDeque – реалізація інтерфейсу Deque, який розширює інтерфейс Queue методами, що дозволяють реалізувати конструкцію LIFO (last-in-first-out). Інтерфейс Deque та реалізація ArrayDeque були додані до Java 1.6. Ця колекція являє собою реалізацію з використанням масивів, подібно до ArrayList, але не дозволяє звертатися до елементів по індексу та зберігання null. Як заявлено в документації, колекція працює швидше ніж Stack, якщо використовується як колекція LIFO, а також швидше ніж LinkedList, якщо використовується як FIFO.

ArrayList – мабуть сама часто використовувана колекція. Вона інкапсулює в собі звичайний масив, довжина якого автоматично збільшується при додаванні нових елементів. Так як ArrayList використовує масив, то час доступу до елементу за індексом мінімально (на відміну від LinkedList). При видаленні довільного елемента зі списку, всі елементи знаходяться «правіше» зміщуються на одну клітинку вліво, при цьому реальний розмір масиву (його ємність, capacity) не змінюється. Якщо при додаванні елемента, виявляється, що масив повністю заповнений, буде створено новий масив, в нього будуть поміщені всі елементи зі старого масиву + новий, для додавання [11-13].

Клас ArrayList призначений для читання об’єктів по індексу. Тож не дарма у назві є слово Array (масив). Після створення колекції на основі ArrayList, прочитати дані можна кількома способами. Наступний приклад демонструє створення ArrayList, його наповнення об’єктами типу String та їх читання за допомогою методу get (int index) та за допомогою ітератора.

Приклад роботи з ArrayList:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class TestArrayList {
   private ArrayList<String> al;
   public static void main(String[] args) {
      TestArrayList test = new TestArrayList();
      test.create();
      test.getData();
      test.iterateData();
   }
   void create() {
      //створюємо і наповнюємо ArrayList
      a1 = new ArrayList<String>();
      al.add(«Привіт»);
      al.add(«студент!»);
   }
   //читаємо дані по індексу
   void getData() {
      for (int i = 0; i < al.size(); i++) {
         System.out.print(al.get(i) + «»);
      }
   }
   // читаємо вміст ArrayList використовуючи iterator
   void iterateData() {
      ListIterator<String> it = al.listIterator();
      while (it.hasNext()) {
         System.out.print(it.next() + «»);
      }
   }

}
```

Передати вміст ArrayList можна у звичайний масив за допомогою методу toArray().

Перегляд даних за допомогою ітератора:

```
ListIterator<String> it=a1.listIterator();
```

Створюється об’єкт-ітератор, посилання на який передається об’єктній змінній it типу ListIterator. ListIterator – це інтерфейс, який розширює

інтерфейс Iterator декількома новими методами. Базові методи інтерфейсу Iterator:

- boolean hasNext() – повертає true, якщо ітерація має наступний елемент;
- E next() – повертає наступний елемент ітерації (буква E вказує, що це може бути елемент будь-якого типу).
- void remove() – знищує останній елемент, що повертався ітератором.

Приклад використання циклу while:

```
while(it.hasNext()){
 System.out.print(it.next() + « »);
}
```

У коді бачимо конструкцію while. Цикл працює доти, поки є елементи в ітераторі. Перевірка здійснюється за допомогою методу hasNext. А вивід елементів здійснюється за допомогою методу next. Перевірка за допомогою hasNext необхідна через те, що в разі відсутності наступного елементу при виклику методу next буде викинуто виняток NoSuchElementExeption [5].

Інтерфейсом ListIterator передбачено ще такі методи як add, hasPrevious, next, nextIndex, previous, previousIndex, set. Назви методів говорять самі за себе. Клас Itr, який реалізовує інтерфейс ListIterator є внутрішнім класом класу AbstractList. ArrayList є нащадком класу AbstractList. Для того, щоб ітератор міг працювати з певним об’єктом, клас даного об’єкту повинен реалізовувати інтерфейс Iterable:

```
public interface Iterable<E>
{
	Iterator<E> iterator();
}
```

Інтерфейс Collection розширює даний інтерфейс.

Можна також перебрати елементи за допомогою перевантаженого циклу for (так званого «for each»):

```
for (String str: a1) {
	System.out.print(str + « »);
}
```

При компіляції даний цикл перетворюється компілятором у цикл із ітератором.

ArrayList можна також перетворити у звичайний масив за допомогою методу toArray():

```
String strArray[] = new String[a1.size()];
strArray = a1.toArray(strArray);
System.out.println(strArray[0]);
```

LinkedList – це структура даних, що являє собою пов'язаний список елементів (об'єктів).

Різниця між ArrayList та LinkedList полягає в тому, що ArrayList реалізований у вигляді масиву, а LinkedList у вигляді пов'язаних між собою об'єктів. ArrayList швидко виконує читання і заміну елементів (посилань на об'єкти), проте, щоб вставити новий елемент в середину ArrayList або видалити існуючий в середині ArrayList здійснюється послідовний зсув цілого ряду елементів масиву. В LinkedList доволі швидко відбувається вставлення нового елементу або видалення існуючого. Це відбувається тому, що в середині реалізації LinkedList змінюються лише посилання на попередній і наступний об'єкти (елементи). Проте доступ до об'єктів по індексу в LinkedList відбувається повільніше ніж в ArrayList. Тож загалом, LinkedList корисний, коли необхідно часто вставляти та видаляти елементи зі списку, а в інших випадках краще використовувати ArrayList.

Існує два конструктури LinkedList:

```
LinkedList()
LinkedList(Collection c)
```

Перший конструктор створює пустий список, а другий – створює пов'язаний список із іншої колекції.

Клас LinkedList розширює клас AbstractSequentalList та реалізує інтерфейси List, Dequeue та Queue. Реалізація останніх двох інтерфейсів (черг) означає, що можна працювати із пов'язаним списком як із стеком з використанням методів pop(), push(), poll(), pollFirst(), pollLast() і тому подібне. Перебирати елементи LinkedList також можна за допомогою ітератора та у циклі for earch.

Розглянемо приклад. Попрацюємо з LinkedList, який міститиме в якості елементів об'єктні змінні типу Car. Спочатку реалізуємо клас Car:

```
public class Car {
   private String name;
   private double price;
   private int year;
   public Car(String name, double price, int year) {
      this.name = name;
      this.price = price;
      this.year = year;
   }
```

```
    @Override
    public String toString() {
        return «\n Car [name=» + name + «, price=» + price + «, year=» + year +
«]»; } }
```

Клас Car містить лише поля, конструктор, який заповнює дані поля та заміщений метод toString(), який видаватиме нам інформацію про автомобіль у вигляді рядка.

Наступний клас демонструє роботу з LinkedList:

```
import java.util.ArrayList;
import java.util.LinkedList;

public class TestLinkedList {
    private LinkedList<Car> ll = new LinkedList<>();
    public static void main(String[] args) {
        TestLinkedList t = new TestLinkedList();
        t.test();
    }
    void test () {
        Car car1 = new Car(«Ferrary», 10800, 1995);
        Car car2 = new Car («Zaporozhets», 2600, 1989);
        ll.add(car1);
        ll.add(car2);
        //додаємо на початок списку
        ll.addFirst(new Car(«Alfa Romeo 155», 11678, 2000));
        ll.remove(car2); // видаляємо об’єкт
        System.out.println(«After car2: » + ll);
        ll.remove(1); // видаляємо елемент по індексу
        System.out.println(«Після видалення першого елементу: »+ ll);
        Car myCar = ll.get(0);
        System.out.println(«Отриманий елемент за індексом [0]: » + myCar);
        ll.set(0, carl); //замінюємо елемент по індексу
        System.out.println(«Замінений елемент за індексом [0] » + ll.get(0));
        ArrayList<Car> arrList = new ArrayList<Car>();
        arrList.add(carl);
        arrList.add(car2);
        ll.addAll(arrList); //додаємо вміст ArrayList у LinkedList
        System.out.println(«Після додавання ArrayList: » + ll);
    }
}
```

4.6. Клас HashSet

Клас HashSet призначений для зберігання даних у вигляді множини невпорядкованих елементів. Якщо потрібна впорядкована множина, то використовуйте TreeSet. HashSet також не гарантує стабільного порядку збереження об'єктів. Тобто при додаванні об'єктів порядок зберігання елементів змінюється. Вони можуть бути збережені як в кінці множити так і в середині. Якщо потрібен один порядок зберігання об'єктів використовуйте LinkedHashSet.

Сам термін «множина» означає, що елементи не будуть повторюватися. Для зберігання і пошуку елементів використовується хеш-код об'єкта. HashSet також може містити значення null. Власне всередині самої реалізації HashSet використовується клас HashMap, який дозволяє зберігати елементи у вигляді двох складових: ключа та хеш-коду. У класі HashSet хеш-код недоступний і використовується неявно для користувача.

Клас HashSet розширює клас AbstractSet та реалізує інтерфейс Set. Також реалізовує інтерфейси Serializable та Clonable.

HashSet має такі конструктори:

```
HashSet()
HashSet (Collection c)
HashSet (int об'єм)
HashSet (int об'єм, float коефіцієнт_заповнення)
```

Коефіцієнт заповнення – це число в межах 0.0 до 1.0, що представляє собою частку заповнення HashSet при якій об'єм HashSet буде збільшений. По замовчуванню в конструкторах, що не задають коефіцієнт заповнення, використовується значення 0.75.

Зробити копію хеш множини можна наступним чином:

```
HashSet<String> hsc = (HashSet)hs.clone();
```

Слід зауважити, що метод clone() здійснює поверхневе (shadow) копіювання, тобто копіюються лише адреси об'єктів, які містяться у екземплярі класу HashSet, а не самі об'єкти.

При використанні ітератор може викидати виняток ConcurrentModificationException, якщо HashSet було змінено. Щоб уникнути таких випадків слід використовувати методи ітератора для зміни HashSet [9-10].

У класу String перевизначені такі методи як equal та hashcode, що дає можливість коректного порівняння їхніх хеш-кодів та значень і уникати випадків, коли будуть додаватися ідентичні рядки, незалежно від того, чи вони зберігаються в різних комірках пам'яті. Проте при використанні власних класів, на практиці для правильного функціонування HashSet (без повторення

об’єктів з однаковим станом, наприклад, ідентичними полями) у класах необхідно замістити методи equals() та hashcode() самостійно. Так, якщо створено класи оболонки, що міститимуть String поле countryName із однаковим значенням, наприклад, «Австрія», то два об’єкти вважатимуться різні, оскільки у них буде різний хеш-код, тож вони будуть додані до HashSet. В таких випадках необхідно продумати методи equals() та hashcode().

Наступний приклад демонструє роботу із HashSet:

```
import java.util.HashSet;
public class TestHashSet {
   HashSet<String> hs = new HashSet<String>();
   public static void main(String[] args) {
      TestHashSet test = new TestHashSet();
      test.test();
   }
   void test() {
      hs.add(«Australia»);
      hs.add(«Ukraine»);
      hs.add(«USA»);
      System.out.println(«1) Три країни: » + hs + « розмір = » + hs.size());
      hs.add(«Australia»); // помилки не буде, однак у HashSet нічого не
додасться
      System.out.println(«2) Після спроби додати Australia ще раз: » + hs);
      hs.remove(«USA»); //видаляємо USA з множини
      hs.add(«Germany»);
      hs.add(«England»);
      hs.add(null);
      hs.add(null); // другий раз не додаються
      System.out.println(«3)» + hs);
      System.out.println(«4) Чи містить множина Germany? » + hs.contains
(«Germany»));
      System.out.println(«5) Чи пуста множина? » + hs.isEmpty());
      //можемо також отримати iterator, або перебрати множину за допомогою
for earch
      for (String str:hs) {
         System.out.println(str);
      }
      hs.clear(); // очищаємо
      System.out.println(«6) Розмір множини після очищення = » + hs.size());
   }
}
```

Клас LinkedHashSet розширює клас HashSet не додаючи ніяких нових

методів. Працює він дещо довше за HashSet проте зберігає порядок в якому елементи додаються до нього. Відповідно це дозволяє організувати послідовну ітерацію вставлення та витягнення елементів. Всі конструктори та методи роботи з LinkedHashSet аналогічні методам класу HashSet.

4.7. Клас TreeSet

Клас TreeSet дозволяє створювати відсортовану множину. Тобто елементи не повторюються та зберігаються у відсортованому порядку. Для зберігання елементів застосовується бінарна деревоподібна структура. Об'єкти зберігаються у відсортованому порядку по зростанню. Час доступу та одержання елементів доволі малий, тому клас TreeSet підходить для зберігання великих об'ємів відсортованих даних, які повинні бути швидко знайдені.

Клас TreeSet розширює клас AbstractSet та реалізує інтерфейс NavigableSet. NavigableSet реалізується на базі TreeMap.

В класі доступні чотири конструктори:

```
TreeSet ()
TreeSet(Collection c)
TreeSet(Comparator компаратор)
TreeSet(SortedSet ss)
```

Третій конструктор дозволяє задавати власний компаратор, відповідно до якого буде відбуватися сортування об'єктів. Так об'єкти класу String не потребують реалізації власного компаратора, проте якщо необхідно зберігати в класі TreeSet розроблені об'єкти, то потрібно задавати компаратор для цих об'єктів. Такий компаратор може реалізовувати сортування об'єктів по певному полю, наприклад по Прізвищу, якщо створений клас зберігає інформацію про осіб. Можна реалізувати ланцюжок компараторів з використанням методу thenComparing() класу Comparator.

TreeSet не може містити значення null. Також TreeSet не синхронізований клас, як і інші класи колекцій при потребі його потрібно синхронізувати з використанням методу Collections.synchronizedSet().

Розглянемо приклад роботи TreeSet з компаратором. Використаємо клас Car, який був попередньо використаний при роботі з LinkedList додавши лише гетер та сетер методи:

```
public class Car {
   private String name;
   private double price;
   private int year;
   public Car (String name, double price, int year) {
```

```
        this.name = name;
        this.price = price;
        this.year=year;
    }
    public String getName() {
        return name;
    }
    public void setName (String name) {
        this.name = name;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice (double price) {
        this.price = price;
    }
    public int getYear() {
        return year;
    }
    public void setYear (int year) {
        this.year = year;
    }
    @Override
    public String toString() {
        return «\n Car [name = » + name + «, price = » + price + «, year = » + year +
«]»;
    }
}
```

Клас компаратора CarsComparator виглядає так:

```
import java.util.Comparator;
public class CarsComparator implements Comparator {
    @Override
    public int compare(Car car1, Car car2) {
        if(car1.getYear() > car2.getYear())
            return 1;
        else if (car1.getYear() < car2.getYear())
            return -1;
        else return 0;
    }
}
```

Програма з TreeSet:

```
import java.util.Comparator;
import java.util.TreeSet;
public class TestTreeSet {
   CarsComparator comp = new CarsComparator();
   TreeSet<Car> ts1 = new TreeSet<>(comp);
   public static void main(String[] args) {
      TestTreeSet t = new TestTreeSet();
      t.test();
   }
   void test() {
      Car car1 = new Car(«Ferrary», 12000, 1988);
      Car car2 = new Car(«Ford», 13000, 1955);
      Car car3 = new Car(«Toyota», 13500, 2003);
      Car car4 = new Car(«Citroen», 12000, 2014);
      Car car5 = new Car(«Mercedes-Benz», 15000, 2011);
      ts1.add(carl);
      ts1.add(car2);
      ts1.add(car3);
      ts1.add(car4);
      ts1.add(car5);
      System.out.println(«Сортування по роках: » + ts1);
      //Зворотній компаратор для TreeSet (Java8)
      TreeSet<Car> ts2 = new TreeSet<>(comp.reversed());
      ts2.addAll(ts1); //додати вміст попереднього TreeSet у новий
      System.out.println(«Зворотне сортування: » + ts2);
   }
}
```

4.8. PriorityQueue

PriorityQueue дозволяє реалізувати чергу на основі пріоритету. Така черга може бути корисна, наприклад, у разі необхідності обслуговування клієнтів згідно пріоритету. При зберіганні чисел в пріоритетній черзі, така черга гарантує, що першим елементом завжди буде найменший елемент. При цьому не гарантується ніякий стабільний послідовний порядок збереження елементів. Після додавання або видалення елементу з пріоритетної черги, порядок зберігання елементів в цій черзі змінюється таким чином, що в голові черги опиняється найменший елемент згідно його природнього порядку або згідно заданого компаратора [10].

PriorityQueue має наступні конструктори:

```
PriorityQueue() //початковий об’єм становить 11
PriorityQueue(int початковий_об’єм)
PriorityQueue(Comparator comparator)
PriorityQueue(int початковий_об’єм, Comparator компаратор)
PriorityQueue(Collection c)
PriorityQueue(PriorityQueue c)
PriorityQueue(SortedSet c)
```

Таким чином в разі потреби можна задати власний компаратор для видачі елементів у потрібному нам порядку.

PriorityQueue не може містити null, вона розширює AbstractQueue та реалізує інтерфейси Serializable, Iterable, Collection, Queue. Реалізація інтерфейсу Queue говорить, що в PriorityQueue доступні такі методи роботи з чергою:

- add(E e) – додати вказаний елемент у чергу;
- element() – отримати, але не видаляти, елемент з голови черги;
- offer(E e) – додати визначений елемент у чергу;
- peek() – отримати елемент з голови черги, але не видаляти його. Повертає null, якщо черга порожня;
- poll() – отримати та видалити елемент з голови черги;
- remove() – отримати та видалити елемент з голови черги.

Крім того в PriorityQueue доступні методи, що наявні і в інших колекціях як то: clear(), comparator(), contains(), iterator(), spliterator(), size(), toArray() тощо.

Приклад використання PriorityQueue:

```
import java.util.PriorityQueue;
public class TestPriorityQueue {
   public static void main(String[] args) {
      PriorityQueue<Integer> pq = new PriorityQueue<>(11);
      pq.add(750);
      pq.add(50);
      pq.offer(1);
      pq.add(300);
      pq.add(25);
      pq.add(2);
      pq.offer(20);
      pq.add(5);
      pq.add(40);
      System.out.println(«Порядок зберігання елементів » + pq);
      //Отримуємо і видаляємо елемент з голови черги
```

```
        while (!pq.isEmpty()) {
            System.out.println(«Отримали: » + pq.poll());
            System.out.println(«Новий порядок: » + pq);
        }
    }
}
```

Таким чином за допомогою методу poll() (його можна також замінити методом remove()), отримуємо елементи масиву у відсортованому порядку, незважаючи на те, що в самій пріоритетній черзі елементи зберігаються у іншому порядку.

4.9. ArrayDeque

ArrayDeque – це клас який забезпечує двосторонню чергу. Іншими словами – це автоматично зростаючий масив, що дозволяє додавати або видаляти елементи з обох боків черги. ArrayDeque може бути використано як стек (LIFO, останній ввійшов – перший вийшов) або ж як черга (FIFO, перший ввійшов – перший вийшов). ArrayDeque не може містити в якості елемента null.

Клас ArrayDeque швидший за клас Stack, якщо використовувати його в якості стеку і швидший за LinkedList, якщо використовувати в якості черги. ArrayDeque розширює клас AbstractCollection та реалізує інтерфейси Deque, Cloneable, Serializable. Таким чином можна використовувати для роботи з даним класом, як методи інтерфейсу Deque (який розширює інтерфейс Queue) так і методи інтерфейсу Collection.

Якщо використовується ітератор, то він буде викидати виняток ConcurrentModificationException, якщо вміст ArrayDeque був модифікований після створення ітератора. Таким чином ітератор необхідно з обережністю використовувати з ArrayDeque.

В ArrayDeque доступні наступні конструктори:

```
ArrayDeque() // пустий ArrayDeque з початковим об’ємом 16 елентів
ArrayDeque(Collection c) // міститиме елементи колекції в порядку, що повертає ітератор даної колекції
ArrayDeque(int numElements) // дозволяє задати початковий об’єм
```

При потребі збільшення об’єму, то він автоматично збільшується вдвічі. Наступний приклад демонструє роботу із ArrayDeque у вигляді стека та у в вигляді черги:

```
import java.util.ArrayDeque;
public class TestArrayDeque {
   public static void main(String[] args) {
```

```
        ArrayDeque<String> ad = new ArrayDeque<>();
        System.out.println(«Використання в якості стеку»);
        ad.push(«Перший»); // метод інтерфейсу Deque
        ad.push(«Другий»);
        ad.push(«Третій»);
        System.out.println(«Преший елемент ArrayDeque: » + ad.getFirst());
        //Отримаємо перший елемент але не видаляємо його
        System.out.println(«Вміст ArrayDeque: » + ad);
        System.out.println(ad.pop()); //Отримаємо та видаляємо з вершини стеку
        System.out.println(ad.pop()); //еквівалентно до removeFirst()
        System.out.println(ad.pop());
        //System.out.println(ad.pop());
        // т.я. ArrayDeque порожній отримаємо NoSuchElementException
        System.out.println();
        System.out.println(«Використання в якості черги»);
        ad.offer(«Перший»);
        ad.offer(«Другий»);
        ad.offer(«Третій»);
        System.out.println(«Перший елемент ArrayDeque: » + ad.getFirst());
        //Отримаємо перший елемент, але не видаляємо його
        System.out.println(«Вміст ArrayDeque: » + ad);
        System.out.println(ad.poll()); //витягнули елемент і видаляємо з голови
        System.out.println(ad.poll());
        System.out.println(ad.poll());
        System.out.println(ad.poll());
    }
}
```

4.10. Інтерфейс Map

Зіставлення Map – це об’єкт який зіставляє ключ із відповідним значенням (key = value). Маючи ключ можна знайти його значення. І ключ, і значення є об’єктами. Ключ має бути унікальним, а значення може повторюватися. Деякі зіставлення можуть приймати null ключ і null значення, інші не можуть. Зіставлення не підтримують Iterable інтерфейс. Тому не дозволено використовувати ітератор чи цикл for each для перебору вмісту зіставлення. Проте можна перетворити зіставлення у інший тип колекції і вже на ньому застосувати ітератор [18].

Основу зіставлень складає інтерфейс Map:

```
interface Map<K, V>
```

де K – визначає тип ключа, а V – визначає тип значення.

Інтерфейс SortedMap розширює інтерфейс Map і забезпечує зберігання

вмісу у порядку зростання, базуючись на значенні ключа.

Інтерфейс NavigableMap розширює SortedMap і забезпечує повернення вмісту базуючись на схожості ключа або ключів.

Інтерфейс Map.Entry – вкладений в Map інтерфейс і забезпечує роботу із вмістом зіставлення.

HashMap розширює AbstractMap та реалізує інтерфейс Map. Клас не додає власних методів, та використовує хеш таблиці для зберігання зіставлення. Він не гарантує порядок елементів, та має наступні конструктори:

```
HashMap( )
HashMap(Map<? extends K, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)
```

Перший конструктор створює HashMap по замовчуванню.

Другий створює хеш зіставлення використовуючи елементи m.

Третій задає ємність хеш зіставлення.

Четвертий задає ємність(по замовчуванню 16) та коефіцієнт заповнення (по замовчуванню 0.75).

Ключі і значення в HashMap є об’єктами. Якщо необхідно використати, наприклад, числа цілого типу (int) то використовуватиметься клас обгортка Integer. Так само і для інших примітивних типів.

Клас **TreeMap** розширює AbstractMap та реалізує інтерфейс NavigableMap. TreeMap не містить методів крім тих, що визначені у NavigableMap та класі AbstractMap. Що забезпечує зберігання зіставлень за допомогою деревовидної структури. TreeMap забезпечує зберігання пар ключ/значення у відсортованому порядку (в порядку зростання), що забезпечує швидкий доступ до елементів. TreeMap по замовчуванню сортується за ключами. Для задання іншого сортування використовують компаратор [5].

TreeMap узагальнений клас і оголошується як:

```
class TreeMap<K,V>
```

де K – задає тип ключа, а V – задає тип значення.

TreeMap визначає наступні конструктори:

```
TreeMap( )
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

Перший конструктор створює пусте деревовидне зіставлення, що буде відсортоване використовуючи природній порядок ключів.

Другий створює пусте зіставлення, що буде відсортоване використовуючи компаратор.

Третій створює деревовидне зіставлення із входжень зіставлення m, що буде відсортоване в природньому порядку.

Четвертий ініціалізує деревовидне зіставлення входженнями із sm з порядком сортуванням того ж sm.

Клас **LinkedHashMap** розширює HashMap і підтримує пов'язаний список записів у зіставленні, в порядку в якому вони додавались. При необхідності можна задати зіставлення в якому записи будуть зберігатись в порядку останнього доступу [7].

LinkedHashMap має наступні конструктори:

```
LinkedHashMap( )
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int capacity)
LinkedHashMap(int capacity, float fillRatio)
LinkedHashMap(int capacity, float fillRatio, boolean Order)
```

Перший конструктор створює LinkedHashMap по замовчуванню.

Другий ініціалізує LinkedHashMap елементами із зіставлення m.

Третій конструктор задає ємність.

Четвертий задає ємність та коефіцієнт заповнення. Ємність по замовчуванню – 16, а коефіцієнт заповнення по замовчуванню становить 0,75. Останній конструктор дозволяє задати порядок заповнення зіставлення, чи записи зберігатимуться в порядку вставлення чи в порядку останнього доступу. Якщо Order є true тоді використовуються порядок зберігання по порядку доступу, якщо false тоді використовується порядок вставлення.

Завдання для самостійного виконання:

1. Обрати 1 варіант:

1) Ввести рядки з файлу, записати їх в стек. Вивести рядки з файлу в зворотному порядку.

2) Ввести число, занести його цифри в стек. Вивести число, у якого цифри йдуть у зворотному порядку.

3) Скласти два багаточлена заданого ступеня, якщо коефіцієнти багаточленів зберігаються в об'єкті HashMap.

4) Створити стек з елементів каталогу. Не використовуючи допоміжних об'єктів, переставити негативні елементи даного списку в кінець, а позитивні – на початок цього списку.

5) Задати два стека, поміняти інформацію місцями.

6) Помножити два багаточлени заданого ступеня, якщо коефіцієнти багаточленів зберігаються у списку.

2. Обрати 1 варіант:

1) Організувати обчислення у вигляді стеку. Виконати попарне підсумовування довільного кінцевого ряду чисел наступним чином: на першому етапі підсумовуються попарно поруч стоячі числа; на другому етапі підсумовуються результати першого етапу, і так далі, доки не залишиться одне число.

2) Визначити клас Stack. Оголосити об'єкт класу. Ввести послідовність символів і вивести її в зворотному порядку.

3) Визначити клас Set на основі множини цілих чисел, n = розмір. Створити методи для визначення перетину і об'єднання множин.

4) Програма отримує N параметрів (аргументи командного рядка). Ці Параметри – елементи вектора. Будується масив типу double, а на базі цього масиву – об'єкт класу DoubleVector. Далі програма виводить в консоль значення елементів вектору в вигляді: Вектор: 2.3 5.0 7.3.

5) Списки I та U містять результати N вимірювань струму і напруги на невідомому опорі R. Знайти число R методом найменших квадратів.

3. Обрати 1 варіант. Дано файл, що містить відомості про іграшки. Вказується назва іграшки (лялька, кубики, м'яч, конструктор), її вартість у копійках і вікові межі дітей, для яких іграшка призначена (наприклад, для дітей від двох до п'яти років). Крім того, для ляльки зазначено її розмір у сантиметрах, для кубиків – їхня кількість у наборі, для м'яча – його вага у грамах, для конструктора – кількість конструкцій, які з нього можна збудувати згідно інструкції. Одержати наступні відомості:

a. Вивести перелік іграшок, ціна яких не перевищує вказану і які підходять дітям 5 років у порядку зростання ціни.

b. Вивести перелік конструкторів у порядку зростання ціни. (Ціну виводити за зразком …грн…коп).

c. Вивести перелік найбільш коштовних іграшок (ціна яких відрізняється від ціни найкоштовнішої іграшки не більш ніж на 10 грн.). Відсортувати у порядку спадання ціни.

d. Перелік іграшок, що підходять дітям від 4 років до 10 років. Виводити в алфавітному порядку.

e. Перелік усіх кубиків, у порядку зростання цін. Ціну виводити за зразком …грн…коп.

f. Вивести перелік іграшок, будь яких, крім м'яча, що підходять дитині 3 років, щоб вартість іграшки не перевищувала задану суму. Перелік виводити у порядку спадання ціни.

g. Вивести перелік м'ячів із ціною, що вказана, або менша за неї, призначених дітям від 3 до 8 років. Перелік виводити у порядку зростання ваги

м’ячів.

h. Вивести перелік ляльок, що підходять дитині 3 років, щоб розмір іграшки не перевищував заданий. Перелік виводити у порядку збільшення розміру ляльки.

i. Вивести перелік кубиків, що підходять дитині 2 років, щоб їх вартість перевищувала задану суму, але була не більше 200 грн. Перелік виводити у порядку зростання цін.

j. Перелік найбільш дешевих іграшок (ціна яких відрізняється від ціни найдешевшої іграшки не більш ніж на 10 % її вартості) у порядку зростання ціни.

Контрольні запитання

1. Назвіть два базові інтерфейси, які лежать в основі Java Collection Framework, та поясніть їхнє основне призначення і відмінність у способі зберігання даних.
2. Інтерфейс Collection розширює інший важливий інтерфейс. Як він називається та яку основну можливість він надає всім колекціям, що його реалізують? Поясніть, що таке ітератор.
3. Опишіть основні характеристики та відмінності між інтерфейсами List та Set. Наведіть приклади їхніх реалізацій.
4. Інтерфейс Map призначений для зберігання даних у форматі «ключ – значення». Порівняйте дві реалізації цього інтерфейсу: HashMap та Hashtable, вказавши ключові відмінності між ними.
5. Розгляньте інтерфейс List та його реалізації ArrayList та LinkedList. В яких випадках доцільно використовувати ArrayList, а в яких – LinkedList? Поясніть причини такого вибору, враховуючи особливості їхньої внутрішньої реалізації.
6. Яке основне призначення інтерфейсу Queue? Поясніть принцип обробки елементів у черзі FIFO. Назвіть основні реалізації інтерфейсу Queue, представлені в тексті.
7. Опишіть особливості PriorityQueue. Чим ця реалізація відрізняється від звичайної черги FIFO? Як можна задати власний порядок сортування елементів у PriorityQueue?
8. Що таке інтерфейс Deque? Яку реалізацію Deque наведено в тексті та які її переваги в порівнянні з Stack та LinkedList при використанні як LIFO та FIFO структури даних?
9. Поясніть різницю між ArrayList та LinkedList з точки зору їхньої внутрішньої реалізації та продуктивності операцій додавання/видалення елементів у середину списку та доступу до елементів за індексом.
10. Які особливості класу HashSet? Чому важливо перевизначати методи equals() та hashCode() при використанні власних класів у HashSet? Чим LinkedHashSet відрізняється від HashSet?

11. Чим відрізняється інтерфейс Set від List у колекціях Java? Наведіть приклади реалізацій обох інтерфейсів.

12. Яке призначення інтерфейсу Queue, та в яких випадках доцільно використовувати його реалізації, зокрема PriorityQueue та ArrayDeque?

13. Поясніть відмінності між класами HashMap та TreeMap. У яких ситуаціях доцільно використовувати кожен з них?

14. Які особливості реалізації має клас HashSet, і як він забезпечує унікальність елементів?

15. Що таке колекційна ітерація в Java і які основні способи доступу до елементів колекцій (через Iterator, for-each, лямбда-вирази)?

Розділ 5. РОБОТА З РЯДКАМИ

5.1. Об’єкти (змінні) типу String

Для представлення даних, в Java існують примітивні типи даних. Серед цих типів немає рядкового типу даних. Немає також масиву символів. Для роботи з рядками в Java використовується клас String [4]. Об’єкт класу String дозволяє виконувати різні операції над рядками символів. Об’єкт (змінну) типу String можна присвоювати іншому об’єкту типу String. Перед використанням, об’єкт типу String обов’язково має бути ініціалізований. Рядки символів представляють собою літерали. Будь-яка рядкова константа (рядковий літерал) є об’єктом класу String.

Приклади рядкових літералів (рядкових констант):

```
«Це рядок символів»
«Два\nрядки»
«Три\nрядки\nсимволів»
«Тема: \’Рядки символів\’»
```

Приклад опису об’єктів типу String:

```
// опис об’єктів (змінних) типу String
String s; // описується об’єкт (змінна) з іменем s типу String
String str, myString;
```

Після опису, об’єкту типу String можна присвоювати рядки символів (літерали):

```
s = «Hello world!»;
str = s; // str = «Hello world!»
myString = «Hello « + «world!»; // myString = «Hello world!»
```

Ініціалізація об’єкта класу String може виконуватися як за допомогою оператора присвоювання змінній класу String рядкової змінної або рядкового літерала, або при створенні об’єкта за допомогою оператора *new* з використанням одного з конструкторів класу String:

```
// початкова ініціалізація змінних типу String
String s1 = «text»;
String s2 = s1; // s2 = «text»
String s3 = «This is a « + s2, s4 = s3 + «!»;
// s3 = «This is a text», s4 = «This is a text!»
```

В Java два рядки перевіряються на рівність:

- за допомогою операцій порівняння ‘= =‘ або ‘!=‘;
- за допомогою методу *equals()* класу String.

Загальна форма методу *equals()*:

```
boolean equals(другий_рядок)
boolean equalsIgnoreCase(другий_рядок)
```

Приклад:

```
// порівняння рядків
boolean b;
String s1, s2;
s1 = «Text»;
s2 = «Text»; // операція порівняння ==
b = s1 == s2; // b = true
b = s1 == «TEXT»; // b = false

// метод equals()
b = s1.equals(s2); // b = true
b = s1.equals(«TEXT»); // b = false
```

Якщо рядки рівні, то метод *equals()* повертає значення *true*. В іншому випадку повертається *false*. Метод *equalsIgnoreCase()* працює аналогічно методу *equals()*. Відмінність цього методу від методу *equals()* полягає в тому, що при порівнянні рядків ігнорується верхній регістр символів.

Використання в операторі умовного переходу *if*:

```
// порівняння рядків
String s1, s2;
s1 = «Text1»;
S2 = «Text2»;

// операція !=
if (s1!=s2)
   System.out.println(«Рядки не рівні»);
else
   System.out.println(«Рядки рівні»);

// метод equals()
if (s1.equals(s2))
   System.out.println(«Рядки рівні»);
```

```
else
   System.out.println(«Рядки не рівні»);
```

Використання методу *equalsIgnoreCase()*:

```
// метод equalsIgnoreCase()
String s1, s2;
boolean b;
s1 = «Java Eclipse»;
s2 = «JAVA ECLIPSE»;
b = s1.equalsIgnoreCase(«JAVA ECLIPSE»); // b = true
b = s2.equalsIgnoreCase(s1); // b = true
```

Оскільки в Java рядки є об'єктами, для порівняння рядків можна використовувати оператор «==» і метод *equals* (Object об'єкт).

- Щоб визначити довжину рядка (кількість символів), потрібно використати метод *length()* класу String. Загальна форма методу **length()**: *int length()*.

Приклад:

```
// метод length()
String s;
s = «Method length()»;
int n;
n = s.length(); // n = 15
```

- Метод **charAt()** – повертає символ, розташований за вказаним індексом рядка. Індекси рядків в Java починаються з нуля: *char charAt(індекс)*.

Приклад:

```
// метод charAt()
String s;
s = «Java Eclipse»;
char c;
c = s.charAt(0); // c = 'J'
c = s.charAt(1); // c = 'a'
c = s.charAt(2); // c = 'v'
c = s.charAt(3); // c = 'a'
```

➢ Метод **codePointAt**(int index) повертає Unicode-символ за заданим індексом. Метод **codePointBefore**(int index) повертає Unicode-символ, який передує заданому індексу.

Приклад:

```
// метод codePointAt()
String s;
s = «Java Eclipse»;
int n;
n = s.codePointAt(0); // n = 74 – код символу ‘J’
n = s.codePointAt(5); // n = 69 – код символу ‘E’

// метод codePointBefore()
n = s.codePointBefore(3); // n = 118 – код символу ‘v’
n = s.codePointBefore(1); // n = 74 – код символу ‘J’
// помилка: «String index out of range: 0»
// n = s.codePointBefore(0);
```

➢ Методи **compareTo()** та **compareToIgnoreCase()** порівнюють два рядки в лексикографічному порядку. Методи доцільно використовувати в алгоритмах сортування рядків. Загальна форма методів:

```
int compareTo(другий_рядок)
int compareToIgnoreCase(другий_рядок)
```

Методи повертають цілочислове значення: < 0, якщо другий рядок слідує після першого в лексикографічному порядку; 0, якщо рядки одинакові; > 0, якщо другий рядок слідує перед першим рядком в лексикографічному порядку. Метод *compareToIgnoreCase()* не враховує регістр символів.

В Java *compareTo()* отримує значення 0, якщо аргумент є рядком лексично рівним порівнюваному рядку; значення менше 0, якщо аргумент є рядком лексично більшим, ніж порівнюваний рядок; і значення більше 0, якщо аргумент є рядком лексично меншим цього рядка. Метод *compareToIgnoreCase()* – в порівнює лексично впорядковані два рядки, ігноруючи регістр букв [4-5].

Приклад:

```
// метод compareTo()
String s1, s2;
int n;
s1 = «Java Eclipse»;
s2 = «»;
```

```
n = s1.compareTo(s2); // n = 12
n = s1.compareTo(«Java»); // n = 8
n = s2.compareTo(«T»); // n = -1

// метод compareToIgnoreCase()
n = s1.compareToIgnoreCase(«java ECLIPSE»); // n = 0, рядки рівні
```

➢ Об'єднання рядків, метод **concat()** дозволяє з'єднувати два рядки між собою. Також з'єднувати рядки можна за допомогою перевантаженої операції '+'.

Загальна форма методу:

String concat(другий_рядок)

де другий_рядок – рядок, що додається до поточного рядка.

Приклад:

```
// метод concat()
String s;
s = «Hello»;
s = s.concat(«world»); // s = «Hello world»
s = s.concat(«!»); // s = «Hello world!»

// операція '+'
s = «Hello»;
s = s + «world»; // s = «Hello world»
s = s + «!»; // s = «Hello world!»
```

Крім методу *concat()* та операції '+' рядки можна об'єднувати з допомогою операції '+='. Це є зручним, коли потрібно об'єднувати рядки з довгими іменами. Немає потреби двічі вводити довге ім'я рядка.

Приклад:

```
// конкатенація рядків, операція += для рядків
// конкатенація об'єктів
String s1 = «Hello»;
String s2 = «world!»;
s1 += s2; // s1 = s1 + s2 = «Hello world!»
// конкатенація рядкових констант
s2 = «3»;
s2 += «.»;
s2 += «14»;
```

```
s2 += «15»; // s2 = «3.1415»
// операція += в поєднанні зі складним виразом
s1 = «5»;
s2 = «»;
s2 += s1 + «*» + s1 + «=2» + s1; // s2 = «5*5=25»
s2 += s2 + s1; // s2 = «5*5=255*5=255»
```

Демонструються різноманітні способи конкатенації (об'єднання) рядків з допомогою операції '+='. Як видно з прикладу, операція s1 += s2 неявно замінюється операцією s1 = s1 + s2. Метод *concat()* в Java об'єднує рядки, шляхом додавання одного рядка в кінець іншого.

➢ Визначення наявності підрядка в рядку. Метод **contains()**, призначений для визначення наявності підрядка в рядку. Загальна форма методу *contains(): boolean contains(другий_рядок)* Метод повертає значення *true*, якщо другий_рядок є підрядком даного рядка. В іншому випадку повертається *false*. Клас String містить метод *contains*, який повертає *true*, якщо задана послідовність символів міститься в рядку.

Приклад:

```
// метод contains()
String s1, s2;
boolean b;
s1 = «Java Eclipse»;
s2 = «Eclipse»;
b = s1.contains(s2); // b = true
b = s2.contains(«eclipse»); // b = false
```

➢ Метод **endsWith()** призначений для визначення того, чи підрядок є закінченням вихідного рядка. Загальна форма методу: *boolean endsWith(другий_рядок).* Метод *endsWith()* – перевіряє, чи закінчується рядок зазначеним виразом.

Приклад:

```
// метод endsWith()
String s1, s2;
boolean b;
s1 = «Java Eclipse»;
s2 = «Eclipse»;
b = s1.endsWith(«Ecl»); // b = false
b = s2.endsWith(s1); // b = false
b = s1.endsWith(s2); // b = true
```

```
b = s1.endsWith(«Java»); // b = false
b = s1.endsWith(«ipse»); // b = true
```

➢ Розбиття рядка на масив цілих чисел. Метод *getBytes()* дозволяє отримати значення рядка у вигляді масиву значень типу *byte*, в якому кожне значення є кодом символу в таблиці символів. Наприклад, рядок «Java» складається з послідовності символів з кодами: 74 – ‘J’, 97 – ‘a’, 118 – ‘v’, 97 – ‘a’.

Метод *getBytes()* має дві форми: *getBytes(String charsetName)* – кодує даний рядок в послідовність байтів, використовуючи *charsetName(кодування)*, зберігає результат в новий масив байтів; *getBytes()* – кодує даний рядок в послідовність байтів, за замовчуванням за допомогою платформи *charset*, зберігає результат в новий масив байтів.

Приклад:

```
// метод getBytes()
String s;
byte[] c;
s = «Java Eclipse»;
c = s.getBytes();
// c = { 74, 97, 118, 97 } – коди символів слова «Java»
```

➢ Визначення першого та останнього входження символу або рядка. Методи **indexOf()** та **lastIndexOf().** Метод *indexOf()* має декілька перевантажених реалізацій. В одній реалізації метод повертає позицію першого входження символу з заданим кодом. В другій реалізації метод повертає позицію першого входження заданого підрядка в рядку. Якщо не знайдено співпадінь, то метод повертає -1.

Загальна форма реалізацій методу *indexOf(): int indexOf(код_символу) int indexOf(другий_рядок).*

Приклад:

```
// метод indexOf()
String s;
int d;
s = «Java Eclipse»;

// 1-й варіант методу
d = s.indexOf(118); // d = 2, позиція символу з кодом 118 – символу ‘v’
d = s.indexOf(130); // d = -1, символа з кодом 130 немає в рядку s
d = s.indexOf(74); // d = 0
```

```
// 2-й варіант методу – позиція 1го входження підрядка
d = s.indexOf(«ava»); // d = 1
d = s.indexOf(«abs»); // d = -1, входження немає
```

➢ Метод **lastIndexOf()** визначає позицію останнього входження символу або підрядка в заданому рядку. Загальна форма двох перевантажених реалізацій методу: *int lastIndexOf(код_символу) int lastIndexOf(другий_рядок).*

Метод **indexOf()** шукає в рядку заданий символ або рядок, та повертає їх індекс (тобто порядковий номер). Метод: повертає індекс, під яким символ або рядок перший раз з'являється в рядку; повертає (-1) якщо символ бо рядок не знайдені. Метод *lastIndexOf(int ch)* – повертає індекс останнього входження зазначеного символу у заданому рядку або -1, якщо символ не зустрічається.

Приклад:

```
String s;
int d;
s = «Java»;
d = s.lastIndexOf(97);
// d = 3, останнє входження символу з кодом 97 – ‘a’
d = s.indexOf(97);
// d = 1, перше входження символу з кодом 97
d = s.lastIndexOf(«va»); // d = 2
d = s.lastIndexOf(«a»); // d = 3
```

➢ Перевірка, чи рядок є пустий. Метод **isEmpty()** повертає значення *true*, якщо рядок пустий “”. В іншому випадку повертається *false*. Загальна форма методу*: boolean isEmpty()* Зробити перевірку, чи рядок є пустим, також можна з допомогою операції = =.

Приклад:

```
// метод isEmpty()
String s;
boolean b;
s = «Java Eclipse»;
b = s.isEmpty(); // b = false
s = «»;
b = s.isEmpty(); // b = true

// операція ‘==‘
b = s==«»; // b = true
s = «Java»;
b = s==«»; // b = false
```

➢ Заміна символів в тексті. Метод **replace()** здійснює заміну символів у тексті. Метод має декілька варіантів реалізацій:

String replace(символ1, символ2)
String replace(рядок1, рядок2)
String replaceFirst(рядок1, рядок2)

де символ1 – символ, який потрібно замінити;
символ2 – символ, який замінюється;
рядок1 – текст, який потрібно замінити;
рядок2 – текст, який замінюється іншим текстом.

Метод *replaceFirst()* замінює тільки перше входження рядок2 у рядок1.

Метод *replace()* заміняє зазначений символ (або підрядок) в рядку на новий.

Приклад:

```
// метод replace()
String s,s2;
s = «TextTextText»; // заміна одного символу
s2 = s.replace('T', 'X'); // s2 = «XextXextXext»
// заміна одного тексту на інший
s2 = s.replace(«T», «TT»); // s2 = «TTextTTextTText»
s2 = s.replace(«Text», «ABC»); // s2 = «ABCABCABC»
// метод replaceFirst()
s2 = s.replaceFirst(«Te», «ABC»); // s2 = «ABCxtTextText»
```

➢ Метод **startsWith()** визначає, чи підрядок, що є параметром методу, являється початком вихідного рядка. Загальна форма методу: *boolean startsWith(другий_рядок)*. Має два варіанти і перевіряє чи починається рядок із зазначеного префіксу, починаючи із зазначеного індексу або з початку рядка (за замовчуванням).

Приклад:

```
// метод startsWith()
String s;
boolean b;
s = «TextTextText»;
b = s.startsWith(«Te»); // b = true
b = s.startsWith(«ext»); // b = false
```

➢ Метод **substring()** виділяє підрядок з рядка. Метод має декілька варіантів реалізацій.

Варіант 1. *String substring(індекс)* У цьому випадку виділяється підрядок починаючи з позиції індекс і до кінця рядка.

Варіант 2. *String substring(індекс1, індекс2)* У цьому випадку виділяється підрядок починаючи з позиції індекс1 і закінчуючи позицією індекс2.

Підрядок починається з символу, заданого індексом, і триває до кінця цього рядка або до endIndex-1, якщо введений другий аргумент.

Приклад:

```
// метод substring()
String s, s2;
// варіант 1
s = «This is a text»;
s2 = s.substring(2); // s2 = «is is a text»
// варіант 2
s2 = s.substring(2,3); // s2 = «i»
s2 = s.substring(2,4); // s2 = «is»
s2 = s.substring(6,13); // s2 = «s a tex»
```

➢ Розбиття рядка на символи. Метод **toCharArray()** дозволяє перевести рядок типу String в масив символів *char[]*. Метод *toCharArray()* створює з рядка масив символів.

Приклад:

```
// метод toCharArray()
String s;
char c[];
s = «Java»;
c = s.toCharArray();
// c = {‘J’, ‘a’, ‘v’, ‘a’}
```

➢ Приведення до потрібного регістру символів. Методи **toLowerCase()** та **toUpperCase()** Метод *toLowerCase()* приводить символи рядка до нижнього регістру. Метод *toUpperCase()* приводить символи рядка до верхнього регістру. Загальна форма методів: *String toLowerCase() String toUpperCase().*

Приклад:

```
// метод toLowerCase()
String s1;
```

```
String s2;
s1 = «Java Eclipse»;
s2 = s1.toLowerCase(); // s2 = «java eclipse»
s2 = s1.toUpperCase(); // s2 = «JAVA ECLIPSE»
```

➢ Перетворення масиву символів *char[]* в рядок String. Метод **copyValueOf()** перетворює масив символів типу *char[]* в рядок символів. Загальна форма методу: *String copyValueOf(масив_символів)*. Має дві різні форми: *public static String copyValueOf (char [] data)* – повертає рядок, який представляє собою послідовність символів в заданому масиві. Та *public static String copyValueOf (char[] data, int offset, int count)* – повертає рядок, який представляє собою послідовність символів у заданому масиві.

Приклад:

```
// метод copyValueOf()
String s;
char[] c = { ‘J’, ‘a’, ‘v’, ‘a’ };
s = String.copyValueOf(c); // s = «Java»
```

➢ Перетворення числа в рядок. Метод **valueOf()** дозволяє переводити значення числового або іншого типу в його рядкове представлення. Повертає відповідний числовий об’єкт, що містить значення передачі аргументу, тобто перетворює в потрібний тип даних. Аргумент можна перетворити в *int, double, float* та інші типи даних, наприклад, можна перетворити рядок в число. Метод *valueOf()* в Java є статичним методом. Метод може сприймати два аргументи, де один є рядком, а інший системою числення [4-5].

Приклад:

```
// метод valueOf()
String s;
double d; // тип double
d = 3.8567;
s = String.valueOf(d); // s = «3.8567»
int t; // тип int
t = -3903;
s = String.valueOf(t); // s = «-3903»
char c = ‘+’; // тип char
s = String.valueOf(c); // s = «+»
boolean b; // тип boolean
b = true;
s = String.valueOf(b); // s = true
```

5.2. Поняття масиву рядків

Як і будь-яка мова програмування, Java може реалізовувати масиви рядків. Будь-який рядок в має тип String. Одновимірний масив рядків має тип *String[]*. Двовимірний масив рядків має тип *String[][]*. Загальна форма оголошення та виділення пам'яті для одновимірного масиву рядків:

```
String[] arrayName = new String[size];
```

де *String* – вбудований в Java клас, що реалізує рядок символів;
arrayName – ім'я об'єкту (екземпляру) типу String. Фактично, *arrayName* є посиланням на об'єкт типу String;
size – розмір масиву (кількість рядків, кількість елементів типу String).

Оголошення одновимірного масиву рядків та виділення пам'яті для нього можна реалізувати іншим чином:

```
String[] arrayName;
arrayName = new String[size];
```

Приклад оголошення та використання одновимірного масиву рядків:

```
// оголошення одновимірного масиву рядків
String[] arrayS = new String[5];
// заповнення початковими значеннями
arrayS[0] = «abcd»;
arrayS[1] = «Hello»;
arrayS[2] = «»; // пустий рядок
arrayS[3] = «bestprog»;
arrayS[4] = «;:\\+=«;// комбінація «\\» замінюється на «\»
// використання у виразах
arrayS[4] = arrayS[1] + « » + arrayS[3]; // arrayS[4] = «Hello bestprog»
arrayS[4] += «.net»; // arrayS[4] = «Hello bestprog.net»

// ініціалізація одновимірного масиву рядків
String[]
M = {«Sunday», «Monday», «Tuesday», «Wednesday», «Thursday», «Friday»,
      «Saturday»};
String s;
s = M[2]; // s = «Tuesday»
s = M[4]; // s = «Thursday»
```

Ініціалізація одновимірного масиву рядків така сама як ініціалізація одновимірного масиву будь-якого іншого типу.

В деяких задачах виникне потреба в оголошенні двовимірного масиву рядків. Загальна форма оголошення двовимірного масиву рядків наступна:

```
String[][] matrName = new String[n][m];
```

де *matrName* – ім’я об’єкту (посилання на об’єкт), що є двовимірним масивом типу String;
n – кількість рядків у масиві *matrName*;
m – кількість стовпців у масиві *matrName*.

Можливий також інший спосіб оголошення та виділення пам’яті для двовимірного масиву рядків:

```
String[][] matrName; // оголошення посилання на 2-мірний масив
matrName = new String[n][m];
```

Приклад оголошення та використання двовимірного масиву рядків:

```
// оголошення двовимірного масиву рядків
String[][] matrS = new String[2][3];
// заповнення масиву значеннями
for (int i=0; i<matrS.length; i++)
  for (int j=0; j<matrS[i].length; j++)
    matrS[i][j] = «matrS[» + i + «][» + j + «]»;
// перевірка
String s;
s = matrS[0][0]; // s = «matrS[0][0]»
s = matrS[1][1]; // s = «matrS[1][1]»
```

Ініціалізація двовимірного масиву рядків нічим не відрізняється від ініціалізації двовимірного масиву будь-якого примітивного типу. Елементами масиву є звичайні рядки. Нижче наведено приклад ініціалізації двовимірного масиву рядків з іменем *M*:

```
// оголошення масиву M з початковою ініціалізацією
String M[][] = new {
  { «a1», «a2», «a3» },
  { «b1», «b2», «b3» },
  { «a1», «c2», «a1» }
};
// перевірка
String s;
s = M[0][1]; // s = «a2»
s = M[1][0]; // s = «b1»
```

Довжина масиву рядків. Властивість **length**. Для одновимірних масивів кількість рядків *n* визначається як:

```
String[] arrayS = new String[25];
int n;
n = arrayS.length;
```

Для двовимірних масивів кількість рядків та стовпців визначається наступним чином:

```
// matrS – двовимірний масив рядків
String[][] matrS = new String[2][3];
int n, m;
n = matrS.length; // n = 2 – кількість рядків
m = matrS[0].length; // m = 3 – кількість стовпців
m = matrS[1].length; // m = 3
```

Пошук заданого рядка в одновимірному масиві рядків:

```
// оголошення масиву рядків
String M[] = new String[5];
String s = «May»; // рядок, який потрібно знайти
boolean f_is;
M[0] = «January»; // заповнення масиву значеннями
M[1] = «February»;
M[2] = «May»;
M[3] = «October»;
M[4] = «December»;
// пошук рядка
f_is = false;
for (int i = 0; i < M.length; i++)
   if (M[i]==s) {
      f_is = true;
      break; }
// виведення результату
if (f_is)
   System.out.println(«Шуканий рядок є в масиві.»);
else
   System.out.println(«Шуканого рядка немає в масиві.»);
```

Сортування одновимірного масиву рядків за алфавітом методом вставки. Для порівняння двох рядків в лексикографічному порядку у класі String розроблено метод **compareTo().**

Загальна форма методу наступна*: int compareTo(другий_рядок).*

Метод повертає: < 0, якщо другий рядок слідує після першого рядка в лексикографічному порядку; = 0, якщо рядки однакові; > 0, якщо другий рядок слідує перед першим в лексикографічному порядку.

Приклад:

```
// сортування масиву рядків методом вставки
String[] M = { «abc», «bde», «abcd», «bcdef», «cdef», «fghij», «aaa» };
String s;
for (int i = 0; i < M.length-1; i++) // сортування
  for (int j = i; j >= 0; j--)
    if (M[j].compareTo(M[j+1])>0) { // обміняти M[j] та M[j+1] місцями
      s = M[j];
      M[j] = M[j+1];
      M[j+1] = s; }
for (int i = 0; i < M.length; i++) // виведення результату
  System.out.println(M[i]);
```

Визначення **кількості входжень заданого рядка** у двовимірному масиві рядків:

```
// обчислення кількості входжень заданого рядка у двовимірному масиві
// оголошення масиву M з початковою ініціалізацією
String M[][] = {
  { «abcd», «abc», «bcd» },
  { «acd», «bcd», «abcd» },
  { «abc», «bc», «cde» }
};
String s = «abc»; // рядок, кількість входжень якого потрібно обчислити
int k = 0; // к-сть входжень, результат
for (int i = 0; i < M.length; i++)
  for (int j = 0; j < M[i].length; j++)
    if (M[i][j]==s)
      k++; // k = 2
```

Заміна рядка у двовимірному масиві рядків. Нехай задано: двовимірний масив рядків з іменем *matrS*; рядок для заміни – *s1*; рядок *s2* – це рядок, який замінює рядок *s1*:

```
// оголошення двовимірного масиву рядків
String[][] matrS = new String[2][3];
// заповнення матриці matrS довільними значеннями
matrS[0][0] = «abc»;
matrS[0][1] = «cba»;
matrS[0][2] = «def»;
```

```
matrS[1][0] = «abc»;
matrS[1][1] = «fff»;
matrS[1][2] = «qqq»;
// заповнення значеннями рядків s1 та s2
String s1 = «abc»; // рядок, який заміняють
String s2 = «mmm»; // рядок, яким замінюється рядок s1
for (int i = 0; i < matrS.length; i++) // цикл обчислення
  for (int j = 0; j < matrS[i].length; j++)
    if (matrS[i][j] == s1)
      matrS[i][j] = s2;
for (int i = 0; i < matrS.length; i++) {// виведення результату
  for (int j = 0; j < matrS[i].length; j++)
    System.out.print(matrS[i][j] + « »);
  System.out.println(); }

//  Отримаємо результат:
// mmm cba def
// mmm fff qqq
```

5.3. StringBuffer і StringBuilder

Клас String являє собою незмінні послідовності символів постійної довжини і часте використання об'єктів даного класу займає багато місця в пам'яті. Клас *StringBuffer* представляє такі послідовності символів, які можна розширювати та змінювати. Тобто дозволяє вставляти символи і підрядки в існуючий рядок і в будь-якому місці. Даний клас набагато економічніший у плані використання пам'яті [4-5, 7].

Існує чотири конструктора класу:

StringBuffer() – резервує місце під 16 символів без перерозподілу пам'яті

StringBuffer(int capacity) – явно встановлює розмір буфера

StringBuffer (String string) – встановлює початковий вміст і резервує 16 символів без повторного резервування

StringBuffer(CharSequence cs) – створює об'єкт, що містить послідовність символів і резервує місце ще під 16 символів

Метод *length()* класу *StringBuffer* – дозволяє отримати поточну довжину об'єкта. Метод *capacity()* – дозволяє отримати поточний обсяг виділеної пам'яті.

Приклад:

```
StringBuffer sb = new StringBuffer(«Коте»);
tvInfo.setText(«Обсяг пам'яті: » + sb.capacity()); // поверне 20
```

У прикладі вище, слово «Коте» складається з чотирьох символів, в

пам’яті виділено запасний простір для додаткових 16 символів. Для такої найпростішої операції виграшу немає, але в складних прикладах, коли доводиться з’єднувати велику кількість рядків, продуктивність різко зростає.

Приклад:

```
String str = «Java»;
StringBuffer strBuffer = new StringBuffer (str);
System.out.println («Обсяг:» + strBuffer.capacity ()); // 20
strBuffer.ensureCapacity (32);
System.out.println («Обсяг:» + strBuffer.capacity ()); // 42
System.out.println («Довжина:» + strBuffer.length ()); // 4
```

У прикладі вище, на самому початку StringBuffer ініціалізується рядком «Java», його обсяг становить 4 + 16 = 20 символів. Потім збільшуємо обсяг буфера за допомогою виклику *strBuffer.ensureCapacity(32)* підвищуємо мінімальний обсяг буфера до 32 символів. Однак фінальний обсяг може відрізнятися в більшу сторону. Так, в даному випадку отримуємо обсяг не 32 і не 32 + 4 = 36, а 42 символи. Справа в тому, що з метою підвищення ефективності Java може додатково виділяти пам’ять. Але в будь-якому випадку незалежно від обсягу довжина рядка, яку можна отримати за допомогою методу *length()*, в *StringBuffer* залишається колишньою – 4 символи (так як в слові «Java» – 4 символи).

Методи класу StringBuffer:

- щоб отримати рядок, який зберігається в StringBuffer, можна використовувати стандартний метод **toString()**;
- **ensureCapacity()** – дозволяє попередньо виділити місце для певної кількості символів, якщо необхідно додавати велику кількість маленьких рядків;
- **setLength(int length)** – встановлює довжину рядка. Значення має бути невід’ємним;
- **charAt(int index)** та **setCharAt(int index, char ch)** – дозволяють витягти значення окремого символу за допомогою методу *charAt()* і встановити нове значення символу за допомогою методу *setCharAt()*, вказавши індекс символу і його значення;
- **getChars()** – дозволяє скопіювати підрядок з об’єкта класу StringBuffer в масив. Необхідно подбати, щоб масив був достатнього розміру для прийому потрібної кількості символів зазначеного підрядка;
- **append()** – з’єднує дані будь-яких інших типів. Є кілька перевантажених версій:

```
StringBuffer append (String string)
StringBuffer append (int number)
```

StringBuffer append (Object object)

Рядкове представлення кожного параметра отримують через метод String.valueOf() і потім отримані рядки склеюються в остаточний рядок.

- **insert()** – вставляє один рядок в інший. Також можна вставляти значення інших типів, які будуть автоматично перетворені в рядки. Потрібно вказати індекс позиції, куди буде вставлятися рядок.

- **reverse()** – дозволяє змінити порядок символів на зворотний.

- **delete()** – видаляє послідовність символів між заданими індексами символів.

- **deleteCharAt()** – видаляє один символ із зазначеної позиції.

- **replace()** – дозволяє замінити один набір символів на інший. Потрібно вказати початковий і кінцевий індекс і рядок заміни.

- **substring()** – дозволяє отримати частину вмісту. Є дві форми методу. У першому випадку потрібно вказати індекс початку позиції, з якої потрібно отримати підрядок. У другому варіанті вказується початковий індекс і кінцевий індекс, якщо потрібно отримати текст із середини рядка.

Клас StringBuilder ідентичний класу StringBuffer і володіє більшою продуктивністю. Однак він не синхронізований, тому його не потрібно використовувати в тих випадках, коли до рядка, який змінюється, звертаються кілька потоків [7].

Створення нового об’єкту:

```
StringBuilder builder = new StringBuilder();
```

Додавання нового фрагменту в існуючий рядок:

```
builder.append(ch); // можна додати один символ
builder.append(sometext); // можна додати готовий рядок
String completedString = builder.toString(); // результуючий рядок
```

З’єднувати рядки ланцюжком:

```
StringBuilder().append(s1).append(s2).append(s3).toString();
```

Класи StringBuffer і StringBuilder схожі між собою, практично двійники, вони мають однакові конструктори, одні і ті ж методи, які однаково використовуються. Єдина їх відмінність полягає в тому, що клас StringBuffer синхронізований і потокобезпечний. Тобто клас StringBuffer зручніше використовувати в багатопотокових застосуваннях, де об’єкт даного класу може змінюватися в різних потоках. Якщо ж мова про багатопотокові додатки не йде, то краще використовувати клас StringBuilder, який не потокобезпечний, але при цьому працює швидше, ніж StringBuffer в

однопоточних додатках. За всіма своїми операціями StringBuffer та StringBuilder нагадують клас String.

Відмінність між String, StringBuilder та StringBuffer:

1. Класи StringBuffer і StringBuilder в використовуються, коли виникає необхідність зробити багато змін в рядку символів.
2. На відміну від String, об’єкти типу StringBuffer і StringBuilder можуть бути змінені знову і знову.
3. StringBuilder був введений починаючи з Java 5.
4. Основна відмінність між StringBuffer і StringBuilder полягає у тому, що методи StringBuilder не є безпечними для потоків (несинхронізовані).
5. Рекомендується використовувати StringBuilder щоразу, коли це можливо, тому що він швидший, ніж StringBuffer.
6. Якщо необхідно забезпечити безпеку потоків, найкращим варіантом є об’єкти StringBuffer.

Завдання для самостійного виконання:

1. Аналіз та обробка рядків. Для кожного з наведених нижче рядків виконайте наступні дії:

- Визначте довжину рядка.
- Виведіть символ, що знаходиться за індексом 3 (якщо індекс існує).
- Перевірте, чи закінчується рядок підрядком «ing».
- Замініть всі входження символу ‘a’ на символ ‘*’.
- Перетворіть рядок на нижній регістр.
- Варіанти рядків:

1) «Programming»
2) «Java»
3) «Coding is fun»
4) «String manipulation»
5) «Application»
6) «Testing»
7) «Development»
8) «Algorithm»
9) «Information technology»
10) «Example»

2: Робота з масивами рядків. Створіть одновимірний масив, що містить 5 різних слів. Для цього масиву виконайте наступні дії:

- Виведіть всі елементи масиву на екран.
- Знайдіть та виведіть індекс першого входження слова «test» (якщо воно є).
- Відсортуйте масив за алфавітом та виведіть відсортований масив.

- Перевірте, чи містить масив хоча б одне слово, що починається на літеру ‘a’ (регістр не враховувати).
- Створіть новий рядок, що є конкатенацією всіх елементів масиву, розділених пробілом.
- Варіанти початкових слів для масиву:

1) {«apple», «banana», «cherry», «date», «fig»}
2) {«code», «debug», «test», «deploy», «release»}
3) {«sun», «moon», «stars», «planet», «galaxy»}
4) {«book», «pen», «notebook», «pencil», «eraser»}
5) {«car», «bus», «train», «plane», «ship»}
6) {«dog», «cat», «bird», «fish», «hamster»}
7) {«red», «green», «blue», «yellow», «orange»}
8) {«one», «two», «three», «four», «five»}
9) {«flower», «tree», «grass», «leaf», «root»}
10) {«computer», «keyboard», «mouse», «monitor», «printer»}

3: Маніпуляції зі StringBuffer. Для кожного з наведених нижче початкових рядків StringBuffer виконайте наступні дії:
- Додайте в кінець рядок «!».
- Вставте в позицію з індексом 5 (якщо існує) рядок «TEST».
- Замініть символи з індексами від 2 до 6 (не включно) на рядок «***».
- Видаліть символ, що знаходиться за індексом 1 (якщо існує).
- Переверніть рядок та виведіть результат.
- Варіанти початкових рядків StringBuffer:

1) new StringBuffer(«example»)
2) new StringBuffer(«stringbuffer»)
3) new StringBuffer(«java»)
4) new StringBuffer(«code»)
5) new StringBuffer(«modify»)
6) new StringBuffer(«text»)
7) new StringBuffer(«append»)
8) new StringBuffer(«insert»)
9) new StringBuffer(«delete»)
10) new StringBuffer(«reverse»)

4: Порівняння продуктивності String, StringBuffer та StringBuilder. Для кожного з наведених нижче сценаріїв виконайте наступне:
- Створіть цикл, який виконує конкатенацію заданої кількості рядків (наведених у варіантах) у змінну типу String, StringBuffer та StringBuilder. Виміряйте час виконання кожної операції (створення об’єкта та виконання циклу конкатенації). Виведіть на екран час виконання для кожного типу. Проаналізуйте отримані результати та поясніть різницю в продуктивності.

- Варіанти сценаріїв (кількість конкатенацій та рядок для конкатенації):

1) Кількість: 1000, Рядок: «a»
2) Кількість: 5000, Рядок: «long»
3) Кількість: 10000, Рядок: «short»
4) Кількість: 2000, Рядок: «medium_text»
5) Кількість: 7500, Рядок: «single»
6) Кількість: 15000, Рядок: «char»
7) Кількість: 3000, Рядок: «another_word»
8) Кількість: 6000, Рядок: «test_string»
9) Кількість: 9000, Рядок: «value»
10) Кількість: 4000, Рядок: «item»

Контрольні запитання

1. Які два способи порівняння рядків на рівність існують в Java, та в чому полягає принципова відмінність між використанням операторів == та != і методів equals() та equalsIgnoreCase() класу String?
2. Окрім методу concat() та операції +, який ще спосіб об'єднання рядків пропонує Java, і в яких випадках його використання може бути більш зручним? Наведіть приклад.
3. Для чого використовуються методи contains(), endsWith(), startsWith() класу String? Наведіть приклади їхнього застосування для перевірки наявності підрядка, закінчення або початку рядка.
4. Для чого призначений метод getBytes() класу String та які дві його форми існують? Поясніть, що саме представляє собою масив типу byte, який повертається цим методом.
5. Яка різниця між методами indexOf() та lastIndexOf() класу String? Наведіть приклади їхнього використання для пошуку як окремих символів, так і підрядків. Що повертають ці методи, якщо пошук не дав результатів?
6. Опишіть процес оголошення та ініціалізації одновимірного масиву рядків у Java. Наведіть приклади присвоєння початкових значень елементам масиву як під час оголошення, так і після нього.
7. Опишіть процес оголошення та ініціалізації двовимірного масиву рядків у Java. Як визначити кількість рядків та стовпців у такому масиві за допомогою властивості length?
8. Наведіть приклад коду, який демонструє пошук заданого рядка в одновимірному масиві рядків. Який метод класу String використовується для порівняння рядків при пошуку?
9. Поясніть принцип сортування одновимірного масиву рядків за алфавітом методом вставки. Який метод класу String є ключовим для порівняння рядків у цьому процесі?

10.Чим класи StringBuffer та StringBuilder відрізняються від класу String у контексті роботи зі змінними рядками? Опишіть основні конструктори та методи класів StringBuffer та StringBuilder.

11.Яка основна відмінність між класами StringBuffer та StringBuilder? У яких випадках рекомендується використовувати StringBuffer, а в яких – StringBuilder? Чим зумовлений вибір на користь одного з цих класів?

Розділ 6. ПОТОКИ

6.1. Основи багатопоточності

Потік (*thread*) – це окремий шлях виконання інструкцій програми. У типовому випадку сучасні додатки запускаються з одним потоком, що є достатнім для більшості задач. Проблеми виникають, коли необхідно одночасно обробляти декілька подій. Наприклад, якщо під час очікування введення з клавіатури програма не здатна оновлювати зображення на екрані, виникає необхідність у паралельному виконанні різних частин коду [3, 5, 7].

Багатопоточність дозволяє виконувати кілька потоків одночасно в межах однієї програми. Це не тотожне одночасному запуску декількох копій програми – в такому випадку операційна система створює окремі процеси з власними ресурсами. У системах на кшталт Unix створення нового процесу за допомогою механізму *fork()* означає дублювання коду і даних, що вимагає значних ресурсів та створює додаткове навантаження на процесор.

На відміну від цього, запуск нового потоку в межах існуючого процесу дозволяє ефективно використовувати ресурси шляхом спільного доступу до батьківського простору даних. Це робить багатопоточність ресурсозберігаючою технологією [19].

Основні характеристики потоків:

- Кожен потік має окремий стек викликів.
- Всі потоки одного процесу поділяють загальну пам’ять.
- Потік у Java не тотожний потоку операційної системи.
- JVM використовує власний планувальник потоків, незалежний від планувальника ОС.
- Існують два типи потоків у Java: потоки-демони (*daemon threads*) та призначені для користувача потоки (*user threads*). Завершення всіх призначених для користувача потоків призводить до завершення програми.

6.2. Створення потоків

У Java існує два основних способи створення потоків: розширення класу та реалізація інтерфейсу. Обидва підходи мають свої переваги та застосовуються в залежності від потреб програми.

Розширення класу Thread

Створення потоку можливе шляхом успадкування класу Thread, який уже містить усі необхідні методи для роботи з потоками. У цьому випадку необхідно перевизначити метод *run()*, який міститиме код, що виконується потоком.

Обмеження: Java не підтримує множинне наслідування класів, тому клас, який успадковується від Thread, не може одночасно наслідувати інші

класи. Це знижує гнучкість при проектуванні складних ієрархій.

Реалізація інтерфейсу Runnable

Більш гнучким способом створення потоку є реалізація інтерфейсу Runnable, який містить лише один метод – *public void run()*. Після реалізації інтерфейсу об'єкт передається конструктора класу Thread, що дозволяє відокремити логіку виконання від механізму запуску потоку.

Приклад:

```
class MyTask implements Runnable {
   public void run() {
      // код виконання потоку
   }
}
Thread t = new Thread(new MyTask());
t.start();
```

Особливості інтерфейсів:

- Містять лише абстрактні методи та константи (*static final* змінні).
- Не можуть реалізовувати методи напряму.
- Клас, що реалізує інтерфейс, зобов'язаний реалізувати всі його методи.
- Можуть розширювати інші інтерфейси, включаючи множинне розширення.
- Не можна створювати екземпляри інтерфейсу напряму через *new*.

Таким чином, клас Thread є повноцінним потоком та надає функціональність для створення й управління потоками. Інтерфейс Runnable визначає сутність, яку можна виконати – її реалізація слугує фундаментом для гнучкої багатопотокової архітектури.

Приклад створення потоку шляхом розширення класу Thread. Для створення потоку шляхом успадкування класу Thread необхідно:

1) Створити клас, який успадковується від Thread.
2) Перевизначити метод *run()*, що міститиме код, який виконується в окремому потоці.
3) Створити екземпляр цього класу.
4) Викликати метод *start()* для запуску потоку.

Застосовується у випадках, коли клас виконує лише одну задачу – виконання в потоці, і не планується наслідування від інших класів.

```
class MyThread extends Thread {
   public void run() {
      for (int i = 0; i < 5; i++) {
```

```
            System.out.println(«Потік працює: » + i);
            try {
                Thread.sleep(500); // затримка на 0.5 секунди
            } catch (InterruptedException e) {
                System.out.println(«Потік перервано»);
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread(); // створення потоку
        thread.start(); // запуск потоку
    }
}
```

Пояснення:

- Метод *run()* містить логіку, яка буде виконуватись у новому потоці.
- Метод *start()* викликає внутрішній механізм JVM для створення окремого потоку, після чого автоматично викликається *run()*.

6.3. Інтерфейс **Runnable** та його роль у створенні потоків

Клас Thread знаходиться у пакеті java.lang, тому його можна використовувати без явного імпорту. Він реалізує абстрактний метод *run()*, який походить з інтерфейсу Runnable. Саме цей інтерфейс задає основу для визначення задач, які мають виконуватись у потоках.

Особливості інтерфейсу Runnable:

- є функціональним інтерфейсом (є лише один абстрактний метод – *run()*),
- не містить реалізації логіки виконання потоку,
- призначений лише для визначення задачі, яка повинна виконуватись у потоці.

Реалізація Runnable є кращою, коли класу потрібно розширити інший клас, оскільки Java дозволяє успадковування лише від одного класу. Таким чином, реалізація Runnable забезпечує гнучкість і дотримання принципів об’єктно-орієнтованого програмування.

Механізм запуску. Інтерфейс Runnable сам по собі не створює окремого потоку. Його реалізація використовується у конструкторі класу Thread, який

відповідає за запуск нового потоку:

```
class MyTask implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(«Виконання задачі в окремому потоці: » + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(«Потік перервано»);
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Runnable task = new MyTask();       // створення задачі
        Thread thread = new Thread(task);   // обгортка в потік
        thread.start();                     // запуск потоку
    }
}
```

Таким чином, інтерфейс Runnable – це інструмент, який дозволяє описати завдання для виконання, але вся реальна робота з керування потоками виконується класом Thread. Такий підхід забезпечує розділення логіки задачі та механізму її виконання, що сприяє кращій структурі коду.

6.4. Інстанціювання та запуск потоків у Java

Інстанціювання (від англ. *instantiation*) – це процес створення екземпляра класу. У контексті потоків у Java це означає створення об’єкта, що реалізує або розширює механізм Thread.

1). Якщо клас розширює Thread:

```
MyThread t = new MyThread();
t.start(); // запуск потоку
```

2). Якщо клас реалізує Runnable:

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
t.start(); // запуск потоку
```

3). Передача одного екземпляра Runnable кільком потокам:

```
MyRunnable r = new MyRunnable();
Thread foo = new Thread(r);
Thread bar = new Thread(r);
Thread bat = new Thread(r);
foo.start();
bar.start();
bat.start();
```

Важливо: завжди викликається *start()*, а не *run()*. Метод *start()* ініціює новий потік і автоматично викликає *run()* у фоновому режимі. Якщо викликати run() напряму, код виконається в основному потоці без створення нового потоку.

6.5. Властивості та стани потоку

Властивості потоку. Кожен потік у Java має низку основних властивостей, які визначають його ідентичність та поведінку:

- ***id*** – унікальний ідентифікатор потоку, який присвоюється автоматично.
- ***name*** – ім'я потоку. Може бути встановлено при створенні або за допомогою методу setName().
- ***priority*** – пріоритет потоку. Визначається числом від Thread.MIN_PRIORITY (1) до Thread.MAX_PRIORITY (10). За замовчуванням використовується Thread.NORM_PRIORITY (5).
- ***daemon*** – статус потоку як потоку-демона. Потоки-демони працюють у фоновому режимі й не перешкоджають завершенню програми.

Властивості потоку не можна змінювати після того, як потік було запущено методом *start()*.

Приклад:

```
Thread t = new Thread(() -> System.out.println(«Потік»));
t.setName(«Робочий потік»);
t.setPriority(Thread.NORM_PRIORITY);
t.setDaemon(false);
System.out.println(«Ім'я: » + t.getName());
System.out.println(«Пріоритет: » + t.getPriority());
System.out.println(«Daemon: » + t.isDaemon());
```

Стани потоку. Потік у Java може перебувати в одному з п'яти основних станів Рис. 6.1. Кожен стан відповідає певному етапу життєвого циклу потоку:

- Новий (***new***). Потік створено, але ще не запущено. Стан триває до виклику *start()*. Потік не вважається «живим».
- Працездатний (***runnable***). Після виклику *start()* потік переходить у стан runnable, тобто він готовий до виконання, але фактично ще не працює. Очікує, коли планувальник вибере його.
- Працюючий (***running***). Потік обрано планувальником, він виконує код методу *run()*. Лише один потік може бути в цьому стані одночасно на одному процесорі.
- Очікуваний / Блокований / Сплячий. Потік тимчасово не може виконуватися:
  - ***sleep()*** – затримка виконання.
  - ***wait()*** – очікування певної умови.
  - ***blocked*** – очікування звільнення ресурсу.

Усі ці стани означають, що потік не готовий до виконання, але не завершений.

- Мертвий (***dead***). Потік завершив виконання або зупинений. Після цього не може бути перезапущений.

Отримання стану:

```
Thread.State state = t.getState();
System.out.println(«Стан потоку: » + state);
```

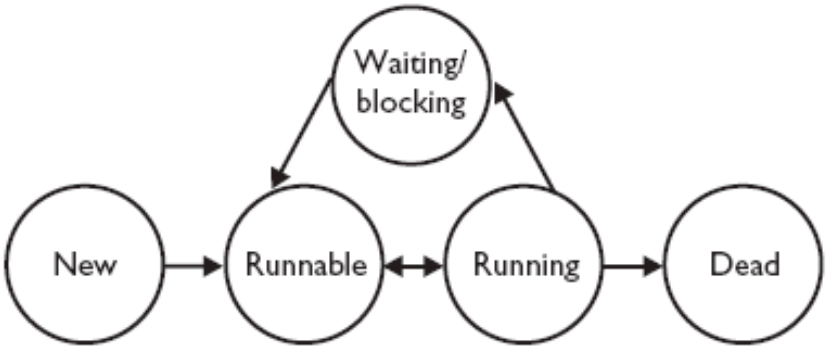

Рисунок. 6.1 – Життєвий цикл потоку

6.6. Взаємодія потоків

У багатопоточному середовищі важливою є можливість керування та координації потоків. Для цього в Java надається набір засобів для створення, зупинки, очікування та синхронізації потоків.

Основні дії з потоками:

1) Створення потоку:

За допомогою **new** Thread**(...)** або через реалізацію Runnable.

2) Запуск потоку:

```
t.start();
```

3) Очікування завершення потоку:

```
t.join(); // блокує поточний потік до завершення t
```

4) Переривання потоку:

```
t.interrupt(); // встановлює флаг переривання
```

5) Призупинення виконання:

```
Thread.sleep(1000); // переводить потік у «сплячий» стан
```

6) Передача управління (*yield*):

```
Thread.yield(); // дозволяє іншим потокам отримати процесор
```

➢ Сон потоку. Методи для зупинки потоку на певний час:

```
Thread.sleep(millis)
Thread.sleep(millis, nanos)
TimeUnit.SECONDS.sleep(1); – з використанням TimeUnit
```

Всі методи зупинки викидають виключення InterruptedException, якщо потік було перервано під час сну.

➢ Планування потоків (Thread Scheduling). У Java є планувальник потоків, який обирає, який потік буде запущено.

Основні чинники:

- Пріоритет потоку: значення від Thread.MIN_PRIORITY (1) до Thread.MAX_PRIORITY (10).
- Daemon-статус: фонові потоки, що не заважають завершенню JVM.

JVM завершує виконання, коли всі користувацькі потоки завершено, навіть якщо фонові (daemon) ще працюють.

Види планувальників:

- З пріоритетами: потокам з вищим пріоритетом надається перевага.
- Без пріоритетів: потік працює доти, поки сам не зупиниться.

➢ Зупинка, призупинення і відновлення:

- *stop()* – зупиняє потік назавжди, не рекомендується до використання.
- *sleep()* – тимчасова зупинка.
- *suspend() / resume()* – застарілі, викликають ризики блокування.

Краще використовувати логіку очікування з умовами (наприклад, *wait/notify* або блокувальники з пакету *java.util.concurrent*).

Очікування завершення потоку. Методи *join()* дозволяють дочекатися завершення потоку:

```
t.join();                  // чекає, поки потік завершиться
t.join(1000);                 // чекає до 1000 мс
t.join(1000, 500);               // чекає до 1000 мс і 500 нс
```

Всі методи можуть викидати InterruptedException.

➢ Переривання потоку.

Для м'якої зупинки потоку використовується механізм флагів:
- *interrupt()* – подає сигнал переривання.
- *isInterrupted()* – перевіряє стан.
- *interrupted()* – перевіряє та скидає флаг.

Код, що використовує методи очікування (*sleep, join, wait*), повинен обробляти InterruptedException.

6.7. Синхронізація потоків і взаємодія через монітори

Проблема одночасного доступу. Коли кілька потоків одночасно звертаються до спільного ресурсу (файл, черга, база даних), можуть виникати конфлікти та помилки, зокрема пошкодження даних або непередбачена поведінка програми. Це явище називається станом гонитви (*race condition*).

Блокування та синхронізація в Java. Будь-який об'єкт у Java може виступати блокуванням.

Синхронізований метод:

```
public synchronized void save() {
   // критична секція
}
```

Еквівалентне синхронізованому блоку:

```
public void save() {
```

```
    synchronized (this) {
        // критична секція
    }
}
```

Для статичних методів:

```
public static synchronized void log() { ... }
```

Еквівалент: `synchronized (ClassName.class) { ... }`

Взаємне блокування та монітори. У Java **монітор** – це механізм контролю одночасного доступу до об'єкта. Кожен об'єкт має один монітор, і тільки один потік може утримувати його в конкретний момент часу.

- Якщо потік входить у synchronized-блок або метод – він захоплює монітор.
- Інші потоки блокуються до моменту звільнення монітора.
- Метод *wait()* тимчасово звільняє монітор.
- Метод *notify()* чи *notifyAll()* не звільняє монітор, а лише сповіщає потоки.

Методи для взаємодії потоків (Object API)

- *wait()* – потік чекає умови.
- notify() – розбудити один потік, що чекає на моніторі.
- notifyAll() – розбудити всі потоки, що чекають на моніторі.

Ці методи можна викликати лише зсередини synchronized-блоку або методу. Інакше – IllegalMonitorStateException.

Приклад: патерн «**виробник-споживач**»

Ідея: потік-виробник кладе дані у буфер, а потік-споживач ці дані забирає.

Інтерфейс черги:

```
interface Buffer {
    void put(int value) throws InterruptedException;
    int get() throws InterruptedException;
}
```

Реалізація з використанням моніторів:

```
class SharedBuffer implements Buffer {
    private int data;
    private boolean available = false;
```

```
    public synchronized void put(int value) throws InterruptedException {
        while (available) wait();     // чекаємо, поки споживач забере дані
        data = value;
        available = true;
        notify();                     // повідомити споживача
    }

    public synchronized int get() throws InterruptedException {
        while (!available) wait();    // чекаємо, поки з'являться дані
        available = false;
        notify();                     // повідомити виробника
        return data;
    }
}
```

Блокування потоків:

- Потоки, які викликають нестатичні synchronized-методи одного й того ж об'єкта, блокують один одного.
- Потоки, які викликають статичні synchronized-методи, завжди блокуються по монітору Class.
- Статичні і нестатичні методи не блокують один одного.
- Якщо використовується синхронізований блок, тут важливо контролювати, щоб lockObject був унікальним для певного ресурсу.

```
synchronized (lockObject) { ... }
```

6.8. Модель пам'яті Java (Java Memory Model)

Потоки та пам'ять. У Java кожен потік має власний стек викликів, де зберігаються [19]:

- Виклики методів;
- Локальні змінні – невидимі для інших потоків, навіть якщо потоки виконують один і той самий код. Кожен потік має свою копію змінних, зберігаючи їх ізольовано у власному стеку.

Купа та розділення об'єктів. Об'єкти створюються в кучі (*heap*) і доступні всім потокам, які мають на них посилання. У результаті:

- Змінні-члени об'єктів доступні для спільного використання;
- Виникає потреба в синхронізації доступу до цих змінних.

Основи Java Memory Model (JMM). Ключові властивості:

- **Атомарність**. Операції виконуються як єдине неподільне ціле. Усі операції з примітивними типами, крім *long* та *double*, атомарні.

- **Видимість**. Зміни, внесені одним потоком, повинні бути видимими іншим потокам.

Це забезпечується через:

- захоплення/звільнення блокувань;
- запуск нового потоку;
- завершення потоку;
- використання *volatile*.

- **Впорядкованість** (*ordering*). Код може виконуватись у зміненому порядку (через оптимізації), але видається послідовним з погляду одного потоку.

Модифікатор volatile. Змінна **volatile** гарантує:

- видимість змін для всіх потоків;
- використання пам'яті напряму, минаючи кеш потоку.

Проте volatile не гарантує атомарності при складних операціях (наприклад, *count++* – не атомарна).

Серіалізація (Serialization) – процес перетворення об'єкта в послідовність байтів, яку потім можна зберігати або передавати.

Навіщо потрібна:

- Передача об'єктів по мережі;
- Збереження в файл;
- Взаємодія з іншими ОС;
- Підтримка Java RMI.

Як реалізувати:

- Клас реалізує інтерфейс Serializable.
- Створюється об'єкт ObjectOutputStream.
- Викликається метод writeObject().

```
ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(«file.ser»));
oos.writeObject(myObject);
```

Відновлення:

```
ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(«file.ser»));
MyClass obj = (MyClass) ois.readObject();
```

Серіалізуються також всі пов'язані об'єкти – формується граф об'єктів.

Локалізація (Localization, l10n) – пристосування програми до мови, формату дати/часу, валюти тощо. Для цього використовується клас java.util.Locale, що підтримує:

- регіональні алфавіти;
- числові формати;

- дати й валюти.

JVM автоматично застосовує регіональні налаштування ОС, але їх можна перевизначити програмно.

Завдання для самостійного виконання:

1. Створення та запуск потоків. Для кожного з наведених нижче варіантів створіть та запустіть по два потоки. Один потік повинен бути створений шляхом розширення класу Thread, а інший – шляхом реалізації інтерфейсу Runnable. Кожен потік повинен виводити на екран своє ім’я (яке ви задасте при створенні) та лічильник від 1 до 5 з затримкою в 500 мілісекунд між виведеннями.

Варіанти імен потоків:

1) Потік 1: «Лічильник А», Потік 2: «Лічильник В»
2) Потік 1: «Перший», Потік 2: «Другий»
3) Потік 1: «RunnableTask», Потік 2: «ThreadExtension»
4) Потік 1: «Alpha», Потік 2: «Beta»
5) Потік 1: «Producer», Потік 2: «Consumer»
6) Потік 1: «FileLogger», Потік 2: «ConsolePrinter»
7) Потік 1: «WorkerOne», Потік 2: «WorkerTwo»
8) Потік 1: «Task_1», Потік 2: «Task_2»
9) Потік 1: «MainThread», Потік 2: «SecondaryThread»
10) Потік 1: «ProcessA», Потік 2: «ProcessB»

2. Синхронізація потоків. Створіть клас Counter з одним цілочисловим полем count та двома методами: increment() та decrement(). Обидва методи повинні бути синхронізовані. Створіть два потоки, один з яких 10 разів викликає метод increment(), а інший – 10 разів викликає метод decrement(). Після завершення обох потоків виведіть на екран кінцеве значення count.

Варіанти початкового значення count в класі Counter:

1) Початкове значення: 0
2) Початкове значення: 5
3) Початкове значення: -3
4) Початкове значення: 10
5) Початкове значення: -5
6) Початкове значення: 1
7) Початкове значення: -1
8) Початкове значення: 7
9) Початкове значення: -7

10) Початкове значення: 2

3. Взаємодія між потоками (Виробник-Споживач). Реалізуйте сценарій «Виробник-Споживач» з використанням спільного буфера (наприклад, ArrayList обмеженого розміру). Створіть клас Producer, який додаватиме цілі числа до буфера через випадкові проміжки часу, та клас Consumer, який забиратиме ці числа з буфера також через випадкові проміжки часу. Використовуйте методи wait() та notify() для синхронізації роботи виробника та споживача, щоб запобігти спробам додавання до повного буфера або забирання з порожнього.

Варіанти параметрів для буфера та кількості операцій:

1) Розмір буфера: 5, Кількість операцій виробника/споживача: 10
2) Розмір буфера: 3, Кількість операцій виробника/споживача: 7
3) Розмір буфера: 7, Кількість операцій виробника/споживача: 12
4) Розмір буфера: 2, Кількість операцій виробника/споживача: 5
5) Розмір буфера: 6, Кількість операцій виробника/споживача: 9
6) Розмір буфера: 4, Кількість операцій виробника/споживача: 8
7) Розмір буфера: 8, Кількість операцій виробника/споживача: 11
8) Розмір буфера: 1, Кількість операцій виробника/споживача: 6
9) Розмір буфера: 9, Кількість операцій виробника/споживача: 13
10) Розмір буфера: 10, Кількість операцій виробника/споживача: 15

Контрольні запитання

1) Поясніть поняття «потік» у контексті програмування. Чим багатопоточність відрізняється від одночасного запуску кількох копій програми (багатопроцесності)? Які основні переваги використання багатопоточності?
2) Опишіть два основні способи створення потоків у Java: розширення класу Thread та реалізація інтерфейсу Runnable. У чому полягають основні відмінності між цими підходами? Який підхід вважається більш гнучким і чому?
3) Інтерфейс Runnable сам по собі не запускає потік. Поясніть механізм використання інтерфейсу Runnable для створення та запуску потоку. Яку роль у цьому процесі відіграє клас Thread та його метод start()? Чому важливо викликати метод start(), а не run() для запуску нового потоку?
4) Назвіть основні властивості потоку в Java. Які значення можуть приймати властивості пріоритету потоку та демонічного статусу? Чи можна змінити властивості потоку після його запуску?
5) Опишіть п’ять основних станів життєвого циклу потоку в Java. Поясніть, що відбувається з потоком при переході між цими станами. Наведіть приклади

ситуацій, коли потік може перебувати у станах «Очікуваний / Блокований / Сплячий».

6) Опишіть основні дії, які можна виконувати з потоками в Java, включаючи запуск, очікування завершення, переривання та призупинення виконання. Поясніть призначення методів join(), interrupt(), sleep() та yield().

7) Що таке «стан гонитви» (race condition) у багатопоточному середовищі? Поясніть, як механізми блокування та синхронізації в Java допомагають запобігти цій проблемі. Наведіть приклади використання синхронізованих методів та синхронізованих блоків.

8) Поясніть концепцію монітора в Java та його роль у взаємодії між потоками. Опишіть призначення методів wait(), notify() та notifyAll(). Які обмеження існують при виклику цих методів?

9) Розкрийте поняття Java Memory Model (JMM). Чим відрізняється зберігання локальних змінних від змінних-членів об’єкта у контексті багатопоточності? Поясніть значення атомарності, видимості та впорядкованості в JMM. Яку роль відіграє модифікатор volatile?

10) Що таке серіалізація в Java та для чого вона використовується? Опишіть основні кроки для серіалізації об’єкта та його відновлення. Що таке локалізація (l10n) і який клас у Java використовується для її підтримки?

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Gosling J., Joy B., Steele G., Bracha G., Buckley A. The Java Language Specification. Java SE 8 Edition. Redwood City, California : Oracle, 2015. 788 p.
2. Олецький О. В. Перші кроки в Java : навч. посіб. Київ, 2017. 144 с.
3. Bloch J. Effective Java. 3rd ed. Addison-Wesley, 2017. 412 p.
4. Sharan K. Beginning Java 9 Fundamentals: Arrays, Objects, Modules, JShell, and Regular Expressions. Montgomery, Alabama : Apress, 2017. 1023 p.
5. Horstmann C. S. Core Java. Vol. I. Fundamentals. 11th ed. Prentice Hall, 2018. 889 p.
6. Grinev S. Mastering JavaFX 10. Build Advanced and Visually Stunning Java Applications. Packt Publishing, 2018. 268 p.
7. Modern Java in Action: Lambdas, Streams, Functional and Reactive Programming. 2nd ed. Manning, 2018. 592 p.
8. Методичні вказівки до лабораторної роботи «Основи об'єктно-орієнтованого програмування мовою Java» з курсу «Програмування та підтримка веб-застосувань» : для студ. спец. 122 «Комп'ютерні науки» / уклад. В. О. Колбасін, Г. Ю. Сидоренко ; НТУ «ХПІ». Харків : НТУ «ХПІ», 2018. 31 с.
9. Cuartielles D. The Java Workshop. Packt Publishing, 2019. 606 p.
10. Кадомський К. К., Ніколюк П. К. Java. Теорія і практика : навч. посіб. Вінниця : ДонНУ, 2019. 197 с.
11. Васильєв О. Програмування мовою Java. Тернопіль : Навчальна книга-Богдан, 2020. 696 с.
12. Vasyliev O. Java. How to Create a Program. Mini-Book 01. Independently published, 2021. 35 p.
13. Flask R. Java for Beginners: A Crash Course to Learn Java Programming in 1 Week. L-Università ta' Malta, 2022. 179 p.
14. Gallardo R. The Java Tutorial: A Short Course on the Basics. 6th ed. Addison-Wesley Professional, 2022. 828 p.
15. Об'єктно-орієнтоване програмування мовою Java : метод. вказівки до лабораторних робіт з курсу «Об'єктно-орієнтоване програмування» : для студ. спец. 122 «Комп'ютерні науки», 126 «Інформаційні системи та технології» / уклад. О. М. Нікуліна, Л. В. Іванов, Н. В. Коцюба ; НТУ «ХПІ». Харків : Друкарня Мадрид, 2022. 64 с.
16. Мартін Р. Чистий кодер. Харків : Фабула, 2023. 256 с.
17. Іванов Л. В., Пашнєв А. А. Мова і платформа Java в інформаційних технологіях : навч. посіб. Харків : НТУ «ХПІ», 2024. 167 с. URL: https://doi.org/ (дата звернення: 25.03.2025).
18. Java Programming. Wikibooks. URL: https://en.wikibooks.org/wiki/Java_Programming/Print_version (дата звернення: 10.01.2025).
19. Eck D. Javanotes: Java Programming Tutorial. Hobart and William Smith

Colleges. URL: https://math.hws.edu/javanotes/ (дата звернення: 10.01.2025).
20. Google Java Style Guide.
URL: https://google.github.io/styleguide/javaguide.html (дата звернення: 25.03.2025).

Навчальне видання

ШАПОВАЛОВА Марія Ігорівна
ВОДКА Олексій Олександрович

МОВА ПРОГРАМУВАННЯ JAVA
Частина 1

Навчальний посібник
для студентів першого (бакалаврського)
рівня підготовки спеціальностей
F1(113) «Прикладна математика» та
F3(122) «Комп’ютерні науки»

Відповідальний за випуск доц. Місюра С. Ю.

Роботу до видання рекомендував доц. Федоров В. О.

В авторській редакції

План 2025 р., поз. 63

Гарнітура Times New Roman. Ум. друк. арк. 5,2

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2.

Електронне видання