

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Київський національний університет будівництва і архітектури

М.І. Цюцюра

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Конспект лекцій
для студентів спеціальностей
122 «Комп'ютерні науки»
та 126 «Інформаційні системи та технології»

Київ 2018

ЗМІСТ

1	ВСТУП.....	10
1.1	Короткі відомості про ООП.....	10
1.2	Короткі відомості про мову Java.....	11
1.3	Питання для самоперевірки.....	12
2	JAVA-ПРОГРАМУВАННЯ НА РІВНІ БАЗОВИХ ТИПІВ.....	13
2.1	Ідентифікатори.....	13
2.2	Коментарі.....	13
2.3	Типи.....	13
2.3.1	Цілочисельні типи.....	13
2.3.2	Дійсні типи.....	14
2.3.3	Логічний тип.....	14
2.3.4	Тип char.....	14
2.4	Операції.....	14
2.4.1	Арифметичні операції.....	14
2.4.2	Операції порівняння та логічні операції.....	15
2.4.3	Логічні операції.....	15
2.4.4	Порозрядні логічні операції.....	15
2.4.5	Операція присвоєння.....	15
2.4.6	Комбіновані присвоєння.....	16
2.4.7	Унарні присвоєння.....	16
2.4.8	Умовна операція (тернарний оператор).....	17
2.4.9	Операція кома.....	17
2.4.10	Порядок виконання операцій.....	17
2.4.11	Приведення типів.....	17
2.5	Управління потоком операцій.....	18
2.5.1	Оператор if...else.....	18
2.5.2	Оператор switch.....	18
2.5.3	Цикли while и do ... while.....	18
2.5.4	Оператори break, continue і мітка.....	19
2.5.5	Цикл for.....	19
2.6	Масиви.....	20
2.6.1	Використання циклу for-each для масивів.....	21
2.6.2	Багатовимірні масиви.....	21
2.7	Питання для самоперевірки.....	21
3	ЗНАЙОМСТВО З БІБЛІОТЕКОЮ JRE.....	22
3.1	Поняття об'єкту і класу.....	22
3.1.1	Об'єкт.....	22
3.1.2	Клас.....	23
3.1.3	Пакет.....	23
3.1.4	Оголошення імпорту.....	23
3.1.5	Статичні поля і методи.....	23
3.1.6	Створення об'єктів.....	24
3.1.7	Операції над посилальними типами.....	25
3.1.8	Структура програми на Java.....	26
3.2	Короткі відомості про runtime бібліотеку Java.....	26
3.2.1	Структура бібліотеки.....	26
3.2.2	Клас java.lang.Object.....	27

3.2.3	Клас java.lang.System.....	28
3.2.3.1	Виведення інформації на консоль	28
3.2.3.2	Виведення інформації про помилки.....	28
3.2.3.3	Введення інформації з консолі	28
3.2.3.4	Доступ до системного часу	29
3.2.4	Класи для роботи з рядками символів.....	30
3.2.4.1	Клас java.lang.String	30
3.2.4.2	Класи java.lang.StringBuilder та java.lang.StringBuffer	31
3.2.5	Класи-обгортки.....	32
3.2.5.1	Класи-обгортки для простих типів.....	32
3.2.5.2	Класи типу java.util.OptionalDouble	33
3.2.6	Форматування даних	34
3.2.6.1	Класи java.text.NumberFormat та java.text.DecimalFormat	34
3.2.7	Класи для роботи з математичними функціями та випадковими числами	35
3.2.7.1	Клас Math	35
3.2.7.2	Клас java.util.Random.....	35
3.2.8	Класи для роботи з часом та датою	36
3.2.9	Клас для обробки масивів.....	37
3.2.9.1	Клас java.util.Arrays.....	37
3.2.9.2	Лямбда-функції	37
3.2.9.3	Тип java.util.stream.Stream.....	38
3.3	Огляд деяких корисних можливостей Eclipse.....	40
3.3.1	Автогенерація коду	40
3.3.2	Автоматичне виправлення помилок.....	40
3.3.3	Меню Source.....	40
3.3.4	Меню Refactor	41
3.4	Питання для самоперевірки	41
4	Об'єктна модель JAVA	42
4.1	Фундаментальні принципи «чистого» ООП	42
4.1.1	Інкапсуляція	42
4.1.2	Поліморфізм.....	43
4.1.3	Спадкування.....	44
4.2	Робота з винятками	45
4.3	Приклад створення об'єктно орієнтованого застосування	48
4.4	Питання для самоперевірки	52
5	Потоки введення-виведення і серіалізація об'єктів	53
5.1	Загальні відомості про введення/виведення даних.....	53
5.2	Візуальні компоненти для роботи з файлами та каталогами	53
5.2.1	Клас JFileChooser.....	53
5.2.2	Вибір файлів за допомогою класу FileDialog	55
5.3	Засоби введення/виведення.....	56
5.3.1	Обмін даними с консоллю	56
5.3.2	Обмін даними з файлами	56
5.3.2.1	Читання рядків із текстового файлу.....	56
5.3.2.2	Читання даних примітивного типу із текстового файлу	57
5.3.2.3	Виведення до текстового файлу	57
5.4	Серіалізація.....	58

5.4.1	Інтерфейс Serializable	59
5.4.2	Ключове слово transient	59
5.4.3	Запис об'єктів у файл і відновлення об'єктів із файлу	59
5.5	Питання для самоперевірки	60
6	ВИКОРИСТАННЯ ІТЕРАТОРІВ	61
6.1	Поняття ітератора	61
6.2	Інтерфейси ітераторів у Java	61
6.2.1	Інтерфейс Enumeration	61
6.2.2	Інтерфейс Iterator	62
6.2.3	Інтерфейс ListIterator	62
6.2.4	Інтерфейс Iterable	63
6.2.5	Розширення інтерфейсу Iterable	63
6.3	Питання для самоперевірки	64
7	ПЕРЕДАЧА СПЕЦИФІЧНИХ ОПЕРАЦІЙ КОРИСТУВАЧА ДО УЗАГАЛЬНЕНИХ МЕТОДІВ....	65
7.1	Узагальнені методи і специфічні операції користувача	65
7.1.1	Шляхи створення класів, що реалізують інтерфейс	66
7.1.2	Шляхи створення об'єкта, що використовується для передачі операції користувача.....	66
7.1.3	Функціональні інтерфейси та лямбда функції.....	67
7.1.4	Ще один приклад з використанням функціональних інтерфейсів	68
7.2	Питання для самоперевірки	69
8	КОЛЕКЦІЇ JAVA	70
8.1	Загальні відомості про колекції	70
8.2	Узагальнення в колекціях	70
8.3	Інтерфейс Collection.....	71
8.3.1	Загальна характеристика інтерфейсу Collection.....	71
8.3.2	Методи інтерфейсу Collection	72
8.3.3	Використання ітератора Spliterator для паралельної обробки колекції.....	75
8.4	Інтерфейс List	76
8.4.1	Методи інтерфейсу List.....	76
8.4.2	Класи, що реалізують інтерфейс List	77
8.4.2.1	Класи Vector та ArrayList	77
8.4.2.2	Клас Stack.....	77
8.4.2.3	Клас LinkedList.....	78
8.4.3	Інтерфейс ListIterator	78
8.5	Інтерфейси колекцій для черг.....	78
8.5.1	Інтерфейс Queue	79
8.5.2	Інтерфейс Deque	79
8.5.3	Класи колекцій для черг	80
8.5.3.1	Клас LinkedList.....	80
8.5.3.2	Клас ArrayDeque.....	81
8.5.3.3	Клас PriorityQueue.....	82
8.6	Колекції, що реалізують інтерфейс Set.....	82
8.6.1	Інтерфейс Set.....	83
8.6.2	Методи equals() и hashCode()	83
8.6.3	Деякі класи, що реалізують інтерфейс Set.....	84
8.6.3.1	Клас HashSet.....	84
8.6.3.2	Клас LinkedHashSet.....	85

8.6.4	Інтерфейси для впорядкованих унікальних множин	85
8.6.4.1	Інтерфейс SortedSet.....	85
8.6.4.2	Інтерфейс NavigableSet.....	85
8.6.5	Класи для впорядкованих унікальних множин	86
8.6.5.1	Клас TreeSet	86

1 ВСТУП

1.1 КОРОТКІ ВІДОМОСТІ ПРО ООП

Об'єктно-орієнтоване програмування з'явилося вже багато років тому, але й сьогодні займає почесне місце серед концепцій написання програм. Першою мовою програмування, в якому були запропоновані принципи об'єктної орієнтованості, була Сімула. Фактично, це була навіть не мова, а надбудова над мовою програмування Алгол. Незважаючи на це, на момент своєї появи (в 1967 році), Сімула використовувала справді революційні ідеї: об'єкти, класи, віртуальні методи та ін. Однак все це не було сприйняте сучасниками як щось грандіозне і Сімула широко використовувалася тільки програмістами, що займалися імітаційним моделюванням (simulation).

Пізніше концепції ООП були розвинені Аланом Кейем і Деном Інгаллсом у мові Smalltalk. Саме ця мова стала першою об'єктно-орієнтованою мовою програмування, що набула широкого популярності, та й то в основному завдяки тому, що вперше дозволяла створювати графічний інтерфейс користувача. Ця мова також є прикладом (можливо єдиним) чисто об'єктно орієнтованої мови.

Головна особливість мов програмування, що є чисто об'єктно орієнтованими, полягає у тому, що єдиною виразною субстанцією мови є клас. Клас визначає перелік властивостей об'єкту, та методів, що описують «поведінку» об'єкта. За допомогою класів створюються об'єкти, які спілкуються між собою за допомогою повідомлень. Кожне повідомлення активізує відповідний метод і таким чином реалізується потрібний алгоритм.

Слід наголосити, що використання повідомлень - єдиний спосіб обміну інформацією між об'єктами і це, мабуть, найпотужніша концепція об'єктно-орієнтованого програмування. Ця концепція не передбачає використання операцій, тобто вираз $a+b$ розглядається як відправлення об'єкту «а» повідомлення «+» з параметром «в». У класі, якому належить об'єкт «а» має бути метод «+» з одним параметром. Саме так виглядає і інтерпретується запис цього виразу у Smalltalk. У сучасних мовах програмування символ «+» та йому подібні не використовують для назв методів, тому подібний вираз може виглядати таким чином: `a.add(b)`.

В даний час об'єктно-орієнтоване програмування є абсолютним лідером в області прикладного програмування (мови Java, C#, C++ та ін.) і є найбільш ефективним інструментом для створення складних корпоративних програмних систем. Але ці мови не є «чистими» з точки зору концепцій ООП.

Головна відмінність цих мов від «чистого» ООП полягає у тому, що у цих мовах використовуються так звані «базові» або «примітивні» типи даних, які не є класами, а відповідні дані не є об'єктами. Це типи для цілих та дійсних чисел, символів та логічних значень. Для даних базових типів не використовуються повідомлення, натомість використовуються операції. Окрім того, для організації циклів, розгалужень та ін. використовуються звичайні оператори на кшталт `if`, `for`, `switch`, що також не вписується у концепцію ООП. Такими нетиповими для ООП конструкціями найбільш захаращена мова C++.

1.2 КОРОТКІ ВІДОМОСТІ ПРО МОВУ JAVA

Java – це середовище програмування, що складається з мови, інструментів розробки і runtime system (системи часу виконання). Runtime system, або Java машина, дозволяє виконувати Java-програми на будь-якій платформі, якщо на цю платформу завантажено відповідну бібліотеку програм.

Роботи, внаслідок яких з'явилася мова Java розпочалися навесні 1991 року. Перед проєктантами стояло завдання розробити операційне середовище для побутових приладів з сенсорними екранами. Середовище повинно було бути розраховано на звичайних людей, а не комп'ютерних фанатів і повинно було працювати на різних, але заздалегідь підготовлених для неї пристроях.

Розробник мови Java James Gosling намагався реалізувати середовище, розширюючи мову C++, але врешті решт усвідомив, що C++ не задовольняє його за низкою причин, зокрема:

- C++ не зобов'язує до об'єктно-орієнтованого програмування;
- в C++ реалізовано пряме управління пам'яттю. Програміст повинен виділяти пам'ять і звільняти її, що є джерелом важких помилок, крім того, в різних ОС це робиться по-різному.

Тому було вирішено створити нову мову з такими особливостями:

- не буде покажчиків та адресної арифметики, що обіцяло істотно підвищити живучість нового середовища;
- не буде переваження операторів, множинного спадкування і автоматичного узгодження типів даних, що є джерелом проблем і помилок в програмах;
- єдиними виразними одиницями в мові будуть об'єкти;
- мова буде інтерпретуючою, бо, наприклад, файл створений для виконання у середовищі Windows не має сенсу для Unix;
- синтаксис буде якомога ближче до C++, що зменшить час навчання досвідчених програмістів на C++.

До 1993 року мова була створена і названа Oak (дуб), але комерційного застосування їй знайти не змогли. Тому у 1994 виникла ідея адаптувати мову для Internet і був створений Web-браузер з вбудованим інтерпретатором мови, який назвали HotJava, а саму мову назвали Java.

А з 1995 фірма Sun почала безкоштовно поширювати Hot Java і Java в Internet. У тому ж році Java була включена в NetScape Navigator, а невдовзі і в Microsoft Explorer.

Сьогодні Java це проста, об'єктно-орієнтована, високопродуктивна, інтерпретуюча, безпечна, розподілена, живуча, архітектурно-нейтральна, багатопотокова і динамічна мова.

Для роботи Java-програм необхідно, щоб на комп'ютері була встановлена бібліотека відповідних програм, що поставляється у вигляді папок jre (java runtime environment) або jdk (java development kit). Папка jre містить мінімально необхідний набір файлів. Набагато більше інформації, зокрема вихідні тексти багатьох класів, містить папка jdk.

Основною якістю Java-платформи, є те, що вона вбудовується на самому верхньому рівні в інших платформах, рисунок 1.1.

Програма Java компілюється у байт-код, і передається до віртуальної Java машини, яка є складовою частиною Java платформи.

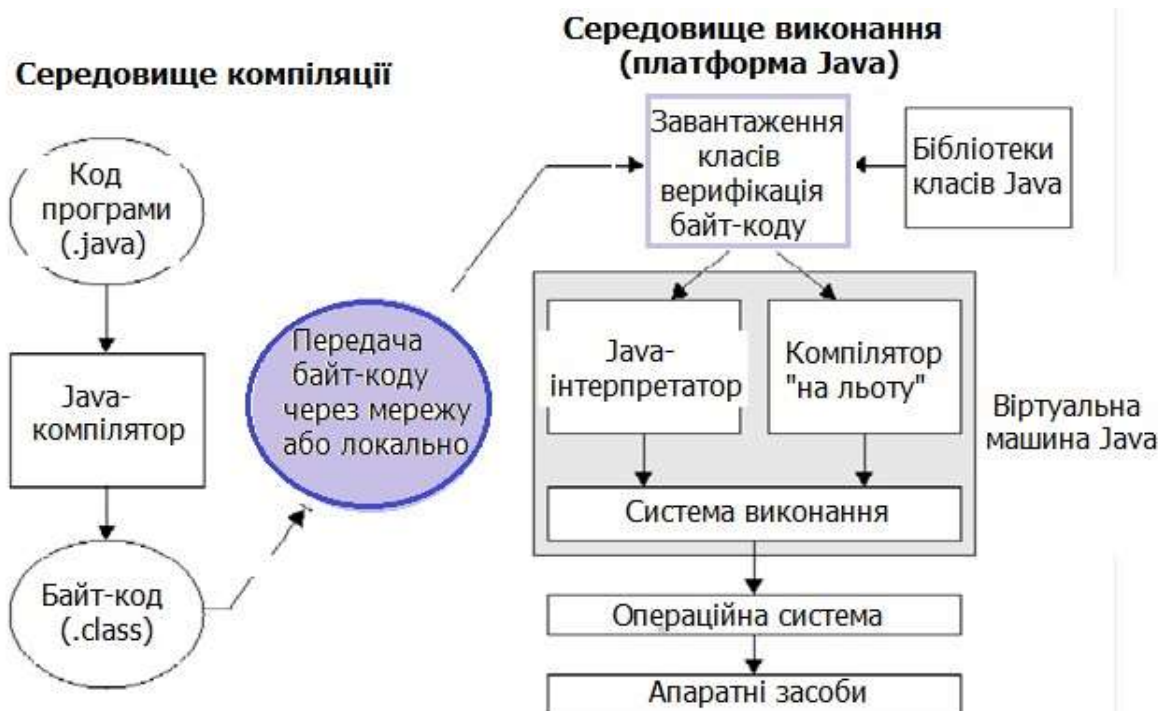


Рисунок 1.1 – Схема виконання Java програми

1.3 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Назвіть деякі з існуючих мов для об'єктно орієнтованого програмування.
2. Яка мова вперше надала засоби для створення графічного інтерфейсу користувача?
3. Чим суттєво Java відрізняється від C++?
4. Яка різниця між jre і jdk?
5. Що необхідно для того, щоб java-програму можна було виконати на деякому цифровому пристрої?
6. Які завдання вирішує компілятор java?

2 JAVA-ПРОГРАМУВАННЯ НА РІВНІ БАЗОВИХ ТИПІВ

2.1 ІДЕНТИФІКАТОРИ

Ідентифікатори починаються з `_` `$` або букви, далі можна 0..9. Букви в широкому сенсі, завдяки тому, що для кодування використовується Unicode. Рекомендується назви змінних починати з маленьких літер і використовувати «верблюжу» нотацію.

Маленькі і великі літери розрізняються.

Тип змінної визначається так само як і у мові C.

Нижче наведено приклад, який можна протестувати на java-сторінці середовища Eclipse (файл з розширенням `.jprage`). Код буде відкомпільовано і виконано, результатом буде ціле число 6.

```
int число=3;  
return число+число;
```

2.2 КОМЕНТАРІ

Звичайні коментарі виглядають так само, як і у мові C.

```
/*Текст коментаря */
```

```
// Текст коментаря
```

Але ще є doc comment:

```
/**
```

```
* Текст коментаря, що є основою для формування Java doc,
```

```
* який використовується у якості “Help”.
```

```
* Таким чином у Java “Help” має написати сам програміст.
```

```
*/
```

2.3 ТИПИ

У Java використовують прості типи, які ще називають базовими або примітивним, і складні типи, які частіше називають посилальними.

Прості – це цілі, дійсні, символічні і логічні.

Складні – це класи та інтерфейси. (У цьому розділі ми їх не розглядаємо).

2.3.1 Цілочисельні типи

`Byte`, `short`, `int`, `long` – усі знакові. Стан переповнення не фіксується. Цілочисельні константи за замовчування відносяться до типу `int`. Щоб визначити константу типу `long` потрібно до константи додати літеру `L`, наприклад, `6L`.

Допустимі шістнадцяткові, вісімкові та двійкові константи.

Ознакою шістнадцяткової константи є префікс `0x`. Для цих констант дозволяється використовувати символ підкреслювання для розділення константи на байти, наприклад, `0x2a_CD_3f`.

Вісімкові константи починаються з цифри 0, наприклад 012 дорівнює десять.

Двійкові константи мають префікс 0b. Для двійкових констант дозволяється використовувати символ підкреслювання для розділення константи на напівбайти, наприклад, 0b0010_0011_1110.

2.3.2 Дійсні типи

Так само як і у мові C в Java є два дійсні типи float та double.

Дійсні константи за замовчування відносяться до типу double. Щоб визначити константу типу float потрібно до константи додати літеру F, наприклад, 0.5F. Дозволяється використовувати «науковий» формат запису констант, наприклад 0.5e-3.

2.3.3 Логічний тип

Назва типу boolean. Константи true, false.

На відміну від C та C++ ніякого зв'язку з цілочисловими типами не має.

2.3.4 Тип char

Це двобайтне ціле без знаку, що має значення від 0 до 65535.

Для кодування символів у Java використовується Unicode.

Для констант використовуються одинарні лапки, наприклад, 'A'.

2.4 ОПЕРАЦІЇ

Операції, що використовуються у Java для простих типів майже такі самі, як і у мові C.

2.4.1 Арифметичні операції

Перелік арифметичних операцій наведено у таблиці 2.1. Усі наведені операції виконуються над даними усіх числових типів, наприклад, результатом обчислення виразу $2.625 \% 0.5$ буде 0.125.

Таблиця 4.1- Арифметичні операції C++

- унарний	Зміна знаку операнду
*	Множення
/	Ділення
%	Остача ділення
+	Додавання
-	Віднімання

Слід також мати на увазі, що операція ділення для цілих чисел повертає також ціле число. Для того, щоб результат ділення був дійсним числом треба, щоб хоча б один операнд був також дійсним числом. Наприклад, $12 / 5$ буде 2, але $12.0 / 5$ буде 2.4.

2.4.2 Операції порівняння та логічні операції

Операції порівняння наведені в таблиці 2.2. Слід зазначити, що на відміну від C, компілятор Java не пропускає помилки, коли для порівняння використовується операція присвоєння.

Таблиця 4.2 – Операції порівняння

Операція	Опис
==	Дорівнює
>	Більше
<	Менше
>=	Більше або дорівнює
<=	Менше або дорівнює
!=	Не дорівнює

2.4.3 Логічні операції

Логічні операції наведено в таблиці 2.3

Таблиця 2.3 – Логічні операції

Операція	Опис
	Або
&&	і
!	Ні
(a b) && !(a&& b)	Виключно або (xor)
b>=4 && b<5	Дужки не потрібні

2.4.4 Порозрядні логічні операції

Порозрядні логічні операції наведено в таблиці 2.4

Таблиця 2.4 – Порозрядні логічні операції

Операція	Опис
	Або
&	і
!	Ні
^	Виключно або (xor)
<<, >>	Зсуви ліворуч та праворуч
>>>	Зсув праворуч разом із знаковим бітом

2.4.5 Операція присвоєння

Операція присвоєння є однією із основних операцій у будь якій мові програмування. У мові Java вона виглядає так само, як і у мові C:

```
<змінна> = <вираз>;
```

де: <змінна> - ім'я змінної, значення якої змінюється в результаті

виконання інструкції присвоювання;

= знак операції присвоювання.

<вираз> – вираз, значення якого присвоюється змінній, ім'я якої вказане ліворуч від символу присвоювання.

Присвоювання виконується наступним чином:

– спочатку обчислюється значення виразу, який знаходиться праворуч від знаку операції присвоювання;

– потім отримане значення записується у змінну, ім'я якої стоїть ліворуч від символу присвоювання.

Операція присвоювання вважається вірною, якщо тип значення, що повертає вираз, відповідає або може бути приведений до типу змінної, яка отримує це значення. Наприклад, змінній типу `double` можна присвоїти значення виразу, тип якого `float` або `int`.

Особливість операції присвоювання полягає у тому, що ця операція **повертає результат**, що дорівнює значенню виразу у правій частині. Тому цю операцію можна використовувати у виразах. Слід тільки мати на увазі, що ця операція має низький пріоритет. Тому присвоєння у виразах слід брати в дужки. Наприклад, вираз `a + (b = c + d)` є допустимим.

2.4.6 Комбіновані присвоєння

Окрім простого оператора присвоєння у мові Java використовуються ще і так звані комбіновані оператори присвоєння.

Ці оператори використовують у тих випадках, коли ліворуч і праворуч від знаку операції присвоєння знаходиться той самий операнд.

Наприклад, замість присвоєння `number = number + d` можна записати `number += d`.

Таке присвоєння можна використовувати у комбінації з будь якою арифметичною операцією та багатьма іншими, наприклад, `*=`, `/=`, `%=`, тощо.

Комбіновані присвоєння скорочують код.

2.4.7 Унарні присвоєння

Ці присвоєння є окремим випадком комбінованого присвоєння. Вони використовуються тоді, коли до змінної треба додати одиницю або відняти її. Відповідно до цього маємо операції інкременту та декременту.

Запис операції унарного присвоєння ще простіший ніж комбінованого. Для того щоб, наприклад, збільшити змінну `number` на одиницю, можна написати `number++`, або `++ number`. Так само можна і зменшувати значення на одиницю: `number--`, або `--number`.

Операції `++` та `--` називають префіксними, якщо знаки цих операцій записуються перед змінною. Якщо ж знаки операції записуються після змінної то операції називають постфіксними.

Різниця між префіксними та постфіксними операціями проявляється у тих випадках, коли ці операції використовуються у виразах разом з іншими операціями. Справа у тому, що префіксні операції мають найвищий пріоритет, а постфіксні – найнижчий. Хай, наприклад, змінна `x` має значення 5. Тоді

значення виразу (3+ ++x) буде дорівнювати 9, а значення виразу (3+x++) буде дорівнювати 8, хоча «x» у обох випадках отримає значення 6.

2.4.8 Умовна операція (теренарний оператор)

Назва походить від слова три. До складу оператора входить три вирази та два символи (?:).

Синтаксис такий:

Вираз_1 ? Вираз_2 : Вираз_3;

Наприклад:

```
stip = ball<4.0 ? 0 : ball<5 ? 600 : 800;
```

2.4.9 Операція кома

У мові Java у деяких випадках дозволяється використовувати операцію кома, що об'єднує декілька визначень, або виразів. Нижче наведено приклад використання цієї операції у циклі з двома параметрами.

```
for (int x=1, y=10; x < y; x++, y--)
```

2.4.10 Порядок виконання операцій

Порядок виконання операцій визначається їх пріоритетами. У таблиці 4.1 наведено пріоритети операцій Java. Чим менший номер пріоритету, тим вищий пріоритет. Деякі операції, можливо, поки що незнайомі, та в подальшому вони будуть розглянуті.

Таблиця 2.5 – Пріоритети операцій Java

Операції	Пріоритет
Префіксні ++ -- () [] .	1
! ~ +a -a (type)	2
* / % (арифметичні)	3
+ - (арифметичні)	4
<< >> >>> (зсуви ліворуч і праворуч)	5
< > <= >= (порівняння)	6
== != (порівняння)	7
& ^ (порозрядні логічні операції)	8,9,10
&& (логічні операції)	11,12
? : (теренарна операція)	13
= += -= *= /= %= &= ^= = >>= (присвоєння)	14
Постфіксні ++ -- , (операція кома)	15

2.4.11 Приведення типів

Результати цілочислових виразів для типів short byte приводяться до int автоматично, без спотворення результату. Також автоматично приводяться результати цілочисельних операцій до дійсного типу.

Але для переходу від `int` до `long`, або від `double` до `float` потрібне явне приведення типів. Явного приведення потребує також перетворення дійсних типів у цілочисельні.

Для приведення до типу потрібно перед виразом у дужках вказати потрібний тип результату, наприклад, `int g = (int)(25 / 4f);`

2.5 УПРАВЛІННЯ ПОТОКОМ ОПЕРАЦІЙ

2.5.1 Оператор `if...else`

Оператор записується і працює так само, як і у мові C, але результат обчислення виразу умови має бути тільки бульовим.

Нижче наведено приклад використання цього оператора.

```
//Обчислення прибуткового податку наприкінці 90-х років
double p, n, z = 2500;
if (z<=150){ n=0.1*z;}
else if (z<=500) {n=0.2*z;}
else if (z<=1000){ n = 0.3*z;}
else if (z<=2000) {n= 0.4*z;}
else n=0.5*z;
return n;
```

2.5.2 Оператор `switch`

Записується і працює так само, як і у мові C. Але вибір типів ширший. У якості параметру перемикача можна використовувати дані типу `String`.

Нижче наведено приклад використання цього оператора.

```
String c="два"; int i;
switch (c) {
    case "один": i=1; break;
    case "два": i=2; break;
    default : i=0;
}
return i;
```

2.5.3 Цикли `while` u `do ... while`

Ці оператори працюють і записуються так само, як і у мові C, але результат обчислення виразу умови має бути тільки бульовим.

Нижче наведено приклад використання оператора `while`.

```
//Підрахунок суми цифр цілого числа
int x=12345;
int sum=0;
while(x!=0){
    sum += x % 10;
```

```
    x /= 10;
}
return sum;
```

2.5.4 Оператори *break*, *continue* і мітка

Використовуються для виходу з будь якого блоку.

break призводить до виходу з циклу. *continue* завершує тільки поточний етап циклу.

break і *continue* можуть використовуватися з міткою. Це дозволяє перервати виконання не тільки поточного, але й зовнішніх блоків.

goto у Java відсутній.

Ось приклад використання цих операторів:

```
//Підрахунок суми парних цифр цілого числа
int x=123456;
int sum=0, z;
label: while(true){
z=x%10;
if((x/=10)==0)break label;
if( z%2!=0) continue label;
sum+=z;
}
return sum;
```

2.5.5 Цикл *for*

Цей цикл реалізується так само як і у мові C.

Нижче наведено приклади використання цього оператора.

```
// Обчислення факторіалу
int n=5;
long fact=1;
for(int i=1; i<=n; i++){
    fact*=i;
};
return fact;
```

```
//Інша реалізація циклу for для обчислення факторіалу
int n=5, i=1;
long fact=1;
for( ; i<=n; fact*=i, i++);
return fact;
```


2.6 МАСИВИ

На відміну від C у Java існує поняття типу для масиву. Для визначення типу масиву просто вказується ім'я типу елементів, за яким йдуть порожні квадратні дужки. Також можна помістити квадратні дужки після ідентифікатора (як у мові C), що має те ж саме значення. Але останнє оголошення не є логічним для Java, бо тип записується перед ім'ям.

В Java масив знає свій розмір і контролює коректність використання індексів.

Для звернення до елемента масиву використовуються квадратні дужки, де вказується значення індексу. Індеси елементів починаються з нуля.

Нижче наведено два варіанти оголошення масивів.

```
int[] a1;  
int a2[];
```

Слід розуміти, що оголошення масиву визначає тільки посилання на масив, значення якого дорівнює null. Тобто оголошення масиву місця у пам'яті для нього не виділяє. І в оголошенні типу не можна вказати розмір масиву.

Щоб виділити пам'ять для масиву необхідно написати вираз ініціалізації. Ініціалізація масивів може бути виконана по різному і в будь-якому місці коду. Нижче наведено приклади ініціалізації:

```
int[] arr;  
arr = new int[10];  
int[] arr1 = new int[20];  
int[] arr3 = { 1, 2, 3, 4, 5 };
```

Елементи масивів базових числових типів за замовчуванням ініціалізуються нулями. Масиви даних логічного типу ініціалізуються значенням false. Масиви посилальних типів за замовчуванням ініціалізуються значенням null.

У Java можна оголосити посилання на деякий масив і присвоїти йому посилання на інший масив. Але у цьому випадку ми будемо мати справу з тим самим масивом, хоча будемо мати два посилання на нього.

Якщо протестувати наступний приклад, то ми побачимо, що зміна початкового елемента у масиві arr2 призводить до зміни такого ж елемента і у масиві arr3. Тобто arr3 той самий масив що й arr2.

```
int[] arr2 = { 1, 2, 3, 4, 5 };  
int[] arr3 = arr2;  
arr2[0] = 1000;  
return arr3;
```

Всі масиви мають внутрішній атрибут length, який містить кількість елементів у масиві. Змінити його не можна, але можна використовувати.

Нижче наведено приклад використання цього атрибуту:

```
int[] arr = { 1, 2, 3, 4, 5 };  
int sum=0;  
for(int i=0; i< arr.length; ++i)
```

```
sum+=arr[i];  
return sum;
```

2.6.1 Використання циклу for-each для масивів

Для масивів в Java можна використовувати також цикл for у стилі for-each, що забезпечує послідовну обробку елементів масиву без використання індексів.

Загальна форма запису циклу for в стилі for-each наведена нижче:

```
for (type змінна: array) statement-block
```

Тут type - тип, var - ім'я змінної, яка буде отримуватиме послідовно один за іншим елементи, що містяться в масиві array. Тип type повинен збігатися (або бути сумісним) з типом елементів, що знаходяться у масиві array.

Ось приклад використання цього оператора:

```
int[] arr = { 1, 2, 3, 4, 5 };  
int sum=0;  
for(long x: arr)  
sum+=x;  
return sum;
```

2.6.2 Багатовимірні масиви

Оголошення та обробка багатовимірних масивів мало чим відрізняється від того, що є у мові C. Слід враховувати тільки ті самі особливості, що і для одно вимірних масивів.

Нижче наведено приклад оголошення та роботи з двовимірним масивом:

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

2.7 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Яка різниця між Java та C у написанні ідентифікаторів?
2. Яка різниця між Java та C у використанні даних логічного типу?
3. Чим відрізняються оператори switch у Java та C?
4. Особливості приведення типів у Java.
5. Оголошення масиву і його ініціалізація.
6. В чому полягають особливості циклу for each і як він записується?

3 ЗНАЙОМСТВО З БІБЛІОТЕКОЮ JRE

3.1 ПОНЯТТЯ ОБ'ЄКТУ І КЛАСУ

3.1.1 Об'єкт

В ООП головним елементом програми є об'єкт. Об'єкт (object) - це елемент програми, що може мати якісь властивості, поведінку і йому притаманна унікальність. У багатьох випадках програмні об'єкти є спрощеним абстрактним відображення реального світу. Так, наприклад, реальне авто в інформаційній базі ДАІ виглядає як програмний об'єкт у якого є номер реєстрації та власник. Власник авто в базі ДАІ виглядає теж, як абстракція, яка має ідентифікаційний код та номер паспорта.

У наведеному прикладі номер реєстрації авто та його власник є властивостями об'єкта авто, а ідентифікаційний код та номер паспорта є властивостями об'єкта власник авто. Складові об'єкта, де зберігаються властивості об'єкта в різних мовах називають по різному – змінні, поля, атрибути, члени. В Java прийнято вживати слово поле (field), у C++ слово член (member).

Поведінка програмних об'єктів проявляється в тому, що вони здатні реагувати на повідомлення. Наприклад, можна об'єкту авто надіслати повідомлення: - «Який у вас номер реєстрації?» і у відповідь об'єкт повідомить нам свій номер. В ООП програмні конструкції, які відповідають за реакцію на повідомлення називають методами (правда, в C++ їх називають член-функції). Різниця між функцією та методом дуже велика. Справа у тому, що функція викликається сама по собі, а метод може бути викликаний тільки для якогось об'єкту. Різниця тут принципова. Уявіть собі, викладач каже в аудиторію – «іди до дошки». Дії, які треба виконати, зрозумілі. Але невідомо, хто має йти. А якщо сказати – “Ковтун, йди до дошки”, усе зрозуміло. Перша частина речення тут об'єкт, друга – повідомлення. І об'єкт Ковтун буде виконувати метод «йди до дошки» з урахуванням своїх властивостей.

Звичайно, до функції можна було б передавати об'єкт, для якого викликається метод, як параметр. Але у цьому випадку буде втрачено багато можливостей, про які мова бути йти далі.

Унікальність програмних об'єктів полягає у тому, що кожному з них виділяється ділянка пам'яті для збереження значень властивостей об'єкта та іншої інформації. В Java кожний об'єкт має своє унікальне ім'я і йому відповідає посилання на виділену ділянку пам'яті.

Працюючи з якимось об'єктом в Java, ми фактично працюємо з посиланням на ділянку пам'яті, що виділена під об'єкт. Внаслідок цього, якщо ми виконаємо операцію присвоєння одного об'єкта іншому, то це буде не два однакових об'єкта, а тільки два однакових посилання. Об'єкт буде один. Так само, як це відбувалося із масивами.

3.1.2 Клас

Клас - це опис об'єктів однакової структури, і якщо в програмі такий клас використовується, то опис присутній в єдиному екземплярі. Об'єктів цього класу може не бути зовсім, а може бути створено як завгодно багато. Таким чином, клас можна розглядати як шаблон, за яким створюються об'єкти.

Слід розуміти, що кожний з об'єктів класу має свій, персональний набір властивостей, натомість методи для усіх об'єктів класу однакові, і можна вважати, що вони знаходяться у класі, а об'єкт просто має посилання на свій клас, що допомагає йому знайти потрібний метод.

Класи Java існують у вигляді файлів двох типів. Файли з розширенням .java містять вихідний текст класу на мові Java. Файли з розширенням .class містять відкомпільований байт код класу.

3.1.3 Пакет

Сучасні програмні проекти є досить складними і можуть включати сотні, а то й тисячі класів. Ці класи розробляють різні групи програмістів і тому проблема унікальності імен класів є вкрай важливою. Мова Java пропонує дуже раціональний шлях вирішення цієї проблеми. Близькі за призначенням класи розташовують у одній файлової папці, надаючи їй характерне ім'я. Далі групи папок об'єднують у папку наступного рівня і так далі. Усі ці папки називають пакетами і ім'я класу складається з повного переліку імен пакетів та самого імені класу, наприклад, `java.util.stream.Stream`.

Імена пакетів зазвичай пишуть маленькими буквами, а імена класів починають з великої літери.

Ім'я пакету, до якого належить клас, записується у першому рядку файлу класу після ключового слова `package`.

3.1.4 Оголошення імпорту

Коли у класі використовуються інші класи, то програміст має записувати їх повні імена, з переліком усього ланцюжка пакетів. Така нотація не дуже зручна як для написання, так і до читання тексту класу.

Ця проблема вирішується за допомогою оголошень імпорту, які розміщуються на початку тексту класу після ключового слова `import`. В оголошенні `import` може бути або повне ім'я класу, або ім'я ланцюжка пакетів із зірочкою, що означає, що імпортуються усі класи пакету. Наприклад:

```
import java.util.ArrayList;  
import java.util.*;
```

Оголошення імпорту дозволяють використовувати у програмі короткі імена класів, що дуже зручно і наглядно.

3.1.5 Статичні поля і методи

Бувають випадки, коли значення деяких властивостей однакові для усіх об'єктів класу. Наприклад, ми створюємо клас студентів конкретного навчального закладу. У цьому випадку властивість «назва навчального закладу»

для кожного об'єкта нашого класу буде та сама, тому виділяти пам'ять для збереження цієї інформації у кожного з об'єктів недоцільно, бо це неефективне використання пам'яті і зайвий клопіт, якщо заклад буде перейменовано. Для таких властивостей пам'ять виділяють у класі, і такі властивості називають статичними (static).

Як вже було сказано, методи викликають через об'єкти і результат виконання методу залежить від властивостей об'єкта. Наприклад, результати виконання повідомлень «Розумний, йди до дошки» та «Недбалий, йди до дошки» можуть бути зовсім різними.

Але бувають методи, які ніяк не пов'язані з властивостями об'єктів, тому для роботи з таким методом нема сенсу створювати об'єкт, що є витратною операцією. В Java такі методи дозволено викликати через клас, і називають такі методи статичними (static). Інколи такі методи називають методами класу на відміну від методів об'єктів.

3.1.6 Створення об'єктів

Якщо ми оголошуємо якусь змінну посилального типу і не присвоюємо їй ніякого значення, то об'єкт при цьому не створюється. Створюється тільки посилання на об'єкт, значення якого дорівнює null.

Зазвичай об'єкти створюються з використанням ключового слова new, причому одне слово new породжує суворо один об'єкт (або взагалі, жодного, якщо виникла помилка). Після ключового слова вказується ім'я конструктора, за допомогою якого ми збираємося породити об'єкт. Ім'я конструктора завжди співпадає з іменем класу. Після назви класу у конструкторі мають бути круглі дужки, в яких перераховуються значення параметрів, що передаються конструктору, якщо вони є. Наприклад

```
java.lang.StringBuilder sb = new java.lang.StringBuilder();
java.util.Scanner sc = new java.util.Scanner(java.lang.System.in);
```

Якщо конструктор відпрацював успішно, то оператор new повертає посилання на створений об'єкт. Це посилання можна зберегти в змінній, передати в якості аргументу на якийсь метод, або використовувати в інший спосіб. JVM завжди займається підрахунком збережених посилань на кожен об'єкт. Як тільки виявляється, що посилань більше немає, такий об'єкт призначається для знищення збиральником сміття (garbage collector). Відновити посилання на такий "втрачений" об'єкт неможливо.

Операція створення об'єкта - одна з таких, що вимагає багато ресурсів. Тому слід уникати непотрібних породжень об'єктів.

У деяких випадках створити правильно проініціалізований об'єкт достатньо складно. Для вирішення такої проблеми створюють класи фабрики об'єктів, до яких включено методи створення об'єктів з відповідними характеристиками, наприклад:

```
java.text.NumberFormat nf
=java.text.NumberFormat.getCurrencyInstance(java.util.Locale.CHINA);
```

У цьому прикладі об'єкт `nf` типу `NumberFormat` створюється за допомогою статичного методу `getCurrencyInstance` класу `NumberFormat`. Це не означає, що для створення об'єкту не використовується конструктор, просто його загорнуто у метод `getCurrencyInstance`.

Але існують способи створення об'єктів без виклику конструктора. Один із таких способів – це використання методу `clone()`, який створює копію ділянки пам'яті, яку займає об'єкт, що клонують. Інший спосіб це використання технології `reflection`, яка дозволяє створити об'єкт без ключового слова `new`, використовуючи лише ім'я класу в текстовому вигляді.

3.1.7 Операції над посилальними типами

Хоча в ООП поняття операції відсутнє, все ж в Java є деякі операції над посиланнями:

- операція крапка - це звернення до полів і методів об'єкта;
- операція присвоювання;
- операція `instanceof` (повертає логічне значення, що визначає, чи є об'єкт екземпляром вказаного класу)
- операції порівняння `==` та `!=` (повертають бульове значення);
- оператор приведення типів (дужки);
- тернарний оператор `? :`
- оператор конкатенації з рядком `+`

Звернення до полів і методів об'єкту або класу можна назвати основною операцією над посилальними величинами. Здійснюється вона за допомогою символу крапка. Наприклад:

```
"abc".charAt(1)
```

```
Math.sin(Math.PI*30/180)
```

Операція присвоєння присвоює значення одного посилання іншому, але новий об'єкт не створюється. Ця операція, так само як і у мові C, повертає результат, значенням якого є посилання, яке було присвоєно.

Операція `instanceof` дозволяє дізнатися, чи належить об'єкт даному класу. Цей оператор має два аргументи. Зліва вказується посилання на об'єкт, а праворуч - ім'я типу, на сумісність з яким перевіряється об'єкт. Наприклад:

```
"abc" instanceof String
```

```
"abc" instanceof Object
```

Для посилання, що дорівнює `null`, оператор `instanceof` завжди поверне значення `false`.

Оператори порівняння `==` і `!=` перевіряють рівність (або нерівність) посилань на об'єкти. Якщо ж потрібне альтернативне порівняння - за значенням, то слід використовувати перевизначений у відповідному класі метод `equals()`.

Для приведення типів використовується пара круглих дужок з іменем класу, до якого приводиться об'єкт. Приводячи типи, слід враховувати, що

кожний клас знає свій «рідний» клас, що відповідає конструктору, за допомогою якого він створювався. Але в програмі він може проходити і під типами, що вище його по ієрархії. Операція приведення буде коректною, якщо класи знаходяться в одній гілці ієрархії.

В операції конкатенації з рядком можуть приймати участь об'єкти будь яких типів. Наприклад:

```
String s = '-' + "abs" + new Double(3) + 5;
```

3.1.8 Структура програми на Java

Програма на Java являє собою сукупність відкомпільованих файлів класів. Для того, щоб програма запрацювала, необхідно, щоб один з цих класів мав статичний метод main. Цей метод є точкою входу до програми.

Для зручності роботи з java програмами, файли класів пакують у .jar архів. Цей файл, окрім файлів .class, може містити файли з вихідними текстами класів, файли ресурсів, бібліотек. Для того, щоб програму можна було виконати, .jar архів має також містити файл MANIFEST.MF, який знаходиться у каталозі META-INF. У файлі MANIFEST зберігається службова інформація, зокрема, ім'я класу, з якого треба починати роботу.

У найпростішому випадку програма являє собою один клас, рисунок 2.1

```
// Вказівка на пакет, якому належить клас
package myFirstProject;

// Заголовок класу
public class MyFirstClass {

    //Заголовок методу
    public static void main(String[] args) {

        System.out.println("Welcome to Java world!");

    }

}
```

Рисунок 3.1 – Найпростіша програма на Java

3.2 КОРОТКІ ВІДОМОСТІ ПРО RUNTIME БІБЛІОТЕКУ JAVA

3.2.1 Структура бібліотеки

Кожний Java проект має посилання на JRE System Library. Ця бібліотека складається з декількох файлів .jar (java архівів), які містять байт-коди класів, що використовуються java-програмами.

Кожний з цих файлів має деревовидну структуру папок, які у Java називають пакетами. Пакети, що є листками дерева, містять файли класів.

Деревовидна структура пакетів забезпечує зручний доступ до класів та спрощує їх ідентифікацію, бо до складу назви класу входять і імена усіх пакетів, що складають ланцюжок доступу до класу.

Найцікавішим з файлів системної бібліотеки для починаючого Java програміста є файл `rt.jar`. Зазирнувши у цей архів, можна побачити безліч пакетів тільки верхнього рівня, а кількість класів мабуть неосяжна. Але лякатися не потрібно. Перш за все, якусь частину класів мабуть використовує тільки сама система. По-друге, якісь групи пакетів стосуються специфічних задач, які зараз нам не цікаві. Але є декілька груп пакетів, які є основою мови Java. Нижче перелічені деякі з них, що будуть найчастіше зустрічатися у нашому курсі:

- `java.lang` – базові елементи мови;
- `java.util` – класи, які ефективно вирішують різноманітні проблеми програмування;
- `java.io` та `java.nio` – послуги введення-виводу;
- `java.awt` та `java.swing` – послуги створення GUI.

3.2.2 Клас `java.lang.Object`

У мові Java класи мають ієрархічну структуру і в корені цього ієрархічного дерева знаходиться клас `Object`. У цьому класі усього дев'ять методів, але ці методи доступні об'єктам будь яких класів. З більшістю із них ми познайомимось пізніше а зараз коротко розглянемо тільки три з них. Тексти методів дуже короткі, тому наведемо їх повний текст.

Перший метод, `equals`, використовується для порівняння об'єктів, тут `this` – це посилання на об'єкт, що викликав метод.

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Як бачимо з тексту методу, порівнюються посилання на об'єкти. Таким чином два об'єкти будуть однакові, якщо вони займають ту саму область пам'яті. Але у класах спадкоємцях цей метод можна перевизначити, встановивши будь які правила порівняння.

Другий метод, `hashCode`, використовується для отримання унікального цілого числа для об'єкта, що викликав цей об'єкт.

```
public native int hashCode();
```

Як бачимо, метод має специфікатор `native`, який означає, що до тексту методу доступу нема, але у деяких джерелах стверджується, що хеш код формується на підставі адреси області пам'яті, яка виділена для об'єкту. Цей метод також може бути перевизначений у будь яких класах, але необхідно стежити, щоб об'єкти, однакові з точки зору `equals`, мали однаковий хеш код.

Третій метод, мабуть найбільш популярний, це метод `toString`.


```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

З тексту методу видно, що він повертає рядок символів, який формується з імені класу та хеш коду у вигляді шістнадцятиричного числа і таким чином унікально характеризує об'єкт. Але цей метод, зазвичай, перевизначають у класах спадкоємцях.

Слід зазначити, що Eclipse має засоби для автоматизованої генерації коду усіх трьох зазначених методів, з урахуванням побажань розробника класу.

3.2.3 Клас *java.lang.System*

Клас *System* містить кілька публічних полів і статичних методів. Об'єкти цього класу не створюються.

Серед полів класу *System* є такі:

- *in*, що містить посилання на засіб для стандартного вводу;
- *out*, що містить посилання на засіб для стандартного виведення;
- *err*, що містить посилання на засіб для виведення інформації про помилки.

Методи класу забезпечують доступ до деяких властивостей системи і змінних оточення, відповідають за завантаження бібліотек, та багато іншого.

Зокрема, клас має дуже корисний метод *arrayCopy* для швидкого копіювання частини масиву системними засобами.

Розглянемо деякі корисні можливості, що пов'язані з класом *System*.

3.2.3.1 Виведення інформації на консоль

Для виведення на консоль найчастіше використовуються методи *print* та *println* об'єкта стандартного виведення *System.out*. Наприклад:

```
int a=5,b=10;
System.out.print("a=" + a + "; b=" + b + "; a + b=");
System.out.println(a + b);
```

3.2.3.2 Виведення інформації про помилки

Для повідомлень про помилки використовуємо об'єкт *System.err*:

```
int x=0;
if(x==0)System.err.println("Некоректне значення параметру, x=0 ??? ");
else System.out.println("y="+100/x);
```

3.2.3.3 Введення інформації з консолі

Для введення інформації з консолі використовується потік стандартного вводу *System.in*. Але цей об'єкт потребує додаткових налаштувань. Зручною надбудовою для цього потоку є клас *java.util.Scanner*.

Об'єкти класу *Scanner* мають різноманітні методи для тестування потоку вводу та введення даних різних типів.

Для тестування використовуються методи `hasNext()`, `hasNextDouble()`, `hasNextInt()` та подібні інші.

Для введення інформації використовуються, зокрема, такі методи:

- `nextLine()` повертає поточний рядок та переходить до наступного рядка;
- `next()` повертає наступну лексему (слово) орієнтуючись на призначений роздільник;
- `nextInt()` повертає наступне ціле число типу `int`;
- `nextDouble()` повертає наступне дійсне число `double`;
- ...
- `useDelimiter(Pattern)` встановлює потрібний роздільник

Нижче наведено приклад, у якому за допомогою об'єкта класу `Scanner` з консолі зчитуються дані різних примітивних типів.

```
System.out.println("Введіть дані за таким шаблоном:");
System.out.println("Прізвище та ім'я/кільк.незадов./сердній бал/");
System.out.println("Наприклад:");
System.out.println("Кіт Базіліо/2/2,4/");
Scanner sc = new Scanner(System.in);
sc.useDelimiter("/");
String name = sc.next();
int n = sc.nextInt();
float b = sc.nextFloat();
sc.close();
System.out.println(name+ "незд: "+n+ "; ср.бал="+b)
```

3.2.3.4 Доступ до системного часу

Метод `currentTimeMillis()` повертає значення поточного часу в мілісекундах від 1 січня 1970 року.

Метод `nanoTime()` повертає значення джерела часу Java машини в наносекундах, що не пов'язано з поточним часом та датою. Використовується тільки для визначення інтервалів часу.

Ось приклад використання цих значень:

```
long time1, time2;
time1 = System.nanoTime();
System.out.println("Системний час " + System.currentTimeMillis()
    + " мілісекунд від 01.01.1970 UTC");
time2 = System.nanoTime();
System.out.println("На визначення часу витрачено " + (time2 - time1)
    + " наносекунд");
```

3.2.4 Класи для роботи з рядками символів

Рядок символів – це одна з найрозповсюджених структур даних. ВJava існує декілька класів. Розглянемо деякі з них.

3.2.4.1 Клас java.lang.String

Це мабуть єдиний клас, об'єкти якого можна створювати літерально.

```
String s = "asdf";
```

Але є можливість і використовувати різноманітні конструктори. Нижче використано два з них:

```
String s = new String ("asdf");  
char[] ach={'s', 'd', 'f', 'g', 'n'};  
String s1 = new String( ach,1,3);
```

Ще одна унікальна особливість рядків – можливість використання операції "+", що здійснює конкатенацію(з'єднання) двох окремих рядків:

```
String s2 = "Рядок1"+" Рядок2";
```

Особливістю класу String є також приналежність його до групи immutable класів, тобто в java об'єкти класу String не можна змінювати, але змінній, яка посилається на певний рядок, можна призначити інший рядок. Навіть коли у рядку замінити одну букву на якусь іншу, в java буде створено новий рядок.

Довжину рядка можна визначити за допомогою методу:

```
public int length().
```

Цей метод повертає кількість пар байтів у рядку і тому результат буде некоректним, якщо рядок містить специфічні символи, які потребують більше місця ніж два байти.

Для того, щоб порівняти два рядки на рівність можна скористатися методом equals(), наприклад:

```
str1.equals(str2);
```

Не слід використовувати операцію порівняння ==, щоб перевірити рядки на рівність. Таким чином лише перевіряється, чи рядки розташовуються за одним місцем в пам'яті, крім того, результат може бути неоднозначним, бо компілятор java однаковим рядкам виділяє пам'ять специфічним чином.

Для порівняння рядків можна також використовувати метод

```
public int compareTo(String anotherString).
```

Метод повертає 0, якщо рядки однакові. Якщо рядок, що викликає метод, менше, ніж anotherString, метод повертає число менше за 0. Якщо рядок більший, ніж anotherString, метод повертає число більше за 0.

Клас `String` надає можливість отримувати символний вигляд для числових даних базових типів `int`, `double`, та `in`. Для цього використовуються статичні методи `valueOf` з аргументами відповідних типів, наприклад:

```
int x= 123;
String sx = String. valueOf(x);
```

За допомогою методу `split` рядок можна розділити на частини, відповідно до заданих роздільників, отримавши масив підрядків. Параметр методу визначає роздільники, що можна задавати у вигляді «regular expresion». У найпростішому випадку це може бути рядок з одного символу-роздільника. Наприклад:

```
"abc,qwerty,zxcvb".split(",")
" bcv    mxbcb ncmxcm mxmxm ".trim().split("\\s+")
```

У останньому прикладі символ `\s` визначає усі стандартні пробільні роздільники [`\t\n\x0B\f\r`], а символ `+` вказує, що вони можуть бути не поодинокі.

Метод `trim` вилучає початкові та хвостові пробіли.

Більше інформації про побудову «regular expresion» можна отримати з документації до класу `java.util.regex.Pattern`.

Взагалі клас `String` має біля півсотні публічних методів, з якими можна познайомитися, відкривши `java.lang.String` з файлу `rt.jar`. Для прикладу наведемо тільки декілька з цих методів.

```
public int indexOf(String str, int fromIndex).
public String substring(int beginIndex, int endIndex).
public String replace(char oldChar, char newChar)
```

3.2.4.2 Класи `java.lang.StringBilder` та `java.lang.StringBufer`

Об'єкти класу `String` працюють неефективно, якщо необхідно часто змінювати рядок. У таких випадках доцільно використовувати клас `StringBuilder`. Цей клас з'явився у `JDK 5.0`. Він майже повністю заміняє більш ранній клас `StringBuffer`, який дещо менш ефективний при роботі з рядками, але натомість підтримує багатопоточність (`multithreading`). Робота з `StringBuilder` та `StringBuffer` практично ідентична.

Для створення об'єктів у класі передбачено декілька конструкторів. Нижче наведено два з них:

```
public StringBuilder()
public StringBuilder(str)
```

Використовуючи останній конструктор можна перетворити об'єкт класу `String` у об'єкт класу `StringBuilder`. Зворотне перетворення можна виконати за допомогою метода `toString(String)` класу `StringBuilder`.

Основними операціями над `StringBuilder`, яких немає у `String`, є методи `append()` та `insert()`, що є переважаними і можуть приймати дані будь-яких

типів. Кожен метод конвертує їхній аргумент у об'єкт String і згодом приєднує або вставляє символи даного рядка до символічної послідовності у об'єкті StringBuilder. Метод append() завжди додає символи у кінець існуючої послідовності, в той час як insert() додає символи всередину символічної послідовності з вказаної позиції.

Окрім цих методів можуть стати у нагоді і такі методи:

```
public StringBuilder delete(int start, int end)
public StringBuilder replace(int start, int end, String str)
public StringBuilder reverse()
```

3.2.5 Класи-обгортки

3.2.5.1 Класи-обгортки для простих типів

Для кожного базового типу існують спеціальні допоміжні класи-обгортки (wrapperclasses).

Для числових типів byte, short, int, long, float і double це Byte, Short, Integer, Long, Float і Double.

Для створення об'єкту відповідного типу можна використати конструктор, куди передати або число відповідного типу, або число у вигляді рядка символів.

Усі числові класи обгортки успадковують абстрактний клас Number, де оголошені методи byteValue, shortValue, intValue, longValue, floatValue, doubleValue. За допомогою цих методів дані будь якого числового класу обгортки можна перетворити у будь який інший числовий примітивний тип, але коректність не гарантується. Можлива втрата точності і навіть знаку. Наприклад:

```
Double x = new Double(2.7);
return x.intValue();
```

Числові класи-обгортки містить також статичні методи, назва яких складається зі слова «parse» та імені класу, наприклад “parseDouble”. Ці методи використовуються для перетворення рядка символів у число відповідного типу. Наприклад:

```
String s="3.14";
return Double.parseDouble(s);
```

Окрім методів, класи обгортки містять константи, які визначають різноманітні характеристики класу, наприклад:

```
return Integer.MAX_VALUE;
return Double.MIN_EXPONENT;
```

Докладніше з константами можна ознайомитися, відкривши вихідні тексти класів-обгортки.

Для типів char і boolean класи обгортки це Character та Boolean.

Клас `Character` містить багато методів для обробки та тестування символів, наприклад:

```
public static boolean isDigit(char ch)
public static boolean isLetter(char ch)
```

На останок розглянемо приклад перетворення рядка символів, що містить числа розділені пробілами у масив чисел:

```
String[] ars = "23.4 13 45 1.1 8 12.3".trim().split("\\s+");
double[] arn = new double[ars.length];
for (int i = 0; i < ars.length; i++) {
    arn[i] = Double.parseDouble(ars[i]);
}
System.out.println(Arrays.toString(arn));
```

3.2.5.2 Класи типу `java.util.OptionalDouble`

У Java8 з'явилася ще одна група класів контейнерів для примітивних типів – `OptionalDouble`, `OptionalInt`, `OptionalLong`. Особливість цих класів полягає у тому, що їх об'єкти можуть містити, а можуть і не містити значення примітивного типу.

Пустий контейнер можна створити за допомогою статичного методу `empty()`, і метод `isPresent()` для цього об'єкту поверне `false`.

Наприклад,

```
java.util.OptionalDouble dOp = java.util.OptionalDouble.empty();
return dOp.isPresent();
```

Для створення об'єкту зі значенням використовується статичний метод `of()` з параметром відповідного типу, наприклад:

```
java.util.OptionalDouble dOp = java.util.OptionalDouble.of(3.14);
return dOp.isPresent();
```

Метод `ifPresent()` об'єктів цього класу дозволяє визначити блок коду, що буде виконуватися, якщо контейнер не пустий. Блок коду записують у вигляді лямбда функції. Про ці функції буде сказано пізніше. Наприклад:

```
OptionalDouble dOp = OptionalDouble.of(3.14);
OptionalDouble dOpEmp= OptionalDouble.empty();
dOpEmp.ifPresent((x)->System.out.println(x));
dOp.ifPresent((x)->System.out.println(x));
```

Метод `orElse()` повертає значення, що є параметром цього методу, якщо контейнер пустий, а інакше повертає значення, що знаходиться у контейнері:

```
OptionalDouble dOp = OptionalDouble.of(3.14);
OptionalDouble dOpEmp= OptionalDouble.empty();
System.out.println(dOp.orElse(6.28));
```

```
System.out.println(dOpEmp.orElse(6.28));
```

Методи `getAsDouble()`, `getAsLong()`, `getAsInt()` повертають відповідні значення примітивних типів.

3.2.6 Форматування даних

3.2.6.1 Класи `java.text.NumberFormat` та `java.text.DecimalFormat`

Абстрактний клас `NumberFormat` надає інтерфейс для форматування і розбору чисел. Конкретна реалізація дій по форматуванню реалізована в спадкоємцях цього класу, зокрема, `DecimalFormat`.

Щоб відформатувати число, слід перш за все створити об'єкт типу `DecimalFormat`, використовуючи рядок шаблону в якості параметру конструктора класу.

Шаблон може складатися з таких символів:

- 0 – кожному з яких буде встановлено цифру або 0, якщо цифри нема;
- # – кожному з цих відповідає або цифра або нічого, якщо цифри нема.
- .
- як десяткова крапка;
- ,
- як роздільник тисяч;
- '
- для того, щоб обмежити буквальні символи або текст.

Після створення об'єкту типу `DecimalFormat`, можна використовувати його для форматування чисел за допомогою методу `format`, до якого в якості параметру передаємо число, що слід відформатувати. Метод повертає рядок символів. Наприклад:

```
java.text.NumberFormat nf = new java.text.DecimalFormat("#,##0.00'коп'");  
String s =nf.format(-1234.5);  
System.out.println( s);
```

Клас `NumberFormat` дозволяє також формувати числа як грошові одиниці, наприклад:

```
NumberFormat nf =  
NumberFormat.getCurrencyInstance(java.util.Locale.CHINA);  
String s =nf.format(1245.3);  
System.out.println( s);
```

Є також можливість переводити числа у проценти:

```
java.text.NumberFormat nf = java.text.NumberFormat. getPercentInstance();  
nf.setMinimumFractionDigits(2);  
String s =nf.format(2.0055);  
return s;
```

3.2.7 Класи для роботи з математичними функціями та випадковими числами

3.2.7.1 Клас Math

Як вже наголошувалося, у Java відсутнє поняття функція, але це не означає, що у цій мові не можна користуватися математичними функціями на зразок `sin` або `abs`. Проблема математичних функцій в Java вирішується за допомогою класу `Math`. Це клас, який має тільки статичні методи та константи. Саме в цих методах і реалізовані математичні функції. Об'єкти цього класу створювати нема сенсу, тому у ньому відсутній публічний конструктор.

Ось деякі з констант та методів класу `Math`:

```
public static final double PI = 3.14159265358979323846;
public static double log(double a)
public static double pow(double a, double b)
public static double random()
public static double min(double a, double b)
```

3.2.7.2 Клас `java.util.Random`

Об'єкти цього класу використовуються для генерації псевдовипадкових чисел, зазвичай, рівномірно розподілених.

Клас має два конструктори.

```
public Random()
public Random(long seed)
```

Перший з цих конструкторів, використовує в якості зерна значення системного таймера в наносекундах. Тому об'єкти, що були створені з проміжком часу, що більший за одну наносекунду, будуть формувати різні випадкові послідовності.

Другий конструктор використовує в якості зерна число, що задано в конструкторі. Це дає можливість відтворювати ту саму випадкову послідовність скільки завгодно разів.

Найбільш популярні методи цього класу – це методи, ім'я яких починається словом «next», після чого йде назва типу, наприклад:

```
public double nextDouble().
```

Ці методи повертають псевдовипадкові числа відповідного примітивного типу у межах діапазону цього типу.

Метод

```
public int nextInt(int bound)
```

повертає цілі додатні числа в межах діапазону від 0 до `bound-1`.

3.2.8 Класи для роботи з часом та датою

В Java існують класи для роботи з датою та часом: `Date`, `Calendar`, `GregorianCalendar`, `TimeZone`, `SimpleTimeZone`, `SimpleDateFormat`, але працювати з ними виявилось не зовсім зручно.

Тому у Java SE 8 з'явилися нові засоби для роботи з датою та часом. Розглянемо деякі з них.

`java.time.LocalDate` - значення дати для локальної часової зони. Даний клас не містить будь-якої інформації про час.

```
LocalDate.now(); => 2014-02-23
```

`java.time.LocalTime` - зберігає інформацію про час для локальної часової зони. Даний клас не містить будь-якої інформації про дату.

```
LocalTime.now(); => 01:53:42.656
```

`java.time.LocalDateTime` - зберігає інформацію про дату і часу для локальної часової зони. Точність до наносекунд.

```
LocalDateTime.now(); => 2014-02-25T01:57:52.250
```

`java.time.ZonedDateTime` - зберігає інформацію про дату і час, із зазначенням часової зони. Точність до наносекунд.

```
ZonedDateTime.now (); => 2014-02-25T02: 09: 53.321 + 03: 00  
[Europe/Minsk]
```

`java.time.Instant` – момент часу з точністю до наносекунд.

```
Instant.now (); => 2014-02-24T23: 18: 45.866Z
```

`java.time.Duration` - застосовується для представлення проміжку часу вираженого в одиницях часу.

```
Instant instant1 = Instant.now ();  
Instant instant2 = instant1.plusSeconds (10);  
Duration.between (instant2, instant1) .getSeconds () => -10  
Duration.between (instant2, instant1) .abs ().getSeconds () => 10
```

`java.time.Period` - застосовується для представлення проміжку часу вираженого в одиницях дати.

```
LocalDate today = LocalDate.now ();  
LocalDate birthday = LocalDate.of (1986, Month.JUNE, 05);  
Period.between (birthday, today) .getYears () => 27
```

`java.time.format.DateTimeFormatter` - використовується для форматування і парсинга значень дати і часу. Для цього використовуються методи `format ()` і `parse ()` відповідно. `DateTimeFormatter` дозволяє використовувати передбачені формати дати і часу, або визначати власні.

```
Locale.setDefault(new Locale("UK"));
DateTimeFormatter myFormat = DateTimeFormatter.ofPattern("dd MMMM
уууу року, HH:mm:ss");
String date = LocalDateTime.now().format(myFormat);
System.out.println("Зараз: " + date); //Зараз: 23 серпня 2014 року, 15:09:35
```

Більш детальний опис нового API тобто класів та їх методів наведений у документації.

3.2.9 Клас для обробки масивів

3.2.9.1 Клас java.util.Arrays

Утилітний клас Arrays містить статичні методи, що реалізують популярні алгоритми обробки масивів. Нижче наведена група методів цього класу для масивів типу double. Такі самі методи є і для інших примітивних типів і є лише один метод для сортування будь яких об'єктів. Цей клас наочно демонструє переваги об'єктно-орієнтованого програмування.

```
public static double[] copyOf(double[] original, int newLength)
public static int binarySearch(double[] a, double key)
public static void fill(double[] a, double val)
public static void sort(double[] a)
public static String toString(double[] a)
public static void setAll(double[] array, IntToDoubleFunction generator)
```

Методи setAll та sort мають аналоги для паралельної обробки масивів setParallelAll та parallelSort. Але використання цих методів доцільно тільки для дуже і дуже великих масивів.

3.2.9.2 Лямбда-функції

Останній, з вище наведених методів, існує у класі Arrays починаючи з версії Java 1.8. Другий параметр цього методу дозволяє використання так званої лямбда функції, яка отримує в якості параметра індекс (Int) поточного елементу масиву, а повертає значення (Double) яке буде записано у поточний елемент масиву.

Лямбда функція складається з трьох частин.

Перша частина – це круглі дужки, у яких наводиться перелік параметрів функції. Типи не вказуються, бо вони визначені типом функції. У методі, що розглядається, параметр має бути int, назва параметру не має значення.

Друга частина – це комбінація з двох символів ->, що є ознакою лямбда.

Третя частина являє собою тіло функції. Якщо функція складається з декількох операторів, то потрібні фігурні дужки і оператор return. Функція може і не використовувати параметр.

Більш докладно лямбда будуть розглянуті пізніше, а зараз просто

наводимо приклад їх використання.

```
int[] arr = new int[25];
Random rnd = new Random();
Arrays.setAll(arr, (i) -> rnd.nextInt(4) + 2);
System.out.println(Arrays.toString(arr));
Arrays.setAll(arr, (i) -> {
    if (arr[i] == 5)
        return 800;
    else if (arr[i] == 4)
        return 600;
    else
        return 0;
});
System.out.println(Arrays.toString(arr));
```

3.2.9.3 Тип `java.util.stream.Stream`

В Java8 з'явилася можливість конвеєрної обробки масивів та інших груп даних. Конвеєрна обробка передбачає, що над групою даних будуть послідовно виконуватися деякі операції.

Для того, щоб скористатися можливостями конвеєрної обробки треба перш за все отримати об'єкт типу `Stream` (стрім) для набору даних. Для масивів такий об'єкт можна отримати за допомогою статичного методу `stream()` класу `Arrays`. Для випадкових чисел стрім можна отримати, наприклад, за допомогою методу `doubles()`.

Слід зазначити, що елементи масиву, для якого створено стрім, у процесі обробки стріму залишаються без змін, змінюється тільки стрім.

Для стрімів можна застосовувати різноманітні варіанти обробки, використовуючи лямбда функції. Особливість багатьох стрім-методів полягає у тому, що вони повертають знов такі стріми. Це дозволяє створити ланцюжок операцій над групою даних, який і є конвеєром.

Нижче наведено деякі з таких методів:

`filter` – вибирає із стріму дані, що відповідають заданій умові;

`distinct` – вилучає із стріму усі дублікати даних;

`sorted` – впорядковує дані стріму;

`map` – змінює дані стріму відповідно до заданого алгоритму.

До методів `filter` та `map` в якості параметра слід передавати лямбда функції, які будуть визначати характер обробки.

Нижче розглянуто приклад, у якому послідовно виконуються такі конвеєрні операції над стрімом для масиву цілих чисел:

- стрім наповнюється випадковими цілими числами від 0 до 9 за допомогою методу `map`;

- зі стріму вибираються тільки парні числа за допомогою методу `filter`;

- вилучають дублікати чисел за допомогою методу `distinct`;

- числа, що залишилися, впорядковуються;
- числа підносяться до квадрату.

```
int[] arr = new int[100];
Random rnd = new Random();
Arrays.stream(arr)
    .map((x)->rnd.nextInt(10))
    .filter((x)-> x % 2 == 0)
    .distinct()
    .sorted()
    .map((x)->x*x)
    .forEach((x)->System.out.println(x));
```

Останній метод, `forEach`, який завершує роботу зі стримом, відрізняється від попередніх тим, що не повертає стрим. Такі методи називають термінальними.

Ще один термінальний метод, `reduce`, допомагає знайти накопичені результати обробки групи даних. На відміну від попереднього, метод `reduce` повертає результат.

У методі `reduce` першим параметром є початкове значення результату, другим параметром є лямбда з двома параметрами, де перший є поточним значенням результату, а другий поточним елементом стриму.

Наведений нижче приклад забезпечує створення масиву цілих чисел, обчислення середнього арифметичного квадратів випадкових чисел та мінімального значення серед цих чисел.

```
int[] arr = new int[3];
Random rnd = new Random();
Arrays.setAll(arr,(x)->rnd.nextInt(3)+2);
System.out.println(Arrays.toString(arr));
double average= Arrays.stream(arr)
    .reduce(0,(sum,x)->sum+=x*x)/(double)arr.length;
System.out.println(average);
double minimum= Arrays.stream(arr)
    .reduce(Integer.MAX_VALUE,(min,x)->x<min ? x :min);
System.out.println(minimum);
```

Концепція стриму та конвеєрної обробки цікаві сама по собі, але мабуть, головною метою тут було запровадження паралельної обробки даних на багатоядерних пристроях. У типі `Stream` є метод `parallel()`. Виклик цього методу забезпечує подальшу паралельну обробку елементів стриму на конвеєрі.

На жаль, на сьогодні, паралельна обробка потребує багато часу на розщеплення масиву, створення пулу тредів для обробки, та об'єднання результатів і тому часто не дає бажаного ефекту.

3.3 ОГЛЯД ДЕЯКИХ КОРИСНИХ МОЖЛИВОСТЕЙ ECLIPSE

Eclipse надає багато можливостей для редагування вихідного коду, які включають в себе підсвічування синтаксису, автодоповнення коду, автогенерацію коду, а також безліч інших корисних функцій. Автодоповнення коду викликається натисканням клавіш Ctrl + Пропуск і дозволяє швидше писати код, використовуючи тільки перші букви конструкції, що створюється.

3.3.1 Автогенерація коду

Автогенерація коду - це можливість автоматичного створення цілих блоків коду з коротких аббревіатур. Для використання необхідно написати відповідну аббревіатуру і натиснути Ctrl + Пробіл (після можливо доведеться вибрати один з варіантів і натиснути Enter для вибору). наприклад:

main → public static void main (String [] args)

syso → System.out.println ()

Список всіх аббревіатур можна переглянути в налаштуваннях програми, крім того, можна додавати свої аббревіатури і генерований по ним код. Автогенерація коду в Eclipse враховує також контекст застосування і намагається підставити відповідні імена змінних, полів і методів у генерований код.

3.3.2 Автоматичне виправлення помилок

Автоматичне виправлення помилок - це засіб, що дозволяє користувачеві виконувати деякі поширені дії по усуненню помилок в програмах (синтаксичних і лексичних) в автоматичному режимі. Щоразу, коли редактор вказує на помилку, є можливість виконати одну з автоматичних дій щодо її виправлення, для цього необхідно встановити курсор на джерело помилки і натиснути Ctrl + 1 (або клікнути на іконку помилки у відповідному рядку). Наприклад, коли у виразі використовується оголошена до цього змінна, є можливість автоматично додати її оголошення, а також провести ряд інших дій. Крім виправлення помилок дана функціональність може бути використана і в інших випадках, наприклад для перейменування ідентифікатора, все залежить від контексту застосування.

Розглянуті вище можливості середовища Eclipse вимагають детального самостійного вивчення в процесі виконання лабораторної роботи.

3.3.3 Меню Source

Меню дозволяє автоматично генерувати цілі послідовності коду, вирішуючи деякі широко поширені завдання, наприклад створення конструкторів, методів доступу, перевизначення методів суперкласу і т.д. Викликається меню поєднанням клавіш Alt + Shift + S (або через навігацію по системі меню). Функція Format з меню Source забезпечує вирівнювання коду, що робить Java-код легким для читання.

3.3.4 Меню Refactor

Це меню містить цілий ряд дій для виконання рефакторингу (переробки) вихідного коду. Рефакторинг - це процес зміни вихідного коду без зміни функціональності програми. Застосовується для поліпшення дизайну, а також читабельності програм. Прикладом рефакторингу є перейменування будь-яких елементів коду в межах всього проекту (або декількох проектів), виділення блоків вихідного коду в методи, виділення інтерфейсів з класів і т.д. Викликається меню рефакторингу за допомогою поєднання клавіш Alt + Shift + T (або через навігацію по системі меню). Зверніть увагу, що дане меню є контекстне-залежним.

3.4 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Клас і об'єкт
2. Статичні поля і методи
3. Створення об'єктів
4. Операції над посилальними типами
5. Структура програми на Java
6. Структура runtime бібліотеки Java
7. Клас `java.lang.Object`
8. Клас `java.lang.System`
9. Клас `java.lang.String`
10. Класи `java.lang.StringBuilder` та `java.lang.StringBuffer`
11. Класи-обгортки для простих типів
12. Класи `java.text.NumberFormat` та `java.text.DecimalFormat`
13. Клас `Math`
14. Клас `java.util.Random`
15. Класи для роботи з часом та датою
16. Клас `java.util.Arrays`
17. Лямбда-функції
18. Тип `java.util.stream.Stream`

4 ОБ'ЄКТНА МОДЕЛЬ JAVA

4.1 ФУНДАМЕНТАЛЬНІ ПРИНЦИПИ «ЧИСТОГО» ООП

Фундаментальні принципи «чистого» ООП полягають у наступному. Єдиним елементом об'єктно-орієнтованої програми є об'єкт. У об'єктів є властивості, значення яких визначають стан об'єкту. Об'єктам притаманна поведінка, яка полягає у здатності реагувати на повідомлення. Реакція на повідомлення забезпечується програмною конструкцією, що зветься методом. Об'єкту може бути послане будь яке повідомлення. Посилка повідомлення - єдиний спосіб обміну інформацією між об'єктами.

Ці принципи на початку 70-х років сформулював Alan Kay, один з розробників мови Smalltalk.

Реалізація цих принципів у мові програмування приводить до того, що програми написані такою мовою мають ряд особливостей, які досить часто також називають принципами ООП.

4.1.1 Інкапсуляція

Перша з особливостей ООП програм – інкапсуляція. Ця особливість природним чином впливає з наведених вище принципів. Об'єкти містять в собі свої дані і можуть взаємодіяти тільки за допомогою посилки повідомлення. Внаслідок цього доступ до внутрішнього стану об'єкту можливий тільки через методи.

Ця особливість суттєво підвищує безпеку об'єкта, бо доступ до атрибутів об'єкту через методи дозволяє контролювати цей процес, або взагалі унеможливити його.

Таким чином у первинному значенні, інкапсуляція – це заборона прямого доступу до атрибутів об'єкта.

Але у сучасних об'єктно орієнтованих мовах, з метою збільшення гнучкості, це поняття дещо трансформувалося. З'явилося поняття «специфікатор доступу», який вказується при оголошенні атрибуту. У Java таких специфікаторів чотири.

– `private`, дозволяє прямий доступ до атрибутів тільки в межах класу. У цьому випадку реалізується інкапсуляція у первинній інтерпретації.

– за замовчуванням, тобто у випадках, коли специфікатор не оголошено. У цьому випадку прямий доступ дозволено тільки у всіх класах пакету. Тому такий доступ інколи називають пакетним.

– `protected`, дозволяє прямий доступ у межах пакету, а також у класах спадкоємця навіть за межами пакету.

– `public`, дозволяє прямий доступ будь звідки. Тобто у цьому випадку принцип інкапсуляції не підтримується.

Специфікатори доступу були також розповсюджені і на методи. Внаслідок цього з'явилися методи з обмеженим доступом.

Таким чином у сучасних мовах програмування поняття інкапсуляція означає обмеження доступу до деяких методів та атрибутів об'єкту.

Досить поширеним визначенням інкапсуляції є вираз «приховування реалізації», але це є лише варіант перекладу англійського слова encapsulation.

Перелік публічних методів та атрибутів визначає так званий «публічний інтерфейс об'єкту» на відміну від приватних складових класу, які «приховані» та використовуються для приватних потреб класу.

4.1.2 Поліморфізм.

Поліморфізм також пов'язаний з поняттям повідомлення. Будь-яке повідомлення може бути надіслане будь якому об'єкту, який сам вирішує обробляти його, чи ні. А якщо обробляти, то яким чином. З цього витікає, що об'єкти різних класів можуть реагувати по різному на ті самі повідомлення, це і є поліморфізм.

Таким чином, можна дати таке визначення поліморфізму – це можливість створення однакового інтерфейсу для різноманітних сутностей.

Поліморфізм – це дуже ефективний інструмент програмування. Він надає можливість створювати узагальнений код. Прикладом того, наскільки ефективний поліморфізм, може слугувати вже відомий клас Arrays. У ньому сім методів сортування масивів примітивних типів, тобто для кожного типу свій метод сортування. А для об'єктів будь якого класу тільки один метод сортування. Оце і є узагальнений код.

Поліморфізм є природним явищем для мов програмування з динамічною типізацією, коли тип аналізується вже на етапі виконання програми. Ось приклад методу для визначення меншого з двох об'єктів:

```
min(a, b){a.less(b) ? a : b}
```

Цей метод придатний для будь яких даних і буде працювати, якщо об'єкт «а» знає, як реагувати на повідомлення less(b), інакше буде помилка часу виконання. До речі, мова Smalltalk довела, що це не так вже й критично.

Але Java, C++, C# є строго (або майже строго) типізованими мовами. Компілятор суворо перевіряє відповідність типів на етапі компіляції і той самий метод ми маємо переписувати так:

```
int min(int a, int b){a.less(b) ? a : b}
long min(long a, long b){a.less(b) ? a : b}
...
```

Як бачимо, типізація є перепорою для реалізації поліморфізму.

Таким чином постає дилема - або типізація або поліморфізм. Але поліморфізм дуже потужний інструмент, і відмовлятися від нього недоцільно. Тому, наприклад, у мові Smalltalk використовували тільки динамічну типізацію.

З іншого боку контроль типів на етапі компіляції програми підвищує надійність створеного коду і відмовлятися від цього теж недоцільно.

Для вирішення цієї проблеми у сучасних мовах програмування було знайдено компромісне рішення. Суть цього рішення полягає у тому, що той самий об'єкт може використовуватися у програмі під різними типами, перелік

яких для нього попередньо визначений.

Так, наприклад, для вирішення проблеми з методом `min` у попередньому прикладі, можна визначити тип `Lessable`, у якому заявлено сигнатуру метода `less()`, реалізація методу навіть не потрібна. Такі методи називають абстрактними. Після цього метод `min` можна переписати так:

```
Lessable min(Lessable a, Lessable b){a.less(b) ? a : b}
```

Тепер використовувати цей метод можна для усіх об'єктів, які окрім свого рідного типу підтримують також і тип `Lessable`.

Звичайно, це створює додаткові складнощі. Треба створювати додаткові типи, треба реалізувати механізм визначення переліку додаткових типів для об'єктів. Але за все треба платити.

4.1.3 Спадкування

Спадкування – це особливість мови, яка передбачає, що клас може мати «підкласи» – спеціалізовані, розширені версії надкласу. Можуть навіть утворюватися цілі дерева спадкування. Підкласи можуть успадковувати атрибути та поведінку своїх батьківських класів, і вводити свої власні атрибути та методи. Успадкування може бути одиничне (один безпосередній батьківський клас) та множинне (кілька батьківських класів). Це залежить від вибору програміста, який реалізовує клас, та мови програмування.

Для мов з динамічною типізацією, спадкування - це усього лише зручний механізм для повторного використання коду. Якщо декілька класів мають щось спільне, то відповідні атрибути та методи доцільно винести до суперкласу, а не дублювати той самий код у кожному з класів.

Що стосується мов із статичною типізацією, то тут спадкування є не тільки механізмом для повторного використання коду, але й підґрунтям, на якому реалізується поліморфізм.

У мові Java вказівка на той факт, що деякий клас `B` є спадкоємцем класу `A` реалізується за допомогою службового слова `extends`,

```
public class B extends A ...
```

Основні риси спадковості у Java такі:

- типи батьківських класів можна використовувати для об'єктів класів спадкоємців;
- у класах спадкоємця доступні усі атрибути батьківських класів, що визначені як `protected`. Так само об'єктам класів спадкоємців доступні усі `protected` методи батьківських класів;
- методи і атрибути визначені у батьківських класів можуть бути перевизначені у класах спадкоємця, хоча це не вважається «правильним» спадкуванням;
- незалежно від того, під яким типом об'єкт було визначено, він знає від якого типу його було породжено;
- для об'єкту можна викликати лише методи того класу, з типом якого його було оголошено;

– незалежно від того, з яким типом було оголошено об'єкт, у разі виклику якогось методу об'єкт починає шукати цей метод у класі, що його породив, і виконує його, якщо знайдено. Якщо метод не знайдено, то об'єкт починає його шукати по гілці ієрархії знизу вгору;

– методи суперкласів можуть бути абстрактними, тобто не мати реалізації тіла, щось на зразок прототипу функції у мові С. У цьому випадку клас стає також абстрактним. Для того, щоб визначити таку особливість класу або методу використовують службове слово `abstract`.

– наявність абстрактних методів у суперкласах дозволяє створювати у цих класах узагальнені методи, де використовуються звернення до абстрактних методів. Виклик таких методів для об'єктів, що створені класами-нащадками, буде призводити до того, що замість абстрактного методу буде викликано відповідний метод з класу спадкоємця і алгоритм виконання узагальненого методу суперкласу буде залежати від того, об'єкт якого класу його викликав.

Таким чином у мові Java поліморфізм може бути реалізовано за рахунок створення у суперкласах методів, які використовують інші методи, що перевизначені у класах нащадках. Але такий спосіб забезпечує поліморфізм тільки у межах однієї ієрархії класів.

Більше можливостей для реалізації поліморфізму надає множинне спадкування, коли один клас може бути спадкоємцем багатьох класів. Але досвід використання множинного спадкування у мові С++ показав, що цей механізм пов'язаний з ризиками створення некоректного коду. Тому в Java множинне спадкування відсутнє, натомість з'явилося поняття інтерфейс (програмний інтерфейс, який не слід плутати з графічним інтерфейсом).

Інтерфейс можна розглядати як абстрактний клас, у якого усі методи абстрактні і нема атрибутів (допустимі тільки статичні константи).

Інтерфейс оголошується так само як і клас, тільки замість слова `class` використовується слово `interface`.

Інтерфейс потребує реалізації, тобто створення коду методів. Реалізуються інтерфейси у класах, і кожен клас може реалізувати скільки завгодно інтерфейсів. Для визначення того факту, що деякий клас В реалізує інтерфейси W, G та Q використовується службове слово `implements`,

```
public class B extends A implements W, G, Q ...
```

Використання інтерфейсів вирішує усі проблеми впровадження поліморфізму.

4.2 РОБОТА З ВИНЯТКАМИ

Виключна ситуація - це проблема, яка може виникнути під час виконання коду, заважаючи подальшому його виконанню, але програмісту, який передбачає, що вона може виникнути, невідомо, що слід робити у такому випадку.

Проблема виключних ситуацій, зокрема, пов'язана з контролем доступу до інкапсульованих даних.

У мові Java прийнято, що у разі виникнення виключної ситуації, слід переривати виконання поточного методу і передавати проблему до методу вищого рівня. Це називають викидом винятку. Якщо у методі вищого рівня відомо, що робити у виключній ситуації, то ця ситуація обробляється. Якщо ж ні, то виняток викидається на наступний рівень.

Таким чином робота з виключною ситуацією передбачає, як мінімум два етапи. Перший – це створення та викид винятка. Другий – це обробка винятку, хоча обробка може полягати знов таки у викиді винятку.

Для реалізації першого етапу цієї технології програміст, що створює метод, де може виникнути виключна ситуація, має вирішити три завдання – створити об'єкт-виняток, викинути його на вищий рівень і повідомити метод вищого рівня про те, що під час виклику його методу може виникнути виключна ситуація

Для створення об'єкту-винятку у Java є ціла ієрархія класів винятків, окрім того програміст може створити власні класи для винятків.

Усі типи винятків є підкласами класу Throwable, який входить в базовий пакет класів Java - java.lang. Тобто він є вершиною ієрархії класів винятків. Його два підкласи Error та Exception утворюють дві основні гілки винятків.

Клас Error з його підкласами - це помилки виконавчого середовища java. І які зазвичай не виникають при нормальній роботі середовища java. Такі винятки зазвичай не можуть бути оброблені в програмі.

Гілка класу Exception - це винятки, які програма повинна вловлювати(catch). Від даного класу та його підкласів можна утворювати власні підкласи. Важливим його підкласом є клас RuntimeException. Винятки даного типу включають такі винятки як ділення на нуль та помилкова індексація масивів.

Актуальну ієрархію класів винятків можна подивитися і уточнити в офіційній документації до JDK[1].

У простішому випадку можна використовувати клас java.lang.Exception, через конструктор якого можна у об'єкт-виняток вкласти текстову інформацію про проблему.

Для викиду створеного винятку використовується службове слово throw.

Для того щоб методи вищого рівня мали можливість реагувати на виняткову ситуацію, у заголовок методу, де така ситуація може відбутися, включають відповідне повідомлення, яке починається службовим словом throws, після якого вказується тип виключної ситуації.

Хай, наприклад, є інкапсульований атрибут x і є публічний метод setX для доступу до цього атрибуту. У методі ми реалізуємо контроль значення атрибуту, який не може бути від'ємним. Код методу може виглядати так:

```
public void setX(int x) throws Exception {
    if (x<0) throw new Exception("Неприпустиме значення "+x);
    this.x = x;
}
```

Перший варіант другого етапу обробки винятку полягає знов таки у викиді винятку. У наведеному нижче прикладі конструктор класу використовує наведений вище метод `setX(int)`, який кидає виняток. Але тут виняток не обробляється, а просто перекидається тому, хто буде користуватися цим конструктором.

```
public MyClass(int x) throws Exception {
    super();
    setX(x);
}
```

Другий варіант другого етапу роботи з винятками передбачає обробку винятку. Для цього використовується конструкція `try{}catch{}` яка складається з блоку перевірки `try` та блоку (або низки блоків) обробки `catch`. Наприклад

```
MyClass myObject;
try {
    myObject = new MyClass(-3);
} catch (Exception e) {
    JOptionPane.showMessageDialog(null, e.getMessage());
    return;
}
```

У наведеному прикладі є звертання до конструктора, що був розглянутий у попередньому прикладі і який викидає виняток. Якщо параметр задано правильно, то об'єкт `myObject` буде створено і програма продовжить роботу.

Якщо ж значення параметру конструктора буде неприпустимим, то виникне виключна ситуація, яка буде оброблятися у блоці `catch`. У наведеному прикладі обробка полягає у виведенні діалогового вікна з повідомленням про помилку, яке містить у собі об'єкт-виняток `e`, та завершенні роботи методу.

Іноді деякі операції необхідно виконати незалежно від того, чи було викинуто виключення в блоці `try`, чи ні. Це може бути, наприклад, закриття файлу або мережевого з'єднання. Для досягнення цієї мети необхідно використовувати блок `finally` після всіх блоків `catch`.

4.2.1 try з ресурсами

При використанні всередині `try` певних ресурсів, наприклад, файлів, при виникненні винятку необхідно було передбачити закриття відкритих ресурсів. Для цієї мети раніше приходилося використовувати блок `finally`. У Java 7 з'явилася конструкція `try з ресурсами (try-with-resources)`. Тепер просто можна створити ресурс у дужках зразу ж після ключового слова `try` і `java` сама потурбується про закриття ресурсу. Приклад:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

4.3 ПРИКЛАД СТВОРЕННЯ ОБ'ЄКТНО ОРІЄНТОВАНОГО ЗАСТОСУВАННЯ

Як приклад розглянемо поетапно просту задачу, додаючи на кожному етапі нові вимоги до застосування. Спочатку задачу будемо розглядати, використовуючи діаграми класів, а після завершення перейдемо до кодування.

Припустимо, що є фірма, яка випускає два види продукції – бруси та циліндри. Потрібно налагодити облік цієї продукції.

Етап 1.

Після першого кроку аналізу можна створити класи для кожного з видів продукції, рисунок 4.1

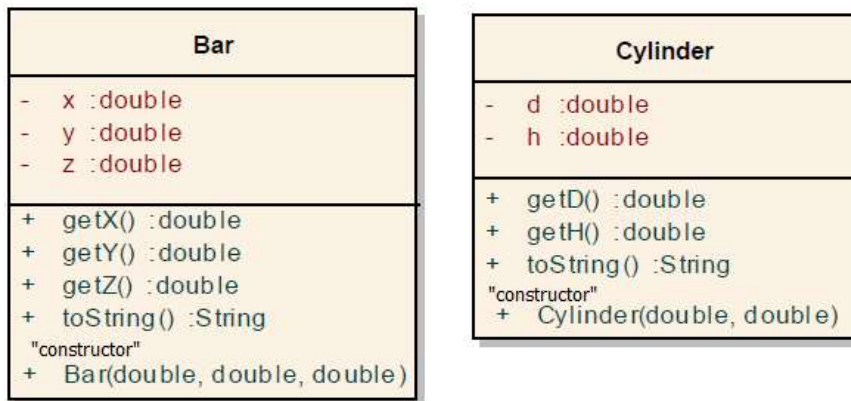


Рисунок 4.1 – Діаграма класів для видів продукції, варіант 1

Створені класи містять мінімум інформації про продукцію, тільки її розміри. Об'єкти класів можна створювати за допомогою конструкторів з параметрами, а інформацію про об'єкт можна отримати через метод toString, або через методи доступу до розмірів.

Етап 2.

Припустимо, що після обговорення першого варіанту діаграми класів з'ясувалося, що вироби можуть бути виготовлені з різного матеріалу і кожен з них має свій реєстраційний номер, який не залежить від виду виробу.

Зважаючи на те, що і матеріал і номер потрібно описувати для обох видів виробів, можна створити суперклас для вже створених класів, і у ньому визначити ці поля із специфікатором protected. Окрім того, потрібен клас для матеріалу. Новий варіант діаграми класів показано на рисунку 4.2.

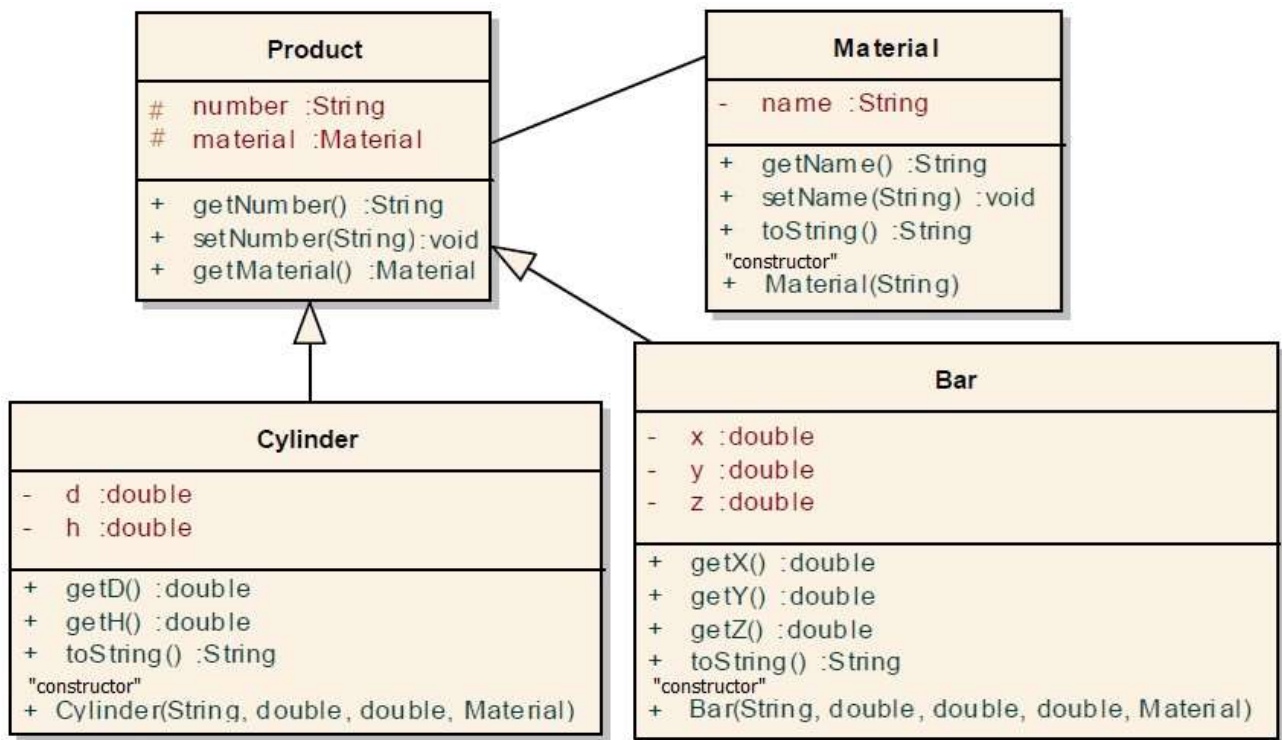


Рисунок 4.2 – Діаграма класів для видів продукції та матеріалу, варіант 2

Клас для матеріалу ми знову таки зробили по мінімуму, задаючи тільки його ім'я. Методи доступу до цього поля дозволяють змінювати його значення.

У класі продукт номер можна змінювати, а матеріал ні.

Довелося також змінити параметри конструкторів у класах для виробів.

Етап 3.

Уявімо тепер, що після обговорення другого варіанту діаграми класів з'ясується, що потрібно обчислювати масу виробів, бо їх вивозять на баржах і потрібно розраховувати завантаження барж.

З курсу фізики відомо, що масу тіла можна обчислити як добуток об'єму на густину матеріалу, з якого зроблено тіло. Таким чином, доведеться доопрацювати клас матеріал і додати туди атрибут «густина матеріалу».

Доведеться також доробити класи для виробів додавши туди методи для обчислення об'єму.

Постає також цікаве питання, де обчислювати масу? Звичайно, можна це робити у класах для виробів. Але ж формула не залежить від виду виробу, тому виникає бажання обчислювати масу у класі Product. Але проблема полягає у тому, що в класі Product відсутній метод для обчислення об'єму і компілятор буде вимагати створити його. Ми його і створимо, але абстрактним. Внаслідок цього і клас Product теж буде абстрактним. Діаграма класів, що утворилася після перелічених змін, наведена на рисунку 4.3.

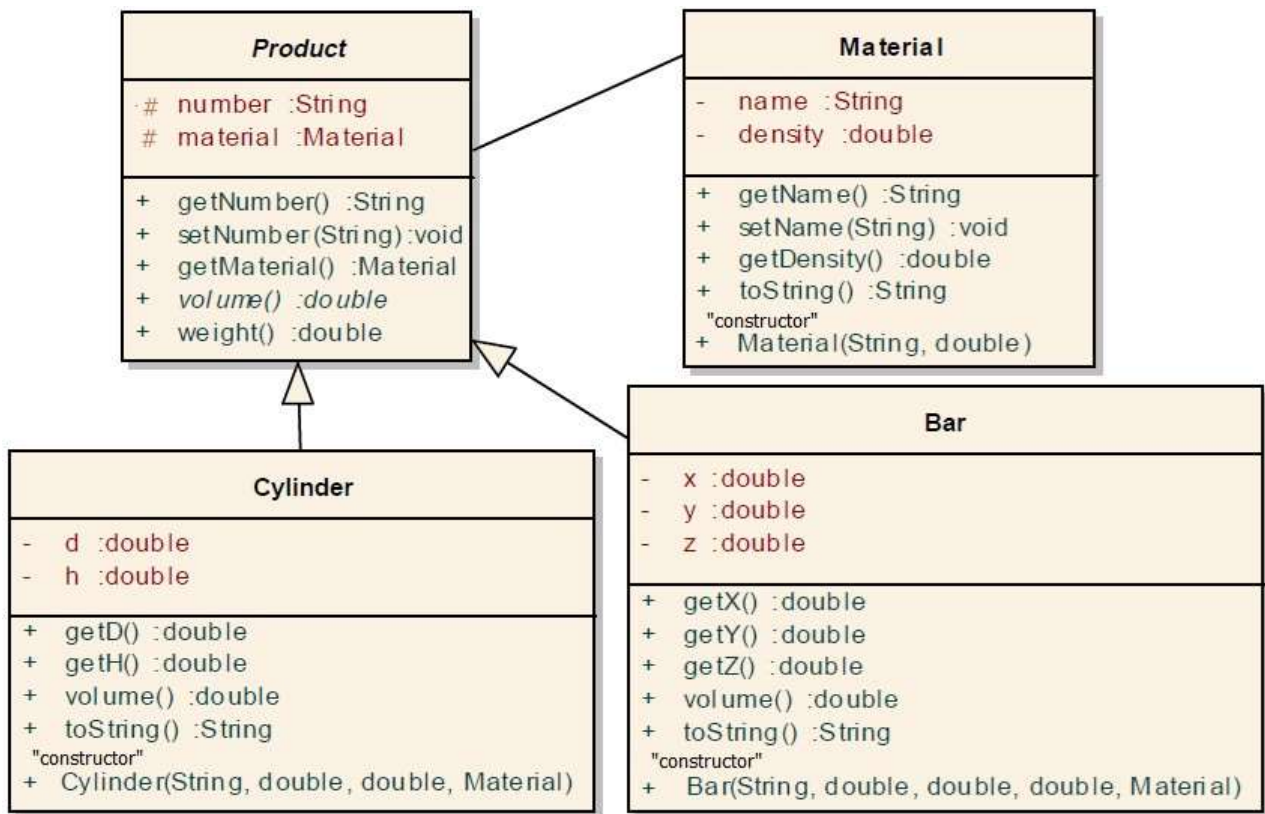


Рисунок 4.3 – Діаграма класів, варіант 3

Після доопрацювання діаграми класів ми отримали узагальнений метод розрахунку маси `weight()` та два поліморфних методи `volume()`. Реалізувати поліморфізм стало можливим завдяки ієрархічній структурі класів.

Етап 4.

Наступний етап модернізації нашої системи буде пов'язаний з тим, що на фірмі з'явився ще один вид продуктів (пакунки з відходами виробництва), який характеризується тільки назвою та вагою, але його теж треба вантажити у баржі, і тому треба якось поєднати з брусами та циліндрами.

Об'єднує ці три типи виробів необхідність доступу до ваги. Тому, окрім класу для пакунків з відходами `Waste`, ми створимо інтерфейс `IWeight`, у якому буде тільки один метод `weight()`, і заявимо цей інтерфейс як у класі `Waste`, так і у класі `Product`, де метод `weight()` вже реалізовано. Після цього уся продукція фірми, що завантажується у баржі, може проходити під типом `IWeight` і можна буде без проблем підраховувати сумарну масу завантаження.

Відповідна діаграма класів нашої системи наведена на рисунку 4.4.

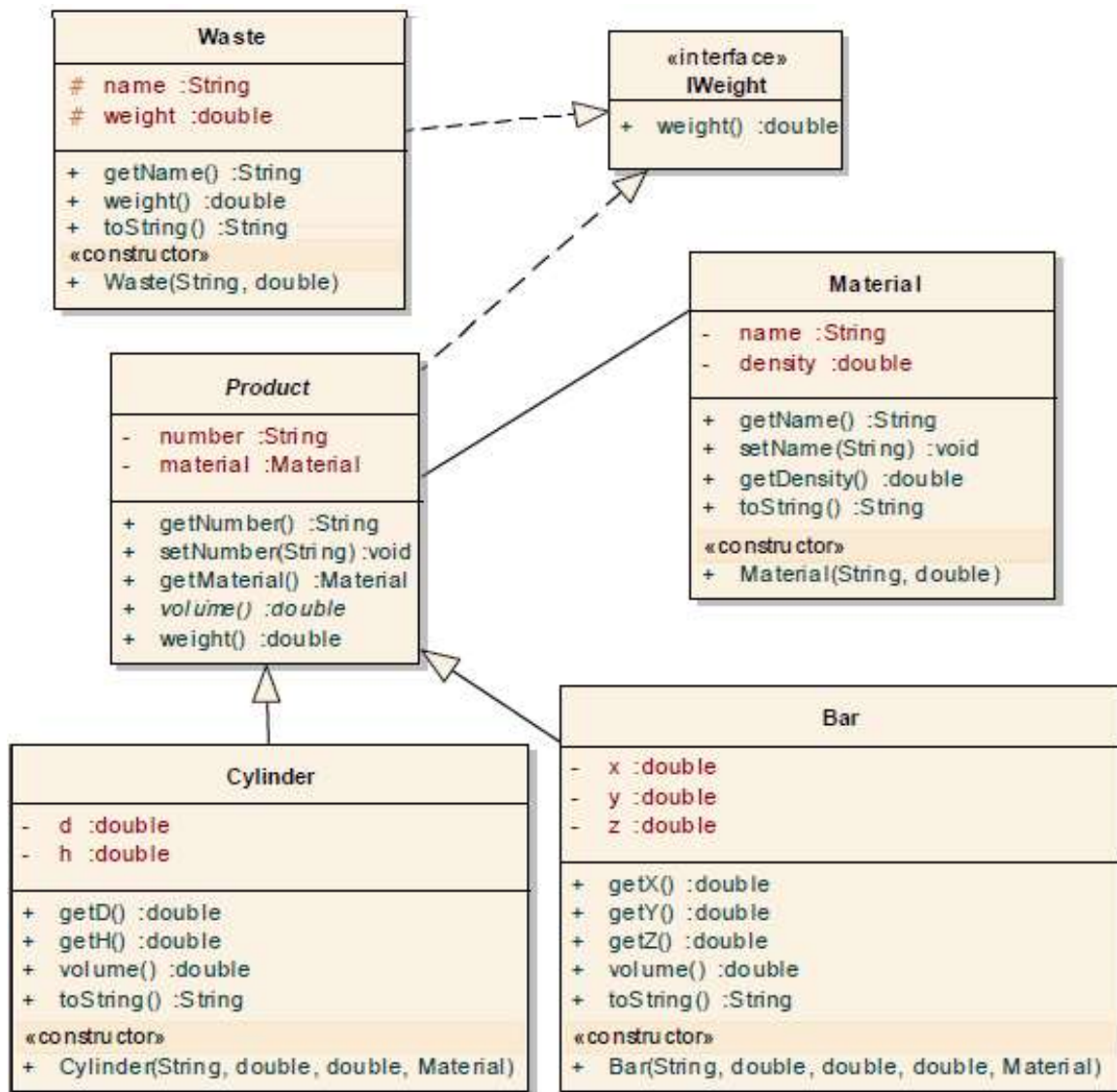


Рисунок 4.4 – Діаграма класів, варіант 4

Етап 5.

Після введення системи в експлуатацію з'ясувалося, що перелік виробів, які було передано на баржу, потрібно впорядковувати за типом виробів та вагою.

Для того, щоб вироби можна було сортувати, необхідно у відповідних класах реалізувати інтерфейс `java.util.Comparable`, який визначає метод `int compareTo(Object)`, що використовується для порівняння об'єктів.

Тип виробів можна визначати за іменем класу, а вага доступна через інтерфейс `IWeight`.

Відповідно до цього зробимо зміни у діаграмі класів. Додаймо інтерфейс `Comparable` з методом `compareTo`, та заявимо реалізацію цього методу у класах `Product` та `Waste`.

Після доопрацювання діаграма має вигляд, що наведено на рисунку 4.5.

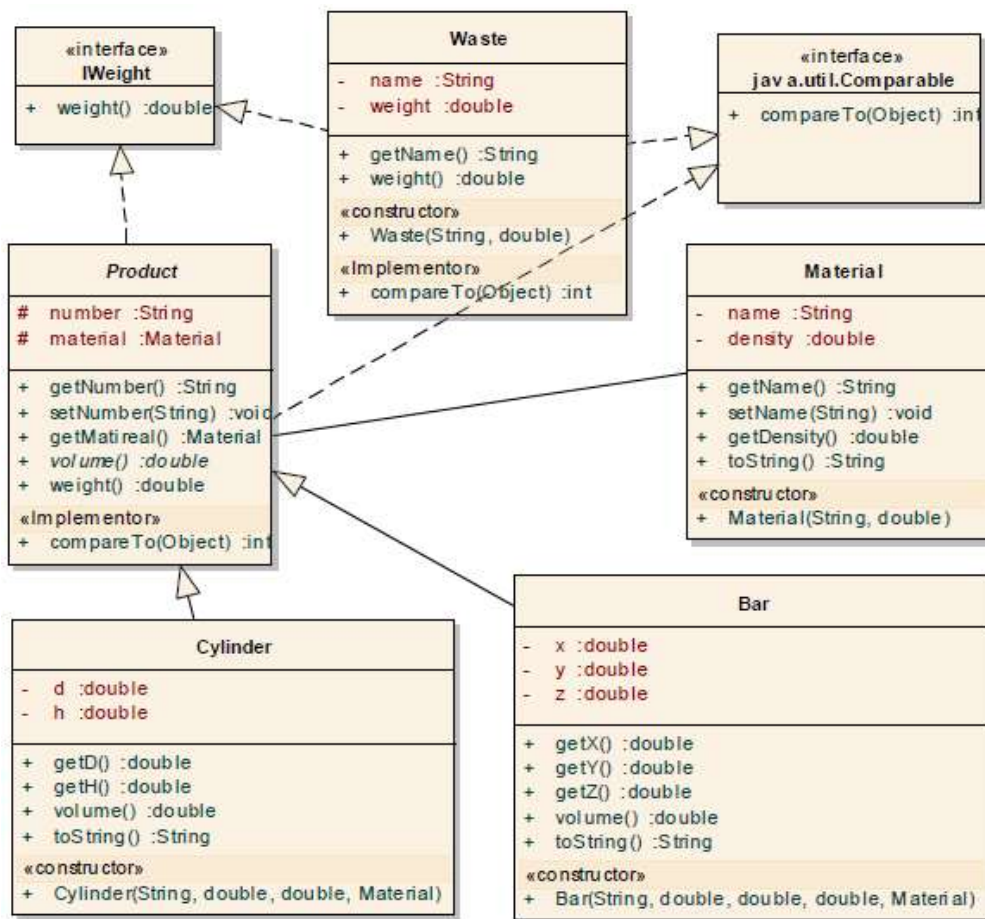


Рисунок 4.5 – Остаточна діаграма класів (Варіант 5)

Усі діаграми класів була створені за допомогою Enterprise Architect, який надає можливість згенерувати відповідні класи.

Подальшу роботу з класами можна продовжити вже в Eclipse і доопрацювати код класів.

Для тестування класів, що були реалізовані, можна створити програму, яка забезпечить введення інформації про об'єкти створення цих об'єктів, накопичення їх у масиві. Після завершення вводу, програма має впорядкувати масив, вивести перелік введених продуктів та підрахувати їх загальну вагу.

Незважаючи на різні класи об'єктів, що створюються, їх можна зберігати в одному масиві, якщо оголосити цей масив з типом IWeight. Окрім того, це надає можливість впорядковувати масив та підраховувати сумарну вагу, викликаючи у циклі метод weight(), який у свою чергу буде викликати методи volume() з різних класів. Це і є прояв поліморфізму.

4.4 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Основні концепції ООП.
2. Інкапсуляція та її реалізація в Java.
3. Спадкування та його реалізація в Java.
4. Поліморфізм та його реалізація реалізації в Java.
5. Формування та обробка виключних ситуацій.

5 ПОТОКИ ВВЕДЕННЯ-ВИВЕДЕННЯ І СЕРІАЛІЗАЦІЯ ОБ'ЄКТІВ

5.1 ЗАГАЛЬНІ ВІДОМОСТІ ПРО ВВЕДЕННЯ/ВИВЕДЕННЯ ДАНИХ

В Java розрізняється два види потоків - потоки даних (stream) і потоки виконання коду (thread).

У даному розділі ми познайомимося з потоками, які вирішують задачу введення/виводу даних.

Складність системи введення/виводу пов'язана з тим, що необхідно врахувати не тільки наявність безлічі джерел і приймачів даних, з якими необхідно підтримувати зв'язок (файли, консоль, мережеві з'єднання), а й реалізації різних форм цього зв'язку (послідовний доступ, довільний, буферизований, двійковий, символний і т. д.). З цієї причини в бібліотеці вводу/виводу Java дуже багато класів. Окрім того, у нових версіях мови в бібліотеці вводу/виводу відбулися значні зміни, покликані поліпшити продуктивність і функціональність.

Тому, щоб досить добре розібратися з бібліотекою введення/виводу Java, слід читати спеціальну літературу.

У цьому ж розділі ми познайомимося тільки з деякими класами бібліотеки Java, що відповідають за введення/виведення, а також навчимося зберігати об'єкти в файлі і відновлювати їх з файлу.

5.2 ВІЗУАЛЬНІ КОМПОНЕНТИ ДЛЯ РОБОТИ З ФАЙЛАМИ ТА КАТАЛОГАМИ

5.2.1 Клас *JFileChooser*

Перед тим як перейти до класів, які здійснюють запис і читання даних, ми розглянемо допоміжні інструменти, щ призначені для роботи з файлами та каталогами.

Елемент інтерфейсу користувача типу *JFileChooser* слугує для відображення списку файлів і вибору користувачем файлу або папки. На рисунку 5.1 показано як він виглядає. Як видно з цього рисунка, його натуральний вигляд виглядає дещо застарілим, але він може підлаштовуватися під операційне оточення. Для цього необхідно зробити такі налаштування у програмі:

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

Це найкраще зробити на самому початку метода `main()` візуальної частини.

Для відкриття діалогу можна використовувати методи `showOpenDialog()` або `showSaveDialog()`. Як параметр в методи відкриття передається посилання на батьківське вікно або компонент. Можна вказувати `null`.

Після відкриття діалогу слід вибрати потрібний файл або папку і натиснути кнопку вибору (Save або Open). Діалог запам'ятовує вибір і закривається.

Після цього можна отримати доступ до результату вибору через метод `getSelectedFile()` або (`getSelectedFiles()`). Ці метод повертають об'єкт типу `File` або масив таких об'єктів.

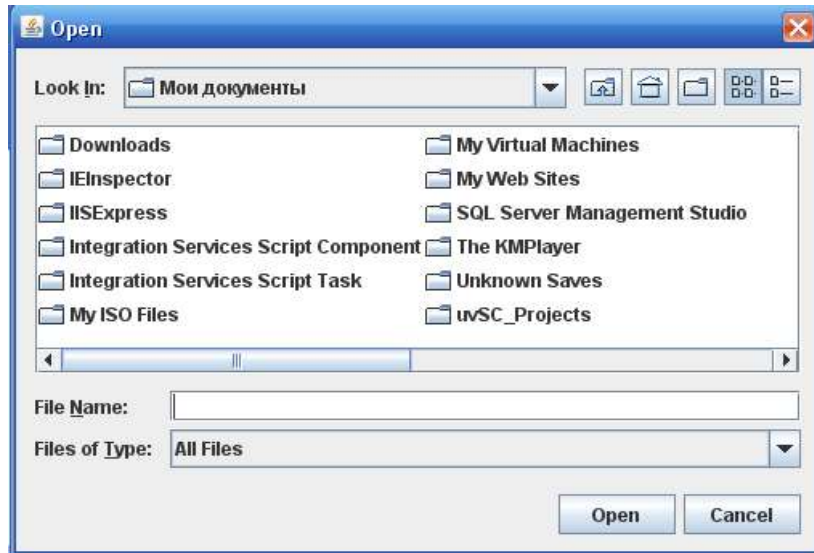


Рисунок 5.1 – Вид компоненту JFileChooser

Об'єкт класу `File` - це не файл, назва класу оманлива. Об'єкти класу `File` це ніби то посередники, які забезпечують доступ до імені вибраного файлу (метод `getAbsolutePath()`), або до імен групи файлів, що знаходяться в каталозі (метод `list()` повертає масив рядків з іменами всіх файлів).

Ось приклад отримання імені вибраного файлу:

```
JFileChooser dialog = new JFileChooser();  
dialog.showOpenDialog(null);  
File f = dialog.getSelectedFile();  
String fName = f.getAbsolutePath();
```

Компонент можна налаштовувати, починаючи від того, чи дозволити обирати кілька файлів чи ні, та закінчуючи завданням своїх назв для різних елементів компоненту. Нижче наведені приклади налаштувань об'єкту `dialog`, створеного в попередньому прикладі:

```

// Дозволити вибір папок та файл
dialog.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
// Встановити назву для кнопки
dialog.setApproveButtonText("Выбрать");
// Вибрати назву діалогу
dialog.setDialogTitle("Выберите файл для загрузки");
// Вибрати тип діалогу
dialog.setDialogType(JFileChooser.OPEN_DIALOG);
// Дозволити вибір декількох файлів
dialog.setMultiSelectionEnabled(true);

```

В компоненті можна задавати фільтр, який визначає які файли відображати користувачеві. Нижче наведено приклад фільтра, який забезпечує відображення файлів з розширенням .txt, об'єкта dialog, створеного в попередньому прикладі:

```

dialog.setFileFilter(new FileFilter() {
    public String getDescription() {
        return " файли типу .java ";
    }
    // У цьому методі може бути будь яка перевірка файлу
    public boolean accept(File f) {
        if (f != null) {
            if (f.isDirectory()) {
                return true;
            }
            return f.toString().endsWith(".java");
        }
        return false;
    }
});

```

Для створення фільтра тут використовується об'єкт безіменного класу, що реалізовує інтерфейс `javax.swing.filechooser.FileFilter`. Зверніть увагу на пакет, де знаходиться потрібний клас. Класи з таким же ім'ям є і в інших пакетах.

5.2.2 Вибір файлів за допомогою класу *FileDialog*

Для роботи з файлами і каталогами в додатках Java, можна користуватися і класом `FileDialog`. Цей клас дозволяє відображати на екрані стандартні діалогові панелі, призначені для вибору файлів.

Щоб створити таку панель, потрібно скористатися одним з конструкторів, передбачених у класі `FileDialog`. В конструктори передається від одного до трьох параметрів.

Через перший параметр конструктору передається посилання на об'єкт, який володіє діалоговою панеллю.

Другий параметр дозволяє вказати довільний заголовок діалогової панелі.

Третій параметр може приймати значення `FileDialog.LOAD`, якщо файл буде вибиратися для завантаження, або `FileDialog.SAVE` в разі збереження файлу.

Після створення діалогової панелі її необхідно відобразити методом setVisible (true).

Початковий каталог для пошуку файлу та ім'я файлу можна встановити методами setDirectory (String dir) і setFile (String fileName).

Замість конкретного імені файлу fileName можна написати шаблон, наприклад, *.java, тоді у вікні будуть видні тільки імена файлів, що закінчуються крапкою і словом java.

Після того як користувач вибере файл, додаток може отримати шлях до каталогу та ім'я файлу, скориставшись методами getDirectory і getFile, визначеними в класі FileDialog.

Ось приклад використання класу FileDialog для отримання імені вибраного файлу:

```
FileDialog fileDialog = new FileDialog(this);
fileDialog.setMode(FileDialog.LOAD);
fileDialog.setVisible(true);
String dr = fileDialog.getDirectory();
String fn = fileDialog.getFile();
if (dr == null || fn == null)
    return;
String fName = dr + fn;
```

Позитивна особливість FileDialog, яка відрізняє його від JFileChooser, полягає в тому, що він запам'ятовує шлях, який востаннє використовувався.

Недолік у тому, що він може використовуватися для вибору тільки одного файлу.

5.3 ЗАСОБИ ВВЕДЕННЯ/ВИВЕДЕННЯ

5.3.1 Обмін даними с консоллю

Java визначає потоки для стандартного вводу, виводу і помилок: System.in, System.out і System.err.

Ми вже часто зустрічалися з процесом виведення на консоль, використовуючи конструкцію System.out.println ().

Що стосується потоку стандартне введення System.in, то він вимагає додаткових надбудов. Зручною надбудовою для цього потоку є клас java.util.Scanner. Приклади використання цього класу розглядалися у розділі 3.

5.3.2 Обмін даними з файлами

5.3.2.1 Читання рядків із текстового файлу

Для читання текстового файлу по рядкам можна використовувати об'єкт класу BufferedReader, який забезпечує буферизацію, що збільшує швидкість читання у кілька разів.

```

JFileChooser dialog = new JFileChooser();
dialog.showOpenDialog(null);
File f = dialog.getSelectedFile();
// Читаємо рядки файлу і виводимо їх на консоль
try {
    BufferedReader in = new BufferedReader(new FileReader(f));
    String s;
    while ((s = in.readLine()) != null) {
        System.out.println(s);
    }
    in.close();
} catch (Exception e) {
    e.printStackTrace();
}

```

5.3.2.2 Читання даних примітивного типу із текстового файлу

Об'єкт типу `Scanner` можна використовувати і для введення даних з файлу. Для цього йому при створенні потрібно передати посилання на об'єкт типу `File`. У наведеному прикладі передбачається, що у файлі знаходяться цілі числа, які розділені пробілами або переходами до нового рядка:

```

JFileChooser dialog = new JFileChooser();
dialog.showOpenDialog(null);
File f = dialog.getSelectedFile();
try {
    Scanner sc = new Scanner(f);
    while (sc.hasNextLong()) {
        long aLong = sc.nextLong();
        System.out.println(aLong);
    }
    sc.close();
} catch (Exception e) {
    e.printStackTrace();
}

```

5.3.2.3 Виведення до текстового файлу

Для виведення у текстовий файл можна використовувати об'єкт класу `BufferedWriter`, передавши йому як параметр об'єкт класу `FileWriter`, налаштований на обраний файл.

Нижче наведено приклад, де у текстовий файл виводиться послідовність випадкових чисел:

```

JFileChooser dialog = new JFileChooser();
dialog.showOpenDialog(null);
File f = dialog.getSelectedFile();
// Виведення випадкових чисел у текстовий файл
try {
    BufferedWriter out = new BufferedWriter(new FileWriter(f));
    for (int i = 0; i < 10; i++) {
        out.write(String.valueOf(Math.random()));
        out.newLine();
    }
    out.close();
} catch (Exception e) {
    e.printStackTrace();
}

```

5.4 СЕРІАЛІЗАЦІЯ

Серіалізація (serialization) - це процес запису стану об'єкта у формі байтового потоку. Пізніше об'єкт можна відновити, використовуючи процес десеріалізації (deserialization).

Сама по собі серіалізація об'єктів цікава тому, що з її допомогою можна здійснити легковагове довготривале зберігання (lightweight persistence). Це означає, що час життя об'єкта визначається не тільки часом виконання програми - об'єкт існує і між запусками програми. Можна взяти об'єкт і записати його на диск, а після, при іншому запуску програми, відновити його у первісному вигляді і таким чином отримати ефект «живучості».

Механізм серіалізації об'єктів був доданий у мову для підтримки двох розширених можливостей. Віддалений виклик методів Java (RMI) дозволяє працювати з об'єктами, що знаходяться на інших комп'ютерах. При посилці повідомлень віддаленим об'єктам необхідно транспортувати аргументи і повертаються значення, а для цього використовується серіалізація об'єктів.

Серіалізація об'єктів також необхідна візуальним компонентам JavaBean. Інформація про стан візуальних компонентів звичайно змінюється під час розробки. Цю інформацію про стан необхідно зберегти, а потім, при запуску програми, відновити. Дану задачу вирішує серіалізація об'єктів.

Об'єкт, як структура даних, являє собою орієнтований граф, можливо циклічний, який задає зв'язки між даним об'єктом і об'єктами, що входять до його складу. Дерево є окремим випадком такого графа. При серіалізації об'єкта, розташованого у вершині такого графа, всі пов'язані з ним об'єкти шикуються у рекурсивну послідовність і серіалізуються. Аналогічно у процесі десеріалізації всі ці об'єкти і їх посилання будуть відновлюватися в правильну ієрархічну структуру.

5.4.1 Інтерфейс *Serializable*

Для того щоб об'єкт можна було серіалізувати, його клас і класи об'єктів, що входять до його складу, повинні реалізувати інтерфейс *Serializable*. Сам інтерфейс ніяких методів не визначає, він тільки вказує, що об'єкт може бути серіалізовано. Якщо клас серіалізовано, то всі його підкласи також серіалізовані.

5.4.2 Ключове слово *transient*

При управлінні процесом серіалізації може виникнути ситуація, при якій автоматичне збереження і відновлення деякого поля небажано - наприклад, якщо у відповідному об'єкті зберігається деяка конфіденційна інформація, наприклад пароль. Навіть якщо інформація у об'єкті описана як закрита (*private*), це не рятує її від серіалізації, після якої можна витягти секретні дані з файлу або з перехопленого мережевого пакета.

Щоб заборонити запис деяких полів об'єкта типу *Serializable*, скористайтеся ключовим словом *transient*. Фактично він означає: «Це не потрібно ні зберігати, ні відновлювати».

5.4.3 Запис об'єктів у файл і відновлення об'єктів із файлу

Щоб записати об'єкт у файл, потрібно створити вихідний потік *FileOutputStream*, який потрібно вкласти у об'єкт *ObjectOutputStream*. Виклик методу *writeObject()* для цього об'єкта здійснює серіалізацію об'єкта, і сформований потік байтів пересилається у вихідний потік даних *FileOutputStream*. Нижче наводиться метод, що використовує ці об'єкти для серіалізації :

```
protected void storeObjectToFile(Object o, File f){
    try {
        FileOutputStream fileStream = new FileOutputStream(f);
        ObjectOutputStream out = new ObjectOutputStream(fileStream);
        out.writeObject(o); // серіалізуємо об'єкт
        out.close(); // закриваємо потік виводу
    } catch (IOException e1) {
        e1.printStackTrace();
    }
}
```

Для відновлення об'єкта необхідно надбудувати об'єкт *ObjectInputStream* для вхідного потоку *FileInputStream*, а потім викликати метод *readObject()*. Цей метод повертає посилання на об'єкт типу *Object*, тому після виклику методу слід провести спадаюче перетворення для отримання об'єкта потрібного типу.

Процедура десеріалізації у чомусь подібна процедурі серіалізації і очевидна з тексту, що наводиться нижче.

```
protected Object restoreObjectFromFile(File f){
    Object o = null;
    try {
        // створюємо потоки вводу для десеріалізації об'єкта
```



```
FileInputStream fStream = new FileInputStream(f);
ObjectInputStream in = new ObjectInputStream(fStream);
o = in.readObject(); // десеріалізуємо об'єкт
in.close(); // закриваємо потік
} catch (Exception e1) {
    e1.printStackTrace();
}
return o;
}
```

Для відновлення об'єкта необхідна наявність класів відповідних відновлюваним об'єктам і версії класів мають бути тими ж, як і при збереженні. Якщо між збереженням і відновленням об'єкта якийсь клас змінився, при відновленні виникне виняткова ситуація.

5.5 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Діалоги вибору файлів.
2. Класи, які забезпечують обмін інформацією з текстовими файлами.
3. Серіалізація і десеріалізація об'єктів.

6.1 ПОНЯТТЯ ІТЕРАТОРА

Ітератор - це об'єкт, який дозволяє послідовно обробляти елементи деякої групи об'єктів. Основна перевага ітератора полягає в тому, що він реалізує єдиний інтерфейс обробки групи об'єктів, незалежно від внутрішньої організації цієї групи. Наприклад, для обробки і масиву, і дерева, і зв'язного списку можна використовувати одні й ті ж самі методи, якщо для перелічених структур даних існує стандартний ітератор.

Доступ до ітераторів отримують зазвичай за допомогою методу екземпляра класу, що містить групу об'єктів. Очевидно, що реалізація ітераторів для різних структур даних різна, але інтерфейс однаковий.

Таким чином, ітератори - це ще один із способів реалізації поліморфізму.

Слід мати на увазі, що ітератор - це об'єкт одноразової дії. Він забезпечує прохід по групі елементів від початку до кінця (або у зворотному напрямку) і на цьому його можливості вичерпуються. Якщо виникає потреба повторного проходу по групі елементів, ітератор слід створити заново.

В даний час усі класи для різноманітних структур даних, що зберігають набори об'єктів, підтримують поняття ітератора. Програміст може створити стандартний ітератор і для своєї структури даних, реалізувавши для неї метод, який повертає об'єкт відповідного типу.

Ітератори у мові Java визначаються інтерфейсами, і існує кілька різновидів цих інтерфейсів.

6.2 ІНТЕРФЕЙСИ ІТЕРАТОРІВ У JAVA

6.2.1 Інтерфейс Enumeration

Інтерфейс Enumeration вважається застарілим. Його було реалізовано у деяких класах перших версій Java (наприклад, класи Vector, Stack, Dictionary, Hashtable) і він використовується й досі. Для виклику ітератора використовується метод `elements()`.

У таблиці 6.1 наведено опис методів цього ітератора.

Таблиця 6.1 Методи інтерфейсу Enumeration	
Methods	Description
boolean hasMoreElement()	Returns true if there are more elements. Otherwise, returns false.
Object nextElement()	Returns the next element. Throws NoSuchElementException if there is not a next element.

6.2.2 Інтерфейс Iterator

Інтерфейс Iterator можна вважати новою версією інтерфейсу Enumeration. Він з'явився у версії Java 1.2. Набір методів розширений за рахунок методу видалення поточного елемента з набору. Крім того назви методів стали більш короткими.

У таблиці 6.2 наведено опис методів цього ітератора.

Таблиця 6.2 Методи інтерфейсу Iterator

Methods	Description
boolean hasNext()	Returns true if there are more elements. Otherwise, returns false.
Object next()	Returns the next element. Throws NoSuchElementException if there is not a next element.
void remove()	Removes the current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next().

6.2.3 Інтерфейс ListIterator

Інтерфейс ListIterator має більш багатий набір методів, ніж Iterator, але його застосування обмежене наборами типу List (індексні послідовності). Для виклику цього ітератора у колекціях типу List використовується метод listIterator().

У таблиці 6.3 наведений опис методів цього ітератора.

Таблиця 6.3 Методи інтерфейсу ListIterator

Methods	Description
void add(Object obj)	Inserts obj into the list in front of the element that will be returned by the next call to next().
boolean hasNext()	Returns true if there is a next element. Otherwise, returns false.
boolean hasPrevious()	Returns true if there is a previous element. Otherwise, returns false.
Object next()	Returns the next element. A NoSuchElementException is thrown if there is not a next element.
int nextIndex()	Returns the index of the next element. If there is not a next element, returns the size of the list.
Object previous()	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
int previousIndex()	Returns the index of the previous element. If there is not a previous element, returns -1.

Продовження таблиці 6.3

<code>void remove()</code>	Removes the current element from the list. An <code>IllegalStateException</code> is thrown if <code>remove()</code> is called before <code>next()</code> or <code>previous()</code> is invoked.
<code>void set(Object obj)</code>	Assigns <code>obj</code> to the current element. This is the element last returned by a call to either <code>next()</code> or <code>previous()</code> .

6.2.4 Інтерфейс *Iterable*

Цей інтерфейс передбачає реалізацію тільки одного методу - `iterator()`, який повертає об'єкт типу `Iterator`. Таким чином, інтерфейс `Iterable` зводиться до інтерфейсу `Iterator`, нічого нового. Але суть цього інтерфейсу в тому, що для груп об'єктів, що реалізують цей інтерфейс, можна застосувати цикл `for-each`.

Цикл `for-each` розроблено для циклічної обробки груп об'єктів, у суворій послідовності, від початку до кінця, без використання індексу. Завдяки зручності використання цей вид циклу швидко набув широкого поширення. В Java цикл `for-each` з'явився, починаючи з версії Java5.

Особливість реалізації циклу `for-each` у Java полягає в тому, що вона реалізує функціональні можливості `for-each`, вдосконалюючи оператор циклу `for`. Перевага такого підходу полягає в тому, що не потрібно нового ключового слова і не порушується працездатність розробленого раніше коду. Цикл `for` у стилі `for-each` (тобто наділеного функціональними можливостями циклу `for-each`) іноді називають поліпшеним циклом `for` (`enhanced for loop`).

Загальна форма запису циклу `for` у стилі `for-each` приведена у наступному рядку:

```
for(type itr-var : iterableObj) statement-block
```

Тут `type` - тип, а `itr-var` - ім'я змінної циклу (`iteration variable`), яка отримуватиме значення елементів, що містяться у `iterableObj`, послідовно один за іншим від початку до кінця. Об'єкт, на який посилається змінна `iterableObj`, повинен бути масивом або об'єктом, що реалізує інтерфейс `Iterable`. У будь-якому випадку тип `type` повинен збігатися (або бути сумісним) з типом елементів, із яких складається об'єкт `iterableObj`.

У кожному проході тіла циклу з об'єкта `iterableObj` витягується черговий елемент і запам'ятовується у змінній `itr-var`. Цикл виконується поки не будуть витягнуті всі елементи.

6.2.5 Розширення інтерфейсу *Iterable*

У восьмій версії Java інтерфейс `Iterable` було розширено двома дефолтними методами. Дефолтні методи інтерфейсів, на відміну від звичайних, мають реалізацію. Фактично, це робить інтерфейс абстрактним класом. На цей крок розробники Java наважились піти заради того, щоб надати можливість використовувати нововведення восьмої версії і зберегти сумісність з попередніми версіями.

На рисунку 6.1 показано дефолтні методи інтерфейсу `Iterable`. Слід

зауважити, що ці методи не можна застосовувати до масивів.

```
Iterable<T>  
  A iterator() : Iterator<T>  
  D forEach(Consumer<? super T>) : void  
  D spliterator() : Spliterator<T>
```

Рисунок 6.1 – Методи інтерфейсу Iterable

6.3 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Призначення ітераторів.
2. Інтерфейс Enumeration.
3. Інтерфейс ListIterator.
4. Інтерфейс Iterable.

7 ПЕРЕДАЧА СПЕЦИФІЧНИХ ОПЕРАЦІЙ КОРИСТУВАЧА ДО УЗАГАЛЬНЕНИХ МЕТОДІВ

7.1 УЗАГАЛЬНЕНІ МЕТОДИ І СПЕЦИФІЧНІ ОПЕРАЦІЇ КОРИСТУВАЧА

У попередніх розділах вже йшла мова про узагальнені методи. Зокрема, створити узагальнений метод вдавалося за рахунок звертання у методі абстрактного класу до абстрактного поліморфного методу, різні реалізації якого знаходилися у класах спадкоємцях. Згадайте приклад з четвертого розділу, де у класі `Product` було створено узагальнений метод `weight`, що використовував абстрактний метод `volume`, який було реалізовано у класах `Timber` та `Cylinder`.

Але можливі і інші шляхи створення узагальненого коду.

Уявімо собі що є якийсь додаток, що вміє відображати графіки функцій. Зрозуміло, що для того, щоб отримати графік конкретної функції треба якимось чином повідомити додатку про цю функцію.

Якщо йти вищезгаданим шляхом спадкування, то треба клас додатку зробити абстрактним, визначити у ньому абстрактний метод обчислення значень функції і викликати його в узагальненому методі відображення функції. А для відображень конкретних функцій створювати класи спадкоємці, з одним методом, де будуть реалізовуватися потрібні функції.

Інший шлях полягає у тому, що у додатку створюється метод, до якого передається потрібна функція в якості параметру. Цей метод теж буде узагальненим, бо буде отримувати різні функції, і відповідно до цього відображати різні графіки.

Оця функція (або метод, чи щось інше), що ми передаємо до узагальненого методу в якості параметру, і є специфічною операцією користувача додатку.

Тож виникає питання, як у Java-метод передати якийсь код в якості параметру?

Таке завдання в кожній мові програмування має свій специфічний шлях вирішення.

У мові C для вирішення такого завдання використовують покажчики на функцію.

В `Smalltalk` можна передати як параметр код обробки у вигляді об'єкта класу `Block`.

В мові `Groovy`, що сумісна з мовою `Java`, так само можна в якості параметру передавати блок коду, що є об'єктом класу `Code`.

В `ObjectPascal` така проблема вирішується за допомогою процедурного типу даних, і як параметр до узагальненого методу передається ім'я процедури або функції відповідного процедурного типу.

В `Java`, на відміну від цих мов, немає таких спеціальних типів. Немає покажчиків на функції та й самих функцій, немає процедурних типів і класів подібних класу `Block` в `Smalltalk`, або `Code` в `Groovy`.

Для вирішення цієї проблеми в `Java` можна скористатися кількома

підходами, але в будь-якому випадку ці підходи потребують наступних етапів:

1. Створення програмного інтерфейсу, що визначає сигнатуру методу, який має бути переданий через параметр узагальненого методу.
2. Створення класу, що реалізує створений програмний інтерфейс.
3. Створення об'єкту створеного класу.
4. Передача створеного об'єкта як параметра в узагальнений метод.
5. В узагальненому методі через цей об'єкт викликати метод, визначений в інтерфейсі і за допомогою нього виконувати необхідну обробку.

На практиці можливі різні реалізації цієї послідовності етапів, але в будь-якому випадку перш за все повинен бути створений інтерфейс, що визначає сигнатуру переданого методу.

Розглянемо деякі з таких підходів припускаючи, що інтерфейс вже створений.

7.1.1 Шляхи створення класів, що реалізують інтерфейс

Використовується декілька підходів до створення класу, головне завдання якого полягає у реалізації якогось інтерфейсу.

Перший підхід полягає в тому, що створюється звичайний клас, який реалізує даний інтерфейс. Переваги такого підходу в тому, що клас реалізації операцій користувача є зовнішнім і дозволяє описувати методи будь-якої складності. Такий підхід доцільно застосовувати, якщо реалізація складна і вимагає частого використання декількома різними класами.

Другий підхід полягає в створенні внутрішнього класу. Переваги такого підходу в тому, що клас реалізації операції користувача розташовується безпосередньо в класі, де викликається узагальнений метод, що дозволяє при реалізації використовувати приватні поля.

Третій підхід передбачає створення безіменного (анонімного) внутрішнього класу, що реалізовує призначену для користувача операцію. Переваги і недоліки такого підходу ті ж що і в попередньому випадку, різниця полягає лише в більш компактному способі оголошення класу реалізації. Безіменний внутрішній клас створюється безпосередньо при створенні об'єкта за допомогою операції `new`. Як ім'я конструктора при цьому використовується ім'я інтерфейсу, а після цього в фігурних дужках записується реалізація класу.

7.1.2 Шляхи створення об'єкта, що використовується для передачі операції користувача

Перший стандартний підхід полягає у визначенні імені об'єкта і потім його ініціалізації через виклик конструктора створеного класу.

Другий подібний до першого, але відрізняється використанням безіменного класу.

Третій спосіб полягає у використанні анонімного об'єкта. У цьому випадку об'єкт створюється безпосередньо під час виклику методу без присвоювання йому імені, що робить конструкцію досить громіздкою. Таке рішення підходить для випадку, коли метод користувача складається з

невеликої кількості простих операторів або викликів інших методів, інакше код буде важко читати. Проте цей підхід досить часто використовується. Подібним чином створюються оброблювачі подій, пов'язаних з компонентами (наприклад, натискання кнопки):

```
// Додаємо до кнопки слухача події (це і є обробник)
btnButton1. addActionListener(
    // Створюємо безіменний об'єкт безіменного класу,
    // що реалізовує інтерфейс ActionListener
    new ActionListener(){
        // Реалізація безіменного класу
        public void actionPerformed(ActionEvent arg0) {
            // Операції з обробки події натискання кнопки
            System.out.println ("actionPerformed");
        }
    });
```

Тут:

`addActionListener ()` - це метод кнопки, через який реєструється обробник події. Для зберігання таких обробників в класі кнопки є колекція налаштована на об'єкти типу `ActionListener`.

`ActionListener` - це інтерфейс, який передбачає реалізацію методу `actionPerformed ()`.

Четвертий підхід заснований на використанні визначених користувачем констант-об'єктів для безіменних внутрішніх класів. Фактично, відмінність полягає лише в способі отримання посилання на реалізацію користувальницької операції. Перевагою даного підходу є можливість реалізації часто використовуваних операцій.

7.1.3 Функціональні інтерфейси та лямбда функції

У Java8 з'явилося поняття «функціональний інтерфейс». Так почали називати інтерфейси, у яких визначено тільки один абстрактний метод. Така назва пов'язана з тим, що у такому випадку інтерфейс ніби то визначає якусь функцію.

Деякі різновиди таких інтерфейсів були стандартизовані, що звільнило програміста від необхідності створювати такі інтерфейси самому.

Функціональні інтерфейси визначені у пакеті `java.util.function`, зокрема це інтерфейси `Function`, `Consumer`, `Predicate`.

Визначена також анотація `@FunctionalInterface`, яка повідомляє компілятору про необхідність контролю кількості абстрактних методів

Внаслідок того, що функціональний інтерфейс має тільки один метод, що підлягає реалізації, назва цього методу однозначно пов'язана з назвою інтерфейсу. Більш того, коли у якийсь метод, у якості параметру треба передати об'єкт типу функціонального інтерфейсу, то цей тип відомий компілятору із сигнатури методу, куди передається параметр.

Ці дві особливості дозволили спростити створення анонімних об'єктів

анонімних класів, що реалізують функціональні інтерфейси.

Анонімний об'єкт анонімного класу, що реалізує функціональний інтерфейс можна створювати у вигляді лямбда функції.

Нагадаємо, що лямбда функція складається з трьох частин.

Перша частина – це круглі дужки, у яких наводиться перелік параметрів функції. Типи не вказуються, бо вони визначені у інтерфейсі. Імена параметрів довільні.

Друга частина – це комбінація з двох символів `->`, що є ознакою лямбда.

Третя частина являє собою тіло методу функціонального інтерфейсу.

Якщо функція складається з декількох операторів, то потрібні фігурні дужки і оператор `return`. Лямбда функція може і не використовувати параметри, але дужки мають бути.

Лямбда функція спрощує створення анонімного об'єкту за рахунок того, що не використовує ні назву методу, ні назву інтерфейсу з конструкцією `new`. Найбільший ефект використання лямбда функції дає для реалізації коротких методів.

Так, наприклад, обробка події натискання на кнопку зазвичай реалізується створенням анонімного об'єкту анонімного класу у такий спосіб:

```
btnNewButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        onClick();  
    }  
});
```

А використовуючи лямбда функції, те саме можна написати простіше:

```
btnNewButton.addActionListener((e)->onClick());
```

7.1.4 Ще один приклад з використанням функціональних інтерфейсів

Припустимо, що є клас `D` з приватним полем `arr`, яке містить масив цілих чисел, та методом `toString` для перетворення масиву у рядок символів:

```
public class D {  
  
    int[] arr = {1,2,3,4,5,6,7,8,9};  
  
    @Override  
    public String toString() {  
        return «D [arr=» + Arrays.toString(arr) + «]»;  
    }  
}
```

Потрібно у цьому класі створити узагальнений метод, який би змінював заданим чином елементи масиву, що задовольняють деякій умові.

Такий метод можна створити, використовуючи функціональні інтерфейси `Function` та `Predicate`.

`Function` дозволить створити код для зміни елементу масиву.

`Predicate` дозволить сформулювати умову, за якої елемент масиву буде змінюватися.

Узагальнений метод, що буде приймати параметри цих типів з назвами `maker` та `condition` буде виглядати так:

```
public void makeIf(Function<Integer,Integer> maker,
                  Predicate<Integer> condition){
    for (int i = 0; i < arr.length; i++) {
        if(condition.test(arr[i]))
            arr[i] = maker.apply(arr[i]);
    }
}
```

Як бачимо, інтерфейс `Function` потребує визначення типів аргументу та функції. У нас і аргумент і функція мають тип `int`, бо це старий і новий елементи масиву. Але функціональні інтерфейси працюють тільки з посилавними типами, тому вказуємо тип `Integer`, а Java сама виконає обгортку(автобоксинг). До методу `apply`, що визначений у інтерфейсі `Function` передаємо аргумент функції, а результатом обчислення функції буде значення, що повертає метод `apply`.

Інтерфейс `Predicate` потребує визначення типу одного параметру. У нашому випадку це тип елемента масиву, тому вказуємо `Integer`. Метод `test` цього інтерфейсу перевіряє, чи відповідає параметр методу заданій умові.

Для того, щоб протестувати роботу узагальненого методу створимо клас з методом `main`:

```
public static void main(String[] args) {
    D d = new D();
    d.makeIf((x)-> x*x, (x)->x%2 == 0);
    System.out.println(d);
}
```

Тут створюється об'єкт раніше визначеного класу `D` і для нього викликається метод `makeIf`, що було створено вище. Першим параметром методу є лямбда-функція, що забезпечує піднесення елемента масиву до квадрату. Другим параметром методу є лямбда-функція, що перевіряє, чи є елемент масиву парним числом.

7.2 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Що таке специфічні операції користувача.
2. Як вирішується проблема передачі специфічних операцій користувача до методів у різних мовах програмування.
3. Що треба зробити в Java-програмі, щоб передати специфічну операцію користувача до методу.
4. Що таке внутрішній клас?
5. Що таке анонімний клас і анонімний об'єкт?
6. Що таке функціональний інтерфейс?
7. Що таке лямбда-функція?
8. Напишіть виклик методу `makeIf` для перетворення усіх парних чисел у непарні, шляхом ділення їх навпіл, а нуль замінити одиничкою.

8.1 ЗАГАЛЬНІ ВІДОМОСТІ ПРО КОЛЕКЦІЇ

Колекція - це група певним чином організованих об'єктів. Поняття колекції вперше з'явилося в мові Smalltalk. По суті це поняття об'єднало всі існуючі структури даних для зберігання наборів об'єктів (масиви, списки, дерева). Особливість колекцій в тому, що незалежно від внутрішньої організації даних вони забезпечують стандартні інтерфейси доступу та обробки даних. Крім того, колекції звільняють користувача від необхідності вирішувати проблему виділення пам'яті і автоматично коригують обсяг необхідної пам'яті, а інтерфейси пропонують користувачеві багатий вибір операцій з обробки даних.

У мові C ++ поняття колекції було замінено поняттям контейнера, крім того, внаслідок особливостей мови, для реалізації контейнерних класів була розроблена спеціальна бібліотека STL (Standard template library), робота з якою вимагає певних навичок.

В Java колекції реалізовані практично так само, як і в Smalltalk, але поняття колекції було дещо звужене. В Java колекцією вважається набір даних, доступ до яких здійснюється або за індексом або за посиланням на елемент набору. Ці колекції об'єднані інтерфейсом Collection.

8.2 УЗАГАЛЬНЕННЯ В КОЛЕКЦІЯХ

Ще одна особливість колекцій - це можливість контролю типу даних. У перших версіях Java всі дані колекцій визначалися типом Object, який накривав всі можливі типи даних. Це дозволяло зберігати в колекціях різнотипні дані, але знижувало рівень безпеки і при виборі елемента колекції його доводилося спеціально типізувати. Тому в більш пізніх версіях мови було додано поняття узагальнення (generic). Узагальнення дозволяють налаштувати колекції на конкретний тип об'єкта, що дозволяє здійснювати контроль при занесенні даних в колекцію і при виборі елемента колекції не потрібно його спеціальної типізації. Для настройки колекції на заданий тип даних, його слід вказати в кутових дужках після імені типу колекції.

Розглянемо кілька прикладів:

```
List<Node> sonList = new Vector<Node>();
```

Тут визначається список sonList інтерфейсного типу List, який ініціалізується об'єктом класу Vector. Узагальнення <Node> вказує на те, що список призначений для зберігання об'єктів типу Node.

Зверніть увагу на те, що узагальнення вказано як при визначенні типу, так і при виклику конструктора. Але в останніх версіях Java дублювати типи даних колекції вже не потрібно:

```
Map<Object, Node> sonMap = new Hashtable<>();
```

У наведеному прикладі визначається карта відображення `sonMap`, яка ініціалізується об'єктом класу `HashTable`. Узагальнення `<Object, Node>` вказує на те, що ключами в карті повинні бути об'єкти типу `Object`, а значеннями - об'єкти типу `Node`.

```
Collection<PriorityQueue<Object>> c;
```

Тут визначається колекція для зберігання черг типу `PriorityQueue`, кожна з яких може зберігати об'єкти будь-якого типу (`Object`).

```
public interface Collection<E> extends Iterable<E>
```

У цьому прикладі визначається інтерфейс `Collection`, що успадковує інтерфейс `Iterable`. Передбачається, що інтерфейси повинні працювати з об'єктами деякого типу `E`, який буде визначений при реалізації інтерфейсу.

```
boolean addAll(Collection<? extends E> c);
```

У цьому прикладі визначається заголовок методу `addAll`, параметром якого може бути колекція типу `Collection`, яка містить елементи будь-яких типів, що успадковують тип `E`.

```
Comparator<? super E> comparator;
```

Тут визначається компаратор, який може порівнювати об'єкти типів `E` або батьківських для `E`.

Детальніше про узагальнення можна послухати в лекціях по посиланнях:

https://www.youtube.com/watch?list=SPB0276A0A62BDEF06&v=87pm79sP Svc&feature=player_detailpage

http://www.youtube.com/watch?feature=player_detailpage&v=MniNZsyjH9E

Що стосується `C++`, то внаслідок відсутності в цій мові єдиного базового класу для всіх типів об'єктів, контейнер завжди налаштовується на певний тип даних.

8.3 ІНТЕРФЕЙС COLLECTION

8.3.1 Загальна характеристика інтерфейсу Collection

Цей інтерфейс є базовим для групи інших інтерфейсів і визначає найбільш загальні методи роботи з колекціями.

Інтерфейс `Collection` розширюють інтерфейси `List`, `Set` і `Queue`. Інтерфейс `Queue` розширюється інтерфейсом `Deque`. Інтерфейс `Set` розширюється інтерфейсами `SortedSet` і `NavigableSet`. На рисунку 8.1 показана діаграма інтерфейсів і класів пакета `java.util`. Імена інтерфейсів показані в рамочках, а імена класів, що реалізують відповідні інтерфейси зображені без рамок.

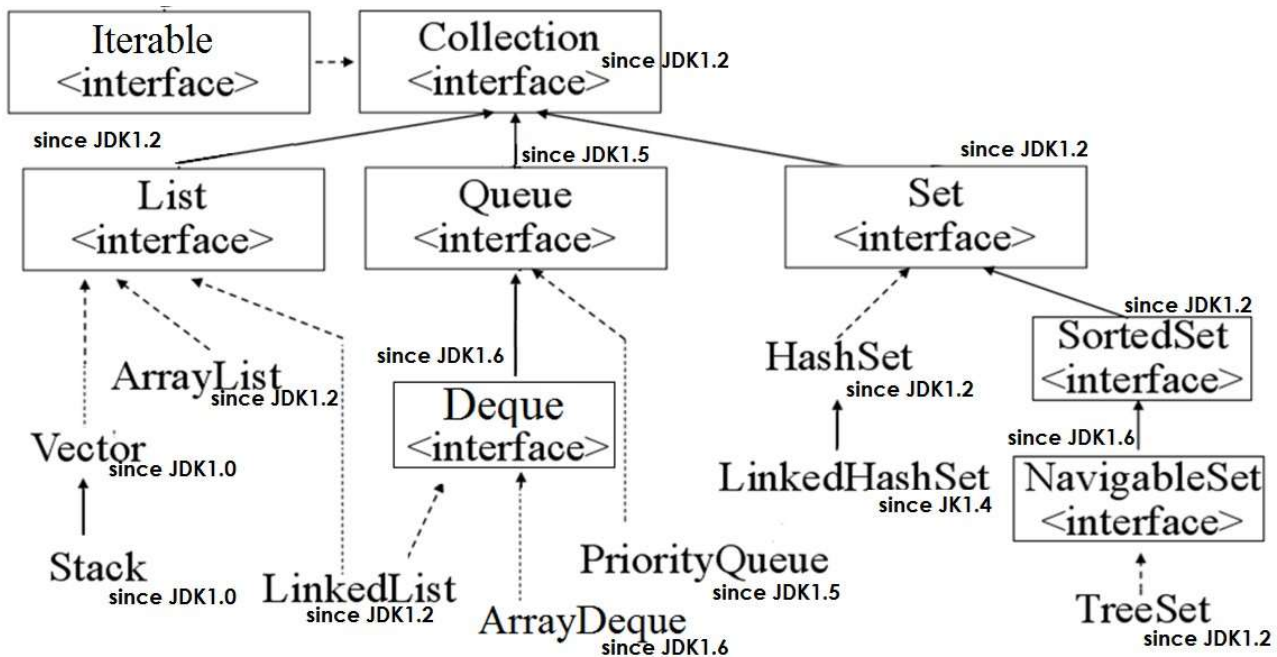


Рисунок 8.1 – Діаграма класів колекцій пакету java.util

Слід зауважити, що на діаграмі 8.1 не показані абстрактні класи колекцій (AbstractCollection, AbstractList, AbstractSet і ін.), які забезпечують часткову реалізацію інтерфейсів.

Конкретні класи колекцій розширюють абстрактні класи та забезпечують повну реалізацію інтерфейсів колекцій. Конкретні класи можуть використовуватися для безпосереднього створення екземплярів цих класів.

Класи, які реалізують інтерфейс List призначені для обробки списків.

Класи, які реалізують інтерфейс Set призначені для обробки множин.

Класи, які реалізують інтерфейс NavigableSet призначені для обробки сортованих множин.

Класи, які реалізують інтерфейс Queue призначені для обробки черг.

На додаток до цих інтерфейсів колекції використовують інтерфейси Comparator, Iterator і ListIterator.

8.3.2 Методи інтерфейсу Collection

На рисунках 8.2 – 8.4 наведено основні методи інтерфейсу Collection. Ці методи можуть бути застосовні до об'єктів всіх класів колекцій. Для зручності методи розбиті на три групи. Перша група складається з методів, що реалізують типові операції з колекціями, а саме – додавання, вилучення одного або усіх елементів, перевірка наявності елемента у колекції, а також визначення розміру або пустоти колекції. Призначення кожного методу зрозуміло з його назви. Останній метод свідчить про те, що для будь якої колекції можна викликати ітератор. Нагадаємо також, що інтерфейс Collection успадковує інтерфейс Iterable, а це означає, що будь-яку колекцію можна обробляти у циклі типу `fore each`.

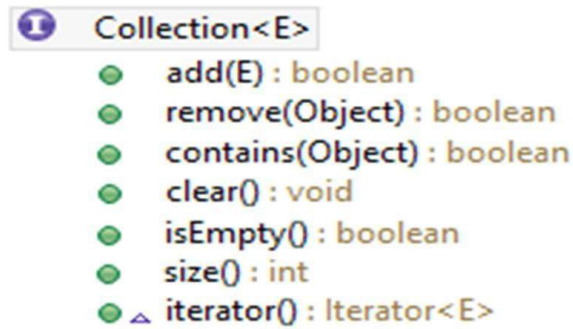


Рисунок 8.2 – Основні методи інтерфейсу Collection

Нижче наведено приклади використання цих методів з результатами.

```

//початковий вміст coll = [ 1, 5, 2, 0 ]
coll.add(3);           // [ 1, 5, 2, 0, 3 ]
coll.remove(1);       // [ 5, 2, 0, 3 ]
coll.contains(0);     // true
coll.size();          // 4
coll.clear();         // [ ]

```

Далі наведено приклади використання ітераторів для колекцій.

Цикл for:

```

for (Iterator itr = coll.iterator(); itr.hasNext();) {
    int x = (int) itr.next(); . . .
}

```

Цикл while:

```

Iterator<Integer> itr = coll.iterator();
while (itr.hasNext()) {
    int x = (int) itr.next(); . . .
}

```

Цикл for each:

```

for (int x : coll){System.out.println(x);}

```

Друга група методів, що наведена на рисунку 8.3 забезпечує групові операції над елементами колекцій, фактично це реалізація стандартних операцій з множинами – об'єднання, знаходження різниці, перетин, перевірка входження.

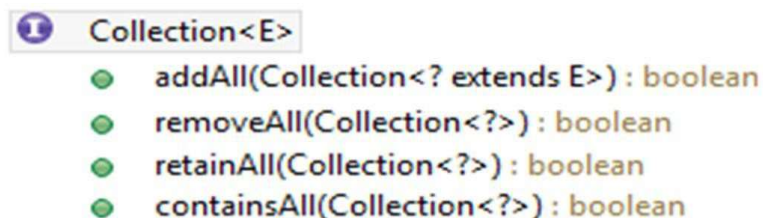


Рисунок 8.3 – Методи інтерфейсу Collection, що визначають операції над множинами

Нижче наведено приклади використання методів для операцій з множинами:

```
//початковий вміст: c1 = [1, 5, 2, 0, 0]; c2 = [2, 3, 0, 2];
c1.removeAll(c2); // c1 = [1, 5]
c1.addAll(c2); // c1 = [1, 5, 2, 3, 0, 2]
c1.retainAll(c2); // c1 = [2, 3, 0, 2]
c1.containsAll(c2); // true
```

Решта методів наведена на рисунку 8.4. Методи `equals` та `hashCode` перевизначають відповідні методи класу `Object`. Це дозволяє порівнювати колекції за змістом. Останні два методи дозволяють перетворити колекцію на масив типу `Object` або на масив заданого типу. В останньому випадку масив попередньо доводиться створювати.

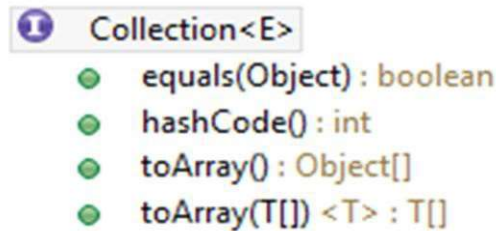


Рисунок 8.4 – Інші методи інтерфейсу `Collection`

У восьмій версії Java до інтерфейсу `Collection` було додано дефолтні методи, рисунок 8.5. Дефолтні методи – це методи інтерфейсу, що мають реалізацію. Це дозволило використовувати для обробки колекцій лямбда функції, стрими та паралельну обробку.

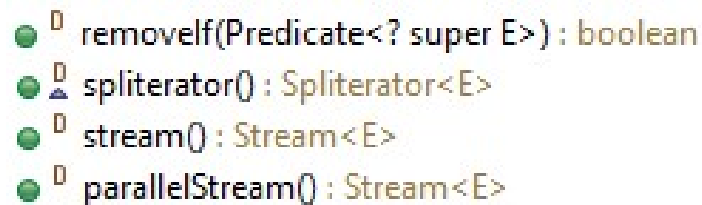


Рисунок 8.5 – Дефолтні методи інтерфейсу `Collection`

Окрім того дефолтні методи, що дозволяють реалізувати цикл `forEach` з використанням лямбда функції, з'явилися в інтерфейсах `Iterable` та `Iterator`.

Нижче наведено приклади використання цих методів.

Приклад використання `forEach`:

```
coll.forEach((x) -> System.out.println(x));
```

Приклад конвеєрної обробки колекції:

```
double d = coll.stream().parallel()
    .sorted()
    .filter((x) -> x % 2 == 0)
    .mapToDouble((x) -> x / 2.0)
    .average() //OptionalDouble
    .getAsDouble();
```


Приклад видалення елементів за умовою:

```
//початковий вміст coll = [3, 5, 2, 0]
coll.removeIf((x)->x>2); // [2, 0]
coll.forEach(System.out::println);
```

8.3.3 Використання ітератора *Spliterator* для паралельної обробки колекції

Дефолтний метод `splitIterator`, що з'явився в інтерфейсі `Collection`, повертає об'єкт типу `Spliterator`, що дозволяє реалізувати паралельну обробку колекцій на багатоядерних пристроях.

Методи інтерфейсу `Spliterator`

- ^A `trySplit(): Spliterator<T>`
- ^D `forEachRemaining(Consumer<? super T>): void`
- ^A `tryAdvance(Consumer<? super T>): boolean`

Нижче наведено приклад створення і використання такого ітератора.

```
// Створюємо колекцію з 8 елементів
Collection<Integer> coll = new Vector<>();
for (int i = 0; i < 8; i++)
    coll.add(i);

// Створюємо масив розщеплених ітераторів
Spliterator<Integer>[] arSplt;
arSplt = new Spliterator[4];

// Отримуємо початковий ітератор для для всієї колекції
// arSplit[0] буде налаштований на елементи 0/1/2/3/4/5/6/7/
arSplit[0] = coll.spliterator();

//Розщеплюємо arSplit[0] навпіл, після чого
//arSplit[1] буде налаштований на елементи 0/1/2/3/
//arSplit[0] буде налаштований на елементи 4/5/6/7/
arSplit[1] = arSplit[0].trySplit();

//Ще раз розщеплюємо arSplit[0] навпіл, після чого
//arSplit[2] буде налаштований на елементи 4/5/
//arSplit[0] буде налаштований на елементи 6/7/
arSplit[2] = arSplit[0].trySplit();

//Розщеплюємо arSplit[1] (0/1/2/3/) навпіл, після чого
//arSplit[1] буде налаштований на елементи 2/3/
//arSplit[3] буде налаштований на елементи 0/1/
arSplit[3] = arSplit[1].trySplit();
```



```

// Створюємо пул потоків
// для паралельного виконання ітераторів
ExecutorService pool =
    Executors.newWorkStealingPool();

// Передаємо до пулу завдання на обробку
// ( клас Task1 наведено далі)
for (int i = 0; i < 4; i++) {
    pool.submit(new Task1(arSplit[i]));
}

//Завдання на обробку розщепленої колекції, варіант 1
private class Task1 extends Thread {
    Spliterator<Integer> siter;
    public Task(Spliterator<Integer> siter) {
        this.siter = siter;
    }

    public void run() {
        siter.forEachRemaining( (x) -> { . . . } );
    }
}

//Завдання на обробку розщепленої колекції, варіант 2
private class Task2 implements Runnable {
    Spliterator<Integer> siter;
    public Task(Spliterator<Integer> siter) {
        this.siter = siter;
    }

    public void run() {
        while(siter.tryAdvance( (x) -> { . . . } ));
    }
}

```

8.4 ІНТЕРФЕЙС LIST

8.4.1 Методи інтерфейсу List

Особливість цього інтерфейсу полягає в тому, що для кожного елемента колекції існує поняття попередній та наступний елементи. Окрім того кожен елемент має свій індекс, що визначає місце елемента у колекції і колекція надає доступ до елементів через індекс. Перший елемент має індекс 0.

Нижче, на рисунку 8.6 наведені методи цього інтерфейсу, що розширюють інтерфейс Collections.

Метод `get` повертає елемент із заданим індексом, метод `set` встановлює нове значення елемента за заданим індексом.

```

public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    boolean addAll(int index, Collection<? extends E> c);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
}

```

Рисунок 8.6 – Основні методи інтерфейсу List

Метод `add` вставляє елемент у задану позицію, і робить зсув решти елементів праворуч.

Метод `remove` вилучає елемент по індексу і зсуває усі наступні елементи ліворуч на 1 позицію.

Метод `addAll` вставляє групу елементів починаючи із заданої позиції, зсуваючи решту елементів праворуч.

Метод `indexOf` повертає індекс першого входження елемента до колекції.

Метод `lastIndexOf` повертає індекс останнього входження елемента.

Метод `listIterator` повертає ітератор типу `ListIterator`.

8.4.2 Класи, що реалізують інтерфейс List

Як впливає з діаграми класів 8.1, інтерфейс `List` реалізують такі класи пакету `java.util`:

- `java.util.Vector`
- `java.util.Stack`
- `java.util.ArrayList`
- `java.util.LinkedList`

8.4.2.1 Класи `Vector` та `ArrayList`

У цих класах для зберігання елементів використовується масив. Якщо виділений обсяг пам'яті для масиву вичерпується, розмір масиву автоматично коригується. Коригування розміру масиву призводить до уповільнення методу додавання. Але вибірка елементів за індексом дуже швидка. Вставка і видалення уповільнені, тому що доводиться зрушувати елементи масиву.

Об'єкти класу `ArrayList` працюють швидше, проте методи класу `Vector` синхронізовані і забезпечують безпечну роботу в багатопоточних додатках.

8.4.2.2 Клас `Stack`

Клас розширює клас `Vector` і забезпечує додатково реалізацію методів роботи з колекцією як зі стеком. Нижче, на рисунку 8.7 наведено методи цього класу, що розширюють клас `Vector`.

```
public boolean empty() { };
public synchronized E peek() { };
public synchronized E pop() { };
public E push(E object) { };
public synchronized int search(Object o) { };
```

Рисунок 8.7 – Основні методи класу Stack

Метод `empty` повертає `true`, якщо стек пустий.

Метод `peek` повертає посилання на елемент, що знаходиться у вершині стеку, але не вилучає його.

Метод `pop` повертає посилання на елемент, що знаходиться у вершині стеку, і вилучає його із стеку.

Метод `push` додає елемент у вершину стеку.

Метод `search` повертає порядковий номер заданого елемента від вершини стеку. Елемент на вершині має номер 1.

8.4.2.3 Клас LinkedList

Цей клас забезпечує зберігання елементів у зв'язному списку. Такий спосіб зберігання не вимагає надлишковою пам'яті, прискорюються операції вставки і видалення, але доступ по індексу уповільнений, оскільки доводиться перебирати елементи від початку до заданого індексу.

Крім інтерфейсу `List`, клас реалізує інтерфейс `Deque`, і може досить ефективно використовуватися для створення черг, тому що в класі передбачено зберігання посилань на початок і кінець списку, що дозволяє швидко додавати елементи в кінець черги і видаляти їх з початку.

Методи інтерфейсу `Deque` розглядаються пізніше.

8.4.3 Інтерфейс ListIterator

Інтерфейс `ListIterator` успадковує інтерфейс `Iterator` і реалізований в колекціях типу `List`. Цей інтерфейс розширює набір методів інтерфейсу `Iterator`. Його методи використовують поняття індексу елемента, а також поняття наступний і попередній. Для виклику цього ітератора використовується метод `listIterator ()`.

Цей ітератор вже розглядався у розділі 6.

8.5 ІНТЕРФЕЙСИ КОЛЕКЦІЙ ДЛЯ ЧЕРГ

Java пропонує розробникові різні способи для створення черг. Ця різноманітність певною мірою пов'язане з розвитком мови.

На рисунку 8.8 зображена діаграма інтерфейсів і класів колекцій пакету `java.util`, які використовуються для створення черг.

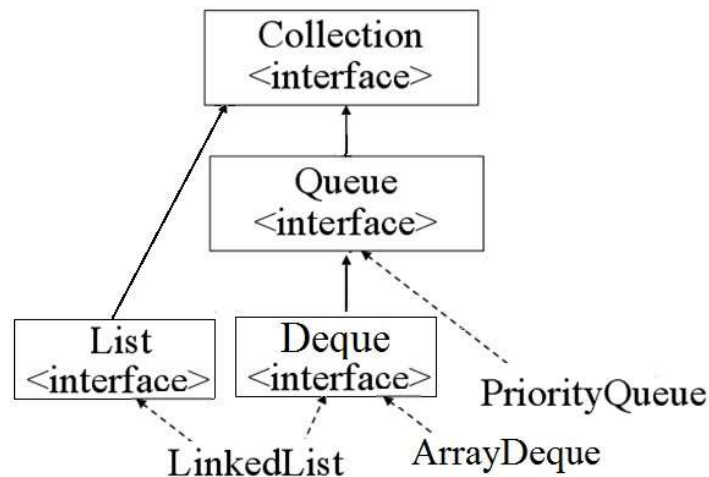


Рисунок 8.8 – Діаграма інтерфейсів і класів колекцій пакету java.util, які використовуються для створення черг

Насамперед слід зазначити, що черги є класичними представниками колекцій і тому інтерфейси, що визначають функціонал класів черг, успадковують інтерфейс Collection.

Специфіка роботи з чергами відображена в інтерфейсі Queue, який було пізніше розширено інтерфейсом Deque.

8.5.1 Інтерфейс Queue

Інтерфейс Queue успадковує інтерфейс Collection і додатково до нього визначає специфічні методи, що використовують при роботі з чергами.

Методи інтерфейсу Queue наведені на рисунку 8.9.

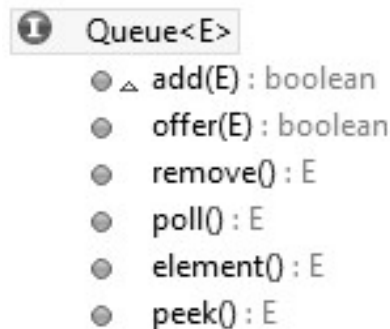


Рисунок 8.9 – Методи інтерфейсу Queue

Методи add і offer додають елемент у кінець черги і відрізняються фактично тільки назвою.

Методи remove і poll вилучають елемент з голови черги і повертають посилання на нього.

Методи peek та element повертають посилання на елемент, що знаходиться в голові черги, але не вилучають його.

8.5.2 Інтерфейс Deque

Цей інтерфейс з'явився в пізніших версіях мови і суттєво розширив набір

методів інтерфейсу Queue. Інтерфейс Deque припускає, що черга є двосторонньою, тобто елементи можна додавати і видаляти з обох кінців черги. Тобто черга одночасно може використовуватися і як стек.

На рисунках 8.10 – 8.12 наведено переліки цих методів. Про призначення методів свідчать їх назви.

- `addFirst(E) : void`
- `addLast(E) : void`
- `offerFirst(E) : boolean`
- `offerLast(E) : boolean`
- `push(E) : void`

Рисунок 8.10 – Методи інтерфейсу Deque, що використовуються щоб додати елемент до черги

- `removeFirst() : E`
- `removeFirstOccurrence(Object) : boolean`
- `removeLast() : E`
- `removeLastOccurrence(Object) : boolean`
- `pollFirst() : E`
- `pollLast() : E`
- `pop() : E`

Рисунок 8.11 – Методи інтерфейсу Deque, що використовуються щоб вилучити елемент з черги

- `peekFirst() : E`
- `peekLast() : E`
- `getFirst() : E`
- `getLast() : E`
- `descendingIterator() : Iterator<E>`

Рисунок 8.12 – Методи інтерфейсу Deque, що використовуються для доступу до елементів

8.5.3 Класи колекцій для черг

Як випливає з діаграми класів 2.1, інтерфейс Queue реалізують такі класи пакету `java.util`.

- `java.util.LinkedList`
- `java.util.ArrayDeque`
- `java.util.PriorityQueue`

8.5.3.1 Клас `LinkedList`

Цей клас донедавна був, судячи з усього, найбільш придатним для

створення черг, тому що в класі передбачено зберігання посилань на початок і кінець списку, що дозволяло швидко додавати елементи в кінець черги і видаляти їх з початку.

8.5.3.2 Клас ArrayDeque

Цей клас з'явився у версії Java 1.6. Його автори припускають, що черги і стеки цього типу будуть працювати швидше, ніж такі ж об'єкти, створені на основі класів `LinkedList` і `Stack`.

Дані в колекції типу `ArrayDeque` зберігаються в масиві. У класі передбачено зберігання індексів хвоста і голови черги, рисунок 8.13. Додавання в кінець черги і видалення останнього проводиться звичайним способом з використанням індексу хвоста. А додавання в початок проводиться без зсуву масиву, шляхом використання умовно негативного індексу, який перетворюється в позитивний, за рахунок відліку від кінця масиву. В результаті такого перетворення елементи, що додаються в початок черги, реально додаються у хвостову частину масиву. Таким чином, при поперемінному додаванні елементів то в начало, то в кінець, індекси хвоста і голови рухаються назустріч один одному, а елементи утворюють як би кільце, перескакуючи з початку в кінець або з кінця в початок. При таких маніпуляціях дані можуть бути або розділені на дві частини, або зберігатися нерозривною групою в середині масиву.

Коли індекс голови досягає значення індексу хвоста, дані переписуються в масив подвоєної довжини. Крім того, для зручності маніпуляцій з індексами, довжина масиву завжди підтримується рівній ступеню двійки. Початкова довжина дорівнює 8.

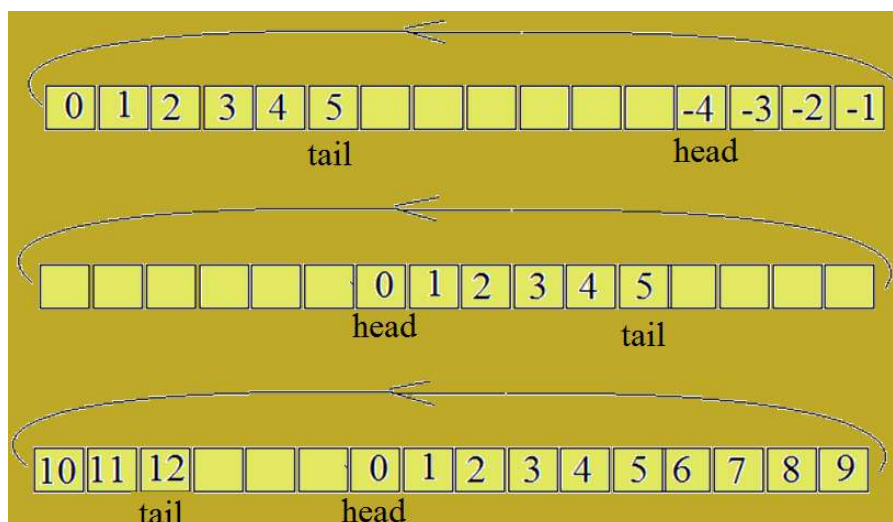


Рисунок 8.13 – Варіанти збереження даних у циклічному масиві

Прискорення роботи в порівнянні з `LinkedList` і `Stack` досягається за рахунок різкого скорочення операцій виділення пам'яті, відсутності операцій зсуву в масиві і використання системних операцій копіювання масивів.

8.5.3.3 Клас PriorityQueue

Цей клас реалізує тільки інтерфейс Queue і дозволяє створити чергу, в голові якої завжди знаходиться найбільший пріоритетний елемент.

Дані зберігаються в масиві, у вигляді збалансованої бінарної купи: два сини елемента черги з індексом $[N]$ зберігаються в елементах з індексами $[2 * N + 1]$ і $[2 * (N + 1)]$, рисунок 8.14. Причому пріоритет кожного вузла вище, ніж пріоритети дітей.



Рисунок 8.14 – Схема бінарної купи

Вилучення елемента з черги робиться з початку масиву, а додається елемент в кінець. Після кожної операції проводиться реорганізація масиву.

Слід мати на увазі, що такий спосіб зберігання не забезпечує впорядкованості всіх елементів черги в масиві, і тому ітератор не повертає елементи в порядку зменшення пріоритету. Однак при послідовному вилучення елементів з черги елементи будуть повертатися в порядку їх пріоритетів.

Поняття «пріоритет» визначається за допомогою об'єкта типу Comparator, який передається через конструктор, або використовується метод compareTo інтерфейсу Comparable, якщо елементи черги його реалізують.

8.6 КОЛЕКЦІЇ, ЩО РЕАЛІЗУЮТЬ ІНТЕРФЕЙС SET

Колекції, що реалізують інтерфейс Set, забезпечують унікальність об'єктів, але не підтримують поняття індексу. Java пропонує розробникові різні способи для створення таких колекцій.

На рисунку 8.14 зображена діаграма інтерфейсів і класів колекцій пакету java.util, які використовуються для створення унікальних множин.

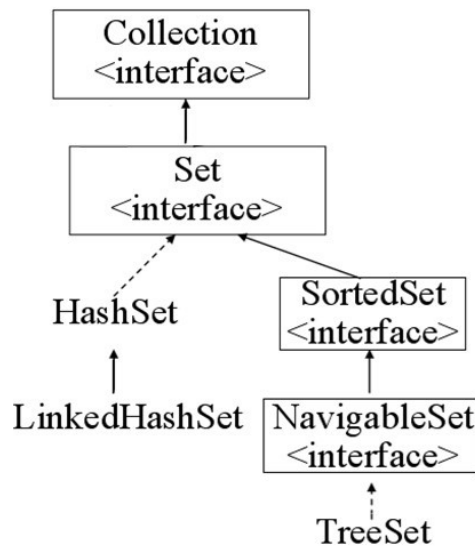


Рисунок 8.14 – Діаграма інтерфейсів і класів пакету java.util для роботи з унікальними множинами

8.6.1 Інтерфейс Set

Методи інтерфейсу Set повторюють методи інтерфейсу Collections. Однак, інтерфейс Set гарантує, що колекція містить унікальні елементи.

Унікальність об'єкта перевіряється за допомогою методу equals() класу, до якого належать елементи колекції, або його суперкласів. Для ідентифікації елементів в колекції використовується їх хеш код. Цей код формується методом hashCode класу до якого належать елементи колекції, або його суперкласів.

8.6.2 Методи equals() u hashCode()

У класі Object є два методи для перевірки ідентичності об'єктів equals () і hashCode ().

Реалізація методу equals () за замовчуванням, надана класом Object, є просто перевіркою тотожності посилань:

```

public boolean equals(Object obj) {
    return (this == obj);
}
  
```

Метод hashCode () є вбудованим, і результат залежить від розміщення об'єкта в пам'яті.

Таким чином, за замовчуванням об'єкти однакові, якщо вони займають одне і те ж місце в пам'яті, тобто це має бути один і той же об'єкт.

Однак часто однаковість об'єктів потрібно визначати за іншими правилами. Наприклад, два об'єкти класу Point (точка) можна вважати однаковими, якщо однакові їх координати, хоча об'єкти займають різне місце в пам'яті. У таких випадках метод equals() слід перевизначити, що і зроблено в класі Point.

Але перевизначаючи метод equals(), слід змінити і метод hashCode(), бо між цими методами існують важливі взаємозв'язки. Якщо два об'єкти тотожні

відповідно до методу equals (), вони повинні мати одне значення hashCode(), хоча зворотне не завжди вірно. Реалізація hashCode (), надана класом Object, виходить з перетворення адреси пам'яті об'єкта в цілочисельне значення. Але в деяких архітектурах адресне поле більше, ніж область значень для int, то, можливо, у двох окремих об'єктів може бути один і той же hashCode (.)

Одночасна зміна методів equals() і hashCode() досить відповідальне завдання, тому Eclipse надає можливість автоматичної генерації цієї пари методів з урахуванням значень полів об'єктів.

Слід зауважити, що питання ідентичності об'єктів могло би вирішуватися виключно методом equals(). Але метод hashCode () існує для підвищення продуктивності роботи з хешованими наборами даних.

8.6.3 Деякі класи, що реалізують інтерфейс Set

8.6.3.1 Клас HashSet

У цьому класі посилання на елементи множини зберігаються у так званих «кошиках». Кожен кошик крім посилання на елемент містить його хеш код та іншу інформацію.

Для зберігання посилань на кошики використовується масив, розмір якого зазвичай істотно перевищує кількість елементів в колекції. При додаванні нового елемента в колекцію для нього створюється кошик і посилання на нього розміщується в масив по індексу, який обчислюється за хеш кодом цього об'єкта. Для отримання індексу хеш код обрізається до діапазону можливих індексів масиву (наприклад, шляхом застосування операції % по модулю довжини масиву або її аналога). В результаті посилання на кошики розташовуються в масиві хаотично.

Хеш код об'єкта унікальний, але у результаті його обробки може виникнути ситуація, коли різним хеш кодами буде відповідати один і той же індекс. Для вирішення цієї колізії кожна корзина має поле next для зберігання посилання на таку ж корзину. Це дозволяє створювати односпрямований список кошиків з однаковими індексами.

Таким чином, якщо у масиві місце для посилання на нову корзину вже зайнято, то посилання на новий кошик розміщується не у масиві, а у кошику, яка знаходиться у кінці списку.

Коли масив заповнюється до деякого критичного значення, наприклад на 75%, створюється масив більшого розміру і туди переписуються елементи колекції. Ця операція називається рехешування. Але оскільки розмір масиву змінився то змінюються і індекси елементів, хоча хеш код залишається постійним. В результаті позиції елементів у масиві змінюється.

Доступ до елементів у таких колекціях дуже швидкий, майже такий же як доступ по індексу масиву. Але доступ елементів за допомогою ітератора уповільнений, тому що доводиться пропускати порожні клітинки масиву, та ще й звертатися до кошиків.

8.6.3.2 Клас `LinkedHashSet`

Особливість колекцій цього класу в тому, що кожен кошик колекції зберігає посилання на елемент, який додається в колекцію після нього і на попередній елемент. Таким чином створюється двонаправлений список кошиків, що зберігає порядок додавання елементів колекції. Колекція зберігає також посилання на початок і кінець цього списку. Це дозволяє організувати перебір елементів в порядку їх додавання і прискорює цей процес.

8.6.4 Інтерфейси для впорядкованих унікальних множин

Java надає можливість створювати впорядковані множини унікальних об'єктів. Для порівняння таких об'єктів при впорядкуванні використовується метод `compareTo()`, якщо об'єкти підтримують інтерфейс `Comparable`. Для об'єктів інших типів можна використовувати компаратор (інтерфейс `Comparator`), який передають колекції через відповідний конструктор.

Унікальність об'єктів також забезпечується використанням методів `compareTo()` або `compare()`. Якщо результат порівняння дорівнює 0, то елемент у колекцію не заноситься. Однак поняття однаковості елементів у сенсі методу `compareTo()` або методу компаратора `compare()` не повинно суперечити методу `equals()`.

8.6.4.1 Інтерфейс `SortedSet`

Цей інтерфейс, рисунок 8.15, визначає методи, які повинні бути реалізовані у впорядкованій множині, і передбачає надання доступу до першого та останнього елементів, а також до впорядкованих частин колекції з початку, кінця та середини колекції.

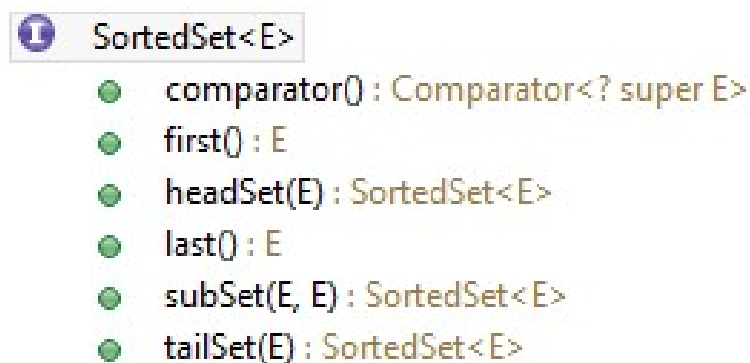


Рисунок 8.15 – Інтерфейс `SortedSet`

8.6.4.2 Інтерфейс `NavigableSet`

Цей інтерфейс, рисунок 8.15, з'явився у версії мови 1.6 і розширив інтерфейс `SortedSet`. Він передбачає надання доступу до елементів що знаходяться вище чи нижче заданого з урахуванням нюансу, якщо елемент дорівнює заданому.