МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чернігівський національний технологічний університет

ОБ'ЄКТНО ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА JAVA

Конспект лекцій з дисципліни
«Об'єктно орієнтоване програмування»
для студентів спеціальностей
121 – «Інженерія програмного забезпечення»
123 – «Комп'ютерна інженерія»

ЗАТВЕРДЖЕНО
на засіданні кафедри
інформаційних та комп'ютерних систем
протокол № 1 від 29.08.19

Чернігів ЧНТУ 2019

Об’єктно орієнтоване програмування на Java. Конспект лекцій з дисципліни «Об’єктно орієнтоване програмування» для студентів спеціальностей 121 – «Програмна інженерія», 123 – «Комп’ютерна інженерія». /Укл.: Бивойно П.Г. – Чернігів: ЧНТУ, 2019. – 136 с.

Укладачі: Бивойно Павло Георгійович, канд. техн. наук, доцент;

Відповідальний за випуск: В.М. Базилевич, зав. кафедрою інформаційних та комп’ютерних систем, канд. економ. наук., доцент

Рецензент: С. О. Нестеренко, канд. техн. наук, доцент кафедри інформаційних і комп'ютерних систем Чернігівського національного технологічного університету

ЗМІСТ

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+в розглядається як відправлення об’єкту «а» повідомлення «+» з параметром «в». У класі, якому належіть 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 проектантами 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 перевантаження 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.

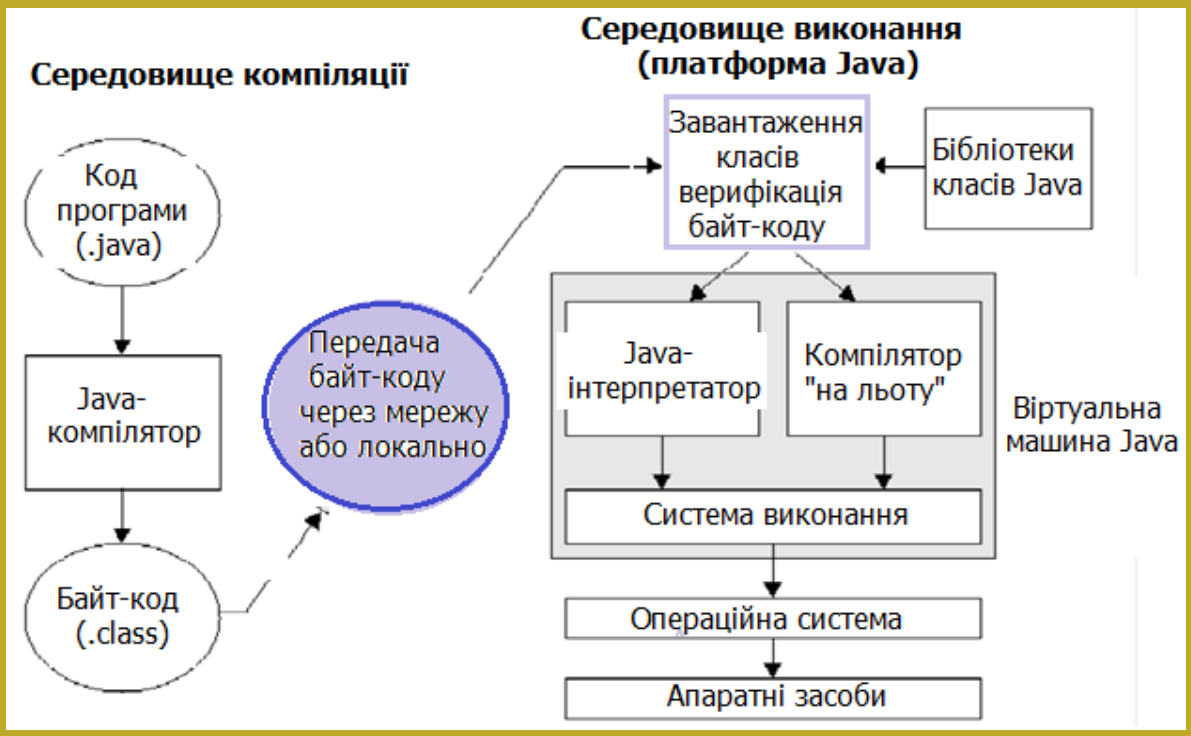


Рисунок 1.1 – Схема виконання Java програми

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 С.

Нижче наведено приклад, який можна протестувати на java-сторінці середовища Eclipse (файл з розширенням .jpage). 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 С.

```
/*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, символьні 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 С 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 С та С++ 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 типі майже такі самі, як і у мові С.

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 С++

- 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, шо 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 С, але результат обчислення виразу умови має бути тільки бульовим.

Нижче наведено приклад використання цього оператору.

```
//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 С. Але вибір типів ширший. У якості параметру перемикача можна використовувати дані типу String.

Нижче наведено приклад використання цього оператору.

```
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 и do … while*

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

Нижче наведено приклад використання оператору 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 С.

Нижче наведено приклади використання цього оператору.

```
// 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 елементу масиву використовуються квадратні дужки, де вказується значення індексу. Індекси 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 значення 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 елементу 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 С. 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 файловій 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 належіть 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 залежіть 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);
– теренарний 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 С, повертає результат, значенням якого є посилання, яке було присвоєно.

Операція instanceof дозволяє дізнатися, чи належіть 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 належіть 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 познайомимося 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 метода 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 шістнадцятирічного числа і таким чином унікально характеризує об’єкт. Але цей метод, зазвичай, перевизначать 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 методі 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. BJava 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 символьний вигляд для числових даних базових типів int, double, та ін. 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 ncmxcnm    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 перевантаженими 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 символьної послідовності у об’єкті StringBuilder. Метод append() завжди додає символи у кінець існуючої послідовності, в той час як insert() додає символи всередину символьної 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-обгорток.

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 форматувати 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 виявилося 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 yyyy 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 (стрим) для набору даних. Для масивів такий об'єкт можна отримати за допомогою статичного методу stream() класу Arrays. Для випадкових чисел стрим можна отримати, наприклад, за допомогою методу doubles().

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

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

Нижче наведено деякі з таких методів:

filter – вибирає із стриму дані, що відповідають заданій умові;

distinct – вилучає із стриму усі дублікати даних;

sorted – впорядковує дані стриму;

map – змінює дані стриму 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 стримом для масиву цілих чисел:

– стрим наповнюється випадковими цілими числами від 0 до 9 за допомогою методу map;

– зі стриму 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 абревіатур. Для використання необхідно написати відповідну абревіатуру 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 абревіатур можна переглянути в налаштуваннях програми, крім того, можна додавати свої абревіатури 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 кликнути на іконку помилки у відповідному рядку). Наприклад, коли у виразі використовується неоголошена 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.StringBilder та java.lang.StringBufer
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 випливає 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 спадкоємцях 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 «a» 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 перепоною 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 спадкоємцях 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 спадкоємцях, 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 B 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

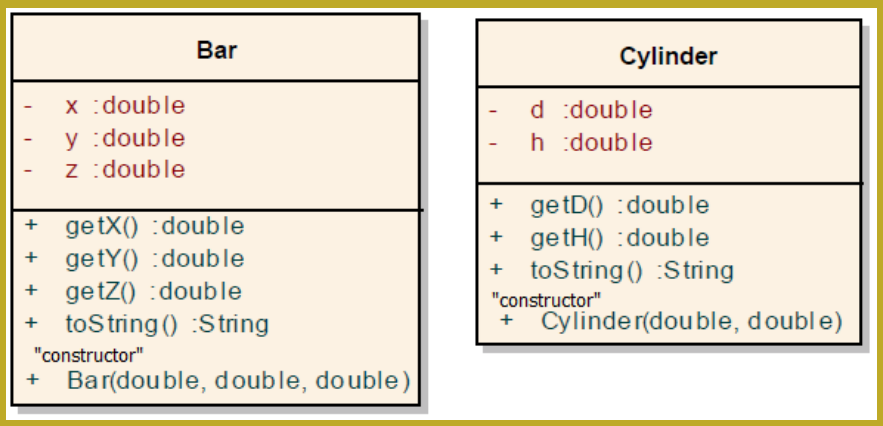


Рисунок 4.1 – Діаграма класів для видів продукції, варіант 1

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.

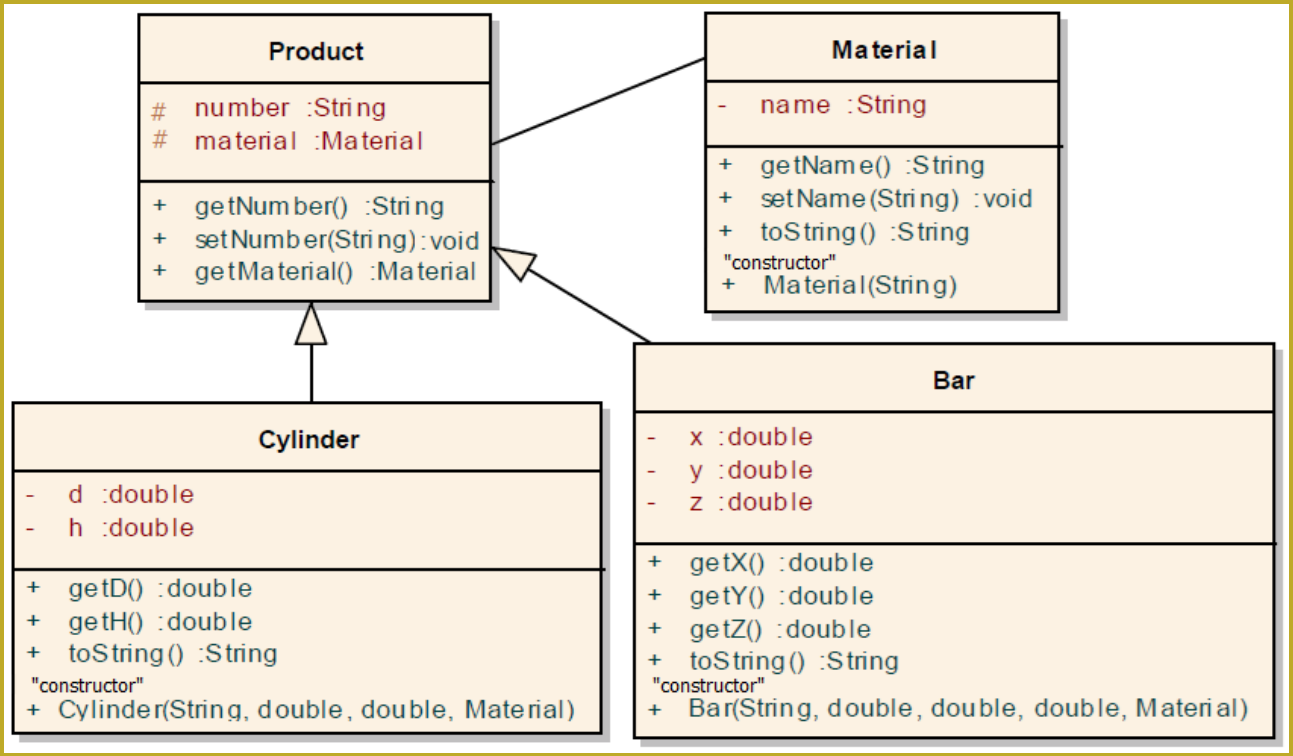


Рисунок 4.2 – Діаграма класів для видів продукції та матеріалу, варіант 2

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'ясовується, 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.

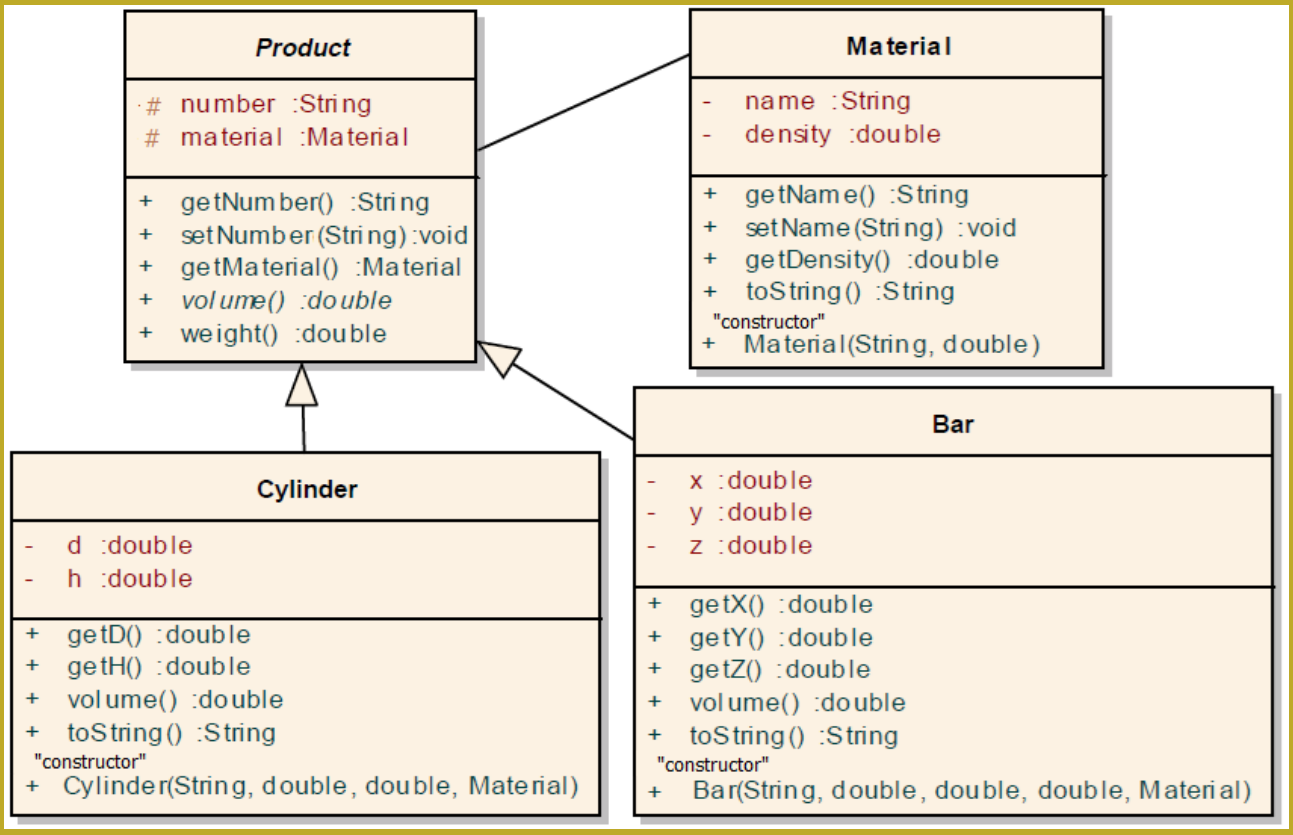


Рисунок 4.3 – Діаграма класів, варіант 3

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.

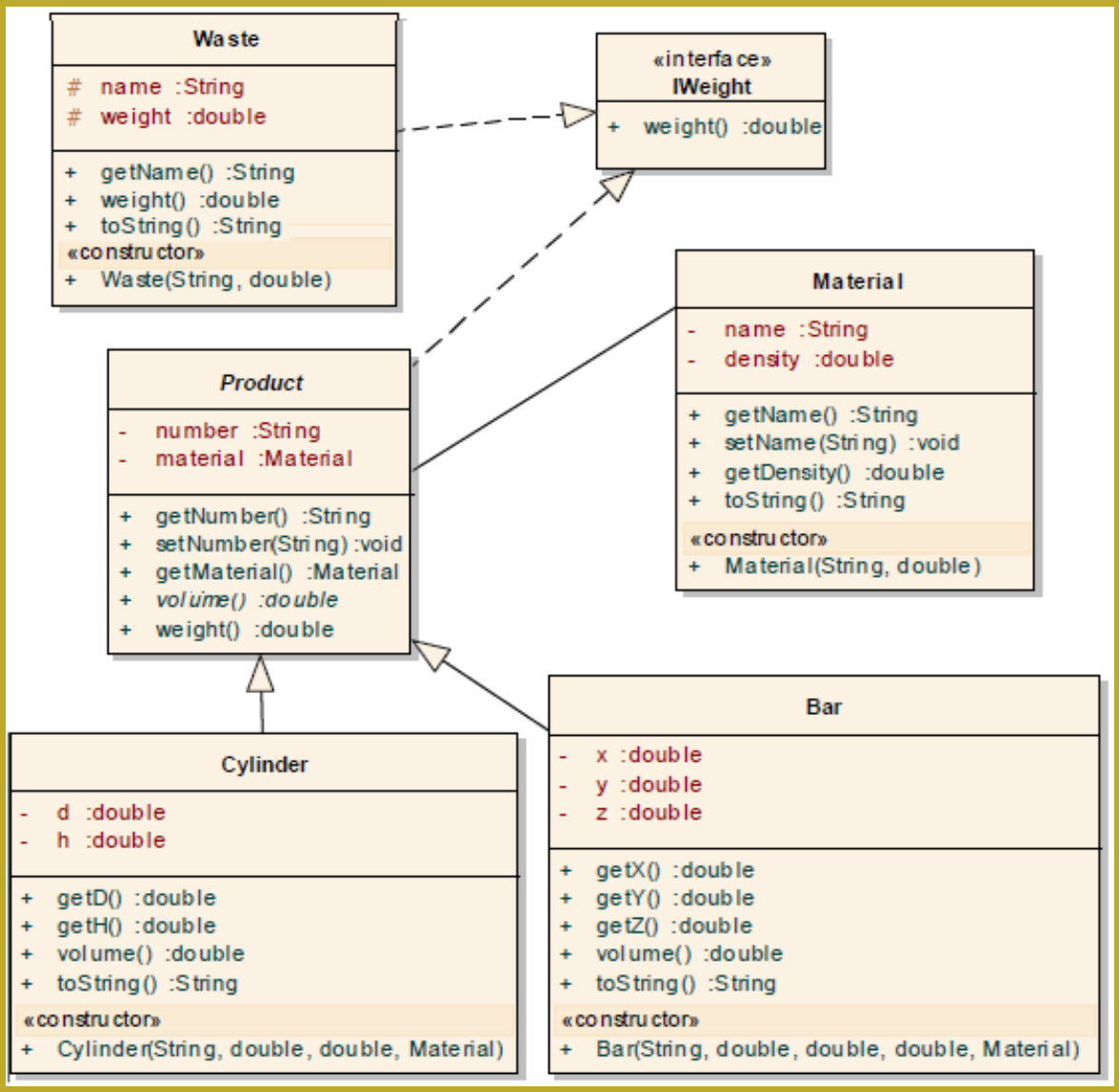


Рисунок 4.4 – Діаграма класів, варіант 4

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.

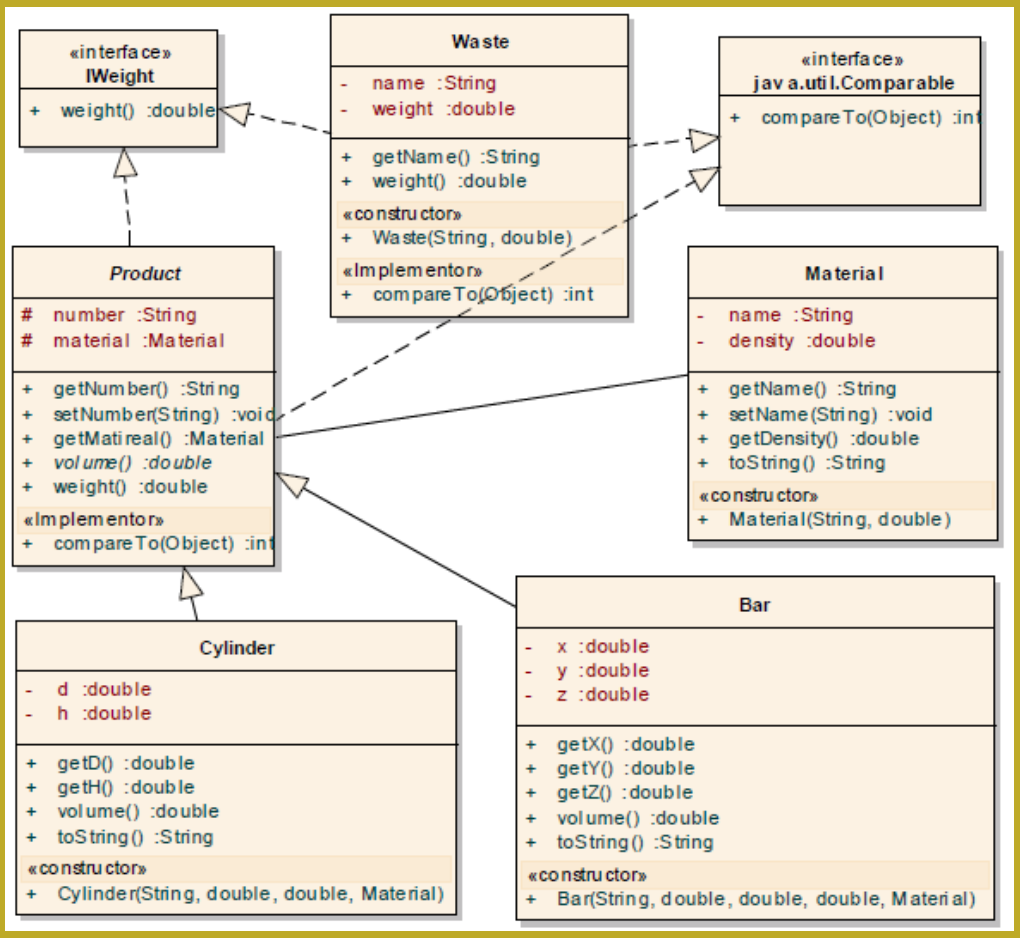


Рисунок 4.5 – Остаточна діаграма класів (Варіант 5)

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, буферізований, двійковий, символьний 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.

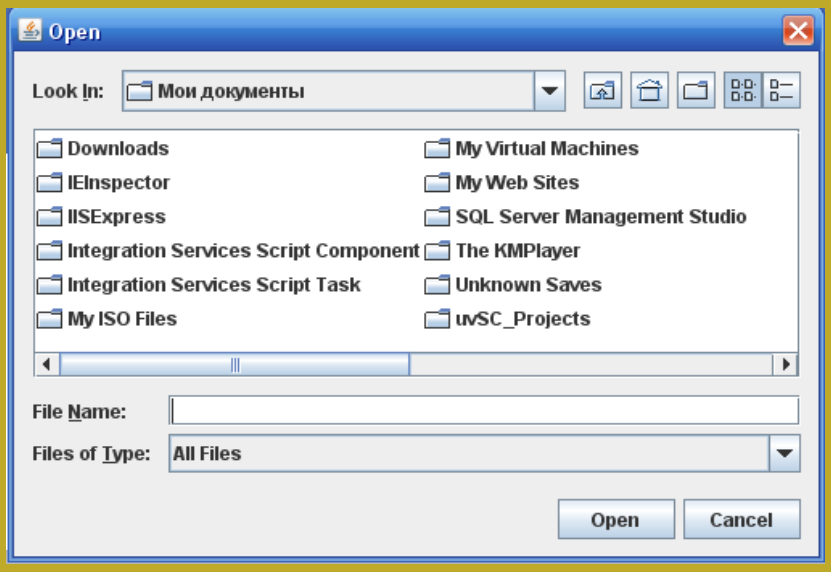


Рисунок 5.1 – Вид компоненту JFileChooser

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                    я
dialog.setApproveButtonText("Выбрать");
// Вибрати назву діалогу
dialog.setDialogTitle("Выберите файл для загрузки");
//  Вибрати тип діалогу
dialog.setDialogType(JFileChooser.OPEN_DIALOG);
// Дозволити вибір декількох файлі
dialog.setMultiSelectionEnabled(true);
```

В компоненті можна задавати фільтр, який визначає які файли відображати користувачеві. Нижче наведено приклад фільтру, 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 діалогової панеллю.

Другий параметр дозволяє вказати довільний заголовок діалогової панелі.

Третій параметра 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.Scaner. 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 буферізацію, 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); // сериализуемо 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 ВИКОРИСТАННЯ ІТЕРАТОРІВ

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 потребою 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

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

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 iterabieObj, послідовно один за іншим від початку до кінця. Об'єкт, на який посилається змінна iterabieObj, 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 iterabieObj 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 методі 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.

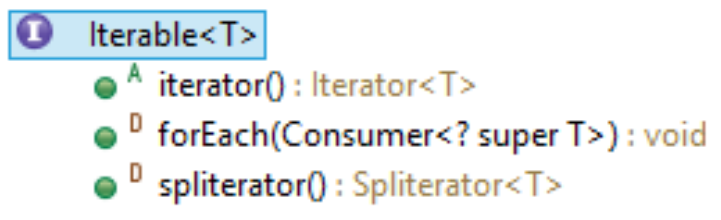


Рисунок 6.1 – Методи інтерфейсу Iterable

6.3 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

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

7 ПЕРЕДАЧА СПЕЦИФІЧНИХ ОПЕРАЦІЇ 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 С 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 Clause.

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. Нема покажчиків на функції та й самих функцій, немає процедурних типів і класів подібних класу Block в Smalltalk , або Clause 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) {
    onButtonClick();
  }
});
```

А використовуючи лямбда функції, те саме можна написати простіше:

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

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 сформувати 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 посилальними 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 елементу масиву, тому вказуємо Integer. Метод test цього інтерфейсу перевіряє, чи відповідає параметр методу заданій умові.

Для того, щоб протестувати роботу узагальненого методу створімо 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 елементу 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 заміняти одиничкою.

8 КОЛЕКЦІЇ JAVA

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 узагальненнях можна послухати в лекціях по посиланнях:

https://www.youtube.com/watch?list=SPB0276A0A62BDEF06&v=87pm79sPSvc&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 рамочок.

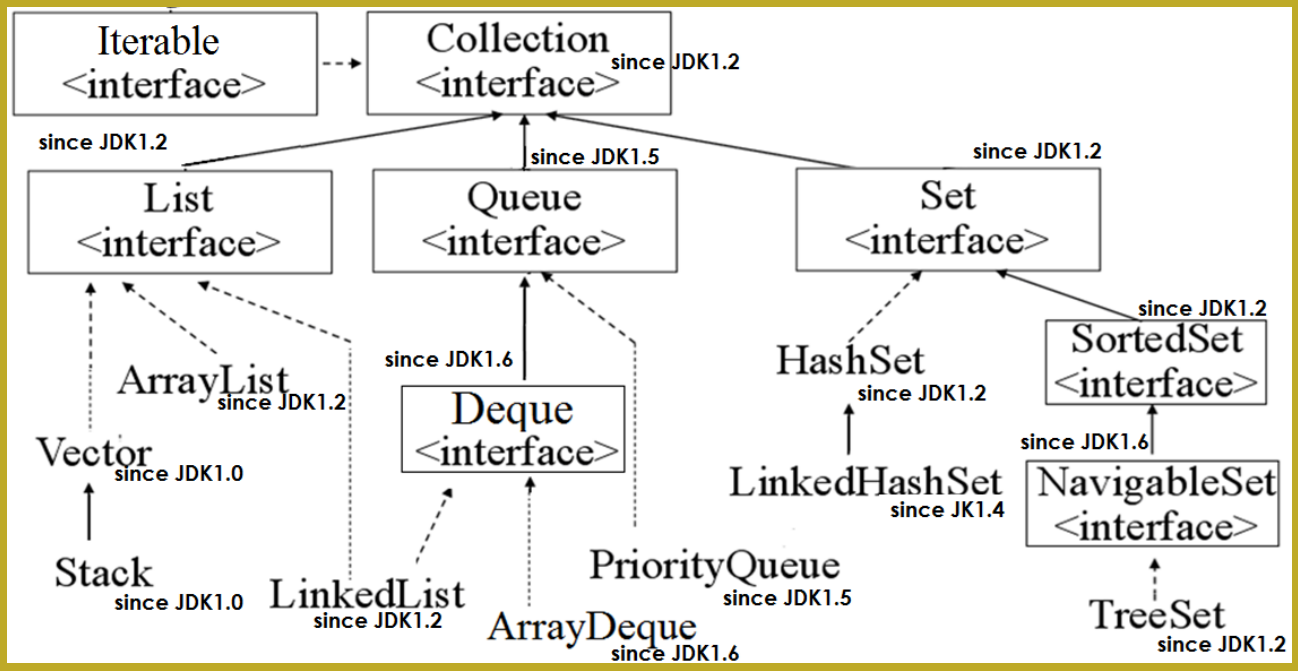


Рисунок 8.1 – Діаграма класів колекцій пакету java.util

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.

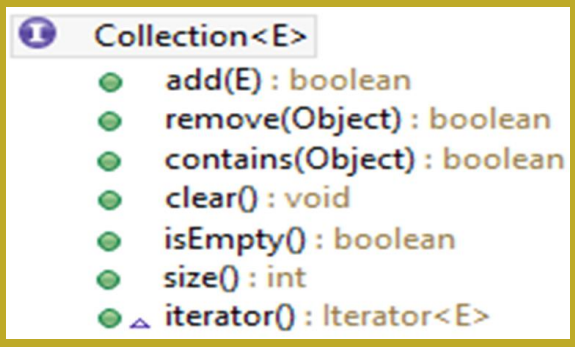


Рисунок 8.2 – Основні методи інтерфейсу Collection

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.

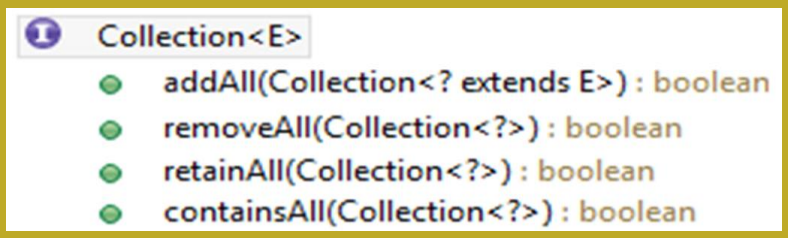


Рисунок 8.3 – Методи інтерфейсу Collection, що визначають операції над множинами

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.

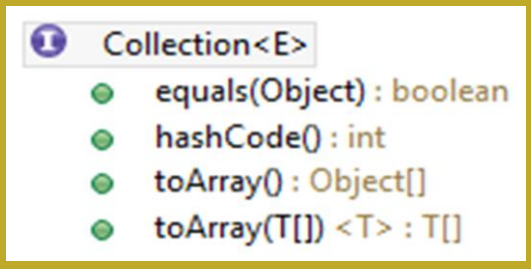


Рисунок 8.4 – Інші методи інтерфейсу Collection

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.

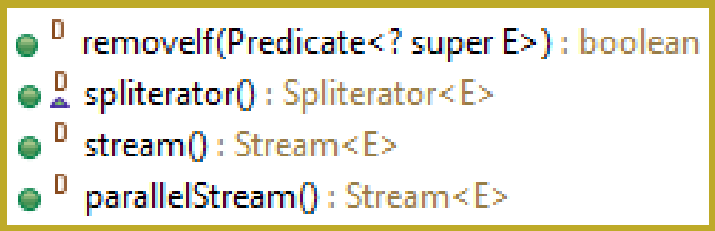


Рисунок 8.5 – Дефолтні методи інтерфейсу Collection

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() //OptonalDouble
 .getAsDouble();
```

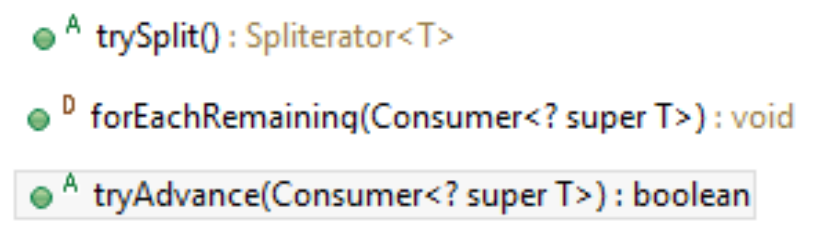
Приклад видалення елементів за умовою:

```
//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



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 index повертає індекс першого входження елемента до колекції.

Метод lastIndex повертає індекс останнього входження елемента.

Метод listIterator повертає ітератор типу ListIterator.

8.4.2 *Класи, що реалізують інтерфейс List*

Як випливає 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 елементу 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 Dequeue, 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 Dequeue 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.

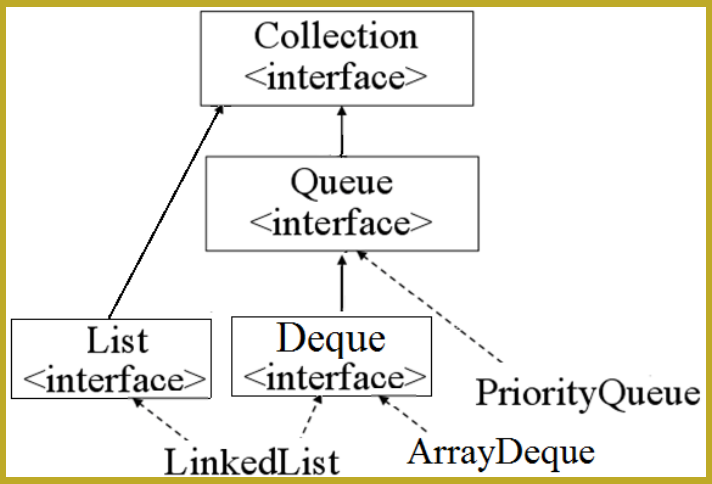


Рисунок 8.8 – Діаграма інтерфейсів і класів колекцій пакету java.util, які використовуються для створення черг

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.

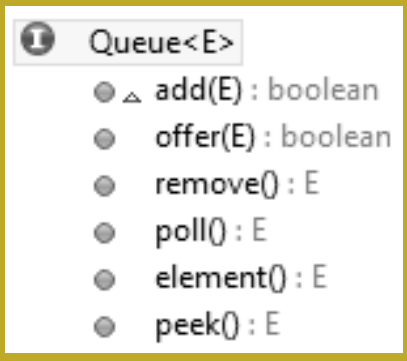


Рисунок 8.9 – Методи іинтерфейсу Queue

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.

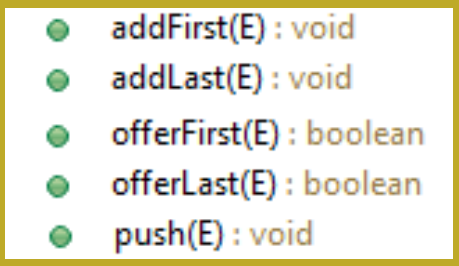


Рисунок 8.10 – Методи інтерфейсу Deque, що використовуються щоб додати елемент до черги

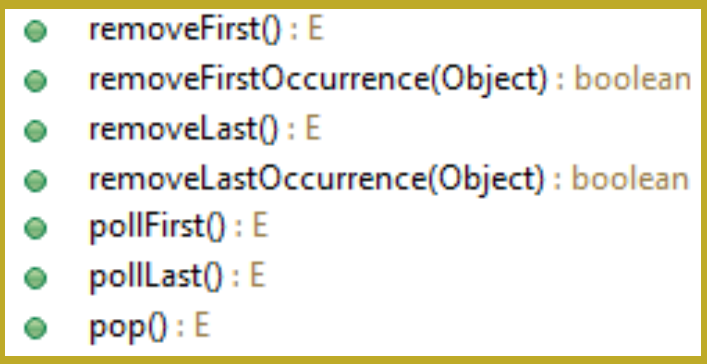


Рисунок 8.11 – Методи інтерфейсу Deque, що використовуються щоб вилучити елемент з черги

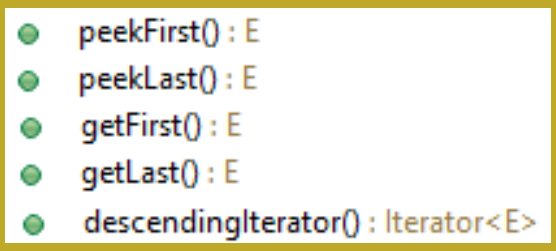


Рисунок 8.12 – Методи інтерфейсу Deque, що використовуються для доступу до елементів

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 можу 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.

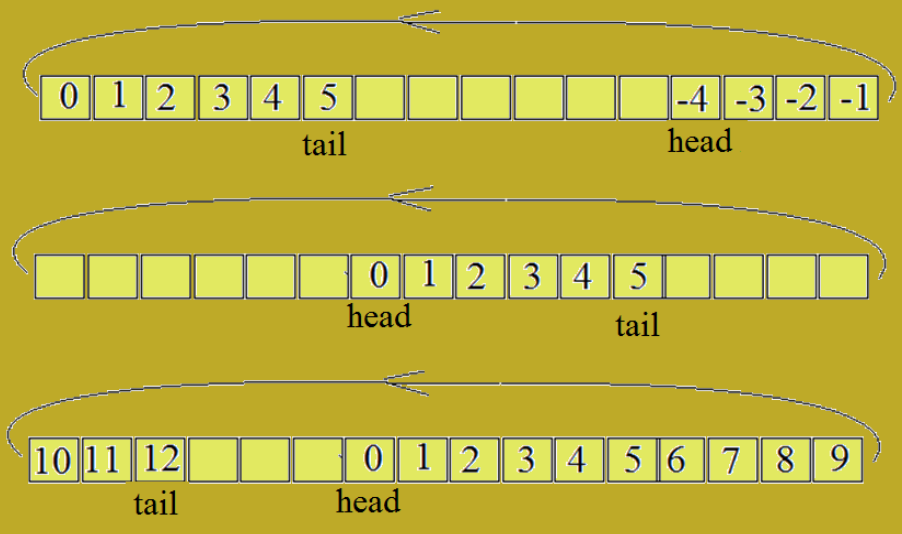


Рисунок 8.13 – Варіанти збереження даних у циклічному масиві

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 вилученні 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.

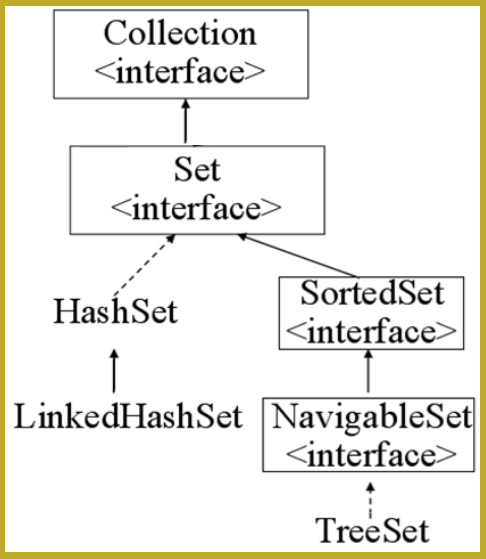


Рисунок 8.14 – Діаграма інтерфейсів і класів пакету java.util
для роботи з унікальними множинами

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() и 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 елементу 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.

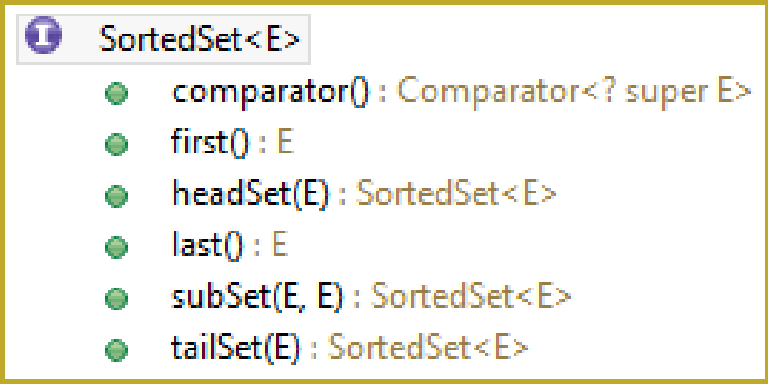


Рисунок 8.15 – Інтерфейс SortedSet

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.

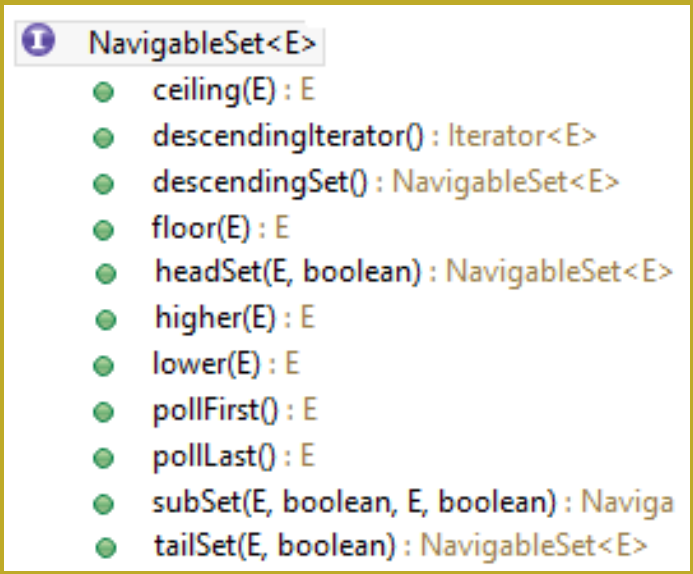


Рисунок 8.15 – Інтерфейс NavigableSet

8.6.5 *Класи для впорядкованих унікальних множин*

8.6.5.1 Клас TreeSet

Клас TreeSet реалізує інтерфейси SortedSet і NavigableSet. Елементи цієї колекції розташовуються у двійковому збалансованому дереві.

Кожен вузол дерева являє собою кошик, в якому зберігається посилання на елемент колекції, а також посилання на кошики лівого і правого синів та на батька. Хеш коди елементів у цьому класі не використовуються.

8.7 КЛАС COLLECTIONS

Утилітний клас Collections містить велику кількість статичних методів, що дозволяють обробляти колекції, а також створювати нові.

Деякі з них можна застосовувати для колекцій будь яких типів, інші мають обмежене застосування, наприклад, тільки для колекцій типу List.

Наводити тут перелік методів класу ми не будемо, з ними можна ознайомитися, відкривши текст класу. Натомість, розглянемо декілька прикладів.

8.7.1 *Методи класу Collections для даних типу Collection*

Ці методи не залежать від внутрішньої організації даних у колекції і оперують тільки з даними.

Нижче наведено декілька прикладів використання таких методів:

```
// Наповнення колекції даними
   Collection<Double> col = new Vector<>();
   Collections.addAll(col, 3d, 3.7, 4.3, 5.);
   col.forEach( System.out::println);
//Визначаємо кількість повторень числа 3.0
   int fr = Collections.frequency(col, 3d);
// Знаходимо найбільше число в колекції
```

```
   double max = Collections.max(col);
// Знаходимо найменше число в колекції
   double min = Collections.min(col);
```

8.7.2 *Методи класу Collections для об'єктів типу List*

Ці методи враховують наявність понять попередній та наступний елемент, що притаманні колекціям типу List.

Нижче наведено декілька прикладів їх використання:

```
//Створюємо колекцію
   List<Integer> list = new Vector<>();
   Collections.addAll(list,1,3,5,3,4,2,14,3,7);
//Міняємо елементи місцями
   Collections.swap(list, 2, 6); //[1,3,14,3,4,2,5,3,7]
//Впорядковуємо колекцію
   Collections.sort(list);   //[1, 2, 3, 3, 3, 4, 5, 7, 14]
//Впорядковуємо у зворотньому напрямку
   Collections.sort(list,(a,b)->b-a); //[14,7,5,4,3,3,3,2,1]
//Знаходимо позицію числа 3 для зворотнього напрямку
   Collections.binarySearch(list,3,(a,b)->b-a));//4
//Перевернути колекцію
   Collections.reverse(list); //[1,2,3 3,3,4,5,7,14]
//Зсув відносно заданої позиції з переносом на початок
   Collections.rotate(list,2); //[7,14,1,2,3,3,3,4,5]
//Перемішування колекції
   Collections.shuffle(list); //[2,3,7,1,5,3,14,3,4]
//Заміна значень елементів колекції
   Collections.replaceAll(list,3,8); //[2,8,7,1,5,8,14,8,4]
//Заповнення колекції однаковими елементами
Collections.fill(list,0);  //[0,0,0,0,0,0,0,0,0]
```

8.8 ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Поняття колекції в мовах програмування.
2. Використання узагальнень в колекціях.
3. Методи інтерфейсу Collection.
4. Діаграма інтерфейсів і класів колекцій інтерфейсу List.
5. Особливості інтерфейсу List і його методи.
6. Особливості класів Vector і ArrayList.
7. Клас Stack і його методи.
8. Особливості класу LinkedList.
9. Діаграма інтерфейсів і класів колекцій пакету java.util, які використовуються для створення черг
10. Методи інтерфейсу Queue.
11. Особливості інтерфейсу Dequeue.
12. Класи LinkedList, ArrayDeque, PriorityQueue.
13. Діаграма інтерфейсів і класів колекцій пакету java.util для роботи з унікальними множинами.
14. Особливості інтерфейсу Set.

15. Методи equals () і hashCode ().
16. Класи HashSet і LinkedHashSet.
17. Інтерфейси SortedSet і NavigableSet.
18. Клас TreeSet.

9 КАРТИ ВІДОБРАЖЕННЯ JAVA

9.1 ЗАГАЛЬНІ ВІДОМОСТІ ПРО АСОЦІАТИВНІ МАСИВИ

Асоціативний масив або карта відображень (map) - це об'єкт, який зберігає асоціації (зв'язку) між ключами і значеннями, або інакше - пари ключ / значення, наприклад <"ЧДТУ", об'єкт-ЧДТУ>, <№ заліковки, об'єкт-СТУДЕНТ>. В якості ключів не можуть використовуватися примітивні типи, можуть використовуватися тільки об'єкти.

Ключі карт повинні бути унікальними.

Діаграма інтерфейсів і класів, що реалізують асоціативні масиви представлена на рисунку 9.1.

До складу інтерфейсів карт відображень входять - Map, Map.Entry і SortedMap і NavigableMap, які призначені:

– Map - для відображення унікальних ключів е значення;

– SortedMap і NavigableMap - розширюють Map і підтримує пари е відсортованому порядку по ключах.

– Map.Entry - внутрішній інтерфейс для опису «кошиків» в яких зберігаються елементи множини (пари ключ / значення).

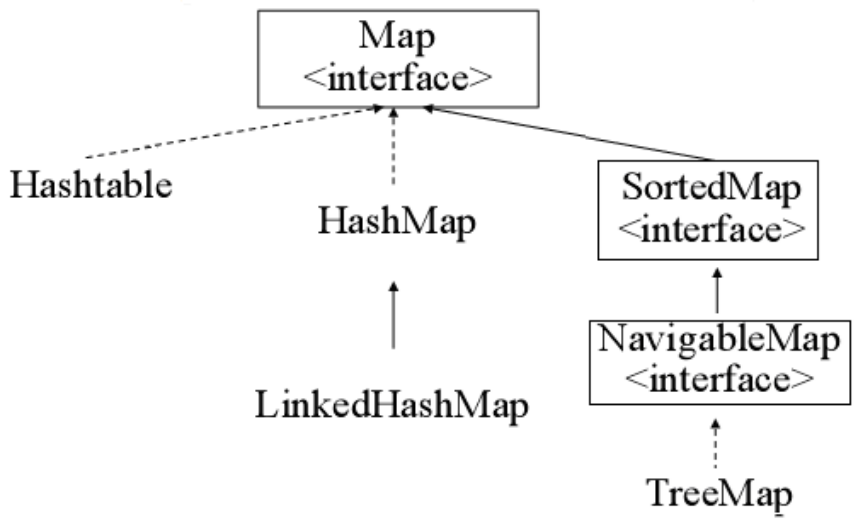


Рисунок 9.1 – Діаграма інтерфейсів і класів, пов'язаних з картами відображень пакету java.util

9.2 ІНТЕРФЕЙСИ КАРТ ВІДОБРАЖЕНЬ

9.2.1 *Інтерфейс Map*

Методи інтерфейсу Map наведені на рисунках 9.2 і 9.3.

Поповнюється масив за допомогою методу put, де в якості параметрів вказують ключ і значення.

Метод putAll дозволяє об'єднати карти.

Метод get дозволяє отримати значення з масиву на підставі заданого ключа не вилучаючи його.

Метод remove вилучає елемент з карти по заданому ключу і повертає значення елементу.

Призначення решти методів з рисунку 9.2 зрозуміло з їх назв.

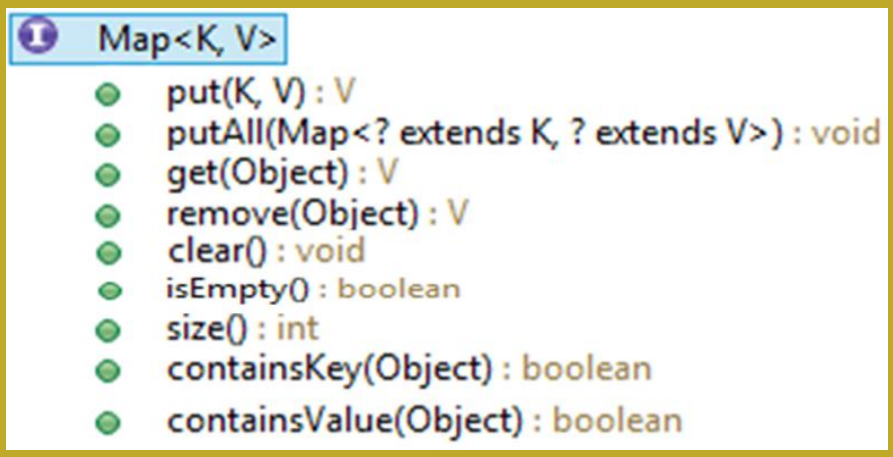


Рисунок 9.2 – Методи інтерфейсу java.util.Map для роботи з елементами колекції

На рисунку 9.3 наведено методи, що дозволяють отримати колекції складових асоціативного масиву.

Метод entrySet повертає множину кошиків масиву, що мають тип Map.Entry.

Метод keyset повертає множину унікальних ключів карти.

Метод values повертає колекцію значень карти.

Методи equals та hashCode перевизначають відповідні методи класу Object.

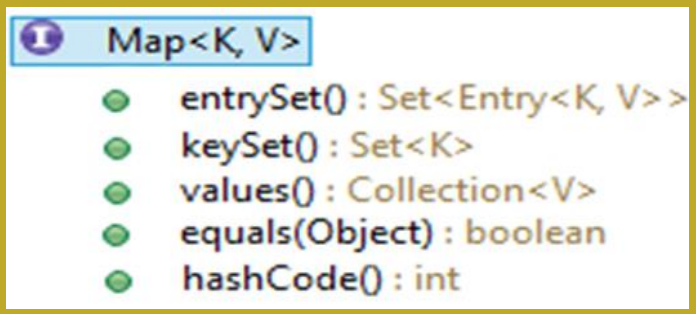


Рисунок 9.3 – Методи інтерфейсу java.util.Map для доступу до структурних складових карти та допоміжні методи

9.2.2 *Допоміжний інтерфейс Map.Entry*

Внутрішній інтерфейс Map.Entry визначає методи спілкування з кошиками, що входять до складу карти.

До найпопулярніших методів інтерфейсу Map.Entry відносяться:
– Object getKey () - повертає ключ пари;
– Object getValue () - повертає значення пари.

9.2.3 *Інтерфейс SortedMap*

Цей інтерфейс визначає методи роботи з впорядкованими картами відображень. За функціональністю ці методи подібні методам інтерфейсу SortedSet. На рисунку 9.4 зображені ці методи.

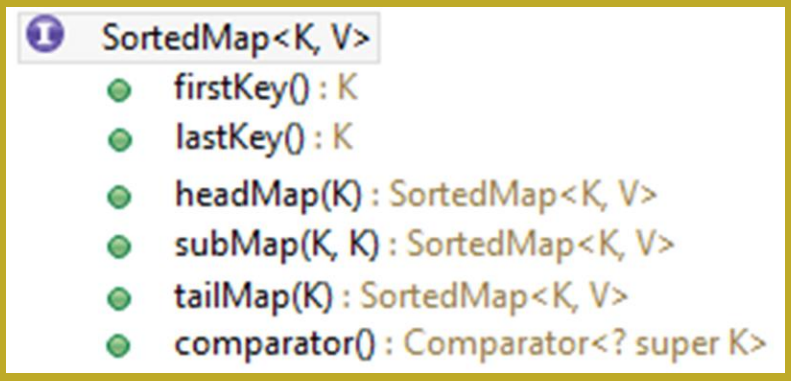


Рисунок 9.4 – Методи інтерфейсу java.util.SortedMap

9.2.4 *Інтерфейс NavigableMap*

Цей інтерфейс розширює SortedMap і містить додаткові методи роботи з впорядкованими картами відображень, що з'явилися у версії 1.6. На рисунках 9.5 - 9.6 зображені ці методи. За функціональністю ці методи подібні методам інтерфейсу NavigableSet.

Рисунок 9.5 – Методи інтерфейсу java.util.NavigableMap для роботи з елементами карти

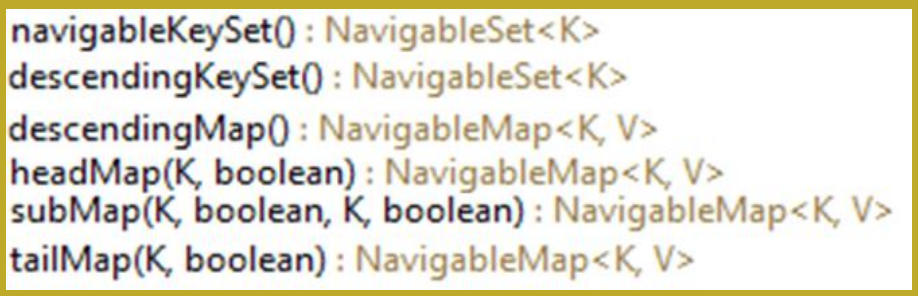


Рисунок 9.6 – Методи інтерфейсу java.util.NavigableMap для роботи зі структурними складовими карти

9.3 КЛАСИ АСОЦІАТИВНИХ МАСИВІВ

Ці класи реалізують методи інтерфейсів асоціативних масивів. Стандартні класи асоціативних масивів Java можна розбити на дві групи - абстрактні і конкретні класи.

Абстрактні класи асоціативних масивів забезпечують часткову реалізацію інтерфейсів і використовуються для створення конкретних класів асоціативних масивів (на рисунку 9.1 абстрактні класи не показані).

Конкретні класи асоціативних масивів розширюють абстрактні класи і забезпечують повну реалізацію інтерфейсів асоціативних масивів. Конкретні класи можуть використовуватися для безпосереднього створення екземплярів цих класів.

До конкретних класів асоціативних масивів відносяться класи:

– HashMap - асоціативний масив у вигляді хеш-таблиці, забезпечує швидкий доступ до об'єктів, стара версія такої карти реалізована у класі HashTable;

– LinkedHashMap - відрізняється від HashMap, тим що зберігає інформацію про порядок додавання елементів.

– TreeMap - асоціативний масив у вигляді збалансованого дерева.

Детальний опис цих класів тут не наводиться, оскільки ці описи фактично збігаються з описами класів інтерфейсу Set. Справа в тому, що всі класи інтерфейсу Set реалізовані на картах відображень. Кошик являє собою об'єкт класу, подібного класу Entry. Єдина різниця полягає в тому, що в множинах не використовується поле value.

Ітератори в колекціях типу Map відсутня, але ці колекції дають доступ до множини своїх кошиків через метод entrySet(), який повертає колекцію типу Set. А тип Set поняття ітератора підтримує. Тому для обходу елементів карти слід отримати множину кошиків і скористатися ітератором для цієї множини.

Окрім того можна отримати множину ключів карти і за допомогою ітератора для цієї множини переглянути усі значення карти.

9.4 ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Поняття карти відображень.
2. Діаграма інтерфейсів і класів пов'язаних з картами відображень пакету java.util.
3. Методи інтерфейсу Map.
4. Характеристика методів інтерфейсу SortedMap.
5. Характеристика методів інтерфейсу NavigableMap.
6. Класи карт відображень.
9. Написати невеликий додаток, що використовує асоціативний масив (наприклад, найпростіший перекладач).

10 ПОТОКИ ВИКОНАННЯ КОДУ

10.1 БАГАТОПОТОЧНІСТЬ

Багатопоточність має на увазі одночасне виконання декількох програм. Розподілом процесорного часу між цими програмами займається операційна система. При цьому слід розрізняти два поняття - потік і процес.

Процес (process) - це об'єкт, який створюється операційною системою, коли користувач запускає додаток. Процесу виділяється окремий адресний простір, причому цей простір фізично недоступний для інших процесів. Процес може працювати з файлами або з каналами зв'язку локальної або глобальної мережі. Коли користувач викликає текстовий процесор або програму калькулятора, створюються нові процеси.

Для кожного процесу операційна система створює один головний потік (thread, тред, нитку), який є потоком команд центрального процесора. При необхідності головний потік може створювати інші потоки, користуючись для цього програмним інтерфейсом операційної системи.

Всі потоки, створені процесом, виконуються в адресному просторі цього процесу і мають доступ до ресурсів процесу. Однак потік одного процесу не має жодного доступу до ресурсів потоку іншого процесу, бо вони працюють в різних адресних просторах. При необхідності організації взаємодії між процесами або потоками, що належать різним процесам, слід користуватися системними засобами, спеціально призначеними для цього.

Найбільш очевидна область застосування багатопоточності - це програмування графічних інтерфейсів користувача (GUI). Нитка корисна тоді, коли необхідно, щоб графічний інтерфейс продовжував відгукуватися на дії користувача під час виконання деякої обробки інформації.

Ще одна популярна область застосування багатопоточності - ігри. В іграх різні потоки можуть відповідати за роботу з мережею, анімацію, розрахунками.

Дуже корисна багатопоточність і при розробці програм моделювання систем з паралельними процесами. Прикладом такого завдання може послужити моделювання роботи операційної системи. Перша об'єктно орієнтована мова Сімула і була створена для вирішення саме таких задач.

10.2 РЕАЛІЗАЦІЯ БАГАТОПОТОЧНОСТІ В JAVA

10.2.1 *Створення потоків в Java*

В Java потік є об'єктом класу java.lang.Thread. Цей клас має усе необхідне для створення і запуску потоків, а також керування їх станом. Для створення потоків клас Thread надає різні конструктори, за допомогою яких можна реалізувати кілька способів створення потоків. Однак у будь якому випадку, код, що має виконуватися у потоці, повинен бути оформлений у вигляді публічного методу run(). Такий метод реалізовано і у класі Thread, проте він майже нічого не робить. Ви можете у цьому переконатися, відкривши його код.

Метод, який реально буде виконуватися у потоці, має бути створений у іншому класі.

Є три можливості опису такого методу:

– створити дочірній клас на базі класу Thread і у ньому перевизначити метод run ();

– реалізувати у деякому класі інтерфейс Runnable, що вимагає реалізації методу run(), а потім створити об'єкт класу Thread, передавши через конструктор цей об'єкт як параметр;

– створити дочірній клас на базі класу TimerTask, у ньому реалізувати метод run(), а потім створити об'єкт класу Timer, встановивши час активації завдання.

Другий спосіб особливо зручний у тих випадках, коли клас, метод якого повинен виконуватися у окремому потоці, успадкований від якогось іншого класу.

10.2.1.1 **Конструктори класу Thread**

За допомогою конструкторів класу Thread, таблиця 6.1, можна створювати потоки різними способами, вказуючи при необхідності для них ім'я і групу. Ім'я призначене для ідентифікації потоку і є необов'язковим атрибутом. Що ж до груп, то вони призначені для організації спільних дій з

Таблиця 6.1 – Конструктори класу Thread

public Thread();	Створення нового об'єкту Thread
public Thread(Runnable target);	Створення нового об'єкту Thread із зазначенням об'єкту, для якого буде викликатися метод run
public Thread(Runnable target, String name);	Аналогічно попередньому, але додатково задається ім'я нового об'єкту Thread
public Thread(String name);	Створення нового об'єкту Thread із зазначенням його імені
public Thread(ThreadGroup group, Runnable target);	Створення нового об'єкту Thread із зазначенням групи потоку і об'єкту, для якого викликається метод run
public Thread(ThreadGroup group, Runnable target, String name);	Аналогічно попередньому, але додатково задається ім'я нового об'єкту Thread
public Thread(ThreadGroup group, String name);	Створення нового об'єкту Thread із зазначенням групи потоку та імені об'єкту

10.2.1.2 **Пріоритети потоків в Java**

Якщо процес створив кілька потоків, то всі вони виконуються паралельно, причому час центрального процесора (або декількох центральних процесорів в мультипроцесорних системах) розподіляється між цими потоками.

Розподілом часу центрального процесора займається спеціальний модуль операційної системи - планувальник. Планувальник по черзі передає

управління окремим потокам, так що навіть у однопроцесорній системі створюється повна ілюзія паралельної роботи запущених потоків.

Розподіл часу виконується по перериваннях системного таймера. Тому кожному потоку дається певний інтервал часу, протягом якого він перебуває в активному стані.

Програми Java можуть вказувати три значення для пріоритетів потоків.

За замовчуванням, створений потік має нормальний пріоритет. Якщо інші потоки в системі мають той же самий пріоритет, то всі потоки користуються процесорним часом на рівних умовах.

У разі необхідності **теоретично** можна підвищити або знизити пріоритет окремих потоків. Потоки з підвищеним пріоритетом мають отримувати більше процесорного часу, а потоки зі зниженим пріоритетом теж отримують можливість працювати, але значно рідше.

10.2.1.3 **Методи класу Thread**

У класі Thread реалізовано багато методів і з ними можна познайомитися, відкривши вихідний текст класу. Тут же ми розглянемо тільки деякі з них.

Метод public void start () використовується для запуску потоку. Не слід його плутати з методом run(). Метод run() містить код, який повинен виконуватися в окремому потоці, а метод start() активізує метод run () та забезпечує його виконання в окремому потоці.

Звичайно, можна і просто викликати метод run (), і він теж буде виконуватися, але тільки не в окремому потоці.

Метод public final void setPriority (int newPriority) використовується для завдання пріоритету потоку. Як параметр в цей метод можна передавати константи NORM_PRIORITY, MAX_PRIORITY і MIN_PRIORITY.

Метод public static void sleep (long millis [, int nanos]) призупиняє потік на заданий час. Цей метод статичний, і його можна викликати через ім'я класу в будь-якому методі.

Метод public final void join() використовується для того, щоб призупинити поточний потік до завершення роботи потоку, якому надіслано повідомлення join(). Тобто join() ніби вставляє інший потік в поточний потік.

10.2.1.4 **Ще один спосіб створення потоку**

При роботі з GUI для відтворення компонентів і реакції на всі події компонентів створюється потік диспетчера подій Abstract Window Toolkit (AWT) з ім'ям AWT_EventQueue. Для того, щоб забезпечити коректну взаємодію між цим потоком та іншими потоками, що пов’язані з GUI, рекомендується створювати потоки, що зачіпають GUI, за допомогою статичного методу invokeLater утилітного класу SwingUtilities. Цей же метод можна викликати через клас EventQueue. Такий потік запуститься на виконання лише після того, як будуть оброблені всі події з черги java.awt.EventQueue. Ось приклад створення і запуску такого потоку:

```
SwingUtilities.invokeLater(new Runnable(){
```

```
        public void run() {
          System.out.println("Hello from SwingUtilities");
        }
      });
```

10.2.1.5 **Засоби для зупинки потоків**

Зазвичай потік зупиняється після завершення методу run (). Але іноді виникають ситуації коли хочеться перервати виконання потоку примусово.

Але в Java нема засобів для примусової зупинки потоку. Вірніше, вони є, але використовувати їх не варто - метод stop оголошений deprecated.

Причина цього полягає в тому, що після примусової зупинки потоку незрозуміло, що робити з ресурсами. Наприклад, потік, що обслуговує мережеве з'єднання з базою даних, зупинився під час робити з даними. Тоді хто буде закривати транзакцію хто буде розблокувати ресурси?

Тому - в Java прийнятий порядок зупинки потоку через повідомлення. Розглянемо цей механізм.

У класу Thread є метод - interrupt. Цей метод дозволяє виставити ознаку, що потік потрібно перервати. Цю ознаку можна перевірити за допомогою методу isInterrupted. Існує також статичний метод interrupted, який робить перевірку для поточного потоку. Виклик цього методу скидає ознаку, що має на увазі відповідальність розробника по обробці цієї ситуації.

Для контролю за перериванням потоку зручніше успадкувати потоки від Thread. У цьому випадку викликати метод interrupt не складе труднощів. Якщо ж використовувати Runnable, то доведеться зберігати посилання на пов'язаний з цим Runnable потік.

Метод interrupt виводить потік зі стану очікування, що було пов’язано з викликом методу sleep або wait, при цьому буде викинуто виключення InterruptedException. Ознака переривання в цьому випадку не виставляється, бо кинутий виняток і є такою ознакою.

Таким чином, якщо очікуються переривання потоку, то у ньому слід періодично вставляти перевірки isInterrupted. Якщо перевірка спрацювала або викинуто виключення під час очікування, треба виконати дії, що забезпечать коректне завершення потоку, після цього вийти з методу run.

10.2.2 *Приклад створення потоків*

10.2.3 *Створення потоку шляхом успадкування класу Thread*

Нижче наведено приклад створення такого потоку, де створено клас MyThread, що успадковує клас Thread, і в ньому перевизначено метод run (), у якому створено цикл накопичення 1 мільярду одиничок. Після накопичення суми результат виводиться на консоль.

```
public class MyThread extends Thread {
      public MyThread(String str) {
            super(str);
      }
      public void run(){
            System.out.println(this.getName() +" started.");
            double sum=0;
            for (long i = 0; i < 1000000000; i++) sum+=i;
            System.out.println(this.getName() +" accum sum "+ sum);
            System.out.println(this.getName() +" finished.");
      }
 }
```

Три потоки класу MyThread створюються і запускаються у класі MainForMyThread:

```
public class MainForMyThread {
     public static void main(String[] args) {
          Thread t1= new MyThread("MyThread_1"); t1.start();
          Thread t2= new MyThread("MyThread_2"); t2.start();
          Thread t3= new MyThread("MyThread_3"); t3.start();
     }
}
```

Якщо запустити кілька разів створений додаток, то можна побачити, що потоки стартують і завершують роботу у довільному порядку.

Цікавим також буде результат дослідження впливу пріоритетів на послідовність завершення роботи потоків

10.2.4 *Створення потоків з використанням інтерфейсу Runnable*

Створення потоку з використанням об’єкту типу Runnable вимагає схожих, але дещо інших дій.

Так само, як і попередньому випадку потрібно створити клас з методом run. Але цей клас має реалізовувати інтерфейс Runnable, а не успадковувати клас Thread.

Наступний крок, як і у попередньому випадку, полягає у створенні об’єкту класу Thread, але за допомогою іншого, непустого конструктора. Потрібно використовувати конструктор з параметром типу Runnable і через цей конструктор передати потоку об’єкт попередньо створеного класу.

Запуск створеного потоку реалізується так само через метод start.

Нижче, як приклад, наведено клас MyRunnable, у якому визначено поле для назви потоку, та посилання на інший потік, завершення якого має чекати даний потік. Таке чекання реалізується за допомогою методу join.

Роботу потік виконує таку саму, як і у попередньому прикладі, тобто знаходить суму великої кількості одиничок.

```
public class MyRunnable implements Runnable {
      Thread pred;
      String name;
      public MyRunnable(String name, Thread pred) {
            this.name = name;
            this.pred = pred;
      }
      public void run() {
            System.out.println(this.name + " started.");
            double sum = 0;
            for (double i = 0; i < 1000000000.0; i++) sum += i;
            System.out.println(this.name + " accum sum " + sum);
            if (pred != null)
                  try {
                        pred.join();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            System.out.println(this.name + " finished.");
      }
}
```

Для тестування створеного класу можна використати клас MainForMyRunnable, де створюється і запускається три потоки:

```
public class MainForMyRunnable {
      public static void main(String[] args) {
            Runnable r1= new MyRunnable("MyRunnable_1", null);
            Thread t1 = new Thread(r1); t1.start();
            Runnable r2= new MyRunnable("MyRunnable_2", t1);
            Thread t2= new Thread(r2);  t2.start();
            Runnable r3= new MyRunnable("MyRunnable_3", t2);
            Thread t3= new Thread(r3); t3.start();
      }
}
```

10.3 ПЛАНУВАННЯ РОБОТИ ПОТОКІВ У ЧАСІ

У Java є засоби, що дозволяють планувати виклики потоків у часі. Це може бути корисним у деяких застосуваннях, зокрема для реалізації мультиплікації.

10.3.1 *Класи Timer і TimerTask*

Об'єкти класу java.util.Timer використовуються для управління об'єктами класу TimerTask.

Для створення об’єктів класу Timer використовуються конструктори з різними характеристиками. Тут ми згадаємо тільки про один з них - public Timer (boolean isDaemon). Цей конструктор дозволяє встановити потік

пов'язаний з таймером, як демон. Потоки демони існують поки існує додаток, з яким вони пов'язані. Завершення додатки закриває і потік демон.

Створений таймер може обслуговувати скільки завгодно завдань, які є об'єктами класу TimerTask, і в яких визначається поведінка потоку, керованого таймером. В класі TimerTask заявлено інтерфейс Runnable, але метод run () не реалізовано. Цей метод повинен бути реалізований в дочірньому класі.

Завдання передаються таймеру за допомогою одного з методів schedule (), які відрізняються параметрами. Наприклад, метод public void schedule (TimerTask task, long delay, long period) забезпечує перший запуск завдання task через delay мілісекунд і повторні запуски через кожні period мілісекунд після першого запуску.

Для зупинки виконання завдання можна викликати метод cancel() класу TimerTask.

Особливість використання класів TimerTask і Timer полягає у тому, що методи run() об’єктів типу TimerTask мають виконуватися швидко. Бажано, щоб вони не містили викликів методів sleep(). Тільки при виконанні цієї умови об’єкт типу Timer зможе підтримувати одночасне виконання таких завдань. Інакше таймер працюватиме некоректно.

10.3.2 *Засоби пакету java.util.concurrent для планування роботи потоків*

У Java 7, у пакеті java.util.concurrent з’явилися нові засоби для планування роботи потоків у часі.

За допомогою утилітного класу-фабрики Executors можна створити об'єкт інтерфейсу ScheduledExecutorService, що реалізує пул потрібної кількості потоків.

Наприклад:

ScheduledExecutorService ses = Executors.newScheduledThreadPool(N);

Далі, за допомогою методу scheduleAtFixedRate(Runnable r), можна екзек’ютеру передавати об'єкти типу Runnable для виконання.

Наприклад:

ses.scheduleAtFixedRate(r1, 100, 14000, TimeUnit.MILLISECONDS)

де r1 - об'єкт типу Runnable,

100 – затримка до початку виконання,

1400 – період повторення,

TimeUnit.MILLISECONDS – константа, що визначає одиницю виміру часу.

Зупинити роботу пулу потоків можна за допомогою методу shutdown(). Наприклад:

ses.shutdown(), або ses.shutdownNow().

В останньому випадку виконання потоку буде перервано одразу і можливий викид винятку InterruptedException. У блоці обробки цього винятку можна реалізувати вихід з методу run() за допомогою оператора return.

Якщо кількість потоків у пулі не менша кількості завдань, то завдання

виконуються паралельно, незалежно від їх складності.

10.4 ОРГАНІЗАЦІЯ ВЗАЄМОДІЇ ПОТОКІВ, ЩО ВИКОНУЮТЬСЯ ПАРАЛЕЛЬНО

Основна складність, з якою стикаються програмісти, що створюють багатопотокові додатки, це організація взаємодії одночасно працюючих потоків.

Однопотокова програма, при запуску отримує в монопольне розпорядження всі ресурси комп'ютера і використовує ці ресурси в тій послідовності, яка відповідає логіці роботи програми.

В багатопотоковій системі можливі ситуації, коли деякий потік змушений чекати завершення виконання етапу алгоритму іншого потоку. Потоки можуть також намагатися звертатися одночасно до одних і тих же ресурсів, що може привести до неправильної роботи програми. Спроба одночасної модифікації спільно використовуваних даних може привести до порушення цілісності цих даних.

Таким чином, в багатопотоковому середовищі необхідна синхронізація спільної роботи потоків для забезпечення коректності використання критичних ресурсів.

В Java передбачені різні засоби для вирішення цього завдання.

10.4.1 *Синхронізація доступу до даних, що використовуються спільно*

Для захисту критичних ділянок програми використовується ключове слово synchronized. Використання ключового слова synchronized дає гарантію, що в даний момент часу якийсь оператор або блок буде виконуватися тільки в одному потоці. Інші потоки автоматично призупиняються при спробі звернення до ресурсу зайнятому іншим потоком.

Синтаксис створення блоку синхронізації при використанні критичного ресурсу object наведено нижче:

```
synchronized(object){
  // critical operations
}
```

У разі потреби можна синхронізувати і метод:

```
 public synchronized void anyMethod(){
     //тіло методу
}
```

Синхронізація використовується як засіб блокування потоків, який не дозволяє одному потоку спостерігати об'єкт в проміжному стані, поки той модифікується іншим потоком.

В межах синхронізованою області потрібно виконувати якомога менше роботи:

- заблокувати ресурс;
- перевірити дані, що використовуються спільно;
- опрацювати дані при необхідності;
- розблокувати ресурс.

У деяких випадках синхронізацію доступу до полів, що спільно використовуються можна опустити, якщо оголосити таке поле з модифікатором volatile. Цей модифікатор гарантує, що будь-який потік, який буде читати це поле, побачить найостанніше записане значення.

10.4.2 *Організація взаємодії потоків*

Як вже було сказано раніше, в багатопотоковій системі можливі ситуації, коли деякий потік змушений чекати завершення виконання етапу алгоритму іншого потоку. Коли це очікування пов'язане із синхронізацією, то зупинка потоку і відновлення його роботи здійснюється автоматично. В інших випадках доводиться програмувати такі зупинки.

Найпростіші способи призупинення роботи потоку вже були розглянуті. Це зупинки за допомогою методів sleep() і join() класу Thread.

Але досить часто доводиться зупиняти потоки до виконання деякої умови. Засобів для реалізації таких зупинок у Java зараз дуже багато.

Розглянемо використання таких засобів на прикладі обчислення суми елементів двовимірного масиву з рядками довільної довжини. Щоб прискорити роботу, будемо паралельно обчислювати суми рядків в окремих потоках. Після цього підсумуємо результати обчислень кожного рядка.

Для успішної реалізації цього прикладу необхідно, щоб головний потік спочатку створив потоки для паралельного обчислення сум рядків, а потім чекав, коли усі потоки повернуть результат своєї роботи. Після цього головний потік може продовжити обчислення.

Це досить простий приклад. Його можна реалізувати і за допомогою методу join. Але ми будемо на цьому прикладі розглядати інші способи.

Перш за все створімо метод для тестування прикладу:

```
public static void main(String[] args) {
    int[][] matr;
    Random rnd = new Random();
    //Випадкова кількість рядків
    int rows = rnd.nextInt(4) +2; //from 2 to 5
    matr = new int[rows][];
    // Створення рядків випадкової довжини
    for (int i = 0; i < matr.length; i++) {
       int cols = rnd.nextInt(1000) + 500;
       matr[i] = new int[cols];
```

```
            //Заповнення рядків випадковими числами
            for (int j = 0; j < cols; j++) {
                matr[i][j] = rnd.nextInt(100);
            }
        }
        // Створюємо паралельного обробника
        (new Thread(()->{
            ParallelCalc pc = new ParallelCalc(matr);
            int sum = pc.calcFullSum();
            System.out.println(sum);
         })).start();
    }
```

Зверніть увагу, паралельного обробника ми теж запускаємо у окремому потоці. Це доцільно робити завжди, бо інакше додаток, у якому викликається потоковий обробник може зупинитися і чекати завершення роботи обробника, що не завжди доречно.

Наступним кроком створімо клас ParallelCalc, об’єкт якого забезпечить паралельну обробку двовимірного масиву через метод calcFullSum().

```
public class ParallelCalc {
  int[][] matr;
  int[] arr; // Масив сум кожного з рядків

  // Конструктор, що приймає масив на обробку
  public ParallelCalc(int[][] matr) {
     this.matr = matr;
     arr = new int[matr.length];
     Arrays.fill(arr, -1);
  }

  // Обчислення суми для одновимірного масиву
  public int calcSum(int[] arr) {
     int sum = 0;
     for (int x : arr)
        sum += x;
     return sum;
  }

  // Тестування завершення роботи
  private boolean isReady() {
     for (int x : arr)
        if (x == -1)
           return false;
     return true;
  }
```

```
    // Клас для створення обробників рядків,
    // що будуть працювати паралельно
    private class SubCalc implements Runnable {
        int i;

        public SubCalc(int i) {
            this.i = i;
        }

        @Override
        public void run() {
            arr[i] = calcSum(matr[i]);
        }

    }

    // Паралельне обчислення суми
    // для двовимірного масиву
    public int calcFullSum() {
        for (int i = 0; i < arr.length; i++) {
            Runnable task = new SubCalc(i);
            Thread t = new Thread(task);
            t.start();
        }
        // Тут треба зупинитись і чекати,
        // щоб метод isReady() повернув true
        while(!isReady()){
            //??????????????????????????
        }
        return calcSum(arr);
    }
}
```

Поля, конструктор і методи, потрібні незалежно від способу організації очікування. Але реалізація методу calcFullSum та методу run класу SubCalc може змінюватися, залежно від реалізації очікування. Окрім того може виникнути потреба у додаткових елементах класу.

Слід зазначити, що наведена реалізація буде працювати, але так організовувати очікування не слід. Адже цикл while у останньому методі буде виконуватися без зупинки і суттєво навантажуватиме процесор безглуздою роботою.

10.4.2.1 **Методи suspend та resume**

У першій версії Java для організації такої взаємодії використовувалися методи suspend та resume класу Thread. Виклик методу suspend у поточному потоці призводив до зупинки виконання коду цього потоку. Для відновлення роботи потоку потрібно було викликати для призупиненого потоку метод resume з іншого потоку. Зазвичай, це мав робити потік, від якого залежало виконання умови. Але вже у другій версії Java методи suspend та resume було визначено deprecate, тому що їх використання у деяких ситуаціях призводило до взаємоблокування потоків (deadlock).

Приклад з використанням цих методів наводити не будемо.

З другої версії Java для тимчасової зупинки потоку до виконання певної умови рекомендується використовувати метод wait() і notify() класу Object.

10.4.2.2 **Методи wait() і notify()**

Метод final void wait () застосовується в тому випадку, коли потрібно змусити потік дочекатися певної умови.

Оскільки як цей метод визначений у класі Object, то він може бути викликаний для будь-якого об'єкта. Єдина вимога до цього об'єкта полягає в тому, що б цей об'єкт на момент виклику методу wait () був синхронізований. В документації такі об'єкти називають моніторами і потік повинен монопольно володіти цим монітором в момент виклику методу wait ().

Для виведення потоку з призупиненого стану використовується метод notify(), який надсилається тому ж монітору, для якого був викликаний метод wait(). Монітор на момент виклику методу notify() теж повинен бути синхронізований.

Виклик методу wait(), як правило, здійснюється всередині циклу, де перевіряється умова очікування. Якщо умова виконується, то метод wait() не викликається. Саме цикл, а не if, потрібен для забезпечення безпеки.

Існує кілька причин, за якими потік може «прокинутися» при невиконанні умов:

– За час від моменту, коли потік викликає метод notify, і до того моменту, коли прокинеться потік, що очікує виконання умови, інший потік може встигнути змінити параметри умови.

– Інший потік може викликати notify, коли умова ще не виконана.

– Потік, що сповіщає про виконання умови, потік може викликати notifyAll, навіть якщо умова пробудження виконано лише для деяких потоків, що чекають.

– Потік може прокинутися і при відсутності сповіщення. Це називається помилковим пробудженням. У багатьох реалізаціях JVM застосовуються механізми управління потоками, у яких помилкові пробудження хоча й не часто, але трапляються.

Стандартна схема використання методу wait виглядає так:

```
synchronized (obj) {
 try {
```

```
      while (< умова не виконана >)
        obj.wait();
    } catch (InterruptedException e) {
      // обробка винятку переривання потоку
    }
      //Виконання дій, що відповідають умові
  }
```

Недолік використання цієї технології полягає у особливостях методів notify та notifyAll. Якщо з монітором пов'язано декілька призупинених потоків, то невідомо, який потік відновить роботу після виклику методу notify для монітору. Якщо ж надсилається метод notifyAll, то активізуються усі потоки, хоча, можливо, у цьому і нема потреби.

Розглянемо використання розглянутих методів для реалізації наведеного вище прикладу.

```
// Монітор для wait() та notify()
Object monitor = new Object();

// Клас для створення обробників рядків,
// що будуть працювати паралельно
private class SubCalc implements Runnable {
   int i;

   public SubCalc(int i) {
      this.i = i;
   }

   @Override
   public void run() {
      arr[i] = calcSum(matr[i]);

      synchronized (monitor) {
         monitor.notify();
      }
   }
}

// Паралельне обчислення суми
// для двовимірного масиву
public int calcFullSum() {
   for (int i = 0; i < arr.length; i++) {
      Runnable task = new SubCalc(i);
      Thread t = new Thread(task);
      t.start();
   }
```

```
// Тут треба зупинитись і чекати,
    // щоб метод isReady() повернув true
    synchronized (monitor) {
        while (!isReady()) {
            try {
                monitor.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    return calcSum(arr);
}
```

10.4.2.3 Організація взаємодії потоків через об'єкти класів Lock та Condition

Використання об'єктів типу Lock та Condition підвищує гнучкість та ефективність реалізації зупинок до виконання умови. Об'єкт типу Lock може містити у собі декілька посилань на об'єкти типу Condition.

Фактично, об'єкт типу Lock дозволяє створити блок синхронізації, межі якого визначаються викликами методів lock() та unLock() інтерфейсу Lock. Оператори синхронізованого блоку рекомендується записувати у межах конструкції try{} finally{}, а метод unLock() викликати у блоці finally{}.

У межах синхронізованого блоку можна звертатися до об'єктів типу Condition, які фактично є моніторами, що пов'язані зі своїм об'єктом Lock. Для створення цих моніторів використовується метод newCondition() інтерфейсу Lock.

Для призупинення потоку використовується метод await(), що надсилається об'єкту Condition.

Для відновлення роботи потоку слід надвслати об'єкту Condition повідомдення signal().

Слід розуміти, що об'єкти типу Condition, нічого не знають про умови, з якими їх пов'язує програміст. Це просто монітори.

Інтерфейс Lock реалізовано у класі ReentrantLock.

Розглянемо використання цієї технології для реалізації наведеного вище прикладу.

```
// Створення об'єктів типу Lock та Condition
Lock synchronizer  = new ReentrantLock();
Condition ready = synchronizer.newCondition();

// Клас для створення обробників рядків,
// що будуть працювати паралельно
private class SubCalc implements Runnable {
```

```
    int i;

    public SubCalc(int i) {
        this.i = i;
    }

    @Override
    public void run() {
        arr[i] = calcSum(matr[i]);
        synchronizer.lock();
        try {
            ready.signal();
        } finally {
            synchronizer  .unlock();
        }
    }
}

// Паралельне обчислення суми
// для двовимірного масиву
public int calcFullSum() {
    for (int i = 0; i < arr.length; i++) {
        Runnable task = new SubCalc(i);
        Thread t = new Thread(task);
        t.start();
    }
    // Тут треба зупинитись і чекати,
    // щоб метод isReady() повернув true
    synchronizer  .lock();
    try {
        while(!isReady()){
            ready.await();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        synchronizer  .unlock();
    }

    return calcSum(arr);
}
```

10.4.2.4 **Використання об’єктів типу CountDownLatch**

Об’єкт типу CountDownLatch можна розглядати як замок, закритий на декілька обертів, або як об’єкт, що містить певну кількість заборон на

продовження виконання коду. Кількість цих заборон встановлюється конструктором. Зупинка перед цим об’єктом забезпечується викликом його методу await(). Кількість заборон можуть зменшувати інші потоки, викликаючи метод countDown(). Виконання призупиненого потоку буде продовжено після того, як кількість заборон зменшіться до нуля.

Нижче розглянуто реалізацію прикладу з використанням об’єкту CountDownLatch.

```
// Визначення об'єкту  CountDownLatch
CountDownLatch latch;

// Клас для створення обробників рядків,
// що будуть працювати паралельно
private class SubCalc implements Runnable {
    int i;

    public SubCalc(int i) {
        this.i = i;
    }

    @Override
    public void run() {
        arr[i] = calcSum(matr[i]);
        latch.countDown();
    }
}

// Паралельне обчислення суми
// для двовимірного масиву
public int calcFullSum() {
// Створення об'єкту  CountDownLatch
CountDownLatch latch = new CountDownLatch(arr.length);
    for (int i = 0; i < arr.length; i++) {
        Runnable task = new SubCalc(i);
        Thread t = new Thread(task);
        t.start();
    }
    // Тут треба зупинитись і чекати,
    // щоб метод isReady() повернув true
    try {
        latch.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return calcSum(arr);
```

```
}
```

10.4.2.5 **Використання об’єктів типу CyclicBarrier**

Цей об`єкт дещо схожий на попередній. Він також містить деяку кількість перепон, що задається у конструкторі, і які встановлюються на шляху потоків. Але технологія використання дещо інша. Кожний з потоків викликає у об’єкту CyclicBarrier метод await(), зменшуючи тим самим кількість перепон на одиницю, і зупиняється до моменту, коли кількість перепон стане дорівнювати нулю. Отже, після відкриття бар’єру усі потоки, що чекали цієї події, відновлюють свою роботу одночасно.

Ще одна особливість бар’єру полягає у тому, що його можна повертати у початковий стан за допомогою методу reset().

Цей об’єкт також можна використати для вирішення задачі, що була сформована на початку підрозділу.

```
// Визначення об'єкту типу CountDownLatch
CyclicBarrier barrier;
// Клас для створення обробників рядків,
// що будуть працювати паралельно
private class SubCalc implements Runnable {
    int i;

    public SubCalc(int i) {
        this.i = i;
    }

    @Override
    public void run() {
        arr[i] = calcSum(matr[i]);
        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}

// Паралельне обчислення суми
// для двовимірного масиву
public int calcFullSum() {
    // Створення об'єкту типу CyclicBarrier
    barrier = new CyclicBarrier(arr.length + 1);
    for (int i = 0; i < arr.length; i++) {
        Runnable task = new SubCalc(i);
        Thread t = new Thread(task);
```

```
            t.start();
        }
        // Тут треба зупинитись і чекати,
        // щоб метод isReady() повернув true
        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
        return calcSum(arr);
    }
```

10.4.2.6 **Використання об’єктів типу Phaser**

Використовується поняття parties (ті, хто приймає участь), їх кількість визначається конструктором

Найчатіше використовуються методи arriveAndDeregister();

```
    arriveAndAwaitAdvance();
    register()
    bulkRegister(parties)
```

```
    // Визначення об'єкту типу Phaser
    Phaser phaser;
    // Клас для створення обробників рядків,
    // що будуть працювати паралельно
    private class SubCalc implements Runnable {
        int i;

        public SubCalc(int i) {
            this.i = i;
        }

        @Override
        public void run() {
            arr[i] = calcSum(matr[i]);
            phaser.arriveAndDeregister();
        }
    }

    // Паралельне обчислення суми
    // для двовимірного масиву
    public int calcFullSum() {
        // Створення об'єкту типу CyclicBarrier
        phaser = new Phaser(arr.length + 1);
        for (int i = 0; i < arr.length; i++) {
```

```
            Runnable task = new SubCalc(i);
            Thread t = new Thread(task);
            t.start();
        }
        // Тут треба зупинитись і чекати,
        // щоб метод isReady() повернув true
        phaser.arriveAndDeregister();

        return calcSum(arr);
    }
```

10.5 РІВНІ ПОТОКОВОЇ БЕЗПЕЧНОСТІ КЛАСІВ

Нижче наводяться рівні безпеки, які можуть забезпечувати класи при роботі з декількома потоками.

Незмінюваний (immutable). Екземпляри такого класу виглядають для своїх клієнтів як константи. Ніякої зовнішньої синхронізації не потрібно. Екземпляри цього класу не можна поміняти. Вся інформація, що міститься в будь-якому його екземплярі, записується в момент його створення і залишається незмінною протягом усього часу існування цього об'єкта. В бібліотеці платформи Java є цілий ряд незмінних класів, в тому числі String, класи-оболонки, BigInteger і BigDecimal. Незмінні класи легше розробляти і використовувати, вони менш схильні до помилок і більш надійні. Єдиний справжній недолік незмінних класів полягає в тому, що для кожного унікального значення їм потрібен окремий об'єкт.

З підтримкою багатопоточності (thread-safe). Екземпляри такого класу можуть змінюватися, проте всі методи мають досить надійну внутрішню синхронізацію, щоб ці екземпляри могли паралельно використовувати кількома потоками без будь-якої зовнішньої синхронізації. Приклади: Random і java.util.Timer.

З умовної підтримкою багатопоточності (conditionally thread-safe). У таких класах деякі методи потребують синхронізації. Наприклад, класи HashTable і Vector будуть підтримувати багатопоточність при умові зовнішньої синхронізації ітераторів.

Сумісний з багатопоточність (thread-compatible). Примірники такого класу можна безпечно використовувати при роботі з паралельними потоками, якщо виклик кожного методу оточити зовнішньою синхронізацією. Прикладом є реалізації колекцій загального призначення, такі як ArrayList і HashMap.

Несумісний з багатопоточність (thread-hostile). Цей клас небезпечний при паралельній роботі з декількома потоками, навіть якщо виклики всіх методів оточені зовнішньою синхронізацією. Зазвичай несумісність пов'язана з тією обставиною, що ці методи міняють деякі статичні дані, які впливають на інші потоки. Прикладом може слугувати метод System.runFinalizerOnExit несумісний з багатопоточність і визнаний застарілим.

Для кожного класу потрібно чітко документувати можливість роботи з декількома потоками. Документуванню класів з умовною підтримкою багатопоточності необхідно приділяти особливу увагу.

10.6 ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Загальні відомості про потоки виконання коду.
2. Способи створення потоків в Java.
3. Конструктори класу Thread.
4. Методи класу Thread.
5. Пріоритети потоків.
6. Інтерфейс Runnable, його призначення і використання.
7. Засоби для зупинки потоків.
8. Що таке синхронізація.
9. Призначення та використання методів wait a notify.
10. Рівні потокової безпечності класів.
11. Використання класів Timer і TimerTask для створення потоків.
12. Використання класу ScheduledExecutorService для планування роботи потоків у часі.
13. Реалізувати дитячу казочку про дітей сороки-ворони, що кашку варила. Кожен з дітей виконував одну з таких робіт: крупу драв, дрова рубав, воду носив, піч топив. А сорока-ворона кашку варила. Потім діти їли кашу. Щоб піч топити потрібні дрова. Щоб кашу варити потрібні крупи, вода і розтоплена піч. А кашу можна їсти після того, як вона звариться. Для реалізації створити основний клас з методом main, де створюються та запускаються потоки. У цьому ж класі створити один або декілька внутрішніх клас для потоків, що реалізують дії дітей та сороки-ворони. У конструктор класу передавати ім’я потоку та об’єкт функціонального інтерфейсу BooleanSupplier, що задає умову початку роботи потоку і буде використовуватися для зупинки за допомогою методу wait. Якщо робота починається без умови передавати null. Можна для очікування, поки буде готова каша, використовувати метод join. Для затримки на виконання роботи використовувати обчислення рандомного числа Фібоначчі від 40 до 45.

11 РЕАЛІЗАЦІЯ МЕХАНІЗМУ ОБРОБКИ ПОДІЙ В JAVA

11.1 ШАБЛОН ПРОЕКТУВАННЯ OBSERVER (СПОСТЕРІГАЧ)

В теорії ООП існує шаблон проектування Observer (спостерігач). Цей шаблон застосовується в тих випадках, коли система володіє такими властивостями:

– існує, як мінімум, один об'єкт, що розсилає повідомлення;

– є не менше одного одержувача повідомлень, причому їхня кількість і склад можуть змінюватися під час роботи програми.

Цей шаблон часто застосовують в ситуаціях, в яких відправника повідомлень не цікавить, що роблять одержувачі з наданою їм інформацією.

При реалізації шаблону «спостерігач» зазвичай використовуються такі класи, рисунок 11.1:

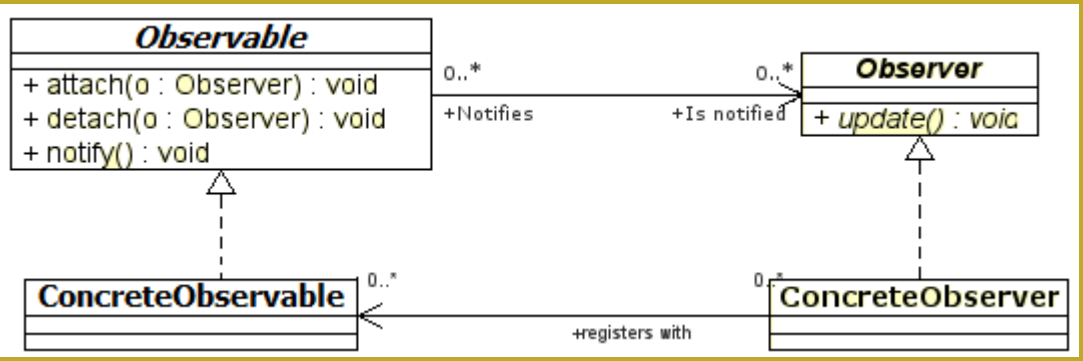


Рисунок 1.1 – Діаграма класів шаблону Observer

Призначення класів діаграми таке:

– Observable – інтерфейс, що визначає методи для додавання, видалення та оповіщення спостерігачів.

– Observer – інтерфейс, за допомогою якого спостережуваний об'єкт оповіщає спостерігачів.

– ConcreteObservable – конкретний клас, який реалізує інтерфейс Observable.

– ConcreteObserver – конкретний клас, який реалізує інтерфейс Observer.

При зміні стану об'єкту, за яким ведеться спостереження, оповіщення спостерігачів може бути реалізоване за такими сценаріями:

– Спостережуваний об'єкт надсилає, кожному із зареєстрованих спостерігачів, всю потенційно важливу інформацію (примусове розповсюдження).

– Спостережуваний об'єкт надсилає, кожному із зареєстрованих спостерігачів, лише повідомлення про те що інформація була змінена, а кожен із спостерігачів, за необхідності, самостійно здійснює запит необхідної інформації у спостережуваного об'єкта (розповсюдження за запитом).

11.2 РЕАЛІЗАЦІЯ ШАБЛОНУ “СПОСТЕРІГАЧ” В JAVA

В Java шаблон “спостерігач” використовується для створення механізму обробки подій, що відбуваються з об’єктами. Але внаслідок того, що опис шаблону з’явився пізніше, ніж з’явилася перша версія Java, то назви інтерфейсів, класів та методів не відповідають рисунку 6.1, хоча суть залишається та сама.

Розглянемо особливості реалізації шаблону «спостерігач» в Java та поняття, що пов’язані з цим.

Подія (event) – це якась дія користувача або зміна стану об'єктів додатку, які можуть викликати у відповідь дії іншої програми.

Це своєрідний спосіб передачі інформації від одних об'єктів до інших.

При цьому ми маємо справу з:

– об'єктом, який породжує (fires), подію;
– власне подією, яка теж є об'єктом;
– об'єктом, який реагує на подію (listener, слухач).

Однією з причин появи поняття «подія» в програмуванні з'явилася можливість написання узагальненого коду і, як наслідок, поява фреймворків і бібліотек стандартних компонентів. На етапі розробки таких засобів програмування, з одного боку, очевидно, що деякі події можуть вимагати обробки, з іншого боку, ще не відомо, хто і як буде реагувати на ці події. Як наслідок, процес програмування, пов'язаний з формуванням події, відокремився від програмування обробки події. Ці процеси відбуваються в різний час і в різних місцях. Причому, код програми, де формується подія, зазвичай недоступний програмісту, який займається обробкою події.

Таким чином виникає декілька проблем, що потребують вирішення, а саме:

– як слухачеві дізнатися про подію?
– як джерелу дізнатися про слухача?
– як врахувати особливості реакції слухача на подію?

Для вирішення цих проблем Java пропонує спеціальний механізм і відповідні засоби. Зрозуміло, що тут необхідні зусилля обох сторін, і джерела, і слухача.

11.2.1 *Клас java.util.EventObject*

Це клас, який може використовуватися безпосередньо для створення найпростіших подій, або як базовий для створення класів подій зі складнішою структурою.

Клас EventObject досить простий, він містить посилання на об'єкт, в якому подія відбулася (source), та методи доступу до цього об'єкта. Оригінальний текст класу представлений нижче.

```
public class EventObject implements java.io.Serializable {

    private static final long serialVersionUID = 5516075349620653480L;

    /**
     * The object on which the Event initially occurred.
     */
    protected transient Object  source;

    /**
     * Constructs a prototypical Event.
     *
     * @param    source
     * The object on which the Event initially occurred.
     * @exception  IllegalArgumentException  if source is null.
     */
    public EventObject(Object source) {
        if (source == null)
            throw new IllegalArgumentException("null source");

        this.source = source;
    }

    /**
     * The object on which the Event initially occurred.
     *
     * @return   The object on which the Event initially occurred.
     */
    public Object getSource() {
        return source;
    }

    /**
     * Returns a String representation of this EventObject.
     *
     * @return  A a String representation of this EventObject.
     */
    public String toString() {
        return getClass().getName() + "[source=" + source + "]";
    }
}
```

Слід зауважити, що в найпростіших випадках об'єкт подію можна взагалі не створювати. Також не обов'язково, щоб класи подій успадковували клас EventObject. Однак, правильніше буде слідувати рекомендаціям розробників мови.

11.2.2 *Дії розробника джерела подій, що надають можливість слухачеві реагувати на події*

– Створити клас для події. В цьому класі крім посилання на джерело події можна передбачити поля для зберігання інформації, що характеризує подію і методи доступу до неї.

– Створити програмний інтерфейс спілкування зі слухачами. Наприклад, ChangeDataListener. Доцільно (але необов'язково) успадковувати стандартний інтерфейс EventListener. Створений інтерфейс буде використовуватися як тип для слухачів події і має визначати сигнатуру методу обробки події. Як параметр у такі методи звичайно передається посилання на подію. Якщо з об'єктом можуть відбуватися різні події, то інтерфейс можна створювати один, але в ньому на кожну подію має бути визначений свій метод. Стандартно такі методи починаються префіксом on, наприклад onChangeData (ChangeDataEvent).

– У класі джерела події створити сховище для посилань на слухачів події, наприклад, changeDataListeners. Для цієї мети можна використовувати навіть масив, але це не дуже зручно. У початкових версіях Java для цього зазвичай використовували колекції синхронізованого класу Vector. Зараз для цього рекомендують використовувати потокобезпечні колекції типу CopyOnWriteArrayList. Для створення самої колекції доцільно використовувати метод відкладеної ініціалізації, при якому колекція буде створюватися при появі першого слухача.

– Створити методи додавання і видалення слухачів, які дадуть можливість слухачам повідомити про себе (зареєструватися), наприклад addChangeDataListener(ChangeDataListener) і removeChangeDataListener (ChangeDataListener).

– Створити метод, що сповіщає зареєстрованих слухачів про виникнення події, викликаючи відповідний метод створеного інтерфейсу для кожного слухача.

11.2.3 *Заходи слухачів, що надають можливість реагувати на події*

– Реалізувати стандартний інтерфейс спілкування з джерелом події, створивши метод реакції на подію. Але інтерфейс слухача події зазвичай не реалізують в основному класі. Для цього створюють внутрішній клас, найчастіше безіменний, за допомогою якого створюють обробник події (handler).

– Далі слід зареєструватися або зареєструвати обробник в джерелі події, використовуючи його метод add ... Listener.

Такий механізм реалізується при обробці стандартних подій візуальних компонентів і він вам відомий.

11.3 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Що така подія, клас EventObject.
2. Заходи джерела подій, які надають можливість реагувати на події.
3. Заходи слухачів, що надають можливість реагувати на події.
4. Відтворити послідовність проходження інформації про подію від виникнення до його обробки.

12 ЗНАЙОМСТВО З БІБЛІОТЕКОЮ SWING

12.1 ДІАГРАМА КЛАСІВ ГРАФІЧНИХ КОМПОНЕНТІВ

Для побудови графічних інтерфейсів користувача у мові Java існує декілька бібліотек. Бібліотека Swing – одна з них. Більшість класів, що забезпечують можливість створення графічних компонент, розташовані в пакеті javax.swing. Окрім того, використовуються деякі класи застарілої бібліотеки java.awt. На рисунку 12.1 зображено спрощену діаграму деяких класів графічних бібліотек java.awt та java.swing.

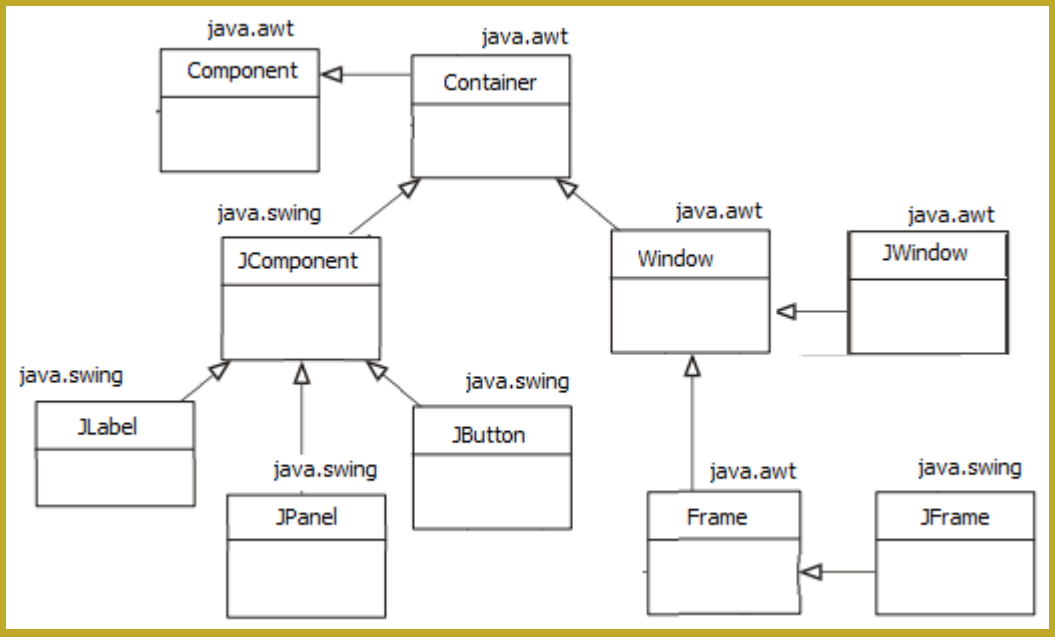


Рисунок 12.1 – Ієрархія деяких класів графічних компонентів

Суперклас java.awt.Component є абстрактним класом, у якому визначені основні атрибути візуальних компонентів. Клас містить велику кількість методів для створення компонентів GUI та подій, що пов'язані з ними.

Породжений від нього підклас Container містить методи типу add (), які дозволяють розташовувати на ньому інші компоненти (об'єкти) і відповідає за їх розміщення.

Від класу Container йде розгалуження на дві групи класів. Гілка, що починається від класу Window, використовується для створення вікон верхнього рівня. Це вікна, які не міститься в іншому вікні. Такі вікна в Java називають фреймами. Для створення фреймів застосовується клас Frame. За його допомогою створюється стандартне вікно. Клас Frame в бібліотеці Swing був розширений класом JFrame.

Друга гілка ієрархії починається класом JComponent. На його основі створено класи, за допомогою яких створюються елементи графічного інтерфейсу.

12.2 ГРАФІЧНІ КОНТЕЙНЕРИ

12.2.1 *Клас javax.swing.JFrame*

Цей клас є основою графічного інтерфейсу. Об’єкти цього класу являють собою звичайні вікна, тобто рамочка з елементами керування, заголовком та іконкою. Ці елементи можна налаштовувати за допомогою відповідних методів. Наприклад:

– метод setTitle(String) налаштовує заголовок вікна;

– метод setResizable(boolean) встановлює, чи можна змінювати розміри вікна.

До складу фрейму завжди входить панель, яка зветься contentPane. Призначення цієї панелі – містити на собі інші елементи управління або панелі. Після створення фрейму ця панель є об’єктом класу JPanel. Доступ до цієї панелі забезпечується методом getContentPane(). За бажанням панель можна змінити на будь який об’єкт типу контейнер, використавши метод setContentPane(Container).

12.2.2 *Класи панелей*

Панелі можна вважати контейнерами наступного за JFrame рівня.

Найбільш популярним контейнером мабуть є панель класу JPanel. На таких панелях зазвичай розташовують групи елементів, об’єднаних спільною задачею. Таке розташування елементів дуже зручне, тому що уся група може розглядатися як одне ціле. Панель не має видимих границь, але за бажанням можна за допомогою методу setBorder(Border) можна такі границі створити.

Панель типу JScrollPane використовується для розташування на ній компонентів, які потребують скролінгу.

Панель JTabbedPane є основою для створення групи панелей із закладками.

Окрім наведених панелей є і інші. З ними можна познайомитися, відкривши вікно візуального редактора.

12.3 МЕНЕДЖЕРИ КОМПОНУВАННЯ

З панелями тісно пов'язано використання менеджерів компоновки (layout manager). Ці об'єкти дозволяють автоматично визначати місце розташування і розміри компонентів всередині панелі і можуть підлаштовувати розміри і положення компонентів у разі зміні розмірів батьківської панелі. Існує кілька типів менеджерів розташування, кожен з яких працює за певним правилом.

Менеджер компоновки для панелі встановлюється методом setLayout(LayoutManager). Нижче розглянемо деякі з них:

– BorderLayout, призначений для розташування 5-ти елементів, що розташовуються, відповідно, в центрі панелі і у 4-х напрямках від центру. Часто використовується для додатків, що виводять основну інформацію в центрі, а решту зон використовують для різного роду елементів управління та рядків додаткової інформації. Розміри кожної з цих 5-ти частин автоматично

задаються пропорційними розміру панелі. Цей менеджер встановлюється за замовчуванням на панелі JPanel.

– CardLayout, менеджер, який розташовує елементи один над іншим, як колоду карт, при цьому на екрані видно тільки останній. Кожний елемент розтягується на всю панель. Для послідовного перегляду (відкриття) кожного елементу використовуються його методи next () і previous ().

– GridLayout, ґратчастий менеджер, який розташовує елементи у таблиці із заданим числом рядків і стовпців. Розміри всіх комірок при цьому однакові, а розмір елементів, що додаються, підлаштовується під розмір комірки.

– FlowLayout, найпримітивніший менеджер. Він розташовує елементи, що додаються, один за одним в рядок. Якщо в рядку не вистачає місця, заповнюється наступний рядок.

– BoxLayout, аналог FlowLayout, але розташовує елементи в напрямку, який задається користувачем (X-горизонталь або Y-вертикаль). На відміну від GridLayout, тут кожен елемент може займати комірку різного розміру.

– GridBagLayout, "розумний" ґратчастий менеджер, що дозволяє задавати індивідуальні властивості кожної комірки, а також об'єднувати кілька комірок для розташування якого-небудь великого елементу. Всі правила розміщення компонентів задаються в об'єкті іншого класу - GridBagConstraints. Цей клас містить одинадцять полів, таблиця 12.1, що визначають розміри компонентів, їх положення в контейнері і взаємне положення, та кілька констант - значень деяких полів. Ці параметри визначаються конструктором, який має одинадцять аргументів, або встановлюються за замовчуванням, якщо використовувати пустий конструктор.

Таблиця 12.1 – Поля класу GridBagConstraints

Поле	Призначення поля
anchor	Напрямок розміщення компонента в контейнері. Константи: CENTER, NORTH, EAST, NORTHEAST, … За замовчуванням CENTER
fill	Управління зміною розміру компонента для заповнення комірки. Константи: NONE, HORIZONTAL, VERTICAL, BOTH. За замовчуванням NONE
gridheight gridwidth	Кількість комірок, що займає компонент у колонці або рядку. Ціле типу int, за замовчуванням 1. Константа REMAINDER означає, що компонент займе залишок колонки, RELATIVE - буде наступним по порядку в колонці.
gridx gridy	Номер комірки в рядку або колонці. Перша комірка має номер 0. За замовчуванням константа RELATIVE, що означає: наступна по порядку.
insets	Поля навкруги контейнера. Об’єкт класу Insets; За замовчуванням поля дорівнюють нулю.

Продовження таблиці 12.1

Поле	Призначення поля
ipadx, ipady	Горизонтальні та вертикальні поля навколо компонентів. За замовчуванням 0.
weightx, weighty	Коефіцієнти пропорційного розтягнення компонентів при зміні розміру контейнера. За замовчуванням 0,0

12.4 ЕЛЕМЕНТИ УПРАВЛІННЯ

Java пропонує декілька класів для реалізації кнопок. Усі вони є спадкоємцями класу AbstractButton. У таблиці 12.2 наведено перелік цих класів та їх призначення.

Таблиця 12.2 – Класи для реалізації кнопок

Клас	Призначення
JButton	Звичайна кнопка.
JCheckBox	Перемикач.
JRadioButton	Радіоклавіша, які зазвичай об’єднують у групу.
JMenuItem	Елемент меню.
JCheckBoxMenuItem	Елемент меню з перемикачем.
JRadioButtonMenuItem	Елемент меню з радіоклавішею.
JToggleButton	Реалізує функціональність JCheckBox та JRadioButton.

Найпопулярніший клас – це клас JButton, який реалізує звичайну кнопку. Напис на кнопці налаштовується методом setText(String) або через вікно properties. Подія натискання на клавішу має назву actionPerformed.

Клас JRadioButton дозволяє створити групу радіоклавіш для реалізації багатоваріантного вибору.

При роботі з радіоклавішами слід враховувати, що для об'єднання радіоклавіш в одну групу необхідно використовувати об'єкт типу GroupButton.

Найпростіше створити групу радіоклавіш за допомогою візуального редактора. Для цього слід виділити групу об'єднуються радіоклавіш використовуючи комбінацію клавіші Ctrl і MouseClickc, а потім в контекстному меню вибрати функцію setButtonGroup. Цей об'єкт можна прописати і вручну, використовуючи конструктор цього класу, і потім кожну радіоклавішу після створення додати в цю групу, використовуючи метод add.

Перевірити стан кнопки можна за допомогою методу isSelected(). Вибір кнопки фіксується подією actionPerformed.

Перевірити стан кнопок у групі можна за допомогою такого коду:

```
Enumeration enm = buttonGroup.getElements();
while (enm.hasMoreElements()) {
	JRadioButton rb = (JRadioButton) enm.nextElement();
	if (rb.isSelected())
		System.out.println(rb.getText()+" isSelected");
}
```

Класи JCheckBox та ToggleButton дозволяють створювати кнопки, що можуть перебувати у двох станах – включено та виключено. Перевірити стан кнопки можна також за допомогою методу isSelected(). Вибір кнопки фіксується подією actionPerformed.

12.4.1 *Кнопки меню*

Кнопки меню є об'єктами класу JMenuItem, що створений на основі звичайної кнопки типу JButton. Відповідно до цього і обробка подій для кнопок меню така ж сама, як і для звичайних кнопок.

Але кнопка меню не використовується сама по собі. Вона є складовою частиною об'єктів меню. Для реалізації меню можна використовувати декілька класів.

Клас JMenuBar використовується для створення головного меню додатку. Об'єкт цього класу стандартно розташовується у верхній частині вікна, займаючи весь рядок. До цього об'єкту стандартно додаються об'єкти типу JMenu, які можуть використовуватися як кнопки, але частіше використовується для розміщення груп елементів типу JMenuItem.

Клас JPopupMenu використовується для створення контекстного меню. Контекстне меню можна пов'язати з різними компонентами. При цьому е до класу додатку автоматично додається статичний метод addPopup. Після створення контекстного меню можна додати до нього потрібну кількість об'єктів типу JMenuItem – кнопок меню. Для зручності користування до меню можна додавати роздільники – об'єкти типу JSeparator.

Окрім кнопок JMenuItem у меню можна використовувати кнопки типу JRadioButtonMenuItem, які є аналогами радіоклавіш.

Більш докладну інформацію про кнопки можна знайти на сайті http://docs.oracle.com/javase/tutorial/uiswing/components/button.html.

12.5 ЕЛЕМЕНТИ ДЛЯ РОБОТИ З ТЕКСТОМ

Java містить декілька класів (Рисунок 12.2) для роботи з текстом.

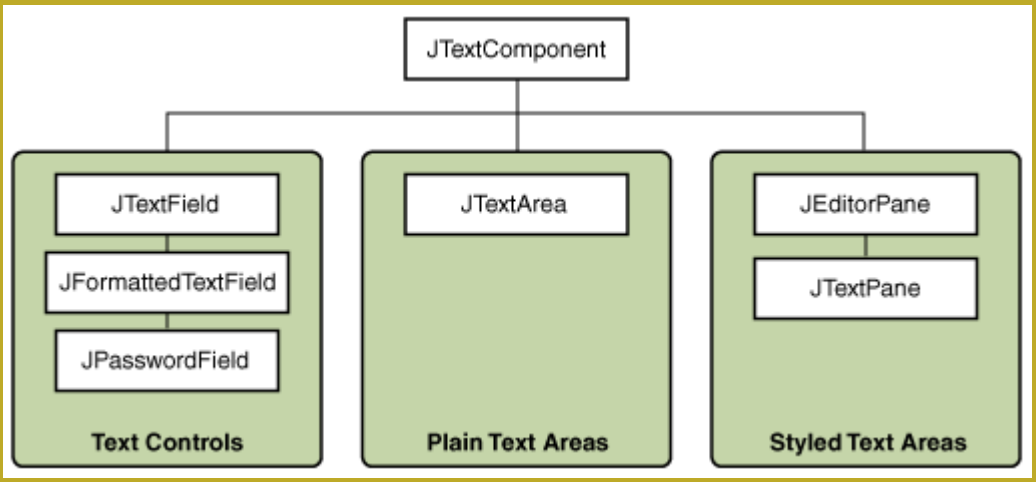


Рисунок 12.2 – Діаграма класів, що забезпечують роботу з текстом

Клас JTextField дозволяє вводити та зчитувати однорядкові тексти за допомогою методів setText(String) та getText(). Зміни тексту викликають подію CaretUpdate.

Клас JTextArea дозволяє працювати з багаторядковим текстом. Тут також працюють методи setText(String) та getText(). Окрім того є метод append(String), за допомогою якого можна додавати текст.

Більш докладно з цими класами та прикладами їх використання можна ознайомитися на сайті, звідки була взята діаграма класів.

http://docs.oracle.com/javase/tutorial/uiswing/components/text.html

12.6 СПИСКИ

Працювати зі списками надають можливість класи JList та JComboBox.

Особливість цих компонентів полягає у тому, що елементи списків містяться не у самому компоненті, і у моделі. Тому для створення елементів списку треба використовувати властивість «model».

Елементи списку можна визначити також через конструктори класів, задаючи у якості параметру масив, що містить елементи списку.

Елементами списку типу JComboBox можуть бути не тільки рядки символів але й об’єкти інших класів. У цьому випадку у списку буде виводитися результат виклику метода toString().

Посилання на вибраний елемент списку можна отримати за допомогою методу getSelectedValue() для об’єктів типу JList, та getSelectedItem() для об’єктів JComboBox.

Можна також отримати індекс вибраного елементу за допомогою методу getSelectedIndex().

Вибір елементу зі списку фіксується подією actionPerformed.

Якщо властивість editable списку JComboBox має значення true, то після натискання клавіші Enter теж формується подія actionPerformed, а метод getSelectedItem() повертає нове значення. При цьому метод getSelectedIndex() повертає -1.

Більш докладно про комбобокс можна почитати на сайті

http://docs.oracle.com/javase/tutorial/uiswing/components/combobox.html

12.7 ДІАЛОГОВІ ВІКНА

Ці компоненти забезпечують короткочасне спілкування з користувачем.

Клас JDialog створено на основі звичайного фрейму на якому розташована панель з двома кнопками «Ok» та «Cancel». Решту користувач класу може формувати на свій розсуд. Діалог викликається методом setVisible(true). Цей же метод з параметром false закриває діалог.

Клас JOptonPane має велику кількість статичних методів, що починаються словом show. Ці методи створюють різноманітні спеціалізовані вікна. Для налаштування вікон використовуються статичні константи класу, наприклад:

```
JOptionPane.showMessageDialog(this, e.getMessage(),
        "Object create error", JOptionPane.ERROR_MESSAGE);
```

12.8 ОБРОБКА ПОДІЙ, ПОВ'ЯЗАНИХ З КОМПОНЕНТАМИ

Графічні компоненти надають можливість дізнаватися про події, що відбуваються з компонентом, і є можливість запрограмувати реакцію на ці події.

Об'єкти, що реагують на події компонентів мають дві назви.

З одного боку це слухачі (listeners). Така назва використовується, коли об'єкт заявляє компоненту, що він хоче дізнаватися про якийсь різновид подій. Цей процес називають реєстрацією слухача у джерела події.

З іншого боку це обробники подій (handler). Така назва використовується, коли об'єкт, дізнавшись про подію, обробляє її (виконує якийсь код).

В Java процес створення такого об'єкту, його реєстрація та реалізація коду реакції на подію зазвичай об'єднують у одну конструкцію з використанням анонімних об'єктів анонімних класів, створених на основі інтерфейсів. Наприклад:

```
okButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            object = createObject();
            setVisible(false);
      }
});
```

У цьому прикладі до кнопки okButton за допомогою методу addActionListener() додається новий слухач типу ActionListenr. Для створення цього об'єкту використовується конструкція new, але ім'я об'єкту не присвоюється, тому об'єкт називають анонімним, або безіменним.

ActionListener – це інтерфейс, але у інтерфейсів не буває конструкторів, тому конструкцію ActionListenr() слід розглядати як конструктор анонімного (безіменного) класу, що реалізує інтерфейс ActionListener. Але для того, щоб такий конструктор спрацював, необхідно одразу після його виклику навести

реалізації методів інтерфейсу, що і зроблено у наведеному прикладі. Тут реалізовано метод actionPerformed(), який буде викликаний кнопкою okButton, коли на неї натиснуть. У якості параметру у цей метод передається об'єкт-подія, який несе інформацію про джерело події та іншу інформацію про неї.

Розглянута конструкція створюється за допомогою Eclipse автоматично. Програмісту залишається дописати тільки тіло методу. У даному прикладі, це тільки два рядки, що виділені більшим шрифтом.

В Java8 розглянута конструкція дещо спрощується за рахунок використання лямбда функції

```
okButton.addActionListener((e) -> {
	object = createObject();
	setVisible(false);
});
```

Ще один варіант реалізації реакції на події кнопок – створення об'єктів типу AbstractAction. Для створення такого об'єкту слід вибрати функцію set Action -> new з контекстного меню кнопки. Результатом буде створення внутрішнього класу і поля, що містить посилання на об'єкт цього класу.

```
private final Action action = new SwingAction();

...
private class SwingAction extends AbstractAction {
	public SwingAction() {
		putValue(NAME, "Ok");
		putValue(SHORT_DESCRIPTION, "Some short description");
	}
	public void actionPerformed(ActionEvent e) {
		//Тут має бути код реакції на натискання кнопки
	}
}
```

Програмісту залишається тільки написати код реакції на подію, реалізувавши тіло методу actionPerformed. Окрім того, можна змінити назви класу, змінної, назву кнопки та короткий опис реакції.

Головна перевага цього способу полягає у тому, що той самий «action» можна пов'язати з декількома кнопками, елементами меню, тощо.

12.9 ПИТАННЯ ДЛЯ САМОПЕРЕВІРКИ

1. Гілки ієрархії графічних компонентів Swing.
2. Загальна характеристика та призначення контейнерів.
3. Загальна характеристика та призначення менеджерів розташування.
4. Загальна характеристика та призначення елементів управління.
5. Поняття про модель для графічного компоненту.
6. Обробка подій графічних компонент.

13 РОБОТА З ГРАФІКОЮ

Докладну інформацію про графічні можливості Java можна знайти за посиланням http://skipy.ru/technics/gui_int.html.

У цьому розділі ми коротко познайомимось з деякими з них.

13.1 КЛАСИ GRAPHICS ТА GRAPHICS2D

Графічні можливості Java базуються на класах пакету java.awt. Серед цих класів особливе місце належіть класу Graphics та його пізнішому розширенню Graphics2D.

Кожний графічний компонент Java має у своєму складі об'єкт типу Graphics або Graphics2D, який називають графічним контекстом (graphics context).

З огляду на те, що графічний контекст суттєво залежить від конкретної графічної платформи, класи Graphics та Graphics2D зроблені абстрактними. Тому не можна безпосередньо створити екземпляри цих класів. Однак кожна віртуальна машина Java реалізує методи цих класів, створює їх екземпляри для кожного графічного компонента і надає доступ до об'єктів класу Graphics. Для цього слугує метод getGraphics() класу Component. Об’єкти цього класу передаються також як аргументи методів paint(Graphics), update (Graphics) та інших.

Для використання розширених можливостей класу Graphics2D, об’єкт типу Graphics, що надається названими вище методами, слід привести до типу Graphics2D, так як зроблено у наведеному нижче прикладі:

```
public void paint (Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
...
}
```

Таким чином графічний контекст є невід’ємною частиною будь якого компоненту. Коли деякий компонент має бути відображений на екрані, викликаються методи paint(Graphics) або update(Graphics) з відповідним йому контекстом Graphics.

13.2 ПРОСТІР КООРДИНАТ

Роботу з графічними об'єктами ускладнює той факт, що різні графічні пристрої можуть мати різні системи координат або роздільну спроможність. У зв'язку з цією обставиною в Java 2D підтримуються два координатних простори:

– простір пристрою, це координатна система вихідного пристрою, такого як екран, вікно або принтер.

– простір користувача, це незалежна від пристрою локальна система координат, яка використовується програмою користувача. Всі геометричні

об'єкти, що малюються методами Java 2D, визначені саме в цій системі координат, початок якої з координатами (0, 0) розташований у верхньому лівому куті області рисування, вісь абсцис направлена вправо, вісь ординат - вниз. Точки координат знаходяться між пікселями.

13.3 КОЛЬОРИ

Контекст містить поточний колір малювання і колір фону. Колір є об'єктом класу java.awt.Color.

Налаштування та доступ до поточного кольору контексту забезпечується методами setColor(Color) та getColor(). Методи setBackground(Color) та getBackground() забезпечують доступ до кольору фону.

Якщо немає потреби ретельно підбирати кольори, то можна скористатися однією з тринадцяти статичних констант, наявних у класі Color: BLACK, BLUE, CYAN, DARKGRAY, GRAY, GREEN, LIGHTGRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW.

Окрім того, за допомогою методів brighter() і darker() можна отримати більш яскравий, або більш темний кольори в порівнянні з поточним кольором, якщо складові кольору не мають граничних значень. Ці методи використовують число 0.7 для зменшення, або збільшення інтенсивності складових кольору. Вони корисні, якщо треба виділити активний компонент або, навпаки, показати неактивний компонент блідіше інших компонентів.

Якщо ж кольори, що визначені константами, не задовольняють програміста, він може використовувати різноманітні конструктори класу Color.

Найпростіші конструктори:

```
Color (int red, int green, int blue)
Color (float red, float green, float blue)
```

Ці конструктори створюють колір, що є сумішшю червоної, зеленої і синьої складових. Ця модель кольорів має назву RGB. Кожна складова змінюється від 0 (відсутність складової) до 255, або 1.0f. Наприклад: наступні два об’єкти визначають чистий яскраво-червоний та чистий яскраво-зелений кольори:

```
Color pureRed = new Color(255, 0, 0);
Color pureGreen = new Color(0.0f, 1.0f, 0.0f);
```

Наступний конструктор задає всі три складові в одному цілому числі:

```
Color (int rgb)
```

Синя складова кольору записується у молодшому байті (біти з 0-го по 7-й). У наступному байті записується зелена складова (біти з 8-го по 15-й). Далі записується червона складова (біти з 16-го по 23-й), наприклад:

```
Color c = new Color (0xFF_8F_48_F0);
```

У цьому прикладі інтенсивність червоної складової дорівнює 0x8F, зеленої - 0x48, синьої - 0xF0.

Біти з 23 по 31 не приймаються до уваги.

Конструктори, що наведено нижче, вводять четверту складову кольору, так звану "альфу", яка визначає прозорість кольору:

```
Color(int red, int green, int blue, int alpha)
Color(float red, float green, float blue, float alpha)
Color(int rgb, boolean hasAlpha)
```

Ця складова проявляє себе при накладенні одного кольору на інший. Якщо альфа дорівнює 255 або 1.0f, то колір абсолютно непрозорий, попередній колір не просвічує крізь нього. Якщо альфа дорівнює 0 або 0.0f, то колір абсолютно прозорий, для кожного пікселя видно тільки попередній колір.

Останній з наведених конструкторів враховує складову альфа, що знаходиться у старшому байті rgb (біти з 24-го по 31-й), якщо параметр hasAipha дорівнює true. Якщо ж hasAipha дорівнює false, то складова альфа вважається рівною 255, незалежно від того, що записано в старших бітах параметру rgb.

Усі попередні конструктори створювали кольори моделі RGB. Якщо ж потрібно створити колір іншої моделі, то слід використовувати наступний конструктор:

```
Color (ColorSpace cspace, float [] components, float alpha)
```

Приклад, наведений нижче, створює колір у моделі HSB:

```
float[] ar={0.5f,0.5f,0.5f};
ColorSpace cs = ColorSpace.getInstance(ColorSpace.TYPE_HSV);
Color c = new Color(cs, ar, 0.5f);
```

Для створення кольору в моделі HSB можна було б скористатися і статичним методом класу Color:

```
getHSBColor (float hue, float saturation, float brightness)
```

Окрім того, два наступних статичних методи класу Color дозволяють переходити з моделі RGB до моделі HSB і назад:

```
float[] RGBtoHSB (int red, int green, int blue, float[] hsb)
int HSBtoRGB (int hue, int saturation, int brightness)
```

Методи класу Color дозволяють також отримати складові поточного кольору. Це наступні методи: getRed(), getGreen(), getBlue(), getAlpha(), getRGB(), getColorSpace(), getComponents().

13.4 МЕТОДИ РИСУВАННЯ

Основний метод рисування в класі Graphics - це метод, який поточним кольором рисує відрізок прямої між точками з координатами (xl, yl) і (x2, y2):

```
drawLine (int xl, int yl, int x2, int y2)
```

Одного цього методу достатньо, щоб, намалювати будь-яку картину по точках, вимальовуючи кожну точку з координатами (x, y) за допомогою методу drawLine (x, y, x, y) і змінюючи кольору від точки до точки. Наведений нижче метод вимальовує прапор України на поверхні заданого компоненту.

```
void drawPrapor(Container c) {
  Graphics2D g = (Graphics2D) c.getGraphics();
  Rectangle r = c.getBounds();
  g.setColor(Color.BLUE);
  for(int y=0; y<r.height / 2; y++ )
    g.drawLine(0, y, r.width,y);
  g.setColor(Color.YELLOW);
  for(int y=r.height / 2; y<r.height; y++ )
    g.drawLine(0, y, r.width,y);
```

Але клас надає і інші методи. Деякі з цих методів наведено нижче.

Для зображення прямокутника зі сторонами, паралельними краям екрану, з координатами верхнього лівого кута (x, y), шириною width пікселів і висотою height пікселів можна використовувати такий метод:

```
drawRect(int x, int y, int width, int height)
```

Для зображення овалу, вписаного у прямокутник із заданими параметрами можна використовувати такий метод:

```
drawOval(int x, int y, int width, int height)
```

З іншими аналогічними методами, які починаються словом draw можна познайомитися у класах Graphics та Graphics2D.

Клас надає також методи, що подібні до попередніх, але починаються словом fill. Вони зображують замкнені фігури залиті поточним кольором. Наприклад:

```
fillRect (int x, int y, int width, int height)
```

Використовуючи наведений метод можна переписати метод зображення прапору України таким чином:

```
void drawPrapor(Container c) {
   Graphics2D g = (Graphics2D) c.getGraphics();
   Rectangle r = c.getBounds();
   g.setColor(Color.BLUE);
   g.fillRect(0, 0, r.width, r.height / 2);
   g.setColor(Color.YELLOW);
   g.fillRect(0, r.height / 2, r.width, r.height);
}
```

Для градієнтного заповнення фігури графічному контексту необхідно передати об’єкт типу Paint через метод setPaint(Paint). Інтерфейс Paint реалізовано у класах GradientPaint та Texturepaint.

Об’єкти класу GradientPaint забезпечують градієнтне заповнення фігури, при якому колір плавно змінюється вздовж прямої, що з'єднує початкову та кінцеву точки, залишаючись постійним уздовж кожної прямої, що

перпендикулярна до прямої що з'єднує початкову та кінцеву точки. Для створення такого об’єкту можна скористатися конструктором:

GradientPaint(float x1, float x2, Color c1, float x2, float y2, Color c2),

де x1, y1, x2, y2 – координати початкової та кінцевої точок заливки,

c1, c2 – значення початкового і кінченого кольору.

Є ще і інший конструктор, що дозволяє налаштувати циклічну зміну градієнту:

GradientPaint(float x1,float x2,Color c1, float x2,float y2,Color c2, boolean c)

Об’єкти класу TexturePaint забезпечують заповненням по заданому зразку.

Для створення такого об’єкту використовується конструктор:

TexturePaint(BufferedImage img, Rectangle2D r),

де img – це зображення зразка, а r – прямокутник, який формує елементи заповнення.

13.5 КЛАС BASICSTROKE

Клас BasicStroke визначає базовий набір атрибутів пера, яке використовує об'єкт Graphics2D для зображення контурів графічних фігур. Значення атрибутів задаються або числами, або константами класу BasicStroke.

Створити перо з потрібними характеристиками можна за допомогою одного з конструкторів класу BasicStroke.

Основний конструктор BasicStroke (float width, int cap, int join, float miter, float [] dash, float dashBegin) задає наступні параметри:

– width, товщина пера в пікселях;

– cap, стиль оформлення кінця лінії, що визначається однією з констант:

a) CAP_ROUND - закруглений кінець лінії;

b) CAP_SQUARE - квадратний кінець лінії;

c) CAP_BUTT - оформлення відсутнє;

– join, спосіб сполучення ліній, що визначається однією з констант:

a) JOIN_ROUND - лінії сполучаються дугою кола;

b) JOIN_BEVEL - лінії сполучаються відрізком прямої, перпендикулярної бісектрисі кута між лініями;

c) JOIN_MITER - лінії просто стикуються.

– miter, відстань між лініями починаючи з якої застосовується сполучення, не може бути менша за 1;

– dash, масив, що задає довжину штрихів та проміжків між ними для пунктирних ліній. Елементи масиву з парними індексами задають довжину штриха в пікселях, елементи з непарними індексами - довжину проміжку і масив перебирається циклічно;

– dashBegin відстань, на яку зсувається початок лінії.

Решта конструкторів задають деякі характеристики за умовчанням:

– BasicStroke (float width, int cap, int join, float miter) - суцільна лінія;

– BasicStroke (float width, int cap, int join) - суцільна лінія, для сполучення JOIN_MITER задається значення miter = 10.0f;

– BasicStroke (float width) - прямий обріз CAP_SQUARE і сполучення JOIN_MITER;

– BasicStroke () - ширина 1.0f.

Після створення пера його можна передати графічному контексту за допомогою методу setStroke(Stroke) та рисувати.

Нижче наведено метод, що вимальовує щось на зразок сонця:

```
public void drawSun(JPanel p, int top, int left, int size) {
    Graphics2D g = (Graphics2D) p.getGraphics();
    g.setColor(Color.RED);
    g.fillOval(left, top, size, size);
    int penSize = size / 4;
    g.setStroke(new BasicStroke(penSize));
    g.setColor(Color.YELLOW);
    g.drawOval(left, top, size, size);
    g.setColor(Color.ORANGE);
    g.setStroke(new BasicStroke(2, BasicStroke.CAP_ROUND,
      BasicStroke.JOIN_MITER, 1, new float[] { 10, 5 }, 0));
    int x1, x2, y1, y2, r1 = (size - penSize) / 2, r2 = 300;
    for (double fi = 0; fi < 2 * Math.PI; fi += Math.PI / 12) {
      x1 = (int) (left + size / 2 + r1 * Math.cos(fi));
      y1 = (int) (top + size / 2 + r1 * Math.sin(fi));
      x2 = (int) (left + size / 2 + r2 * Math.cos(fi));
      y2 = (int) (top + size / 2 + r2 * Math.sin(fi));
      g.drawLine(x1, y1, x2, y2);
    }
 }
```

13.6 ВИВЕДЕННЯ ТЕКСТУ

Графічний контекст дозволяє також виводити в область рисування тексти. Щоб налаштувати шрифт для тексту можна скористатися методом setFont(Font).

Для виведення тексту в область малювання поточним кольором і шрифтом, починаючи з точки (x, y) у класі Graphics є кілька методів, але ми тут згадаємо тільки про один. Це метод **drawString (String s, int x, int y)** , що виводить рядок s.

Параметри «x» та «y» у цьому методі – це координати лівої нижньої точки першої букви тексту(початок базової лінії (baseline) виведення тексту.

Нижче наведено приклад виведення тексту заданим шрифтом.

```
Graphics2D g = (Graphics2D) panel.getGraphics();
int fontSize = 50;
g.setFont(new Font(Font.DIALOG, Font.ITALIC, fontSize));
g.drawString("Слава Україні!", 0, fontSize);
```

У Java 2D окрім імені, стилю і розміру, шрифт отримав багато додаткових атрибутів, наприклад, перетворення координат, підкреслення або

перекреслення тексту, виведення тексту справа наліво. Колір тексту і його фону є тепер атрибутами самого тексту, а не графічного контексту. Можна задати різну ширину символів шрифту, надрядкові і підрядкові індекси. Атрибути встановлюються константами класу TextAttribute через відповідний конструктор.

13.7 ІНТЕРФЕЙС SHAPE

Інтерфейс Shape оголошує сукупність методів для опису та інспектування 2D геометричних об'єктів.

Логічні методи contains() дозволяють перевірити, чи не лежить в багатокутнику задана аргументами методу точка, відрізок прямої або цілий прямокутник зі сторонами, паралельними сторонам екрана.

Логічні методи intersects() дозволяють перевірити, чи не перетинається чи з даними багатокутником відрізок прямої, заданий аргументами методу, або прямокутник зі сторонами, паралельними сторонам екрана.

Методи getBounds () і getBounds2D () повертають прямокутник, цілком містить в собі даний багатокутник.

Інтерфейс Shape реалізовано у групі класів пакету java.awt.geom.

Ці класи визначають як загальні графічні примітиви (форми), такі як дуги, прямокутники, еліпси, так і більш складні фігури і навіть групи фігур.

За допомогою цих класів користувач може сконструювати практично будь-яку геометричну форму і відтворити її на екрані за допомогою виклику методів draw() або fill() контексту Graphics2D.

Прості фігури створюються за допомогою таких класів як Line2D, Arc2D, QuadCurve2D, Rectangle2D, Ellipse2D та інших. Цікава особливість названих класів полягає у тому, що самі ці класи абстрактні, а для створення об’єктів цих класів слід користуватися конструкторами їх внутрішніх класів, що мають назви Float і Double. Вибір одного з них визначає тип чисел, що задають координати фігури.

Наприклад, щоб створити пряму лінію, можна скористуватися таким конструктором:

```
Shape line = new Line2D.Float(2f, 12f, 27f, 122f);
```

Складні фігури зручніше створювати за допомогою класу GeneralPath.

13.8 КЛАС GENERALPATH

Клас GeneralPath є нащадком класу Path2D.Float і є зручним засобом створення плоских геометричних фігур складної форми. Об’єкти цього класу схожі на колекції, що містять послідовність простих геометричних фігур (segments), і навіть надають доступ до ітератора типу PathIterator.

Фактично, об’єкт класу GeneralPath задає маршрут для об’єкту BasicSroke, що буде вирисовувати задану фігуру. Маршрут складається з послідовності точок і правил їх з’єднання.

Для створення об'єкту цього класу можна скористатися конструктором за замовчуванням GeneralPath().

У створеному об'єкті є атрибут «поточна точка». Початкові значення координат цієї точки – (0, 0), але її можна перемістити у точку (x0, y0) методом moveTo (float x0, float y0).

Від поточної точки до точки (x, y) можна провести:

– відрізок прямої методом lineTo (float x, float y);

– криву Безьє другого порядку методом quadTo(float x1, float yl, float x, float y), яка будується на основі поточної точки та двох заданих;

– криву Безьє третього порядку методом curveTo (float xl, float yl, float x2, float y2, float x, float y), яка будується на основі поточної точки та трьох заданих.

Поточною точкою після цього стає точка (x, y). Початкову і кінцеву точки можна з'єднати методом closePath ().

Як приклад, можна навести метод, що повертає фігуру у вигляді хмарки:

```
protected Shape createCloud() {
    GeneralPath gp = new GeneralPath();
    gp.moveTo(60,200);
    gp.curveTo(-40, 250, 5, 50, 80, 100);
    gp.curveTo(130, 0, 180, -40, 220, 50);
    gp.quadTo(400, 120, 250, 150);
    gp.curveTo(230, 250, 80, 120, 60, 200);
    gp.closePath();
    return (Shape)gp;
}
```

Об'єкти класу GeneralPath можна також створювати на основі вже існуючих фігур. Для цього використовується конструктор GeneralPath (Shape sh).

Можна також об’єднати декілька фігур у одному об’єкті. Для цього використовується метод append (Shape sh, boolean connect). Якщо параметр connect дорівнює true, то нова фігура з'єднується з попередніми фігурами за допомогою поточного пера.

13.9 РЕЖИМИ РИСУВАННЯ

Клас Graphics надає можливість використовувати два режими рисування. Перший режим – це звичайний режим, коли нове зображення вимальовується поточним кольором, незалежно від кольору фону. Цей режим встановлюється за замовчуванням, або може бути відновлений методом setPaintMode().

Другий режим має назву XOR і встановлюється методом setXORMode(Color). Цей режим використовує альтернативний колір, що передається у метод setXORMode як параметр. Колір фігури, що зображується з використанням такого режиму, визначається шляхом об’єднання бітів альтернативного кольору та кольору фону за допомогою логічної операції XOR.

При використанні такого режиму намальоване зображення зникає, якщо його повторно відобразити у тому ж самому місті.

Для того, щоб зображення, що вимальовується у цьому режимі, мало потрібний колір, слід визначити альтернативний колір, як результат XOR об'єднання потрібного кольору і кольору фону. Щоб отримати альтеративний колір для потрібного кольору можна скористатися таким методом:

```
private Color altColor(Graphics2D g, Color c) {
     Color back = g.getBackground();
     int rgb = back.getRGB() ^ c.getRGB();
     return new Color(rgb);
}
```

Як приклад використання альтернативного кольору наведемо метод, якій забезпечує переміщення червоного овалу по панелі:

```
public void movingOval(JPanel panel) {
    Graphics2D g = (Graphics2D) panel.getGraphics();
    Dimension d = new Dimension(50, 50);
    int x = 0, y = (panel.getHeight() - d.height) / 2;
    g.setXORMode(altColor(g, Color.RED));
    while (x < panel.getWidth() - d.width) {
         g.fillOval(x, y, d.width, d.height);
         try {
           Thread.sleep(100);
         } catch (InterruptedException e) {}
         g.fillOval(x, y, d.width, d.height);
         x += 5;
    }
}
```

13.10 МЕТОД PAINT(GRAPHICS)

Цей метод оголошено у класі Component і його можна викликати для будь якого компоненту. Метод відповідає за відображення компонентів при появі їх на екрані або після зміни їх розмірів чи переміщенні по екрану. Для цього він використовує три методи - paintComponent(Graphics), paintBorder(Graphics) і paintChildren(Graphics). Усі методи мають специфікатор доступу protected і їх можна перевизначити у класах спадкоємцях. Призначення цих методів таке:

paintComponent забезпечує відображення самого компоненту. Саме цей метод необхідно перевизначити для того, щоб змінити зовнішній вигляд компоненту.

paintBorder – використовується для зображення рамки компоненту.

paintChildren - відображає дочірні елементи.

Наведемо приклад, як змінити зовнішній вигляд компонента. Для цього створимо анонімний клас, що успадковує клас JPanel, у якому і перевизначимо метод paintComponent(Graphics).

У прикладі, що наведено нижче, створюється панель, фон якої буде завжди виглядати як український прапор.

```
public JPanel getPanel() {
 if (panel == null)
          return new JPanel() {
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Rectangle r = getBounds();
        g.setColor(Color.BLUE);
        g.fillRect(0, 0, r.width, r.height / 2);
        g.setColor(Color.YELLOW);
        g.fillRect(0, r.height / 2, r.width, r.height);
      }
 };
 return panel;
}
```

13.11 АФІННІ ПЕРЕТВОРЕННЯ

Афінні перетворення дозволяють виконувати різноманітні маніпуляції над плоскою поверхнею, і відповідно над фігурою, що на ній зображена. Це може бути розтягнення або стиснення, поворот чи переміщення.

Афінне перетворення площини визначається як об'єкт класу AffineTransform. Найпростіший спосіб для створення такого об’єкту – це використання пустого конструктора. Далі можна налаштовувати створений об’єкт на виконання потрібних перетворень за допомогою різноманітних методів. Ось деякі з них:

– translate(double dx, double dy), переміщує фігуру на задану відстань по осям координат;
– scale(double sx, double sy), змінює розміри фігури по осям, відповідно зо заданих масштабних коефіцієнтів;
– rotate(double ang, double x0, double y0), повертає фігуру на кут ang відносно точки з координатами x0, y0.

Для отримання трансформованої фігури можна використовувати метод createTransformedShape(Shape sh) об’єктів класу AffineTransform.

Як приклад, розглянемо метод, що відображає на панелі хмару, як рухається, змінюється у розмірах та й ще обертається. Для створення первинного зображення хмари використовується метод createCloud, що було розглянуто раніше. Можна використовувати і інше зображення.

```
private Shape sh;
public void movingCloud(JPanel p) {
 new Thread(() -> {
   Graphics2D g = (Graphics2D) p.getGraphics();
   sh = createCloud();
   while (sh.getBounds().getWidth() > 5) {
     Rectangle r = sh.getBounds();
     //Формування градієнтного кольору
     GradientPaint grd = new GradientPaint( r.x,  r.y, Color.WHITE,
          r.x + r.width/3, r.y + r.height/3, Color.LIGHT_GRAY, true);
     g.setPaint(grd);
     //Відображення хмари
```

```
            EventQueue.invokeLater(() -> g.fill(sh));
            //Затримка у часі
            try {
              Thread.sleep(150);
            } catch (Exception e) { }
            //Відновлення фонового зображення панелі
            p.repaint(r);
            //Формування афінного перетворення
            AffineTransform at = new AffineTransform();
            at.translate(40, 0);
            at.scale(0.9, 0.95);
            at.rotate(0.08, r.getX() + r.getWidth() / 2,
                 r.getY() + r.getHeight() / 2);
            //Зміна зображення хмари
            sh = at.createTransformedShape(sh);
          }
        }).start();
      }
```

Для нормальної роботи методу його код потрібно виконувати в окремому потоці. Саме такий потік і створюється у методі.

Для відображення хмари також створюється потік за допомогою методу invokeLate(Runnable).

Старе зображення хмари вилучається за допомогою методу repaint(Rectangle).

Афінне перетворення формується за рахунок послідовного виклику трьох методів – переміщення, масштабування та повороту.

Наприкінці циклу за допомогою афінного перетворення створюється нова фігура хмари.

13.12 КЛАС BUFEREDIMAGE

Клас успадковує абстрактний клас java.awt.Image і об’єкти цього класу зберігають у собі графічні зображення.

Для створення об’єктів цього класу можна використовувати різні конструктори, найпростіший з яких виглядає так:

BufferedImage(int width, int height, int imageType),

де width та height – розміри зображення у пікселях, а imageType – це одна з констант, що визначає тип зображення, наприклад, `BufferedImage.TYPE_INT_ARGB`.

Завантажити зображення до об’єкту класу BuferedImage можна декількома способами.

Перший спосіб базується на тому факті, що об’єкт класу BuferedImage має власній графічний контекст типу Graphics, доступ до якого можна отримати через метод getGraphics(). Завдяки цьому для створення зображень можна використовувати усі розглянуті вище засоби для рисування. Однак слід розуміти, що при цьому зображення створюється не на екрані, а в буфері, і не пов’язано з подіями на екрані.

Другий спосіб дозволяє завантажити зображення до буферу з файлів, що

мають розширення .jpg або .png. Для завантаження можна використати статичний метод read() класу javax.imageio.ImageIO, який не тільки завантажує файл, але й створює об'єкт типу BuferedImage. У цей метод можна передавати параметри різних типів, але найкраще передавати параметр типу java.net.URL, а самі файли зберігати у проекті, що створюється. У такому випадку файли будуть завантажені у .jar файл і не буде проблем з їх пошуком.

Нижче наведено приклад створення об'єкту img типу BuferedImage та завантаження файлу d2.jpg до цього об'єкту. Файл знаходиться у папці other під папкою src проекту.

Для створення URL використовується клас `TestWithFoto` з папки src.

```
URL url = TestWithFoto.class.getResource("/other/d2.jpg");
BufferedImage img = null;
{
 try {
    img = ImageIO.read(url);
 } catch (IOException e1) {
    e1.printStackTrace();
 }
}
```

Для того, щоб відобразити вміст об'єкту типу BuferedImage на екрані можна скористатися методом drawImage графічного контексту. Слід тільки використовувати графічний контекст контейнеру, на якому буде відображатися зображення, а не контекст іміджу.

Метод drawImage виглядає так:

```
drawImage(Image img, int x, int y, ImageObserver observer)
```

де img – це імідж, що має бути відображений, x та y – це координати точки на поверхні контейнера, з якої починається відображення зображення, observer – це посилання на об'єкт, який може реагувати на завершення завантаження іміджу. Допустимо значення null або this.

Розміри зображення можна налаштувати під розміри компоненту, на поверхні якого він відображується. Для цього можна використати метод

```
getScaledInstance(int width, int height, int hints),
```

де `width` та `height` – потрібні розміри зображення, а `hints` – це константа, що встановлює правила візуалізації.

Нижче наведено приклад створення панелі, на поверхні якої відображується фото, що завантажено у імідж img:

```
public JPanel getPanel() {
 if (panel == null)
    return new JPanel() {
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2d =(Graphics2D) g;
        Image scaledImg = img.getScaledInstance(getWidth(),

 getHeight(),Image.SCALE_SMOOTH);
```

```
            g2d.drawImage(scaledImg, 0, 0, this);
        }
    };
 return panel;
}
```

13.13 ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Класи Graphics та Graphics2d і їх методи.
2. Клас Color та його конструктори.
3. Клас BasicStroke та його налаштування.
4. Клас GeneralPath та його методи для рисування.
5. Використання методу paintComponent(Graphics).
6. Режими рисування.
7. Клас Image і афінні перетворення.
8. Пояснення до тексту самостійно створеного додатку.

СПИСОК ЛІТЕРАТУРИ

Базова

1. Шилд Гербердт. Java. Полное руководство, 8-е изд. Пер. с англ. — М.: ООО “И.Д. Вильямс”, 2012. — 1104 с . : ил.
2. Java. The complete Reference. Ninth Edition. Oracle Press. 2014.
3. Goetz Brain. Java concurrency in practice. Addison Wesley, 2010.
4. Simon Kendal. Object oriented programming using Java. Ventus Publishing ApS, 2009. – 209 с.
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. – СПб.:Питер,2001

Допоміжна

1. Стелтинг Стивен, Маасен Олав. Применение шаблонов Java. Библиотека профессионала. : Пер. с англ. — М.: Издательский дом "Вильямс", 2002. — 576 с.: ил.
2. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++: Пер. с англ. – М.:Диалект,1999
3. Волш А.И. Основы программирования на Java для World Wide Web. – К.: Диалектика, 1996. – 512 с.
4. Эккель Б. Философия Java. Библиотека программиста. – СПб.: Питер, 2003
5. Блох Д. Java. Эффективное программирование. – Лори (SUN published), 2003