МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

К.О. Ткаченко, О.А. Ткаченко, О.І. Ткаченко

ПРОГРАМУВАННЯ МОВОЮ JAVA

**Курс лекцій**

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського  
як навчальний посібник для студентів, які навчаються  
за спеціальністю 121 «Інженерія програмного забезпечення»

(освітня програма «Інженерія програмного забезпечення мультимедійних та  
інформаційно-пошукових систем»)

Київ

КПІ ім. Ігоря Сікорського  
2024

Рецензент: Коцюбівська Катерина Іванівна, канд. техн. наук, доц.

Відповідальний

редактор

Рибачок Наталя Антонівна, канд. техн. наук, доц.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського  
(протокол № 7 від 09.05.2024 р.)

за поданням Вченої ради факультету прикладної математики  
(протокол №10 від 29.04.2024 р.)

Електронне мережне навчальне видання

Ткаченко Костянтин Олександрович, канд. екон. наук, доцент

Ткаченко Олександр Андрійович, канд. фіз.-мат. наук, доцент

Ткаченко Ольга Іванівна, канд. фіз.-мат. наук, доцент

**ПРОГРАМУВАННЯ МОВОЮ JAVA**КУРС ЛЕКЦІЙ

Програмування мовою Java: конспект лекцій з дисципліни «Кросплатформне програмування» [Електронний ресурс] : навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення» (освітня програма «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»)/ К.О. Ткаченко, О.А. Ткаченко, О.І.Ткаченко; КПІ ім. Ігоря Сікорського. - Електронні текстові дані. - Київ: КПІ ім. Ігоря Сікорського, 2024. - 280 с.

Навчальний посібник розроблено для ознайомлення студентів з технологіями програмування мовою Java для розроблення програмного забезпечення загального призначення, в тому числі, для мобільних пристроїв, мережевих застосунків, мультимедійних та інформаційно-пошукових систем та інших. Навчальний посібник призначений для студентів, які навчаються за спеціальністю 121 Інженерія програмного забезпечення факультету прикладної математики КПІ ім. Ігоря Сікорського.

© К.О.Ткаченко, О.А.Ткаченко, О.І.Ткаченко, 2024

© КПІ ім. Ігоря Сікорського, 2024

# Вступ

Кросплатформне програмне забезпечення — це тип програмного забезпечення, яке працює на різних операційних системах або пристроях, які часто називають платформами.

Основні переваги кросплатформних програм:

* Охоплення ширшої категорії користувачів.
* Швидша та гнучка розробка, отже, скорочення часу виходу програмного продукту на ринок.
* Багатий функціонал і широкі можливості налаштування.
* Зниження витрат на розробку: розроблена одна програма, яка буде доступна на кількох платформах.
* Використання сучасних кросплатформних фреймворків, які пропонують нативний досвід. Таким чином, кінцевий продукт буде таким же ефективним, як і нативний застосунок.

Для полегшення створення кросплатформних програм було розроблено багато мов програмування і фреймворків, серед яких найбільш відомими є Java, JavaScript**,** Ruby, React Native, Xamarin та інші.

Вибір мови чи фреймворку багато в чому залежить від цілей і вимог, яким має відповідати програмне забезпечення, що буде розроблятися. Кожна з кросплатформних мов підтримують має свої переваги, недоліки та обмеження. Вибираючи фреймворк або мову для свого проекту, треба зосередитися на характеристиках майбутнього програмного забезпечення та порівняти функції та вимоги з тим, що пропонує певна технологія.

Однією з найвідоміших кросплатформних мов є Java є, оскільки вихідний код програми компілюється у проміжний байт-код, який потім виконується на віртуальній машині Java (JVM), яка працює з конкретною апаратною платформою. Java — це об’єктно-орієнтована мова програмування загального призначення. Як універсальна кросплатформна мова вона використовується для розробки мобільних застосунків, для розробки веб-застосунків корпоративного масштабу завдяки своїй стабільності та продуктивності. Мова постійно вдосконалюється і тому має перспективу ще довго залишитись популятною серед розробників програмного забезпечення.

# 

# Частина 1. Основи мови програмування JAVA

## ****Загальна характеристика мови**** Java

### 1.1.1 Історія створення мови JAVA

Мова програмування Java народилася в 1991 р. в лабораторіях компанії SunMicrosystems. Розробку проекту започаткував Джеймс Ґослінґ, сам проект мав назву «Green» (Зелений). Створення першої робочої версії, яка мала назву «Oak» (дуб), було виконано за 18 місяців. Оскільки виявилось, що ім'я «Oak» уже використовувалось іншою фірмою, то в результаті тривалих суперечок навколо назви нової мови з поміж ряду запропонованих було вибрано назву Java, у 1995 р. мову було офіційно перейменовано.

Головним мотивом створення Java була потреба в мові програмування, яка б не залежала від платформи (тобто від архітектури комп’ютера) і яку можна було б використовувати для створення програмного забезпечення, яке вбудовується в різноманітні побутові електронні прилади, такі як мобільні засоби зв'язку, пристрої дистанційного керування тощо.

Досить скоро майже всі популярні тогочасні веб-оглядачі отримали можливість запускати «безпечні» для системи Java апплети всередині веб-сторінок. У грудні 1998 р. SunMicrosystems випустила Java 2 (спершу під назвою J2SE 1.2), де було реалізовано декілька конфігурацій для різних типів платформ. Наприклад, J2EE призначалася для створення корпоративних застосунків, а значно урізана J2ME - для приладів з обмеженими ресурсами, наприклад, мобільніх телефонів. У 2006 році в маркетингових цілях, версії J2 було перейменовано у JavaEE, JavaME та JavaSE.

13 листопада 2006 року Sun випустили більшу частину Java в якості вільного та відкритого програмного забезпечення згідно з умовами GNUGeneralPublicLicense (GPL). 8 травня 2007 корпорація закінчила процес, в результаті якого всі вихідні коди Java були випущенні під GPL, за винятком невеликої частини коду, на який Sun не мала авторського права.

Період становлення Java збігся у часі з розквітом міжнародної інформаційної служби WorldWideWeb (WWW або Web). Ця обставина відіграла вирішальну роль у майбутньому Java, оскільки Web теж вимагала платформо-незалежних програм. Як наслідок, були зміщені акценти в розробці Sun з побутової електроніки на програмування для Інтернет.

Зараз мовою Java поновлює компанія Oracle, яка придбала SunMicrosystems у 2009 році.

### 1.1.2 Відмінності мов JAVA та С++

Основні засади створення мовиJava:

* Незалежність від конкретної комп’ютерної платформи та типу процесору. Компілятор Java видає не код виконання, а байт-код, який виконується не операційною системою, а віртуальною машиною Java (JVM).
* Необхідність створення ефективних програм для Internet.

Порівняно з мовою С++ в мові Java немає:

* Вказівників
* Структур і об’єднань
* Перевантаження операторів
* Автоматичного приведення типів з втратою точності
* Глобальних змінних і функцій
* Значень аргументів за умовчуванням
* Деструкторів
* Оператора goto
* Передачі об’єктів за значенням (тільки за посиланням)
* Адресної арифметики

В мові Java є:

* Багатопотоковість
* Пакети
* Інтерфейси (аналог абстрактного класу в С++)
* Вбудований рядковий тип String
* Документаційний коментар
* Всі масиви динамічні.

Відмінність властивостей:

* В С++ true і false – можуть бути числами, в Java – тільки літерали.
* В С++ специфікатор рівня доступу застосовується до груп полів, в Java – для кожного поля окремо

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Дайте коротку історичну довідку про створення і розвиток мови Java.
2. Як реалізована об’єктно-орієнтована спрямованість мови?
3. Чим Java відрізняється від С++?
4. Що означає принцип незалежністі Java-застосунків від архітектури комп’ютера?
5. Які функціі виконує віртуальна машина Java?
6. Як поєднується інтерпретованість з високою продуктивністю?

## Лексичні основи мові JAVA

### 1.2.1 Структура програми на мові JAVA

Головним елементом у програмі на мові Java є конструкція, що позначається ключовим словом *class* (клас). *Клас* – це сукупність даних і методів (функцій), що обробляють дані.

Нижче наведена загальна форма вихідного файлу з програмою на мові Java:

один оператор package (необов'язковий)

будь-яка кількість операторів import (необов'язкові)

оголошення одного відкритого (public) класу

будь-яка кількість закритих (private) класів пакета (необов'язкові)

*Пакет (package)* — це контейнер, який використовується для того, щоб ізолювати імена класів. Наприклад, ви можете створити клас List, вставитии його в пакет і не думати після цього про можливі конфлікти, які могли б виникнути, якщо б хто-небудь ще створив клас з ім'ям List.

Оператор *import* служить для підключення потрібних пакетів і класів.

Розглянемо найпростішу програму на Java, яка дозволяє вивести на екран вітання "Hello, world!"

public class Hello

{

public static void main (String args[]){

System.out.println("Hello world!");

}

}

/\*

Коментар Java,

займає чотири рядки

\*/

//Коментар займає один рядок

У програмі на Java повинен бути присутнім хоча б один клас (в нашому випадку це клас *Hello*).

Оскільки клас *Hello* оголошений з модифікатором *public*, то ім'я файлу з вихідним кодом повинно збігатися з ім'ям класу. Для класів з модифікатором за замовчуванням імена файлів можуть бути будь-якими (розширення обов'язково - *.java*).

Метод *main* оголошений в класі Hello – це єдиний метод програми. Єдиний оператор в цьому методі являє собою виклик методу *println*(), який виводить рядок "Hello world!" на екран.

Символ ';' в Java є роздільником операторів.

Виконання будь-якої програми, написаного на Java, починається з виклику методу *main*().

Змінні являють собою імена області пам'яті, у яких знаходяться дані визначених типів. В залежності від виду даних для них можуть застосовуватися змінні різних типів.

Типи даних змінних визначають внутрішнє подання даних в обчислювальних машинах, діапазон або набір значень, які може приймати змінна, а також операції, в яких можуть використовуватися змінні певного типу.

### 1.2.2 Лексичні основи мови

#### Пробіли та коментарі

Java — мова, яка допускає довільне форматування тексту програм. Немає необхідності вирівнювати текст спеціальним чином. Між окремими лексемами (між якими немає операторів або роздільників) повинно бути, принаймні, по одному пробілу, символу табуляції або символу переходу на наступний рядок.

Існує три різновиди коментарів: коментарі в одному рядку, коментарі у кількох рядках і коментарі для документування. Коментарі, що займають один рядок, починаються з символів // і закінчуються в кінці рядка. Такий стиль коментування корисний для розміщення коротких пояснень до окремих рядків коду:

а = 5; // присвоєння значення змінної

Для більш докладних пояснень ви можете скористатися коментарями, розміщеними на декількох рядках, почавши текст коментарів символами /\* і закінчивши символами \*/ При цьому весь текст між цими парами символів буде розцінений як коментар і транслятор його проігнорує. Наприклад:

/\*

Тут може бути будь-який текст

у кількох рядках

....

\*/

Третя, особлива форма коментарів, призначена для сервісної програми *javadoc*, яка використовує компоненти Java-транслятора для автоматичної генерації документації по програмі. Угода, що використовується для коментарів цього виду, така: для того, щоб розмістити перед оголошенням відкритого (public) класу, методу або змінної документуючий коментар, потрібно почати його з символів /\*\* (коса риска та дві зірочки). Закінчується такий коментар точно так само, як і звичайний коментар — символами \*/.

Програма *javadoc* вміє також розрізняти в коментарях деякі спеціальні змінні, імена яких починаються з символу @ — це так звані теги документації. Вони дозволяють посилатися на документацію до інших класів, успадковувати документацію базового класу, містить інформацію про автора, описує параметри методів та інше. Результатом роботи *javadoc* буде HTML-файл.

Ось приклад використання коментарів, які використовуються для автоматичного створення документації:

/\*\*

Це коментар до класу

Цей клас призначений для ...

...

\*/

class MyClass {

/\*\*

Це коментар до методу

У цього методу два параметра:

@param key - параметр

@param value - параметр

\*/

void put (String key, Object value) {...

#### Зарезервовані ключові слова

Зарезервовані ключові слова — це спеціальні ідентифікатори, які в мові Java використовуються для того, щоб ідентифікувати вбудовані типи, модифікатори і засоби управління виконанням програми. Ці ключові слова спільно з синтаксисом операторів і роздільників входять в опис мови Java. Вони можуть застосовуватися лише за призначенням, їх не можна використовувати в якості ідентифікаторів для імен змінних, класів або методів. Наприклад, *abstract, саѕе, const, else, float, if, int* і т. д

#### Ідентифікатори

Ідентифікатори використовуються для іменування класів, методів і змінних. В якості ідентифікатора може використовуватися будь-яка послідовність рядкових і прописних букв, цифр і символів \_ (підкреслення) і $ (долар). Ідентифікатори не повинні починатися з цифри, щоб транслятор не переплутав їх із числовими літеральними константами. Java — мова, чутлива до регістру букв. Це означає, що, приміром, Value і VALUE — різні ідентифікатори.

#### Літерали

Константи в Java задаються їх літеральним представленнями. Цілі числа, числа з плаваючою точкою, логічні значення, символи і рядки можна розташовувати в будь-якому місці коду.

Цілі літерали

Цілі числа використовується в звичайних програмах найбільш часто. Крім десяткових цілих можуть використовуватися також числа з основою 8 і 16 — вісімкові і шістнадцяткові. Java розпізнає вісімкові числа по цифрі 0, яка ставиться перед числом. Нормальні десяткові числа не можуть починатися з нуля, так що використання в програмі зовні допустимого числа 09 призведе до повідомлення про помилку при трансляції, оскільки 9 не входить в діапазон 0..7, допустимий для знаків вісімкового числа. Шістнадцяткова константа розрізняється по символам 0х або 0Х, які стоять попереду числа. Діапазон значень шістнадцяткової цифри — 0.. 15, причому в якості цифр для значень 10.. 15 використовуються букви від *А* до *F* (або від *а* до *f*). За допомогою шістнадцяткових чисел ви можете в короткій і зрозумілій формі представити значення, орієнтовані на використання в комп'ютері, наприклад, написавши 0xffff замість 65535.

Цілі літерали є значеннями типу *int*, яке в Java зберігається в 32-бітовому слові. Якщо вам потрібне значення, яке по модулю більше, ніж приблизно два мільярди, необхідно скористатися константою типу *long*. При цьому число буде зберігатися в 64-бітовому слові. Для довгих цілих чисел ви можете приписати праворуч рядкову чи прописну літеру *L*, вказавши таким чином, що дане число відноситься до типу long. Наприклад, 0x7ffffffffffffffL або 9223372036854775807L — це значення, найбільше для числа типу *long*.

Літерали з плаваючою точкою

Числа з плаваючою точкою представляють десяткові значення, у яких є дробова частина. Їх можна записувати або в звичайному, або науковому форматах. У звичайному форматі число складається з деякої кількості десяткових цифр, що стоять до десяткової крапки і розташованих за нею десяткових цифр дробової частини. Наприклад, 2.0 або 3.14159 — це допустимі значення чисел з плаваючою точкою, записаних у стандартному форматі. В науковому форматі після перерахованих елементів додатково вказується літера *е* або *Е*, а за нею десятковий порядок. Порядок визначається позитивним або від'ємним десятковим числом. Приклади чисел в науковому форматі: 6.022е23, 314159Е-05, 2е+100.

В Java числа з плаваючою точкою за замовчуванням розглядаються, як значення типу *double*. Якщо вам потрібно константа типу *float*, то праворуч до литералу треба приписати символ *F* або *f*. Можна додавати і до литералів типу *double* символ *D* або *d*. Значення за замовчуванням (типу *double*) зберігаються в 64-бітових словах, менш точні значення типу *float* — у 32-бітових.

Логічні літерали

У логічної змінної може бути лише два значення — *true* (істина) і *false* (неправда). Логічне значення *true* і *false* не перетворюється ні в яке числове подання. Ключове слово *true* в Java не дорівнює 1, a *false* не дорівнює 0. В Java значення *true* і *false* можуть присвоюватися тільки змінним типу *boolean* або використовуватися у виразах з логічними операторами.

Символьні літерали

Символи в Java — це індекси в таблиці символів UNICODE. Вони являють собою 16-бітові значення, які можна перетворити у цілі числа та до яких можна застосовувати оператори цілочисельної арифметики, наприклад, оператори додавання і віднімання. Символьні літерали розміщуються всередині пари апострофів (' '). Всі видимі символи таблиці ASCII можна прямо вставляти всередину пари апострофів: - 'a', 'z', '@'. Для символів, які не можна ввести безпосередньо, передбачено кілька керуючих послідовностей (Табл.1.2.1).

Рядкові літерали

Рядкові літерали в Java— це довільний текст, укладений у пару подвійних лапок (""). Хоча рядкові літерали в Java реалізовані досить своєрідно (Java створює об'єкт для кожного рядка), зовні це ніяк не можна побачити. Приклад рядкового літералу: "Hello World!". Всі керуючі послідовності, вісімкові і шістнадцяткові форми запису, які визначені для символьних літералів, працюють так само і в рядках. Рядкові літерали в Java повинні починатися і закінчуватися в одному і тому ж рядку вихідного коду.

Таблиця 1.2.1. Керуючі послідовності символів

|  |  |
| --- | --- |
| Керуюча послідовність | Опис |
| \ddd | Вісімковий символ (ddd) |
| \uxxxx | Шістнадцятковий символ UNICODE (xxxx) |
| \' | Апостроф |
| \" | Лапки |
| \\ | Зворотна коса риска |
| \r | Повернення каретки (carriage return) |
| \n | Перехід на наступний рядок (line feed, new line) |
| \f | Перехід на наступну сторінку (form feed) |
| \t | Горизонтальна табуляція (tab) |
| \b | Повернення на крок (backspace) |

#### Роздільники

Прості роздільникиі впливають на зовнішній вигляд і функціональність програмного коду (Табл. 1.2.2)

#### Змінні

*Змінна* - це основний елемент зберігання інформації в Java-програмі. Змінна характеризується комбінацією ідентифікатора, типу і області дії. Залежно від того, де ви оголосили змінну, вона може бути локальною, наприклад, для коду всередині циклу *for*, або це може бути змінна екземпляра класу, доступна всім методам даного класу.

Основна форма оголошення змінної така:

тип ідентифікатор [= значення], [ ідентифікатор [= значення] ]...;

*тип* - це або один з вбудованих типів, тобто, *byte, short, int, long, char, float, double, boolean*, або ім'я класу, або інтерфейсу. Нижче наведено кілька прикладів оголошення змінних різних типів. Деякі приклади включають в себе ініціалізацію початкового значення. Змінні, для яких початкові значення не вказані, автоматично не ініціалізуються нулем.

int a, b, с; //Оголошуються три цілих змінних а, b, с.

int d = 3, e, f = 5; // Оголошуються три цілих змінних, ініціалізуються d і f.

byte z = 22; //Оголошується і ініціалізується змінна z.

double pi = 3.14159; //Оголошується і ініціалізується змінна pі.

char x = 'x'; // Оголошується і ініціалізується змінна х (отримує значення 'х').

Таблиця 1.2.2. Роздільники мови Java

|  |  |  |
| --- | --- | --- |
| Символи | Назва | Для чого застосовуються |
| ( ) | Круглі дужки | Виділяють списки параметрів в оголошенні і виклику методу, також використовуються для встановлення пріоритету операцій у виразах, виділення виразів в операторах управління виконанням програми і в операторах приведення типів. |
| {} | Фігурні дужки | Містять значення, які автоматично ініціалізують масиви, також використовуються для обмеження блоку коду в класах, методах і локальних областях дії. |
| [ ] | Квадратні дужки | Використовуються в оголошеннях масивів і при зверненні до окремих елементів масиву. |
| ; | Крапка з комою | Розділяє оператори. |
| , | Кома | Розділяє ідентифікатори в оголошеннях змінних, також використовується для зв'язку операторів в заголовку циклу *for*. |
| . | Точка | Відділяє імена пакетів від імен підпакетів і класів, також використовується для відділення імені змінної або методу від імені об’єкту. |

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Яка структура програми на мові Java?
2. Які є різновиди коментарів?
3. Наведіть приклади зарезервованих ключових слів.
4. Якими можуть бути ідентифікатори?
5. Наведіть приклади літералів різних типів.
6. Як позначають керуючі послідовності символів?
7. Які роздільники використовуються в мові?
8. Для чого потрібні змінні і як їх створювати?

## Консольне введення/виведення

### 1.3.1 Консольне виведення

Узагальнене поняття джерела введення відноситься до різних способів отримання інформації: читання дискового файлу, символів з клавіатури, або отримання даних з мережі. Аналогічно, під узагальненим поняттям виведення також можуть розумітися дискові файли, мережеве з'єднання і т.п. Ці абстракції дають зручну можливість для роботи з введенням-виведенням (I / O), не вимагаючи при цьому, щоб кожна частина коду розуміла різницю між, скажімо, клавіатурою і мережею. У Java ця абстракція називається *stream* і реалізована в декількох класах пакету *java.io*. Введення інкапсулироване в класі *InputStream*, виведення - в *OutputStream*. У Java є декілька спеціалізацій цих абстрактних класів, які враховують відмінності при роботі з дисковими файлами, мережевими з'єднаннями і навіть з буферами в пам'яті.

Для отримання даних, введених користувачем, а також для виведення повідомлень використовується і клас *System*. Цей клас розташовується в пакеті *java.lang*, який автоматично підключається до програми, тому нам не треба додатково імпортувати даний пакет і клас.

Для створення потоку виведення в класі *System* визначено об'єкт *out*. В цьому об'єкті визначено метод *println*, який дозволяє вивести на консоль деяке значення з подальшим переведенням консолі на наступний рядок:

System.out.println ( "Hello world");

Для форматованого виведення можна використовувати оператор *printf ()* (успадкований від мови С):

int printf (char \* format, <список виведення>);

Перший параметр - символьний рядок, який задає специфікацію формату. Список виведення - перелік змінних і виразів, значення яких будуть виведені.

Якщо після виведення рядка не потрібно переводити курсор на наступний рядок, можна використовувати метод print.

Наприклад, за допомогою printf можемо вивести на екран два числа наступним чином:

int x = 5;

int y = 6;

System.out.printf ( "x =% d; y =% d \ n", x, y);

В даному випадку символи % d - це специфікатор, який використовується для виведення цілочисельних значень типу int.

Крім специфікатора %d ми можемо використовувати ще декілька специфікаторів для виведення інших типів даних:

% x: шістнадцяткових чисел

% f: чисел з плаваючою точкою

% e: чисел у експоненціальному вигляді

% c: одиночного символу

% s: рядкових значень.

Наприклад:

String name = "Іван";

int age = 30;

float height = 1.7f;

System.out.printf ( "Ім'я: %s Вік: %d років Зріст: % .2f метрів \ n", name, age, height);

При виведенні чисел з плаваючою точкою ми можемо вказати кількість знаків після коми, для цього використовуємо специфікатор, наприклад, % .2f, де .2 вказує, що після коми буде два десяткових знака. У результаті ми отримаємо наступне:

Ім'я: Іван Вік: 30 років Зріст: 1,70 метрів

### 1.3.2 Консольне введення

Для реалізації консольного введення в класі *System* визначено об'єкт *in*. Однак використання тільки об'єкта System.in часто пов'язане з труднощами, тому, як правило, використовують клас *Scanner*, який, в свою чергу використовує System.in. Нижче показана програма, яка здійснює введення цілого числа і виведення його на екран:

іmport java.util.Scanner; // Клас Scanner знаходиться в пакеті java.util

 public class FirstApp {

public static void main (String [] args) {

Scanner in = new Scanner (System.in);

int num;

num = in.nextInt ();

System.out.println (num);

 }

}

Клас *Scanner* має ще декілька методів, які дозволяють отримати введені користувачем значення:

next (): зчитує рядок до першого пробілу

nextLine (): зчитує весь рядок

nextInt (): зчитує число int

nextDouble (): зчитує число double

hasNext (): перевіряє, чи було введено слово

hasNextInt (): перевіряє, чи було введено число int

hasNextDouble (): перевіряє, чи було введено double

Крім того, клас Scanner має ще такі методи: nextByte, nextShort, nextFloat, nextBoolean, які зчитують дані відповідних типів.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Які існують класи та методи введення-виведення?
2. Які засоби форматування існуть у методах консольного виведення?
3. Що таке специфікатор формату і як його задати?
4. Яке призначення класу *Scanner*?
5. Наведіть приклади методів класу *Scanner*?
6. Які бібліотеки необхідно підключати при використанні функцій введення-виведення?

## Типи даних

В Java існують такі різновиди типів даних змінних:

- прості аба примітивні типи (primitive types). До них відносяться стандартні, вбудовані в мову типи для представлення числових значень, поодиноких символів і булевих (логічних) значень. Всі примітивні типи мають визначений розмір пам'яті, яку вони займають. Це аналог типів даних мови С/С++. Вони не є класами, а введені в мову для підвищення продуктивності обробки даних;

- оболонки (обгортки) типів (wrapper type) – це класи, які «огортають» примітивні типи і мають різні корисні методи для роботи з цими типами;

- посилальні типи (reference type) - типи, визначені користувачем (класи, інтерфейси) і типи масивів. Всі посилальні типи є динамічними типами, для них виділяється пам'ять під час роботи програми.

В цьому розділі будуть розглянуті перші два різновиди типів даних. Посилальні типи будуть обговорюватись в наступних розділах.

### 1.4.1 Прості типи даних

*Прості або примітивні* типи залишені в Java щоб зручніше і швидше виконувати дії зі звичайними числами, а не з їх «обгортками» - об'єктами класів, які їм відповідають. Примітивні, чи базові типи поділяються на цілі типи, типи з крапкою, що плаває, символьні і булеві. Примітивні типи приведені в таблиці 1.4.1.

У багатьох інших мовах (наприклад, у С та у С++) формат і розміри примітивних типів даних залежать від платформи (DOS, Win 32 і т. д.). У Java розміри і формат стандартизовані і не залежать від платформи.

Константи цілого типу записуються як послідовності десяткових цифр. Тип константи за умовчанням - int. Він може бути уточнений додаванням наприкінці константи літер L чи l (тип long). Цілі константи можуть записуватися у вісімковій системі числення, у цьому випадку першою цифрою повинна бути цифра 0, число може містити тільки цифри 0...7. Цілі константи можна записувати й у шістнадцятковій системі числення, у цьому випадку запис константи починається із символів 0x чи 0X. Для позначення цифр понад 9 використовуються латинські літери a, b, c, d, e та f (великі або маленькі). Наприклад:

int octal = 023; // 19

int hex = 0xEF; // 239

У Java 7 з'явилася можливість визначати також двійкові константи (з використанням префіксів 0B або 0b):

int m = 0b110011; // 51

Крім того, групи розрядів у константах можна розділяти знаком підкреслення, наприклад:

int n = 1\_048\_576;

Константи типу char беруть в одиночні лапки (апострофи), значення константи задається або знаком з поточного набору символів, або цілою константою, якій передує зворотна коса риска (символ із заданим кодом). Є ряд спеціальних символів, що можуть використовуватись як значення константи типу char (такі подвійні символи називаються керуючими послідовностями):

'\n' - новий рядок,

'\t' - горизонтальна табуляція,

'\r' - переведення на початок рядку,

'\'' - одиночні лапки (апостроф),

'\"' - подвійні лапки,

'\\' - зворотна коса риска (backslash).

Для зберігання даних символьного типу в пам'яті використовується таблиця Unicode.

Константи дійсних типів можуть записуватись у формі з крапкою або в експонентному форматі і за умовчанням мають тип**double**. При необхідності тип константи можна уточнити, записавши наприкінці суфікс f чи F для типу **float**, суфікс d чи D для типу **double**. Наприклад:

1.5f // 1.5 типу float

2.4E-2d // 0.25 типу double

Значення з плаваючою крапкою не можна присвоювати цілим змінним. Перетворення до більш вузького типу (дійсного до цілого, числа з подвійною точністю до числа з одинарною точністю) пов'язано з ризиком утрати даних і повинне здійснюватися явно, а не за замовчуванням

int i = 10;

float f = i; // Таке перетворення допускається

long l = f; // Помилка!

long l1 = (long) f; // Явне приведення типів

Числа без десяткової крапки інтерпретуються як цілі (типу *int*). Числа з плаваючою крапкою мають тип *double*. Для приведення їх до більш вузьких типів використовується явне приведення типів:

float f = 10.5; // Помилка!

float f1 = (float) 10.5; // Явне приведення типів

float f2 = 10.5f; // Уточнення типу константи. Помилки немає

У Java немає беззнакових (unsigned) цілих типів.

Булевим змінним можна присвоювати тільки константи*true* та *false.* Змінні типу *boolean* не можна неявно чи явно приводити до інших типів і навпаки.

Константа-рядок складається із символів, які беруть у подвійні лапки. Наприклад:

"Це рядок"

Результат додавання рядку до змінної іншого типу забезпечує перетворення значення у представлення у вигляді рядка. Зокрема, такий підхід застосовують для виведення значень декількох змінних. Наприклад:

int k = 1;

double d = 2.5;

System.out.println(k + " " + d); // 1 2.5

### 1.4.2 Визначення локальних змінних

Локальні змінні визначаються (створюються) всередині методів. Опис локальних змінних у Java здійснюється аналогічно C++. Наприклад:

int i = 11;

double d= 0, x;

float f;

int j, k;

Локальні змінні можуть бути визначені в будь-якому місці усередині тіла функції, а також у вкладеному блоці. У Java не можна у внутрішньому блоці визначати імена, вже описані в зовнішньому блоці:

{

int i = 0;

{

int j = 1; // Змінна j визначається у внутрішньому блоці

int i = 2; // Помилка! Змінна i визначена в зовнішньому блоці

}

}

У Java не можна оголошувати змінні без їхнього створення.

Ключове слово *final* стосовно до імен змінних означає, що вони не можуть бути змінені.

final int h = 0;

Слово **const** зарезервоване, але не використовується.

#### Змінні типу var

Однією з цікавих можливостей, що з'явилися в Java 10, є визначення типу локальної змінної за допомогою слова *var* (*type inference*).

Для того, щоб використовувати *type inference*, потрібно описати змінну з використання конструкції виду:

var varName = <some init value>;

При цьому програмісту не потрібно явно вказувати тип змінної, цей тип буде визнаено компілятором на основі значення, яким ініціалізується змінна:

var test = "the type of this var will be String";

У представленому вище виразі змінної *test* буде присвоєно тип *String*.

У Java ключове слово *var* можна використовувати при оголошенні локальних змінних і індексів в удосконаленому циклі *for-loop*. Якщо *var* використовується спільно з оголошенням локальної змінної, то змінна повинна бути инициализирована будь-яким значенням.

Приклад з використанням циклу:

int [] arr = {1, 2, 3, 7, 8};

var sum = 0;

for (var i = 0; i <arr.length; i ++) {

sum + = arr [i];

}

Приклад показує, що тепер немає необхідності явно описувати тип змінної, через яку ми звертаємося до елементів масиву. Компілятор сам вирахує тип для i.

Важливо відзначити, що *var* в Java 10 є зарезервірованниі ім'ям типу, а не ключовим словом. У зв'язку з цим змінні і методи можна назвати ім'ям *var*, а ось для іменування своїх класів і інтерфейсів використовувати *var* не можна. Оскільки *var* не є ключовим словом, наступний фрагмент коду є коректним:

var var = 42; // локальна змінна var з обчисленим типом int,

// ініціалізувати значенням 42

В Java 10 можливості *var* досить обмежені. Наприклад, змінні, оголошені як *var*, не можуть використовуватися в сигнатурі методів і інших конструкціях, відмінних від оголошень локальних змінних і *for*-циклів. Крім того, *var* можна використовувати при ініціалізації змінної масивом через {}, ініціалізації декількох змінних в одному рядку, а також в ряді інших випадків.

Також варто відзначити, що при ініціалізації змінної через *null, var* використовувати не можна.

// var x = null; // код не скомпілюється

String x = null;

### 1.4.3 Класи – оболонки типів

*Класи-оболонки або обгортки типів (wrapper classes)* Java є об'єктними поданнями восьми примітивних типів в Java. Всі класи-оболонки в Java є *final*, тобто після створення об’єкту класу його не можна змінювати. Починаючи з Java 5 автоупакування і розпакування дозволяє легко конвертувати примітивні типи в їх відповідні класи-оболонки і навпаки.

*Класи-оболонки типів* спрощують програму, тому що:

* просто створюються, тестуються та використовуються;
* потоконебезпечні – не мають проблеми синхронізації;
* не потрібен конструктор копіювання;
* не потрібна реалізація клонування;
* зручні в якості ключів в колекціях Map та Set ;
* перевіряються один раз при побудові.

Ці класи мають додаткові методи обробки даних. З класами-оболонками не можна виконувати арифметичні дії. Потрібно спочатку отримати доступ до значень базових типів, укладених в клас-оболонку.

На рис. 1.4.1 показана одна з гілок ієрархії класів Java. Числові класи мають спільного попередника - абстрактний клас *Number*, в якому описані шість методів, які повертають числове значення, що міститься в класі, перетворене у відповідний примітивний тип: byteValue (), doubleValue (), floatValue (), intValue (), longValue (), shortValue (). Ці методи перевизначені в кожному з шести числових класів-оболонок.

Крім методу порівняння об'єктів *equals* (), перевизначеного з класу *object*, є метод *compareTo* (), що порівнює числове значення, що міститься в даному об'єкті, з числовим значенням об'єкта - аргументом методу *compareTo* (). В результаті роботи методу отримуємо одне з таких цілих значень:

* 0, якщо значення рівні;
* від’ємне число (-1), якщо числове значення в даному об'єкті менше, ніж в об'єкті-аргументі;
* додатне число (+1), якщо числове значення в даному об'єкті більше числового значення, що міститься в аргументі.

У кожному з шести числових класів-оболонок є статичні методи перетворення рядка символів типу *String*, який представляє число, в відповідний примітивний тип: Byte.parseByte (), Double.parseDouble (), Float.parseFloat (), Integer.parseInt (), Long. parseLong (), Short.parseShort (). Ці методи корисні усюди, де числа представляються рядками цифр зі знаками плюс або мінус і десяткового точкою.

Табл.1.4.1 Прості типи даних.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ключове слово | Тип | Розмір | Межі значень | Значення по замовчуванню |
| byte | байт | 8 біт | Від –27 до 27-1 | 0 |
| short | коротке ціле | 16 біт | Від –215 до 215-1 | 0 |
| int | ціле | 32 біт | Від –231 до 231-1 | 0 |
| long | довге ціле | 64 біт | Від –263 до 263-1 | 0L |
| float | плаваюча точка | 32 біт | Від 1.7\*10-38 до 1.7\*1038 | 0.0f |
| double | подвійна точність | 64 біт | Від -1.40239846E-45 до 3.40282347E+38 | 0.0d |
| char | символьний | 16 біт | букви, цифри, символи, засновані на 16-бітному наборі символів Unicode  від Unicod 0 до Unicod 216-1 | null |
| boolean | логічний (булевий) |  | true або false | false |
| void | «пусте» значення |  |  |  |

#### Класи Double і Float

*Double і Float* - підкласи класу *Number*. На застосунок до методів, оголошених в суперкласі, ці класи містять декілька інших методів, які полегшують роботу зі значеннями *double* і *float*. У кожного з класів є конструктори, що дозволяють форматувати об'єкти значеннями типів *double* і *float*, крім того, для зручності користувача, ці об'єкти можна форматувати і об'єктом *String*, що містить текстове представлення дійсного числа.

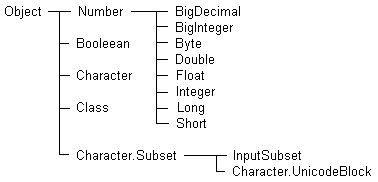


Рис. 1.4.1. Класи-оболонки примітивних типів

Можна скористатися статичними методами *floatToIntBits* () і *doubleToLongBits* (), що перетворюють дійсне значення в ціле. Дійсне число задається як аргумент методу. Зворотнє перетворення цілого числа в дійсне можна виконати методами *intBitsToFloat* () і *longBitsToDouble* ().

Наведений нижче приклад ілюструє створення представників класу *Double* за допомогою обох конструкторів.

class DoubleDemo {

public static void main (String args []) {

Double d1 = new Double (3.14159);

Double d2 = new Double ( "314159E-5");

System.out.println (d1 + "=" + d2 + "->" + d1.equals (d2));

}}

Як можна бачити з результату роботи цієї програми, метод *equals* повертає значення *true*, а це означає, що обидва використаних в прикладі конструктора створюють ідентичні об'єкти класу *Double*.

3.14159 = 3.14159 -> true

Створити об’єкт класу-оболонку також можна за допомогою статичного методу valueOf (String):

Integer intObject3 = Integer.valueOf("11");

#### Нескінченність і невизначеність

У специфікації IEEE для чисел з плаваючою точкою є два значення типу *double*, які трактуються спеціальним чином: *нескінченність* і *NaN* (Not a Number - *невизначеність*).

У класі *Double* є константи POSITIVE\_INFINITY, NEGATIVE\_INFINITY, NaN і тести для перевірки обох цих умов, причому в двох формах - у вигляді методів (статичних), яким значення *double* передається як параметр, і у вигляді методів, які перевіряють число, яке зберігається в об'єкті класу *Double*.

• isInfinite (d) повертає true, якщо абсолютне значення зазначеного числа типу double нескінченно велике.

• isInfinite () повертає *true*, якщо абсолютне значення числа, що зберігається в даному об'єкті *Double*, нескінченно велике.

• isNaN (d) повертає *true*, якщо значення зазначеного числа типу *double* не визначено.

• isNaN () повертає *true*, якщо значення числа, що зберігається в даному об'єкті *Double*, не визначено.

Черговий приклад створює два об'єкти *Double*, один з нескінченним, інший з невизначеним значенням.

class InfNaN {

public static void main (String args []) {

Double d1 = new Double (1/0.);

Double d2 = new Double (0/0.);

System.out.println (d1 + ":" + d1.isInfinite () + "," + d1.isNaN ());

System.out.println (d2 + ":" + d2.isInfinite () + "," + d2.isNaN ());

}}

Нижче наведено результат роботи цієї програми:

Infinity: true, false

NaN: false, true

#### Класи Integer і Long

Клас *Integer* - клас-оболонка для чисел типів *int, short і byte*, a клас *Long* - відповідно для типу *long*. Крім успадкованих методів свого суперкласу *Number*, класи *Integer* і *Long* містять методи для розбору текстового представлення чисел, і навпаки, для представлення чисел у вигляді текстових рядків. Різні варіанти цих методів дозволяють вказувати систему числення, яка використовується під час перетворення. Зазвичай використовуються двійкова, вісімкова, десяткова і шістнадцяткова системи числення.

Статичними методами *toBinaryString (), toHexString () і toOctalString ()* класів *Integer* і *Long* можна перетворити цілі значення типів *int* і *long*, заданими в якості аргументів методу, в рядок символів у двійковому, шістнадцятковому або вісімковому поданню числа.

Метод *toString (int)* - перетворює передане в якості параметра ціле число в текстове представлення в десятковій системі числення, а метод *toString (int, radix)* перетворює передане в якості першого параметра ціле число в текстове представлення в системі числення, яка задається другим параметром.

#### Клас Character

*Character* - клас-оболонка типу *char*. У нього є декілька корисних статичних методів, за допомогою яких можна виконувати для символу різні перевірки і перетворення.

Статичний метод *digit (char ch, in radix)* переводить цифру *ch* у системі числення з основою radix в її числове значення типу *int*.

Статичний метод *forDigit (int digit, int radix)* виконує зворотне перетворення цілого числа *digit* в відповідну цифру (типу *char*) в системі числення з основою *radix*.

Основа системи числення має перебувати в діапазоні від Character.MIN\_RADIX до Character.MAX\_RADIX.

Метод *toString* () переводить символ, що міститься в класі, в рядок з тим же символом

Ось ще деякі з методів:

* *isLowerCase (char ch)* - повертає *true*, якщо символ-параметр знаходиться у нижньому регістрі.
* *isUpperCase (char ch)* - робить те ж саме, що й попередній метод, але якщо символ знаходиться у верхньому регістрі.
* *isDigit (char ch) і isSpace (char ch)* - повертають *true* для цифр і пробілів, відповідно.
* *toLowerCase (char ch) і toUpperCase (char ch)* - виконують перетворення символів з верхнього в нижній регістр і навпаки.
* *isDefined* () - з'ясовує, чи визначено символ в кодуванні Unicode;
* *isIsoControl* () - визначає, чи є символ керуючим;
* *isJavaIdentifierPart* () - з'ясовує, чи можна використовувати символ в ідентифікаторах;
* *isJavaIdentifierStart* () - визначає, чи може ідентифікатор починатися з символу;
* *isLetter* () - перевіряє, чи є символ буквою Java;
* *IsLetterOrDigit* () - перевіряє, чи є символ буквою або цифрою Unicode;
* *isTitieCase* () - перевіряє, чи є символ титульним;
* *isUnicodeIdentifierPart* () - з'ясовує, чи можна використовувати символ в іменах Unicode;
* *isWhitespace* () - з'ясовує, чи є символ пробільним.

В клас *Character* вкладені класи *Subset* і *UnicodeBlock*, причому клас *Unicode* і ще один клас, *inputSubset*, є розширеннями класу *Subset*, як це видно на рис. 1.4.1. Об'єкти цього класу містять підмножини *Unicode*.

#### Клас Biglnteger

Всі примітивні числові типи мають обмежений діапазон значень. Для того щоб можна було виконувати обчислення з цілими числами будь-якої розрядності, до складу Java API введено клас *BigІnteger*, що зберігається в пакеті *java.math*. Цей клас розширює клас Number, отже, в ньому перевизначені методи doubleValue (), floatValue (), intValue (), longValue (). Методи byteVaiue () і shortValue () не перевизначені, а прямо успадковуються від класу Number.

Дії з об'єктами класу *BigInteger* не призводять до переповнення. Якщо результат виконання операції – велике число, то число розрядів для нього просто збільшується. Числа зберігаються в двійковій формі з додатковим кодом.

Перед виконанням операції числа вирівнюються по довжині поширенням знакового розряду.

Шість конструкторів класу створюють об'єкт класу *BigDecimai* з рядка символів (знака числа і цифр) або з масиву байтів.

Дві константи - ZERO і ONE - моделюють нуль і одиницю в операціях з об'єктами класу BigІnteger.

Метод toByteArray () перетворює об'єкт в масив байтів.

Більшість методів класу BigІnteger моделюють цілочисельні операції та функції, повертаючи об'єкт класу BigInteger:

* abs () - повертає об'єкт, який містить абсолютне значення числа, що зберігається в даному об'єкті this;
* add (x) - операція this + х;
* and (x) - операція this & х;
* andNot (x) - операція this & (~ х);
* divide (x) - операція this / x;
* divideAndRemainder (х) - повертає масив з двох об'єктів класу Biglnteger, що містять частку і залишок від ділення this на х;
* gcd (x) - найбільший спільний дільник, абсолютних значень об'єкта this і аргументу х;
* mах (х) - найбільше із значень об'єкта this і аргументу х;
* min (x) - найменше з значень об'єкта this і аргументу х;
* mod (x) - залишок від ділення об'єкта this на аргумент методу х;
* modinverse (x) - залишок від ділення числа, зворотного до об'єкту this, на аргумент х;
* modPow (n, m) - залишок від ділення об'єкта this, зведеного в ступінь n, на m;
* multiply (х) -операція this \* х;
* negate () - зміна знака числа, що зберігається в об'єкті;
* not () - операція ~ this;
* ОR (х) - операція this | х;
* pow (n) - операція зведення числа, що зберігається в об'єкті, в ступінь n;
* remainder (х) -операція this% х;
* shiftLeft (n) - операція this <<n;
* shiftRight (n) - операція this >>n;
* signum () - функція sign (x);
* subtract (x) - операція this - x;
* xor (x) - операція this ^ x.

#### Клас BigDecimal

Клас *BigDecimal* знаходиться в пакеті *java.math*.

Кожен об'єкт цього класу зберігає два цілочисельних значення: *мантиссу* дійсного числа у вигляді об'єкта класу *BigInteger*, і невід'ємних десятковий *порядок* числа типу *int*.

Наприклад, для числа 76.34862 буде зберігатися мантиса 7 634 862 в об'єкті класу *BigInteger*, і порядок 5 як ціле число типу *int*. Таким чином, мантиса може містити будь-яку кількість цифр, а порядок обмежений значенням константи Integer.MAX\_VALUE. Результат операції над об'єктами класу *BigDecimal* округлюється по одному з восьми правил, визначених наступними статичними цілими константами:

* ROUND\_CEILING - округлення в сторону більшого цілого;
* ROUND\_DOWN - округлення до нуля, до меншого за модулем цілого значення;
* ROUND\_FLOOR - округлення до меншого цілого;
* ROUND\_HALF\_DOWN - округлення до найближчого цілого; середнє значення округляється до меншого цілого;
* ROUND\_HALF\_EVEN - округлення до найближчого цілого, середнє значення округляється до парного числа;
* ROUND\_HALF\_UP - округлення до найближчого цілого, середнє значення округляється до більшого цілого;
* ROUND\_UNNECESSARY - передбачається, що результат буде цілим, і округлення не знадобиться;
* ROUND\_UP - округлення від нуля до більшого по модулю цілого значення.

У класі *BigDecimal* чотири конструктора:

* BigDecimal (BigInteger bi) - об'єкт буде зберігати велике ціле bi, порядок дорівнює нулю;
* BigDecimal (BigInteger mantissa, int scale) - задається мантиса *mantissa* і невід'ємний порядок *scale* об'єкта; якщо порядок *scale* негативний, виникає виняткова ситуація;
* BigDecimal (double d) - об'єкт буде містити дійсне число подвоєної точності d; якщо значення d нескінченне або NaN (невизначене), то виникає виняткова ситуація;
* BigDecimal (String val) - число задається рядком символів val, який повинен містити запис числа за правилами мови Java.

При використанні третього з перерахованих конструкторів виникає неприємна особливість, зазначена в документації. Оскільки дійсне число при переведені в двійкову форму представляється, як правило, нескінченим двійковим дробом, то при створенні об'єкта, наприклад, *BigDecimal*(0.1), мантиса, що зберігається в об'єкті, виявиться дуже великою. Але при створенні такого ж об'єкта четвертим конструктором, *BigDecimal*("0.1"), мантиса буде дорівнювати 1.

У класі перевизначені методи doubleValue (), floatValue (), intValue (), longValue ().

Більшість методів цього класу моделюють операції з дійсними числами. Вони повертають об'єкт класу *BigDecimal*. Тут буква х позначає об'єкт класу *BigDecimal*, буква *n* - ціле значення типу int, буква *r* - спосіб округлення (у вигляді однії з восьми перерахованих вище констант):

* abs () - абсолютне значення об'єкта this;
* add (x) - операція this + х;
* divide (х, r) - операція this / г з округленням за способом r;
* divide (х, n, r) - операція this / г зі зміною порядку і округлення за способом r;
* mах (х) - найбільше з this і х;
* min (x) - найменше з this і х;
* movePointLeft (n) - зсув вліво на n розрядів;
* movePointRight (n) - зсув праворуч на n розрядів;
* multiply (х) - операція this \* х;
* negate () - повертає об'єкт з протилежним знаком;
* scale () - повертає порядок чісла;
* setscale (n) – встановлює новий порядок n;
* setscale (n, r) - встановлює новий порядок n і округлює число при необхідності за способом r;
* signumo - знак числа, що зберігається в об'єкті;
* subtract (х) - операція this - х;
* toBigInteger () - округлення числа, що зберігається в об'єкті до toBigInteger ();
* unscaiedvalue () - повертає мантиссу числа.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Перерахуйте відомі вам типи даних.
2. Які типи даних називають простими (примітивними)?
3. Що таке оболонки (обгортки) типів?
4. Що собою являє тип даних char і коли його доцільно використовувати?
5. Що собою являє тип даних boolean і коли його доцільно використовувати?
6. Перерахуйте числові типи даних.
7. Яке призначення класів Biglnteger і BigDecimal?

## 1.5. Арифметичні та логічні оператори

### 1.5.1 Арифметичні оператори

*Оператори* в мові Java - це спеціальні символи, які повідомляють транслятор про те, що ви хочете виконати операцію з деякими операндами. Деякі оператори вимагають одного операнда, їх називають *унарними*. Одні оператори ставляться перед операндами і називаються *префіксними*, інші - після, їх називають *постфіксними*. Більшість же операторів ставлять між двома операндами, такі оператори називаються *інфіксними* *бінарними* операторами. Існує *тернарний* оператор, який працює з трьома операндами.

Арифметичні оператори використовуються для обчислень (див. Табл. 1.5.2). Їх операнди повинні мати числові типи. Використовувати ці оператори для роботи з логічними типами не можна, а для роботи з типом *char* можна, оскільки в Java тип *char* - це підмножина типу *int* (Табл.1.5.1).

Арифметичні оператори працюють як з цілими літералами, так і зі змінними.

 int a = 1 + 1;

int b = a \* 3;

Оператор ділення по модулю ( його часто називають *mod)*, позначається символом *%*. Цей оператор повертає залишок від ділення першого операнда на другий. Цей оператор в Java працює не тільки з цілими, але і з типами з плаваючою точкою. Наведена нижче програма ілюструє роботу цього оператора.

int x = 42;

double у = 42.3;

System.out.println ( "x mod 10 =" + x% 10);

System.out.println ( "y mod 10 =" + у% 10);

Для кожного з арифметичних операторів є форма, в якій одночасно із заданою операцією виконується присвоювання. Нижче наведено приклад, який ілюструє використання подібного різновиду операторів.

class OpEquals {

public static void main (String args []) {

int a = 1;

int b = 2;

int с = 3;

a + = 5;

b \* = 4;

c + = a \* b;

с% = 6;

System.out.println ( "a =" + a);

System.out.println ( "b =" + b);

System.out.println ( "c =" + c);

}}

**Результат**

а = 6

b = 8

с = 3

Таблиця 1.5.1. Арифметичні оператори Java

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Результат | Оператор | Результат |
| + | Додавання | + = | Додавання з присвоюванням |
| - | Віднімання (також унарний мінус) | - = | Віднімання з присвоюванням |
| \* | Множення | \* = | Множення з присвоюванням |
| / | Ділення | / = | Ділення з присвоюванням |
| % | Ділення по модулю | % = | Ділення по модулю з присвоюванням |
| ++ | Інкремент | -- | Декремент |

Оператори інкремента і декремента (++ і --) - є скороченим варіантом запису для складання або віднімання операнда з одиницею. Ці оператори унікальні в тому плані, що можуть використовуватися як в префиксной, так і в постфіксній формі. Наступний приклад ілюструє використання операторів інкремента і декремента.

class IncDec {

public static void main (String args []) {

int a = 1;

int b = 2;

int c = ++ b;

int d = a ++;

c ++;

System.out.println ( "a =" + a);

System.out.println ( "b =" + b);

System.out.println ( "c =" + c);

System.out.println ( "d =" + d);

}}

**Результат**

a = 2

b = 3

c = 4

d = 1

### 1.5.2 Бітові оператори

Для цілих числових типів даних — long, int, short, char і byte, визначено додатковий набір операторів, за допомогою яких можна перевіряти і змінювати стан окремих бітів відповідних значень. В таблиці наведена зведення таких операторів. Оператори бітової арифметики працюють з кожним бітом як з самостійною величиною (Табл.1.5.2).

Таблиця 1.5.2. Бітові оператори Java

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Результат | Оператор | Результат |
| ~ | побітове унарне заперечення (NOT) |  |  |
| & | побітове І (AND) | &= | побітове І (AND) з присвоюванням |
| | | побітове АБО (OR) | |= | побітове АБО (OR) з присвоюванням |
| ^ | побітове виключне АБО (XOR) | ^= | побітове виключне АБО (XOR) з присвоюванням |
| >> | зсув праворуч | >> = | зсув праворуч з присвоюванням |
| >>> | зсув праворуч з заповненням нулями | >>>= | зсув праворуч з заповненням нулями з присвоюванням |
| << | зсув ліворуч | <<= | зрушення ліворуч з присвоюванням |

Нижче наведено приклад програми, яка маніпулює бітами.

class Bitlogic {

public static void main(String args []) {

String binary[] = { "OOOO", "0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000", "1001", "1010", "1011", "1100", "1101",

"1110", "1111" };

int a = 3; // 0+2+1 або двійкове 0011

int b = 6; // 4+2+0 або двійкове 0110

int c = a | b;

int d = a & b;

int e = a ^ b;

int f = (~a & b) | (a & ~b);

int g = ~a & 0x0f;

System.out.println(" a = " + binary[a]);

System.out.println(" b = " + binary[b]);

System.out.println(" ab = " + binary[c]);

System.out.println(" a&b = " + binary[d]);

System.out.println(" a^b = " + binary[e]);

System.out.рrintln("~a&b|а^~Ь = " + binary[f]);

System.out.println(" ~a = " + binary[g]);

} }

**Результат**

a = 0011

b = 0110

a | b = 0111

a & b = 0010

a ^ b = 0101

~a & b | a & ~b = 0101

~а = 1100

Оператор << виконує зсув ліворуч всіх бітів свого лівого операнда на число позицій, заданий операндом, який стоїть праворуч. При цьому частина бітів у лівих розрядах виходить за межі і втрачається, а відповідні праві позиції заповнюються нулями.

Оператор >> означає в мові Java зсув праворуч. Він переміщує всі біти свого лівого операнда праворуч на число позицій, заданих правим операндом. Коли біти лівого операнда висуваються за саму праву позицію, вони губляться. При зсуві праворуч звільняються старші (ліві) розряди і вони заповнюються попереднім вмістом знакового розряду. Таку поведінку називають розширенням знакового розряду.

Так само, як і в випадку арифметичних операторів, у всіх бінарних бітових операторів є форма, що дозволяє автоматично привласнювати результат операції лівому операнду. Це так звані «бітові оператори присвоювання» (зі знаком рівності в кінці оператора).

### 1.5.3 Оператори порівняння

Для того, щоб можна було порівнювати два значення, в Java є набір операторів, список яких наведено в таблиці 1.5.3.

Таблиця 1.5.3. Оператори порівняння

|  |  |
| --- | --- |
| Оператор | Результат |
| == | дорівнює |
| ! = | не дорівнює |
| > | більше |
| < | менше |
| > = | більше або дорівнює |
| <= | менше або дорівнює |

Значення будь-яких типів можна порівнювати, використовуючи оператор перевірки на рівність == і нерівність ! =. У мові Java, так само, як в С/C ++ перевірка на рівність позначається послідовністю (==). Один знак (=) - це оператор присвоювання.

### 1.5.4 Булеві логічні оператори

Булеві логічні оператори, зведення яких наведено в таблиці, оперують лише з операндами типу *boolean*. Всі бінарні логічні оператори сприймають в якості операндів два значення типу *boolean* і повертають результат того ж типу (Табл.1.5.4). Програма, наведена нижче, практично повністю повторює вже знайомий вам приклад BitLogic. Тільки тут ми працюємо з булевими логічними значеннями.

class BoolLogic {

public static void main (String args []) {

boolean a = true;

boolean b = false;

boolean c = a | b;

boolean d = a & b;

boolean e = a ^ b;

boolean f = (! a & b) | (A &! B);

boolean g =! a;

System.out.println ( "a =" + a);

System.out.println ( "b =" + b);

System.out.println ( "a | b =" + c);

System.out.println ( "a & b =" + d);

System.out.println ( "a ^ b =" + e);

System.out.println ( "! A & b | a &! B =" + f);

System.out.println ( "! A =" + g);

}}

**Результат**

а = true

b = false

a | b = true

a & b = false

a ^ b = true

! A & b | a &! B = true

! A = false

#### Оператори швидкої оцінки логічних виразів

Існують два додаткових доповнення до логічних операторів. Це - альтернативні версії операторів AND і OR, які застосовуються для швидкої оцінки логічних виразів (short circuit logical operators). Як відомо, якщо перший операнд оператора OR має значення *true*, то незалежно від значення другого операнда результатом операції буде величина *true*. Аналогічно в разі оператора AND, якщо перший операнд - *false*, то значення другого операнда на результат не впливає - він завжди буде дорівнювати *false*. Якщо ви використовуєте оператори && і || замість звичайних форм & і |, то Java не перевіряє операнд логічного виразу, який стоїть праворуч, якщо відповідь зрозуміла зі значення лівого операнда. Загальноприйнятою практикою є використання операторів && і || в переважній більшості випадків оцінки булевих логічних виразів. Версії операторів & і | застосовуються тільки в бітової арифметиці.

#### Тернарний оператор

Загальна форма оператора така:

вираз1? вираз2: виразЗ

В якості першого операнда — *вираз1*— може бути використаний будь-який вираз, результатом якого є значення типу *boolean*. Якщо результат дорівнює *true*, то виконується оператор, заданий другим операндом, тобто, *вираз2*. Якщо ж перший операнд дорівнює *false*, то виконується третій операнд — *вираз3*. Другий і третій операнди повинні повертати значення одного типу і не повинні мати тип *void*.

У наведеній нижче програмі цей оператор використовується для перевірки дільника перед виконанням операції ділення. У разі нульового дільника повертається значення 0.

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Результат | Оператор | Результат |
| & | логічне І (AND) | & = | логічне І (AND) з привласненням |
| | | логічне АБО (OR) | = | логічне АБО (OR) з привласненням |
| ^ | логічне виключає АБО (XOR) | ^ = | логічне виключає АБО (XOR) з привласненням |
| || | оператор OR швидкої оцінки виразів | == | дорівнює |
| && | оператор AND швидкої оцінки виразів | ! = | не дорівнює |
| ! | логічне унарне заперечення (NOT) | ?: | тернарний оператор |

class Ternary {

public static void main(String args[]) {

int a = 42;

int b = 2;

int c = 99;

int d = 0;

int e = (b == 0) ? 0 : (a / b);

int f = (d == 0) ? 0 : (c / d);

System.out.println("a =" + a);

System.out.println("b =" + b);

System.out.println("c =" + c);

System.out.println("d =" + d);

System.out.println("a / b =" + e);

System.out.println("c / d =" + f);

} }

а = 42

b = 2

c = 99

d = 0

a / b = 21

c / d = 0

Таблиця 1.5.4. Булеві логічні оператори Java

### 1.5.5 Пріоритети операторів

У Java діє певний порядок, або пріоритет, операцій. У таблиці вказані оператори в порядку зменшення їх пріоритету виконання.

Круглі дужки ( ) використовуються для явної установки пріоритету. Квадратні дужки [ ] використовуються для індексування змінної масиву. Оператор «.» (точка) використовується для виділення елемента з посилання на об'єкт.

Оскільки вищий пріоритет мають круглі дужки, завжди можна додати в вираз кілька пар дужок, якщо у є сумніви з приводу порядку обчислень.

Наприклад, для виразу, а >> b + 3 еквівалентним буде а >> (b + 3), оскільки у оператора + більш високий пріоритет, ніж у оператора зсуву. Тому якщо вам треба спочатку виконати операцію зсуву, то слід написати так: (а >> b) + 3.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Наведіть приклади префіксних, постфіксних і інфіксних операторів Java.
2. Наведіть приклади унарних і бінарних операторів Java.
3. Які арифметичні оператори з присвоюванням ви знаєте?
4. Які існують оператори інкременту і декременту?
5. Які бітові оператори ви знаєте?
6. Які булеві логічні оператори ви знаєте?
7. Яке призначення операторів швидкої оцінки логічних виразів?
8. Наведіть приклад використання тернарного оператора.
9. В якому порядку виконуються оператори у формулах?

## Класи і об’єкти

### 1.6.1 Класи

Базовим елементом об'єктно-орієнтованої мови програмування Java є *клас*. Для того, щоб створити клас в Java, достатньо мати вихідний файл, в якому буде присутній ключове слово *class*, слідом за ним — допустимий ідентифікатор і пара фігурних дужок для його тіла.

**Клас** - це опис деякої множини об'єктів із загальними атрибутами, методами, відносинами.

**Клас** - це абстрактний тип даних, який визначає форму і поведінку об'єкта.

**Дані** (атрибути) об'єкта називаються **полями**.

**Функції**, що визначають поведінку об'єкта (що описують операції, що виконуються над даними) називаються **методами**.

**Загальний опис класу**

{<доступ>} **class** <ідентифікатор> {

{<доступ>} <член класу>

}

Ім'я вихідного файлу Java повинно зберігатися з іменем класу в ньому.

Клас визначає загальну структуру об'єктів, які йому належать. Наприклад, клас СТУДЕНТ описує загальні характеристики, властиві всім студентам (прізвище, ім'я, спеціальність, факультет, курс, його екзаменаційні бали та інші). На підставі класу можна створити *об'єкт* цього класу (або іноді кажуть *примірник* або *представник* класу), тобто у нашому прикладі це буде вже конкретний студент з усіма його фактичними характеристиками, зазначеними в класі. Можна сказати, що клас — це шаблон для створення об'єкта.

Характеристики класу описуються змінними. Крім того в класі можуть бути *методи* (функції), які виконують зазначені дії в рамках даного класу, тобто вони утворюють функціональний інтерфейс. Наприклад, у класі СТУДЕНТ може бути метод, який обчислює його середній бал за сесію.

Нижче наведена розширена форма визначення класу.

class iм’я\_класу extends ім’я\_суперкласу

{

type змінна1\_об’єкту;

type змінна2\_об’єкту;

...

type зміннаN\_об’єкту;

type iм’я методу1(список\_параметрів)

{

тіло методу;

}

type ім’я методу2(список\_параметрів)

{

тіло методу;

}

...

type ім'я методуN(список\_параметрів)

{

тіло методу;

}

}

Ключове слово *extends* вказує на те, що «iм’я\_класу» — це підклас класу «ім’я\_суперкласу».

На вершині класової ієрархії Java стоїть вбудований клас *Object*. Якщо ви хочете створити підклас безпосередньо цього класу, ключове слово *extends* і наступне за ним ім'я суперкласу можна не вказувати — транслятор вставить їх в ваше визначення автоматично.

**Методи класу Object**

|  |  |
| --- | --- |
| *Назва* | *Опис* |
| **clone()** | Створення клону об’єкта |
| **equals(Object obj)** | Порівнняння двох об’єктів |
| **finalize()** | Виконання завершальної роботи перед видаленням об'єкта збиральником сміття |
| **hashCode()** | Генерація унікального ідентифікатора об’єкта |
| **toString()** | Повернення рядка символів з описом об’єкта (автоматично викликається методами *print()* і *println()*) |
| **wait()** | Перевдення потоку в стан очикування |
| **notify()** | Відновлення (повідомлення) одного з потоків, який визвав метод *wait()* на тому ж об’єкті |
| **notifyAll()** | Відновлення (повідомлення) всіх потоків, які викликали метод *wait()* на тому ж об’єкті |
| **getClass()** | Повертає опис класу об’єкта |

#### Змінні класу

Дані включаються (інкапсулюються) в клас шляхом оголошення змінних між відкриваючою і закриваючою фігурними дужками, що обмежують у визначенні класу його тіло. Ці змінні оголошуються так само, як оголошуються локальні змінні. Єдина відмінність полягає в тому, що їх треба оголошувати поза методів, у тому числі поза методу *main*, з якого починається виконання програми. Нижче наведено фрагмент коду, в якому оголошено клас *Point (точка)* з двома змінними типу *int* (координати точки).

class Point

{

int х, у;

}

### 1.6.2 Об’єкти

Оператор *new* створює об'єкт (екземпляр) зазначеного класу і повертає посилання на новостворений об'єкт. Нижче наведено приклад створення і присвоювання змінної р екземпляра класу *Point*.

Point р = new Point();

Додамо до класу *Point* метод *main* і, тим самим, отримаємо закінчену програму:

class Point

{

int х, у;

public static void main(String args[]) {

Point p = new Point();

р. х = 10;

p.у = 20;

System.out.println("x =" + р. х + " = " + p.y);

}

}

Тут створено клас *Point*, в методі main якого створюється об'єкт цього класу р - реальна точка. Координатам цієї точки присвоюються конкретні значення 10 і 20. Крапка в операторі р.х = 10; використовується для доступу до змінної х класу р, а також може використовуватися для звернення до методів об'єкта.

Наведена нижче програма складається з двох класів. У класі *MyObject* створюється два різних об'єкта класу *Point* і для кожного з них встановлюються свої власні значення координат.

public class MyObject {

public static void main(String args[]) {

Point p1 = new Point();

Point p2 = new Point();

p1.x = 10;

p1.y = 20;

p2.x = 42;

p2.y = 99;

System.out.println("x =" + p1.x + " y= " + p1.y);

System.out.println("x =" + p2.x + " y= " + p2.y);

}

}

class Point {

int x, y;

}

У цьому прикладі було створено два об'єкти класу *Point* і їх змінним *х* та *у* присвоєні різні значення. Таким чином ми продемонстрували, що змінні різних об'єктів незалежні. Нижче наведено результат, отриманий при виконанні цієї програми.

х = 10 y = 20

х = 42 y = 99

### 1.6.3 Посилання. Збирання сміття

У Java немає типів указівників. Імена змінних непримітивних типів по суті є іменами посилань на відповідні об'єкти. Усі непримітивні типи мають назву типів-посилань. Розіменування не потрібно: звертання до примітивних типів завжди здійснюється за значенням, а до непримітивних - по посиланню. Типи-посилання ніколи не можуть бути приведені до примітивних і навпаки. Java також не підтримує адресної арифметики.

Спеціальне ключове слово null використовують для того, щоб показати, що змінна типу-посилання ні на що не посилається. Константа null може бути присвоєна змінній будь-якого типу-посилання.

Об'єкти, на які вказують посилання, повинні бути розміщені в динамічній пам'яті за допомогою операції **new**:

SomeType st = new SomeType();

Присвоювання значення одного посилання іншому не забезпечує копіювання об'єктів. Після присвоювання два посилання посилаються на один об'єкт.

SomeType a = new SomeType();

SomeType b = new SomeType();

a = b;

Об'єкт, на який посилалося посилання a, загублений.

На відміну від С++, звільнення пам'яті від непотрібних об'єктів не потрібно. У Java немає операції *delete*. Для звільнення пам'яті використовується спеціальний механізм, який має назву збирання сміття. Цей механізм базується на підрахунку посилань на об'єкти. Кожен об'єкт має свій лічильник посилань. Коли посилання копіюється в нову змінну типу-посилання, лічильник збільшується на одиницю. Коли посилання виходить з області видимості, чи перестає вказувати на об'єкт, лічильник зменшується на одиницю. Коли віртуальній машині Java не вистачає оперативної пам'яті, запускається збирач сміття. Він переглядає список об'єктів і видаляє з пам'яті всі об'єкти, для яких кількість посилань дорівнює 0.

Операція == стосовно до змінних-посилань здійснює порівняння адрес, а не вмісту об'єктів.

Аргументи функцій типу-посилання передаються у функції за посиланням. Всередині функції створюється нове посилання на той самий об'єкт. Цей об'єкт можна змінити у функції та після повернення з функції використовувати його значення.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що називається класом?
2. Що називається об’єктом класу?
3. Чим відрізняється оголошення об’єкту від його визначення?
4. Як співвідносяться поміж собою поняття об’єкта та класу?
5. Яке призначення слова «extends» в визначенні класу
6. Що означають терміни «підклас» і «суперклас»?
7. Який клас стоїть на вершині класової ієрархії Java?
8. Яка різниця між вбудованим і користувацьким класами?

## Методи і конструктори

### 1.7.1 Методи

*Методи (функції)* - це підпрограми, приєднані до конкретних визначень класів. Вони описуються усередині визначення класу на тому ж рівні, що і змінні об'єктів. При оголошенні методу задаються тип даних результату, який повертається методом і список параметрів. Загальна форма оголошення методу така:

тип і’мя методу (список формальних параметрів)

{

тіло методу

}

Тип результату, який повинен повертати метод може бути будь-яким, в тому числі і типом *void* - у тих випадках, коли повертати результат не вимагається. Список формальних параметрів - це послідовність пар *тип*-*ідентифікатор*, розділених комами. Якщо у методу параметри відсутні, то після імені методу повинні стояти порожні дужки. У прикладі нижче метод *init* присвоює змінним класу значення, які є його параметрами.

class Point

{ int х, у;

void init(int a, int b)

{

х = а;

y = b;

}

}

Не забороняється оголошувати формальні параметри методів, чиї імена збігаються з іменами змінних представників. Розглянемо в якості прикладу іншу версію методу *init*, в якому формальними параметрами є імена *х* і *у*, а для доступу до однойменних змінним поточного об'єкта використовується посилання *this*.

class Point

{

int х, у;

void init(int х, int у)

{

this.x = х;

this.у = у }

}

class TwoPointsInit {

public static void main(String args[]) {

Point p1 = new Point();

Point p2 = new Point();

p1.init(10,20);

p2.init(42,99);

System.out.println(“x = “ + p1.x + " = " + p1.y);

System.out.println("x =" + p2.x + " = " + p2.y);

} }

Оскільки імена параметрів та імена змінних класу в даному випадку збігаються, то ми використовуємо ключове слово *this*. Це ключове слово-посилання на поточний об'єкт. Тому у виразі this.x = x; перша частина this.x означає, що x - це змінна поточного класу, а не назва параметра x. Якщо б у нас параметри і поля називалися по-різному, то використовувати слово *this* було б необов'язково.

У Java немає глобальних функцій. Аналогом глобальної функції є статичний метод. Опис статичної функції у найпростішому випадку має таку структуру:

static тип\_результату ім'я\_функції(список\_формальних\_параметрів) тіло

Параметри (аргументи) функції, що вказуються в списку у визначенні функції, називаються *формальними*. Параметри, що вказуються під час виклику функції, називаються *фактичними*. Під час виклику функції виділяється пам'ять під її формальні параметри, потім кожному формальному параметру присвоюється значення фактичного параметра.

Тіло функції являє собою складений оператор (блок). Наприклад, визначення статичної функції, що обчислює суму двох цілих чисел, може бути таким:

static int sum(int a, int b)

{

int c = a + b;

return c;

}

Можна взагалі обійтися без змінної с:

static int sum(int a, int b)

{

return a + b;

}

Виклик функції може бути здійснений у виразі в описі або в тілі іншої функції. Під час виклику функції вказується її ім'я і список фактичних параметрів без зазначення їхніх типів:

int x = 4;

int y = 5;

int z = sum(a, b);

int t = sum(1, 3);

Функція може бути без параметрів:

int zero()

{

return 0;

}

Викликаючи таку функцію, також необхідно використовувати дужки:

System.out.println(zero());

Статичні функції в межах класу викликаються з застосуванням лише імені функції та списку фактичних параметрів. Для того, щоб викликати статичну функцію іншого класу, необхідно вказувати його ім'я і далі через точку ім'я функції. В такий спосіб можна звертатись до імен у межах пакету, а також до класів і функцій пакету java.lang. Цей пакет містить класи з дуже корисними функціями. Наприклад, клас Math надає велику кількість математичних функцій.

Якщо класи знаходяться в інших пакетах (не у java.lang), для виклику цих функцій слід або застосовувати префікс - *ім'я\_пакету.ім'я\_класу*. Якщо необхідні класи вкладених пакетів, префікс буде ще більш складним. Для того, щоб запобігти використанню повних імен, застосовується конструкція import. Конструкції import розташовують на початку файлу з вихідним кодом, або безпосередньо після конструкції *package*.

Для забезпечення доступу до класу чи інтерфейсу з іншого пакету є три варіанти:

1. Імпорт класу чи інтерфейсу (**import ім**'я\_пакету.імпортоване\_ім'я).
2. Імпорт усього пакету (**import**).
3. Статичний імпорт - імпорт статичних елементів зазначеного класу.

Наступний приклад демонструє перші два варіанти:

import java.io.FileReader; // Імпорт класу (інтерфейсу)

import java.util.\*; // Імпорт усього пакету

public class TestClass {

public static void main(String[] args) {

java.io.FileWriter fw; // Повне ім'я

FileReader fr; // Прямий доступ до імпортованого імені

ArrayList al; // ArrayList входить у пакет java.util

. . .

}

}

Не слід включати в програму твердження **import** java.lang.\*. Цей пакет імпортується автоматично.

Додаткова форма імпорту, яка з'явилася у версії Java 5, дозволяє імпортувати тільки статичні елементи зазначеного класу. Наприклад, у програмах, які реалізують математичні обчислення доцільно додавати статичний імпорт елементів класу *Math*:

import static java.lang.Math.\*;

Тепер статичні елементи можна використовувати без додаткового кваліфікатору, наприклад:

double d = sin(1);

### 1.7.2 Конструктори

Нижче показано приклад методу *init* для присвоєння початкових значень (ініціалізації) змінним класу.

class Point

{Int х, у;

void init (int a, int b)

{

х = а;

y = b;

}

 }

Ініціалізувати всі змінні класу щоразу, коли створюється його черговий представник - досить виснажлива справа навіть в тому випадку, коли в класі є функції, подібні методу *init.* Для цього в Java передбачені спеціальні методи, звані конструкторами. *Конструктор* - це метод класу, який ініціалізує новий об'єкт після його створення. Ім'я конструктора завжди збігається з ім'ям свого класу. У конструкторів немає типу даних результату, що повертається - ніякого, навіть *void*. Замінимо метод *init* з попереднього прикладу конструктором.

class Point

{

int х, у;

Point (int х, int у) {

this.x = х;

this.у = у;

}

}

Тут для доступу до однойменних змінним поточного об'єкта використовується посилання *this*.

Створимо тепер об'єкт класу Point і встановимо початкові значення його змінних за допомогою конструктора:

class PointCreate {

public static void main (String args []) {

Point p = new Point (10,20);

System.out.println ( "x =" + p.x + "у =" + p.у);

}

}

### 1.7.3 Суміщення (перевантаження) методів і конструкторів

Мова Java дозволяє створювати кілька методів і конструкторів з однаковими іменами, але з різними списками параметрів. Така техніка називається *суміщенням* або *перевантаженням (overloading)*. Якщо списки формальних параметрів збігаються, але типи значень, що повертаються, різні, компілятор повідомляє про помилку. Якщо списки формальних параметрів двох функцій розрізняються числом чи параметрів їхніми типами, то ці дві функції вважаються перевантаженням однієї функції і повинні мати різні визначення.

Як приклад наведена версія класу Point, в якій суміщення методів використано для визначення альтернативного конструктора, який ініціалізує координати х і у значеннями за замовчуванням (-1).

class Point

{

int х, у;

Point (int х, int у)

{

this.x = х;

this.у = у;

}

Point ()

{

х = -1;

у = -1;

}

}

class PointCreateAlt {

public static void main (String args []) {

Point p = new Point ();

System.out.println ( "x =" + p.x + "у =" + p.y);

}}

У цьому прикладі об'єкт класу Point створюється не під час виклику першого конструктора, як це було в попередньому прикладі, а за допомогою другого конструктора без параметрів. Ось результат роботи цієї програми:

х = -1 у = -1

Потрібний конструктор буде викликатися відповідно до кількості і типу параметрів, зазначених в операторі new.

Перевантажені функції використовуються в тих випадках, якщо кілька функцій виконує схожі дії над об'єктами різних типів і зручно дати однакові імена всім цим функціям:

static int max(int a, int b) { } // Вибір максимального з двох цілих чисел

static int max(double x, double y, double z) { } // Вибір максимального з

//трьох дійсних чисел

Для того, щоб визначити, яку саме функцію варто викликати, порівнюються кількість і типи фактичних параметрів, зазначені у виклику, з кількістю і типами формальних параметрів всіх описів функцій з даним ім'ям. У результаті викликається та функція, у якої формальні параметри щонайкраще зіставилися з параметрами виклику, чи видається помилка, якщо такої функції не знайшлося.

Параметри функції за умовчанням, які можуть бути описані у С++, у Java не застосовуються.

**Приклад**

В якості прикладу суміщення методів наведена версія класу Point, в якій використано визначення двох методів *sumа*, перший з яких складає два числа a і b, а другий - подвоює a.

class Point {

int s;

void suma (int a, int b) {

s = а + b;

System.out.println ( "s =", s);

}

void suma (int a) {

s = a + а;

System.out.println ( "s =", s);

}

public static void main (String args []) {

suma (2,3);

suma (2);

}}

#### Оператор return

У будь-якому місці програмного коду методу можна поставити оператор *return*, який призведе до негайного завершення роботи і передачі управління коду, який викликав цей метод.

Інструкція **return** у тілі функції забезпечує завершення роботи функції. Значення виразу після **return** стає значенням функції, яке ця функція повертає.

Функція може не повертати ніякого результату. Для позначення цього використовується тип **void**..

void hello()

{

System.out.println("Hello!");

}

У цьому випадку в тілі функції **return** може бути відсутнім. Якщо інструкція **return** присутня, то після неї не повинно бути ні якого виразу. Таку функцію можна викликати тільки окремою інструкцією.

hello();

Нижче наведено приклад, який ілюструє використання оператора *return* для повернення значення і передавання управління методу *main* після обчислення відстані в методі *distance*. В клас Point додані два методи *distance*, які повертають відстань між двома точками. Одному з суміщених методів в якості параметрів передаються координати точки х і у, а іншому ж ця інформація передається у вигляді параметра-об'єкта Point.

**Приклад**

class Point {int х, у;

Point (int х, int у) {

this.x = х;

this. y = y;

}

double distance (int х, int у) {

int dx = this.x - х;

int dy = this.у - у;

return Math.sqrt (dx \* dx + dy \* dy);

}

double distance (Point p) {

return distance (p.x, p.y);

}}

class PointDist {

public static void main (String args []) {

Point p1 = new Point (0, 0);

Point p2 = new Point (30, 40);

System.out.println ( "p1 =" + p1.x + "," + p1.y);

System.out.println ( "p2 =" + p2.x + "," + p2.y);

System.out.println ( "p1.distance (p2) =" + p1.distance (p2));

System.out.println ( "p1.distance (60, 80) =" + p1.distance (60, 80));

}}

**Результат**

р1 = 0, 0

р2 = 30, 40

р1.distance (p2) = 50.0

p1.distance (60, 80) = 100.0

### 1.7.4 Передача параметрів методам

#### Передача параметрів по значенню

Існує два способи, якими можна передавати аргументи методу - по значенню і по посиланню. У Java простий (примітивний) тип даних передається методу за значенням. Посилальні типи даних передаються за посиланням.

Якщо параметри передаються до функцій за значенням, то значення фактичних параметрів копіюються в пам'ять, відведену для формальних параметрів. При цьому значення, з якими працює функція - це її власні локальні копії фактичних параметрів і їхня зміна на ці параметри не впливає. Таким чином, при передачі за значенням вміст фактичних параметрів не змінюється:

static void f(int k)

{

k++; // k = 2;

}

public static void main(String[] args) {

int k = 1;

f(k);

System.out.println(k); // результат k = 1;

}

Інший приклад передачі параметрів за значенням:

// Значення примітивних типів передаються за значенням.

class Test {

void meth (int i, int j) {

i \* = 2;

j / = 2;

System.out.println ( "а і b всередині методу meth:" + a + "" + b);

}

public static void main (String args []) {

int a = 15, b = 20;

System.out.println ( "а і b перед викликом:" + a + "" + b);

ob.meth (a, b);

System.out.println ( "а і b після виклику:" + a + "" + b);

}

}

**Результат**

а і b перед викликом: 15 20

а і b всередині методу meth 30 10

а і b після виклику: 15 20

Операції, які виконуються всередині методу *meth* () не впливають на значення а і b, використані при зверненні до методу. Їх значення не змінилися на 30 і 10.

#### Передача параметрів за посиланням

Другий спосіб передачі аргументу - за посиланням. При використанні цього підходу параметру передається посилання на аргумент (а не його значення). Усередині методу це посилання використовується для звернення до реального аргументу. Це означає, що зміни, внесені до параметра, будуть впливати на оригінальне значення, використане до виклику метода.

При передачі об'єкта методу використовується спосіб - за посиланням. При створенні змінної типу класу створюється лише посилання на об'єкт. Таким чином, при передачі параметра-посилання метод бере цей параметр, який буде посилатися на об'єкт. Зміни об'єкта всередині методу впливають на об'єкт, використаний як аргумент. Наприклад, розглянемо таку програму:

// Об'єкти передаються по посиланню

class Test

{Int a, b;

Test (int i, int j) {

a = i; b = j;

}

// Передача об'єкта в якості параметра методу

void meth (Test o) {

o.a \* = 2;

o.b / = 2;

}

}

class CallByRef {

public static void main (String args []) {

Test ob = new Test (15, 20);

System.out.println ( "ob.а і ob.b перед викликом:" + ob.a + "" + ob.b);

ob.meth (ob);

System.out.println ( "ob.а і ob.b після виклику:" + ob.a + "" + ob.b);

}

}

Ця програма генерує наступний висновок:

ob.a і ob.b перед викликом: 15 20

ob.a і ob.b після виклику 30 10

В даному випадку дії всередині методу meth () впливають на об'єкт, використаний як аргумент.

### 1.7.5 Вбудовані методи (функції)

Бібліотека класів мови вміщує набір допоміжних службових класів-утиліт, які широко використовуються в інших вбудованих пакетах Java. Ці класи розміщені в пакетах *java.lang*, *java.util* та інших. Їх методи використовуються для роботи з наборами об'єктів, для взаємодії з системними функціями низького рівня, для роботи з математичними функціями, для генерації випадкових чисел, маніпуляцій з датами і часом та інше.

Наприклад, клас *Math* (належить пакету *java.lang*) містить методи для обчислень різних математичних функцій, в тому числі тих, які використовуються в геометрії і тригонометрії. Крім того, в ньому є константи, що використовуються у математичних обчисленнях: — Е (приблизно 2.72) і PI (приблизно 3.14159).

Нижче наведені деякі тригонометричні, степеневі, показникові та логарифмічні функції

* sin(double а) - синус кута а, заданого в радіанах.
* cos(double а) - косинус кута а, заданого в радіанах.
* tan(double а) - тангенс кута а, заданого в радіанах.
* asin(double r) - кут, синус якого дорівнює r.
* acos(double r) - кут, косинус якого дорівнює r.
* atan(double r) - кут, тангенс якого дорівнює r.
* atan2(double a, double b) - кут, тангенс якого дорівнює відношенню а/b.
* pow(double y, double x) - у, зведене в ступінь х.
* exp(double x) - е в ступені х.
* log(double x) - натуральний логарифм х.
* sqrt(double x) - квадратний корінь х.

Функції округлення:

* ceil(double а) повертає найменше ціле число, значення якого більше або дорівнює а.
* floor(double а) повертає найбільше ціле число, значення якого менше або дорівнює а.
* rint(double а) повертає значення а типу double з відкинутою дробовою частиною.
* round(float а) повертає закруглене до найближчого цілого значення а.
* round(double а) повертає закруглене до найближчого довгого цілого значення а.

Крім того, в класі *Math* є методи для отримання модуля, знаходження мінімального та максимального значень для різних типів даних:

* abs(a) - модуль (абсолютне значення) числа а.
* max(a, b) - найбільший зі своїх аргументів.
* min(a, b) - найменший зі своїх аргументів.

Для виконання округлення використовуються також методи класів *BigDecimal* і *RoundingMode*.

**Приклад**

Текст програми для обчислити значення виразів з використанням методів класу *Math*:

public class Lab1 {

public static void main(String[] args) {

double x,y,a,b,c,d,t1,t2;

a=12.5;

b=1.3;

c=14.1;

x=3.3;

y=1.1;

t1=a\*x/y+b/Math.pow(y,2)\*Math.log10(y\*x+c);

t2=Math.log((Math.sqrt((c-b)\*(c+b))\*Math.tan(a\*x)+2)/(Math.sqrt((c-b)\*(c+b))\* Math.tan(a\*x)-2))/2/a/b;

System.out.println ("t1= "+t1);

System.out.println ("t2= "+t2);

}

}

**Результат**

t1= 38.84158789775305

t2= 0.020985550983192488

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що називається методом?
2. Що називається конструктором?
3. Яка різниця між методом і конструктором?
4. Охарактеризуйте синтаксис методу.
5. Що означає термін «перевантаження» методу або конструктору?
6. Які значення може повертати метод?
7. Яке призначення аргументів методу?
8. Які є способи передачі аргументів методу?
9. Які вбудовані методи ви знаєте?

## Перерахування

*Перерахування (enum) -* це особливий клас, який вміщує набір визначених констант.

**Приклад**

***enum*** *Month {****JAN****,****FEB****,****MAR****,****APR****,****MAY****};*

За замовчуванням перерахування

* є спадкоємцями класу **java.lang.Enum,** а тому не можуть розширювати інші класи;
* реалізують інтерфейс **java.lang.Comparable,** а значить можуть бути відсортованими;

Частина методів визначено як **final**, а значить їх не можна перевизначити (*name(), ordinal(), equals(), hashCode(), compareTo(), clone()*);

Опис перерахування може доповнюватися конструкторами, змінними примірника і методами (розміщуються завжди після списку констант, який завершується крапкою з комою);

Конструктор для перерахування повинен мати доступ або **package-private**, або **private**.

**Властивості об’єкта перерахованого типу:**

* безпека типу;
* компактний, ефективний опис значень перечислення;
* без проблемна інтеграція з іншими властивостями мови;
* ефективність продуктивності.

Використовують перерахуваня там, де треба представити фіксований набір констант:

* *природні види перечислень* (наприклад, планети в сонячної системи);
* *набори даних, для яких відомі всі можливі значення під час компіляції* (пункти меню, прапори командного рядка.).

**Методи Enum**

|  |  |
| --- | --- |
| ***Назва*** | ***Функція*** |
| static …[] **values**() | повертає масив всіх значень перечислення в порядку їх об’явлення |
| int **ordinal**() | повертає порядковий номер (позицію) поточної константи |
| String **name**() | повертає значення поточної константи у вигляді рядка символів |
| static Enum **valueOf**(String name) | повертає константу перечислення з вказаниим значенням |

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Коли доцільно використовувати *перерахування*?
2. Якими способами можна описати *перерахування*?
3. Які властивості має об’єкт перерахованого типу?
4. Які існують методи класу *enum?*

## Оператори управління виконанням програми

### 1.9.1 Умовні оператори

#### Оператор if-else

В узагальненій формі цей оператор записується наступним чином:

if (логічний вираз) оператор1; [ else оператор2;]

Розділ *else* необов'язковий. На місці *оператор1* або *оператор2* може стояти вкладений оператор у фігурних дужках. Логічний вираз — це будь-який вираз, який повертає значення типу *boolean*.

А ось повна програма, в якій для визначення, до якої пори року відноситься той чи інший місяць, використовуються оператори *if-else*.

class IfElse {

public static void main(String args[]) {

int month = 4;

String season;

if (month == 12 || month == 1 || month == 2) {

season = "зима";

}

else if (month ==3 || month == 4 || month == 5) {

season = "весна";

}

else if (month == 6 || month == 7 || month == 8) {

season = "літо";

}

else if (month == 9 || month == 10 || month == 11) {

season = "осінь";

}

else {

season = " неіснуючий номер місяця ";

}

System.out.println("Місяць № "+ month + "- це " + season + ".");

} }

**Результат**

Місяць № 4 - це весна.

#### Оператор break

Цей оператор повідомляє виконуючому середовищу, що слід припинити виконання блоку і передати управління оператору, наступному за даними блоком. Для іменування блоків в мові Java використовуються мітки. Оператор *break* при роботі з циклами і в операторах *switch* може використовуватися без мітки. У такому випадку мається на увазі вихід з поточного блоку.

Наприклад, у наступній програмі є три вкладених блоки, і у кожного - своя унікальна мітка. Оператор break, який стоїть у внутрішньому блоці с, що викликає перехід на оператор, наступний за блоком b. При цьому пропускаються два оператора *println*.

class Break {

public static void main(String args[]) { boolean t = true;

a: { b: { c: {

System.out.println("Перед оператором break");

if (t)

break b;

System.out.println("He буде виконано"); }

System.out.println("He буде виконано"); }

} }

System.out.println("Після b");

}

В результаті виконання програми ви отримаєте наступний результат:

Перед оператором break

Після b

#### Оператор switch

Оператор *switch* забезпечує спосіб переходу на різні частини програмного коду в залежності від значення однієї змінної або виразу. Загальна форма цього оператора така:

switch ( вираз ) {

case значення1:

break;

case значення2:

break;

case значенням:

break;

default:

}

Результатом обчислення виразу може бути значення будь-якого простого типу, при цьому кожне із значень, зазначених в операторах *case*, має бути сумісним з типом виразу в операторі *switch*. Всі ці значення повинні бути унікальними літералами. Якщо ж ви вкажете в двох операторів case однакові значення, транслятор видасть повідомлення про помилку.

Якщо значення виразу не відповідає жодному з операторів *case*, управління передається коду, розташованому після ключового слова *default*. Зазначимо, що оператор *default* необов'язковий. У разі, коли жоден з операторів *case* не відповідає значенню виразу і в *switch* відсутній оператор *default* виконання програми продовжується з оператора, наступного за оператором *switch*.

Всередині оператора *switch* (а також всередині циклічних конструкцій) *break* без мітки призводить до передачі управління на код, що стоїть після оператора *switch*. Якщо *break* відсутній, після поточного розділу *case* буде виконуватися наступний. Іноді буває зручно мати в операторі *switch* кілька суміжних розділів *case*, не розділених оператором *break*.

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

int month = 4;

String season;

switch (month) {

case 12:

case 1:

case 2:

season = "зима";

break;

case 3:

case 4:

case 5:

season = "весна";

break;

case 6:

case 7:

case 8:

season = "літо";

break;

case 9:

case 10:

case 11:

season = "осінь";

break;

default:

season = "неіснуючий номер місяця";

}

System.out.println("Місяць № "+ month + "- це " + season + ".");

} }

### 1.9.2 Оператори циклу

#### Оператор while

В Java є три конструкції для організації циклів: *while* (з передумовою), *do-while* (з після-умовою) *for* (з лічильником). Є також і модифікація оператора циклу з лічильником, який називають *for.. each* («для кожного»).

Цикл while багаторазово виконується до тих пір, доки значення логічного виразу рівне *true*. Нижче наведені загальні форми оператора *while*:

while (умова завершення) оператор;

або

while (умова завершення)

{

<група операторів>

}

Нижче наведено приклад циклу while для виведення десяти чисел.

class WhileDemo {

public static void main(String args[]) {

int n = 10;

while (n > 0) {

System.out.println("Число n =" + n);

n--;

}

}

}

#### Оператор do-while

Іноді виникає потреба виконати тіло циклу принаймні один раз — навіть у тому випадку, коли логічний вираз з самого початку приймає значення *false*. Для таких випадків в Java використовується циклічна конструкція *do-while*. Її загальні форми запису такі:

do оператор while (умова завершення) ;

або

do {

<група операторів>

} while (умова завершення);

У наступному прикладі тіло циклу виконується до першої перевірки умови завершення. Це дозволяє поєднати код ітерації з умовою завершення:

class DoWhile {

public static void main(String args[]) {

int n = 10;

do {

System.out.println("Число n =" + n);

} while (--n > 0);

} }

#### Оператор for

Нижче приведена загальна форма запису оператора *for*.

for ( ініціалізація; умова завершення; ітерація )

тіло;

Будь-який цикл, записаний за допомогою оператора *for*, можна записати у вигляді циклу *while* або *do-while*, і навпаки. У канонічній формі циклу *for* відбувається збільшення цілого значення лічильника з мінімального значення до певної межі.

class ForDemo {

public static void main(String args[]) {

for (int i = 1; i <= 10; i++)

System.out.println("i =" + i);

} }

Змінні можна оголошувати всередині розділу ініціалізації оператора *for*. Змінна, оголошена всередині оператора *for*, діє в межах цього оператора.

Іноді виникають ситуації, коли розділи ініціалізації або ітерації циклу *for* вимагають декількох операторів. Оскільки складений оператор в фігурних дужках у заголовок циклу *for* вставляти не можна, Java надає альтернативний шлях. Застосування коми (,) для розділення декількох операторів допускається тільки всередині круглих дужок оператора *for*. Нижче наведено приклад циклу *for*, в якому в розділах ініціалізації та ітерації вставлено кілька операторів.

class Comma {

public static void main(String args[]) {

int a, b;

for (a = 1, b = 4; a < b; a++, b--) {

System.out.println("a =" + a);

System.out.println("b =" + b);

}

} }

Результат виконання цієї програми показує, що цикл виконується всього два рази.

а = 1

b = 4

а = 2

b = 3

#### Оператор for..each

У деяких ситуаціях при роботі з масивами або колекціями виникає потреба перебрати всі їх елементи. У цьому випадку зручно використовувати оператор циклу, в якому не потрібно стежити за зміною значення лічильника:

int A[]=new int[3];

int B[]={9,8,7,6};

for (Object a : A)

System.out.println(a);// тут будуть виведені три нулі – елементи масиву А

for (Object b : B)

System.out.println(b);// тут будуть виведені числа 9,8,7,6 – елементи масиву В

Оператор *break* при роботі з циклами, як і в операторі *switch*, може використовуватися для виходу з поточного блоку.

#### Оператор continue

У деяких ситуаціях виникає потреба достроково перейти до виконання наступної ітерації, проігнорувавши частину операторів тіла циклу, ще не виконаних у поточній ітерації. Для цієї мети в Java передбачений оператор *continue*. Нижче наведено приклад, в якому оператор *continue* використовується для того, щоб в кожному рядку друкувалися два числа.

class ContinueDemo {

public static void main(String args[]) {

for (int i=0; i < 10; i++) {

System.out.print(i + " ");

if (i % 2 == 0) continue;

System.out.println("");

}

}}

Якщо індекс парний, цикл триває без переходу на новий рядок (оператор println не виконується). Результат виконання цієї програми такий:

0 1

2 3

4 5

5 7

8 9

В операторах *break* і *continue* можна використовувати мітку, що вказує на вкладений цикл, в якому ви хочете достроково припинити виконання поточної ітерації. Для ілюстрації наведена програма, що використовує оператор *continue* з міткою для виведення трикутної таблиці множення для чисел від 0 до 9:

class ContinueLabel {

public static void main(String args[]) {

outer: for (int i=0; i < 10; i++) {

for (int j = 0; j < 10; j++) {

if (j > i) {

System.out.println("");

continue outer;

}

System.out.print(" " + (i \* j));

}

}

}}

Оператор *continue* в цій програмі призводить до завершення внутрішнього циклу з лічильником j і переходу до чергової ітерації зовнішнього циклу з лічильником i. У процесі роботи ця програма виводить наступні рядки:

0

0 1

0 2 4

0 3 6 9

0 4 8 12 16

0 5 10 15 20 25

0 6 12 18 24 30 36

0 7 14 21 28 35 42 49

0 8 16 24 32 40 48 56 64

0 9 18 27 36 45 54 63 72 81

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Наведіть приклади операторів розгалужень, поясніть їх особливості.
2. В чому переваги використання оператора switch по відношенню до if?
3. Яке призначення ключового слова default оператора switch?
4. Які оператори можна використовувати для перевірки декількох умов?
5. Який процес називають циклічним? Назвіть його складові.
6. Які існують оператори для організації циклів?
7. Який синтаксис операторів циклу з перед-умовою?
8. Який синтаксис операторів циклу з після-умовою?
9. Поясніть різницю між операторами циклів for… та do while….
10. Поясніть визначення терміну "нескінченний цикл", як запобігти його виникненню.
11. Охарактеризуйте цикл із заданою кількістю повторень.
12. Які є оператори циклу по рахівнику?
13. Які є оператори циклу по умові?
14. Як вийти з циклу по рахівнику, не чекаючи, коли змінна циклу досягне заданого значення?
15. Як вийти з циклу по умові?
16. Як виконати внутрішні оператори циклів по умові хоч би один раз?

## Масиви

### 1.10.1 Одновимірні масиви

*Масив* (*array*) – це група однотипних змінних. Масив може складатися з нульової або ненульової кількості елементів іншого масиву; тоді перший масив називається багатовимірним.

Елементи масиву можуть належати до примітивного типу, наприклад float, char або int. Крім того, елементи можуть мати тип масиву, класу або інтерфейсу.

Змінні, що входять в масив, називають елементами масиву. На елемент масиву можна послатися, вказавши його місце в масиві (за номером елементу масиву, який називають *індексом*).

Індекси масиву починається з нуля і збільшуються до значення, яке на одиницю менше довжини масиву.

Синтаксис оголошення масиву:

тип ім’я\_масиву[];

Тип – тип даних елементів, що складають масив. Тип та кількість елементів масиву визначають обсяг пам'яті, необхідний для розміщення масиву.

В якості імені\_масиву може виступати будь-який дозволений ідентифікатор.

При оголошенні масиву пара квадратних дужок, які визначають масив, може розташовуватися як після імені масиву так і перед його ім'ям. Наприклад, такі оголошення масиву цілих чисел ідентичні:

int d[];

int []d;

Тут оголошено масив *d* без виділення області пам'яті, для розміщення його елементів. Таке оголошення означає тільки те, що змінна d буде служити посиланням на масив, що складається з цілих чисел.

У наступному прикладі оголошуються відразу кілька масивів одного й того ж типу:

int[] array1,array2, array3;

Заповнювати значеннями елементи масиву можна також по-різному:

int[] b={3,2,1};

String s1="One";

String s2="Two";

String c[]={s1,s2};

Для того, щоб зарезервувати пам'ять під масив, використовується спеціальний оператор new. У наведеному нижче рядку коду за допомогою оператора new для масиву *month\_days* виділяється пам'ять для зберігання дванадцяти цілих чисел.

month\_days = new int [12];

Отже, тепер *month\_days* — це посилання на дванадцять цілих чисел. Нижче наведено приклад, в якому створюється масив, елементи якого містять число днів в місяцях року (не високосного).

class Array {

public static void main (String args []) {

int month\_days[];

month\_days = new int[12];

month\_days[0] = 31;

month\_days[1] = 28;

month\_days[2] = 31;

month\_days[3] = 30;

month\_days[4] = 31;

month\_days[5] = 30;

month\_days[6] = 31;

month\_days[7] = 31;

month\_days[8] = 30;

month\_days[9] = 31;

month\_days[10] = 30;

month\_days[11] = 31;

System.out.println("Квітень має " + month\_days[3] + " днів.");

} }

При запуску ця програма друкує кількість днів у квітні, як це показано нижче.

Квітень має 30 днів.

В результаті роботи наступної програми, ви отримаєте такий самий результат, як і від попередньої.

class AutoArray {

public static void main(String args[]) {

int month\_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

System.out.println("Квітень має " + month\_days[3] + " днів.");

} }

Java суворо стежить за тим, щоб ви випадково не записали або не спробували отримати значення, вийшовши за межі масиву. Якщо ж ви спробуєте використовувати в якості індексів від'ємні числа або числа, які більше або дорівнюють кількості елементів в масиві, то отримаєте повідомлення про помилку під час виконання програми.

Наступний приклад — знаходження суми, добутку і середнього арифметичного елементів масиву.

int i;

float av=0;

int s=0;

double p=1;

int nn=0;

/\* заповнення масиву числами \*/

int Ar[]={2,-4,1,-8,11,-3};

for(i=0;i<6;i++)

{

s=s+Ar[i];

p=p\*Ar[i];

nn++;

}

av=(float)s/nn;

System.out.println("Сума = "+s);

System.out.println("Добуток = "+p);

System.out.println("Середнє арифметичне = "+av);

Наступна форма циклу **for** дозволяє спростити повний обхід масивів.

int[] nums = {1, 2, 3, 4, 5, 6};

for (int i = 0; i < nums.length; i++)

System.out.println(nums[i]);

Альтернативну форму циклу можна написати так:

int[] nums = {1, 2, 3, 4, 5, 6};

for(int n : nums)

System.out.println(n);

Альтернативна форма може бути застосована. лише для читання значень елементів, а не для їхньої модифікації.

Масиви читають з клавіатури поелементно. Наступний приклад демонструє читання кількості та значень елементів з клавіатури.

public class ArrayTest {

public static void main(String[] args) {

System.out.println("Уведіть кількість елементів масиву:");

java.util.Scanner s = new java.util.Scanner(System.in);

int size = s.nextInt();

double[] a = new double[size];

System.out.println("Уведіть елементи масиву:");

for (int i = 0; i < a.length; i++)

a[i] = s.nextDouble();

// Робота з масивом

// ...

}

}

Присвоювання імені масиву іншому імені масиву приводить тільки до копіювання посилання, але не самого масиву. Для копіювання елементів масиву необхідно організувати цикл. Клас System надає найпростіший шлях копіювання одного масиву в іншій - використання статичного методу *arraycopy*():

System.arraycopy(a, a\_from, b, b\_from, size);

Це еквівалентно такому циклу:

for (int i = a\_from, j = b\_from; i < size + a\_from; i++, j++) {

b[j] = a[i];

}

Масив, у який здійснюється копіювання, повинен мати необхідні розміри. Функція *arraycopy*() не створює нового масиву. Весь масив a можна скопіювати в b таким викликом:

System.arraycopy(a, 0, b, 0, a.length);

Для заповнення масивів можна використовувати статичні методи класу Arrays, реалізованого в пакеті java.util. Для всіх примітивних типів і типу Object реалізовані функції fill() у двох видах:

public static void fill(тип[] a, тип val)

public static void fill(тип[] a, int fromIndex, int toIndex, тип val)

Тут *тип* означає один з фундаментальних (примітивних) типів або тип Object. Перший варіант використовується для заповнення всього масиву, другий - частини, при чому елемент з номером toIndex не включається в послідовність. Наприклад:

public class FillArray {

public static void main(String[] args) {

int[] a = new int[6];

java.util.Arrays.fill(a, 0, 4, 12); // Інші елементи дорівнюють 0

for (int x : a)

System.out.print(x + " ");

System.out.println();

java.util.Arrays.fill(a, 100); // Всі елементи дорівнюють 100

for (int x : a)

System.out.print(x + " ");

System.out.println();

}

}

Порівняти два масиви чи частину їх можна за допомогою функцій групи equals():

public static boolean equals(тип[] a, тип[] a2)

Тут *тип* означає один з фундаментальних (примітивних) типів або тип Object. Масиви порівнюються поелементно. Два масиви також вважаються еквівалентними, якщо обидва посилання - **null**. Наприклад:

import java.util.Arrays;

public class ArraysComparison {

public static void main(String[] args) {

double[] a = null, b = null;

System.out.println(Arrays.equals(a, b)); // true

a = new double[] { 1, 2, 3, 4 };

b = new double[4];

System.out.println(Arrays.equals(a, b)); // false

System.arraycopy(a, 0, b, 0, a.length);

System.out.println(Arrays.equals(a, b)); // true

b[3] = 4.5;

System.out.println(Arrays.equals(a, b)); // false

}

}

За допомогою функції sort() можна здійснити сортування масиву чисел за зростанням. Наприклад:

import java.util.Arrays;

public class ArraySort {

public static void main(String[] args) {

int[] a = new int[] { 11, 2, 10, 1 };

Arrays.sort(a); // 1 2 10 11

for (int x : a)

System.out.print(x + " ");

System.out.println();

}

}

Функція sort() реалізована для масивів усіх примітивних типів та рядків. Рядки впорядковуються за алфавітом. Можна також сортувати частину масиву. Як і для функції fill(), вказується початковий і кінцевий індекси послідовності, яку слід відсортувати. Кінцевий індекс не включається в послідовність. Наприклад:

int[] a = {7, 8, 3, 4, -10, 0};

java.util.Arrays.sort(a, 1, 4); // 7 3 4 8 -10 0

У відсортованих масивах можна виконати пошук за допомогою методів класу Arrays. Група функцій binarySearch(), реалізована для всіх примітивних типів і типу Object, повертає індекс знайденого елемента або від'ємне значення, якщо елемент відсутній.

public static int binarySearch(тип[] a, тип key)

Масиви-параметри передаються у функції за посиланням. Після повернення з функції елементи можуть містити змінені значення:

public class SwapElements {

static void swap(int[] a) {

int z = a[0];

a[0] = a[1];

a[1] = z;

}

public static void main(String[] args) {

int[] b = {1, 2};

swap(b);

System.out.println(b[0]); // 2

System.out.println(b[1]); // 1

}

}

У Java 1.5 з'явилася додаткова можливість створення функцій зі змінним числом параметрів визначеного типу. Всередині функції такі параметри інтерпретуються як масив:

static void printIntegers(int... a) {

for (int i = 0; i < a.length; i++)

System.out.println(a[i]);

}

Викликати таку функцію можна у два способи: передаючи список аргументів типу елемента масиву, або передаючи масив цілком:

public static void main(String[] args) {  
 printIntegers(1, 2, 3);  
 int[] arr = {4, 5};  
 printIntegers(arr);  
}

Параметри такого виду обов'язково повинні бути останніми в списку.

### 1.10.2 Багатовимірні масиви

В Java існують масиви масивів, які поводять себе подібно багатовимірним масивам. Наведений нижче код створює традиційну матрицю з шістнадцяти елементів типу *double*, кожен з яких ініціалізується нулем. Внутрішня реалізація цієї матриці — це масив масивів *double*.

double matrix [][] = new double [4][4];

Наведений нижче інший фрагмент коду ініціалізує таку ж кількість пам'яті, як і в попередньому прикладі, але пам'ять під другу розмірність відводиться в чотирьох рядках. Це зроблено для того, щоб наочно показати, що матриця насправді являє собою вкладені масиви.

double matrix [][] = new double [4][];

matrix [0] = new double[4];

matrix[1] = new double[4];

matrix[2] = new double[4];

matrix[3] = new double[4];

У наступному прикладі створюється матриця розміром 4 на 4 з елементами типу double, причому її діагональні елементи (ті, для яких х==у) заповнюються одиницями, а всі інші елементи залишаються рівними нулю.

class Matrix {

public static void main(String args[]) {

double m[][];

m = new double[4][4];

m[0][0] = 1;

m[1][1] = 1;

m[2][2] = 1;

m[3][3] = 1;

System.out.println(m[0][0] +" "+ m[0][1] +" "+ m[0][2] +" "+ m[0][3]);

System.out.println(m[1][0] +" "+ m[1][1] +" "+ m[1][2] +" "+ m[1][3]);

System.out.println(m[2][0] +" "+ m[2][1] +" "+ m[2][2] +" "+ m[2][3]);

System.out.println(m[3][0] +" "+ m[3][1] +" "+ m[3][2] +" "+ m[3][3]);

}

}

Запустивши цю програму, ви отримаєте наступний результат:

1 0 0 0

0 1 0 0

0 0 1 0

0 0 0 1

З прикладу видно — щоб значення елемента масиву було нульовим, не потрібно його ініціалізувати, це робиться автоматично.

Для встановлення початкових значень масивів існує спеціальна форма ініціалізатору, придатна і в багатовимірному випадку. У програмі, наведеній нижче, створюється матриця, кожен елемент якої містить добуток номера рядка на номер стовпця. Всередині ініціалізатора масиву можна використовувати не тільки літерали, але і вирази.

class AutoMatrix {

public static void main(String args[]) {

double m[][] = {

{ 0\*0, 1\*0, 2\*0, 3\*0 }, { 0\*1, 1\*1, 2\*1, 3\*1 }, { 0\*2, 1\*2, 2\*2, 3\*2 },

{ 0\*3, 1\*3, 2\*3, 3\*3 } };

System.out.println(m[0][0] +" "+ m[0][1] +" "+ m[0][2] +" "+ m[0][3]);

System.out.println(m[1][0] +" "+m[1][1] +" "+ m[1][2] +" "+ m[1][3]);

System.out.println(m[2][0] +" "+m[2][1] +" "+ m[2][2] +" "+ m[2][3]);

System.out.println(m[3][0] +" "+m[3][1] +" "+ m[3][2] +" "+ m[3][3]);} }

Запустивши цю програму, ви отримаєте наступний результат:

0 0 0 0

0 1 2 3

0 2 4 6

0 3 6 9

### 1.10.3 Класи Arrays та ArrayList

Клас *Arrays* забезпечує різні методи для роботи з масивами: присвоювання значень, перевірка еквівалентності, сортування, пошук. Ось приклад використання цього класу для сортування масиву:

int B[]={9,6,8,7};

Arrays.sort(B);

for (Object b : B)

System.out.println(b);

Клас-колекція *ArrayList* **–** динамічний масив, здатен автоматично виділяти додаткову пам'ять при необхідності розширення масиву.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Коли доцільно використовувати масиви змінних?
2. Як створити у програмі масив?
3. Як звернутись до елементу масиву?
4. Які помилки можуть виникати при роботі з масивами?
5. Продемонструвати способи введення значень елементів масиву у програму.
6. Пояснити, як накопичувати суми і добутки елементів усього масиву

або деяких його елементів.

1. Як організувати знаходження елементів масиву, що задовольняють певним умовам?
2. Пояснити, як сформувати новий масив з деяких елементів вихідного масиву.

## Обробка виняткових ситуацій

### 1.11.1 Поняття виключення

*Виключення* в Java - це об'єкт, який описує винятковий стан (помилку), який виник в якій-небудь ділянці програмного коду. Коли виникає виключення, створюється об'єкт класу *Exception*. Цей об'єкт пересилається в метод, що обробляє даний тип виняткової ситуації. Виключення можуть створюватись як системою, так і самим програмістом для того, щоб повідомити про деякі нештатні ситуації.

До механізму обробки виключень в Java мають відношення п’ять ключових слів: - *try, catch, throw, throws і finally*. Схема роботи цього механізму наступна. Ви намагаєтеся (*try*) виконати блок коду, і якщо при цьому виникає помилка, система збуджує (*throw*) виключення, яке в залежності від його типу ви можете перехопити (*catch*) або передати обробникові по замовчуванню (*finally*).

Нижче наведена загальна форма блоку обробки виключень.

try {

// Блок коду

}

catch (ТипВиключення1 е) {

// Обробник виключень типу ТипВиключення 1}

catch (ТипВиключення2 е) {

// Обробник виключень типу ТипВиключення 2

throw (e) // повторне збудження виключення}

finally {

}

### 1.11.2 Типи виключень

У вершині ієрархії виключень стоїть клас *Throwable*. Кожен з типів виключень є підкласом класу *Throwable*. Два безпосередніх спадкоємця класу *Throwable* ділять ієрархію підкласів виключень на дві різні гілки. Одна з них - клас *Ехception* - використовується для опису виняткових ситуації, які повинні перехоплюватися програмним кодом користувача. Інша гілка дерева підкласів *Throwable* - клас *Error*, який призначений для опису виняткових ситуацій, які при звичайних умовах не повинні перехоплюватися у програмі користувача.

**Опис деяких виключень, які обробляються в програмі**

|  |  |
| --- | --- |
| ***Виключення*** | ***Опис*** |
| ArithmeticException | Арифметична помилка типу ділення на ноль |
| ArraylndexOutOfBoundsException | Індекс масива - поза межами |
| ArrayStoreException | Призначення елементу масиву несумісного типу |
| ClassCastException | Неприпустиме приведення типів |
| UlegalArgirnentException | При виклику методу використано невірний аргумент |
| IllegalMonitorStateException | Незаконна операція монітора, типу очикуваня на розблокованному потоці |
| UlegalStateException | Середовище або застосунок знаходяться в некорректному стані |
| UlegalThreadStateException | Операція не сумісна з поточним станом потоку |
| IndexOutOfBoundsException | Деякий тип індексу знаходиться поза межами |
| NegativeArraySizeException | Масив створювався з від’ємним розміром |
| NullPointerException | Неприпустиме використання нульового посилання |
| NumberFormatException | Недопустиме перетворення рядка в числовий формат |
| SecurityException | Спроба порушити захист |
| StringIndexOutOfBoundsException | Спроба індексувати поза межами рядка |
| OnsupportedOperationException | Зустрілась операція, яка не підтримується |

Об'єкти-виключення автоматично створюються виконуючим середовищем Java в результаті виникнення певних виняткових станів. Наприклад, черговий приклад містить вираз, при обчисленні якого виникає ділення на нуль.

class Exc0 {

public static void main (string args []) {

int d = 0;

int a = 42 / d;

}}

Ось результат виконання прикладу.

java.lang.ArithmeticException: / by zero

at Exc0.main (Exc0.java:4)

Тут типом збудженого виключення не був ні *Exception*, ні *Throwable*. Це був підклас класу *Exception*, а саме: *ArithmeticException*, що пояснює, яка помилка виникла при виконанні програми. Ось інша версія того ж класу, в якій виникає та ж виняткова ситуація, але на цей раз не в програмному коді методу *main*.

class Exc1 {

static void subroutine () {

int d = 0;

int a = 10 / d;

}

public static void main (String args []) {

Exc1.subroutine ();

}}

Результат роботи цієї програми показує, як обробник виключень виконуючої системи Java виводить вміст всього стеку викликів.

java.lang.ArithmeticException: / by zero

at Exc1.subroutine (Exc1.java:4)

at Exc1.main (Exc1.java:7)

### 1.11.3 Розділи try і catch

Для створення блоку програмного коду, який потрібно захистити від виключень, використовується ключове слово *try*. Відразу ж після *try*-блоку поміщається блок *catch*, що задає тип виключення, яке ви хочете обробляти.

class Exc2 {

public static void main (String args []) {

try {

int d = 0;

int a = 42 / d;

}

catch (ArithmeticException e) {

System.out.println ( "ділення на нуль");

}

}}

Метою більшості добре сконструйованих *catch*-розділів повинна бути обробка виняткової ситуації, яка виникла та приведення змінних програми в якийсь розумний стан такий, щоб програму можна було продовжити так, ніби ніякої помилки і не було (в нашому прикладі виводиться попередження - ділення на нуль).

#### Кілька розділів catch

У деяких випадках один блок програмного коду може збуджувати виключення різних типів. Для того, щоб обробляти подібні ситуації, Java дозволяє використовувати будь-яку кількість *catch*-розділів для *try*-блоку. Найбільш спеціалізовані класи виключень повинні йти першими, оскільки жоден підклас виключень не буде досягнутий, якщо поставити його після суперкласу. Наступна програма перехоплює два різних типи виключень за допомогою двох спеціалізованих обробників - розділів *catch*.

class MultiCatch {

public static void main (String args []) {

try {

int a = args.length;

System.out.println ( "a =" + a);

int b = 42 / a;

int c [] = {1};

c [42] = 99;

}

catch (ArithmeticException e) {

System.out.println ( "ділення на 0:" + e);

}

catch (ArrayIndexOutOfBoundsException e) {

System.out.println ( "індекс за межами масиву:" + e);

}

}}

Цей приклад, запущений без параметрів, викликає збудження виняткової ситуації ділення на нуль. Якщо ж ми поставимо в командному рядку один або кілька параметрів, тим самим встановивши *а* (довжину масиву args []) в значення більше нуля, наш приклад виконає оператор ділення, але в наступному операторі буде порушено виключення виходу індексу за межі масиву *ArrayIndexOutOf Bounds*. Нижче наведені результати роботи цієї програми, запущеної і тим і іншим способом.

а = 0

ділення на 0: java.lang.ArithmeticException: / by zero

a = 1

індекс за межами масиву: java.lang.ArrayIndexOutOfBoundsException: 42

#### Вкладені блоки try

Оператори *try* можна вкладати один в одного аналогічно тому, як можна вкладати оператори циклу. Якщо у оператора *try* низького рівня немає розділу *catch*, відповідного порушеному виключенню, стек буде розгорнуто на один щабель вище, і в пошуках підходящого обробника будуть перевірені розділи *catch* зовнішнього оператора *try*. Нижче приклад, в якому два оператори *try* вкладені за допомогою виклику методу procedure з методу main.

class MultiNest {

static void procedure () {

try {

int c [] = {1};

c [42] = 99;

}

catch (ArrayIndexOutOfBoundsException e) {

System.out.println ( " індекс за межами масиву:" + e);

}}

public static void main (String args []) {

try {

int a = args.length ();

System.out.println ( "a =" + a);

int b = 42 / a;

procedure ();

}

catch (ArithmeticException e) {

System.out.println ( "ділення на 0:" + e);

}

}}

### 1.11.4 Оператор throw

Оператор *throw* використовується для збудження виключення програмістом. Для того, щоб зробити це, потрібно мати об'єкт підкласу класу Throwable, який можна або отримати як параметр оператора *catch*, або створити за допомогою оператора *new*. Нижче наведена загальна форма оператора *throw*.

throw Об'єктТипуThrowable;

При досягненні цього оператора нормальне виконання коду негайно припиняється, так що наступний за ним оператор не виконується. Найближчий навколишній блок *try* перевіряється на наявність обробника *catch* відповідного збудженому виключенню. Якщо такий знайдеться, управління передається йому. Якщо ні, перевіряється наступний з вкладених операторів *try*, і так до тих пір поки або не буде знайдений відповідний розділ *catch*, або обробник виключень виконуючої системи Java не зупинить програму, вивівши при цьому стан стека викликів. Нижче наведено приклад, в якому спочатку створюється об'єкт-виключення, потім оператор *throw* збуджує виняткову ситуацію, після чого те ж виключення збуджується ще раз кодом розділу *catch,* який перехопив його перший раз.

class ThrowDemo {

static void demoproc () {

try {

throw new NullPointerException ( "demo");

}

catch (NullPointerException e) {

System.out.println ( "перехоплення всередині demoproc");

throw e;

}}

public static void main (String args []) {

try {

demoproc ();

}

catch (NulPointerException e) {

System.out.println ( "ще одне перехоплення:" + e);

}

}}

У цьому прикладі обробка виключення проводиться два рази. Метод *main* створює контекст для виключення і викликає *demoproc*. Метод *demoproc* також встановлює контекст для обробки виключення, створює новий об'єкт класу *NullPointerException* і за допомогою оператора *throw* збуджує це виключення. Виключення перехоплюється в наступному рядку всередині методу *demoproc*, причому об'єкт-виключення доступний коду обробника через параметр e. Код обробника виводить повідомлення про те, що збуджено виключення, а потім знову збуджує його за допомогою оператора *throw*, в результаті чого воно передається обробнику виключень в методі *main*. Нижче наведено результат, отриманий при запуску цього прикладу.

перехоплення всередині demoproc

ще одне перехоплення: java.lang.NullPointerException: demo

### 1.11.5 Ключове слово throws

Якщо метод здатний порушувати виключення, які він сам не обробляє, він повинен оголосити про таку поведінку, щоб методи, які його викликають могли обробити ці виключення. Для створення списку виключень, які можуть збуджуватися методом, використовується ключове слово *throws*. Якщо метод в явному вигляді (тобто за допомогою оператора *throw*) збуджує виключення відповідного класу, тип класу виключення повинен бути зазначений в операторі *throws* в оголошенні цього методу. З огляду на це наш колишній синтаксис визначення методу повинен бути розширений в такий спосіб:

тип ім’я\_методу (список аргументів) throws список\_виключень {}

Нижче наведено приклад програми, в якій метод *procedure* намагається збудити виключення, не забезпечуючи ні програмного коду для його перехоплення, ні оголошення цього виключення в заголовку методу. Такий програмний код не буде трансльованим.

class ThrowsDemo1 {

static void procedure () {

System.out.println ( "inside procedure");

throw new IllegalAccessException ( "demo");

}

public static void main (String args []) {

procedure ();

}}

Для того, щоб ми змогли успішно транслювати цей приклад, нам доведеться повідомити транслятору, що *procedure* може порушувати виключення типу *IllegalAccessException* і в методі *main* додати код для обробки цього типу виключень:

class ThrowsDemo {

static void procedure () throws IllegalAccessException {

System.out.println ( "inside procedure");

throw new IllegalAccessException ( "demo");

}

public static void main (String args []) {

try {

procedure ();

}

catch (IllegalAccessException e) {

System.out.println ( "caught" + e);

}

}}

**Результат**

inside procedure

caught java.lang.IllegalAccessException: demo

### 1.11.6 Розділ finally

Іноді потрібно гарантувати, що певну ділянку коду буде виконано незалежно від того, які виключення були порушені і перехоплені. Для створення такої ділянки коду використовується блок *finally*. Навіть в тих випадках, коли в методі немає розділу *catch,* відповідного збудженому виключенню, блок *finally* буде виконаний до того, як управління перейде до оператору, наступному за розділом *try*. У кожного розділу *try* повинен бути принаймні один розділ *catch* або блок *finally*. Блок *finally* дуже зручний для закриття файлів і звільнення будь-яких інших ресурсів, захоплених для тимчасового використання на початку виконання методу. Нижче наведено приклад класу з двома методами, завершення яких відбувається з різних причин, але в обох випадках перед виходом виконується код розділу *finally*.

class FinallyDemo {

static void procA () {

try {

System.out.println ( "inside procA");

throw new RuntimeException ( "demo");

}

finally {

System.out.println ( "procA's finally");

}}

static void procB () {

try {

System.out.println ( "inside procB");

return;

}

finally {

System.out.println ( "procB's finally");

}}

public static void main (String args []) {

try {

procA ();

}

catch (Exception e) {}

procB ();

}}

У цьому прикладі в методі *procA* через порушення виключення відбувається передчасний вихід з блоку *try*, але по шляху «назовні» виконується розділ *finally*. Інший метод *procB* завершує роботу виконанням оператора *return*, який стоїть в *try*-блоці, але й при цьому перед виходом з методу виконується програмний код блоку *finally*. Нижче наведено результат, отриманий при виконанні цієї програми.

inside procA

procA's finally

inside procB

procB's finally

#### *Підкласи Exception*

Тільки підкласи класу *Throwable* можуть бути збуджені або перехоплені. Інші використовуватися в якості виключень не можуть. Найбільш загальний шлях для використання виключень - створення своїх власних підкласів класу *Exception*. Нижче наведена програма, в якій оголошено новий підклас класу *Exception*.

class MyException extends Exception {

private int detail;

MyException(int a) {

detail = a;

}

public String toString() {

return "MyException[" + detail + "]";

}

}

class ExceptionDemo {

static void compute(int a) throws MyException {

System.out.println("called computer” + a + ".");

if (a > 10)

throw new MyException(a);

System.out.println("normal exit.");

}

public static void main(String args[]) {

try {

compute(1);

compute(20);

}

catch (MyException e) {

System.out.println("caught" + e);

}

} }

**Результат**

called compute(1).

normal exit.

called compute(20).

caught MyException[20]

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Поясніть поняття «виключення».
2. Перерахуйте відомі вам типи виключень.
3. Яке призначення розділів try і catch?
4. Для чого створюють вкладені розділи try?
5. Коли треба створювати кілька розділів catch?
6. Яке призначення оператора throw?
7. Коли використовують ключове слово throws?
8. Яке призначення розділу finally?

## Пакети

У Java немає заголовних файлів. Для боротьби з можливими конфліктами імен уся глобальна область видимості розділяється на *пакети*, що поєднують групи файлів.

*Пакет (package)* - це контейнер, який використовується для того, щоб ізолювати імена класів. Наприклад, ви можете створити клас *List*, вкласти його в пакет і не думати після цього про можливі конфлікти, які могли б виникнути якби хто-небудь ще створив клас з ім'ям *List*.

*Пакет* - це набір взаємозалежних файлів, які здійснюють взаємодію один з одним через загальний доступ до функцій і даних і через загальний простір імен. Усі вихідні файли одного пакету повинні міститися в одній теці. Пакети можуть вкладатися один в інший. Кожен клас Java знаходиться в будь-якому пакеті. Імена пакетів повинні збігатися з іменами тек, у яких знаходяться файли вихідного тексту (чи скомпільовані файли).

Всі назви класів, які використовуються в програмах по замовчуванню будуть розташовуватися в одному просторі імен (*name space*), якщо додатково не вказати ім’я пакета. Щоб уникнути конфліктних ситуацій з іменами треба піклуватися про те, щоб у кожного класу було своє унікальне ім'я. Пакети - це механізм, який служить як для роботи з простором імен, так і для обмеження видимості.

Перший оператор, що може з'явитися в вихідному файлі Java (не враховуючи порожніх рядків і коментарів) найчастіше - це оператор *package (пакет)*, який повідомляє транслятору, в якому пакеті повинні міститися класи, визначені в цьому файлі:

package package\_name;

Пакети створюють різні простори імен, в яких зберігаються імена класів. Якщо оператор *package* не вказано, класи потрапляють в безіменне простір імен, що використовується за замовчуванням, межі якого визначаються поточною текою. Якщо ви сповіщаєте, що клас належить певному пакету, наприклад:

рackage java.awt.image;

то і вихідний код цього класу повинен зберігатися в каталозі *java / awt / image*.

Каталог, який транслятор Java розглядатиме, як кореневий для ієрархії пакетів, можна задавати за допомогою змінної оточення СLASSPATH. За допомогою цієї змінної можна задати кілька кореневих каталогів, розділяючи їх за допомогою крапки з комою (;), як в звичайному PATH.

При спробі розмістити клас в пакет, ви відразу натрапите на жорстку вимогу точного збігу ієрархії каталогів з ієрархією пакетів. Ви не можете перейменувати пакет, не перейменувавши каталог, в якому зберігаються його класи.

Після оператора *package*, але до будь-якого визначення класів у вихідному Java-файлі, може бути присутнім список операторів *import*. Загальна форма оператора *import* така:

іmport пакет1 [.пакет2]. (ім’я\_класу | \*);

Тут *пакет1* - ім'я пакета верхнього рівня, *пакет2* - це необов'язкове ім'я пакета, вкладеного в перший пакет і відокремлене крапкою. І, нарешті, після вказівки шляху в ієрархії пакетів, вказується або ім'я класу, або метасимвол «зірочка». «Зірочка» означає, що, якщо Java-транслятору буде потрібний який-небудь клас, для якого пакет не вказано явно, він повинен переглянути весь вміст пакету із зірочкою для пошуку необхідного класу. У наведеному нижче фрагменті коду показані обидві форми використання оператора *import*:

import java.util.Date

import java.io. \*;

Використовувати без потреби форму запису оператора *import* з використанням «зірочки» не рекомендується, тому що це може значно збільшити час трансляції коду (на швидкість роботи і розмір програми використання «зірочки» не впливає).

Всі вбудовані в Java класи, які входять в комплект поставки, зберігаються в пакеті з ім'ям *java*. Базові функції мови зберігаються у вкладеному пакеті *java.lang*. Весь цей пакет автоматично імпортується транслятором в усі програми. Це еквівалентно розміщенню на початку кожної програми оператора:

import java.lang. \*;

Якщо в двох пакетах, що підключаються за допомогою форми оператора *import* із зірочкою, є класи з однаковими іменами, проте ви їх не використовуєте, транслятор на це не відреагує. А ось при спробі використовувати такий клас, ви відразу отримаєте повідомлення про помилку, і вам доведеться переписати оператори *import*, щоб явно вказати, клас якого пакета ви маєте на увазі. Наприклад:

class MyDate extends Java.util.Date {}

Через наявність пакетів Java повинна вміти працювати з чотирма категоріями видимості між елементами класів:

* Підкласи в тому ж пакеті.
* Не підкласи в тому ж пакеті.
* Підкласи в різних пакетах.
* Класи, які не є підкласами і не входять в той самий пакет.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що таке пакет і яке його призначення?
2. Які вбудовані в мову пакети ви знаєте?
3. Які є категорії видимості між елементами класів, що забезпечуються пакетами?

## Абстрактні класи і інтерфейси

### 1.13.1 Абстрактні класи

Бувають ситуації, коли потрібно визначити клас, в якому задана структура якоїсь абстракції, але повна реалізація всіх методів відсутня. У таких випадках ви можете за допомогою модифікатора типу *abstract* оголосити, що деякі з методів обов'язково повинні бути заміщені в підкласах. Будь-який клас, що містить методи *abstract*, також повинен бути оголошений як *abstract*. Оскільки у таких класів відсутня повна реалізація, їх представників не можна створювати за допомогою оператора *new*. Крім того, не можна оголошувати абстрактними конструктори і статичні методи. Будь-який підклас абстрактного класу або зобов'язаний надати реалізацію всіх абстрактних методів свого суперкласу, або сам повинен бути оголошений абстрактним.

abstract class A {

abstract void callme();

void met() {

System.out.println("метод met реалізовано у класі А");

} }

class B extends A {

void callme() {

System.out.println("метод callme реалізовано у класі В");

} }

class Abstract {

public static void main(String args[]) {

A a = new B():

a.callme();

a.met();

} }

У нашому прикладі для виклику реалізованого у підкласі класу А методу *callme* і реалізованого в класі А методу *met* використовується динамічне призначення методів.

**Абстрактний клас -** це клас, який оголошений як абстрактний, при цьому він може містити, а може і не містити абстрактні методи.

Абстрактні класи не можуть бути використані для створення екземплярів, але вони можуть бути суперкласами.

Можна сказати, що це клас, який визначає узагальнену форму усімх його підкласів (тобто такий клас визначає тільки природу методів).

*Особливості абстрактного класу:*

* Якщо клас включає абстрактні методи, то він обов'язково повинен бути оголошений як абстрактний;
* Якщо абстрактний клас - це суперклас, то підклас забезпечує реалізацію всіх абстрактних методів свого батьківського класу;
* Якщо підклас не забезпечує реалізацію всіх абстрактних методів свого батьківського класу, то він також повинен бути оголошений як *абстрактний*.

**Приклад**

abstract class Animal {

public abstract void move();

}

abstract class Reptiles extends Animal {

}

class Boa extends Reptiles {

@Override

public void move(){

System.out.println("Boa move");

}

}

public class Main {

public static void main(String[] arg) {

Animal myAnimal = new Boa();

myAnimal.move();

}

}

*Абстрактний метод* - це метод, який оголошений без реалізації (без тіла) і закінчується крапкою з комою. Наприклад:

public abstract void move();

Особливості абстрактного методу:

* Якщо клас включає абстрактні методи, то він обов'язково повинен бути оголошений як абстрактний;
* Якщо абстрактний клас - це суперклас, то підклас забезпечує реалізацію всіх абстрактних методів свого батьківського класу;
* Якщо підклас не забезпечує реалізацію всіх абстрактних методів свого батьківського класу, то він також повинен бути оголошений як абстрактний.

### 1.13.2 Інтерфейси

*Інтерфейс* у Java - це тип, схожий на клас, який може містити тільки константи, оголошення методів і вкладені типи.

**Опис інтерфейсу:**

{public} **interface** <ідентифікатор> { <тіло інтерфейсу> }

**Реализація інтерфейсу**:

{<доступ>} **class** <ідентифікатор> **implements** <інтерфейс> {, інтерфейс>}

{ <реалиіація класу> }

Інтерфейси утворюють "контракт" між класом і зовнішнім світом, і цей контракт налаштовується під час роботи компілятора.

Якщо клас реалізовує інтерфейс, то всі методи, визначені цим інтерфейсом, повинні з'явитися в його вихідному коді для успішної компіляції.

Інтерфейс - це аналог абстрактного классу, який має додаткові вимоги:

* Об'єкти інтерфейсів не можуть бути створені;
* Інтерфейси можуть бути тільки реалізовані в класах;
* Інтерфейси можуть бути розширені в інших інтерфейсах.
* Кожен клас може бути створений тільки від одного базового класу, але при цьому реалізовувати один чи кілька інтерфейсів.
* Клас, що реалізує інтерфейс, повинен забезпечити реалізацію всіх методів, оголошених в інтерфейсі.
* В іншому випадку такий клас буде абстрактним і повинен бути оголошений зі специфікатором **abstract**.
* Методи, визначені в інтерфейсі, є за замовчуванням абстрактними і відкритими. У класі, що реалізує інтерфейс, такі методи повинні бути оголошені як **public.**

Починаючи з Java 8, методи за замовчуванням і статичні методи можуть мати реалізацію у визначенні інтерфейсу (з ключовим словом *default*).

Інтерфейси Java створені для підтримки динамічного вибору (resolution) методів під час виконання програми. Інтерфейси схожі на класи, але на відміну від останніх у інтерфейсів немає змінних представників, а в оголошеннях всіх методів відсутня реалізація. Клас може мати будь-яку кількість інтерфейсів. Все, що потрібно зробити — це реалізувати в класі повний набір методів всіх інтерфейсів. Сигнатури таких методів класу повинні точно збігатися з сигнатурами методів інтерфейсу, що реалізується в цьому класі. Інтерфейси володіють своєю власною ієрархією, що не перетинається з класовою ієрархією успадкування. Це дає змогу реалізувати один і той самий інтерфейс в різних класах, ніяк не пов'язаних по лінії класової ієрархії успадкування. Саме в цьому і полягає головна сила інтерфейсів. За допомогою інтерфейсів можна реалізувати множинне успадкування.

#### Оператор interface

Визначення інтерфейсу схоже з визначенням класу, відмінність полягає в тому, що в інтерфейсі відсутні оголошення даних і конструкторів. Загальна форма інтерфейсу наведена нижче:

interface ім'я {

тип ім’я\_final1-змінної = значення;

тип\_результату ім’я\_методу1(список параметрів);

}

Зверніть увагу — у методів, що оголошуються в інтерфейсі відсутні оператори тіла. Оголошення методів завершується символом ; (крапка з комою). В інтерфейсі можна оголошувати і змінні, при цьому вони неявно оголошуються *final* - змінними. Це означає, що клас реалізації не може змінювати їх значення. Крім того, при оголошенні змінних в інтерфейсі їх обов'язково потрібно ініціалізувати константними значеннями. Нижче наведено приклад визначення інтерфейсу, що містить єдиний метод з ім'ям *callback* і одним параметром типу *int*.

interface Callback {

void callback(int param);

}

#### Оператор implements

Оператор *implements* — це доповнення до визначення класу, що реалізує деякий інтерфейс(и).

class ім'я \_класу [extends суперклас] [implements інтерфейс0 [,інтерфейс1...]]

{ тіло класу }

Якщо в класі реалізується кілька інтерфейсів, то їхні імена розділяються комами. Нижче наведено приклад класу, в якому реалізується визначений нами інтерфейс:

class Client implements Callback {

void callback(int p) {

System.out.println("callback called with" + p);

} }

У черговому прикладі метод *callback* інтерфейсу, визначеного раніше, викликається через змінну - посилання на інтерфейс:

class TestIface {

public static void main(String args[]) {

Callback с = new Сlient();

c.callback(42);

} }

**Результат**

callback called with 42

Методи в інтерфейсі (що не були оголошені як **default** або **static**) неявно є абстрактними і відкритими, тому модифікатор **abstract** і **public** не потрібно вказувати в його описі. Всі константи, визначені в інтерфейсі, неявно є **public** **static** **final**.

*//Приклад неявного опису методів і константи в інтерфейсі:*

public interface Behavior Animal {

int SIGNAL = 5;

void move();

void vocalize();

void preen();

}

Клас, який реалізує (розширює) інтерфейс, повинен представити повну реалізацію всіх методів, оголошених в інтерфейсі, але клас може містити і власні методи.

Якщо клас розширює інтерфейс, але повністю не реалізує його методи, то цей клас повинен бути оголошений як *абстрактний*.

При реалізації методи інтерфейсу повинні бути описані з модифікатором доступу **public**.

**Приклад**

interface Shape {

double PI = Math.PI;

double getSquare();

}

class Circle implements Shape {

double radius;

Circle(double x) { radius = x; }

public double getSquare() {

return PI\*radius\*radius;

}

}

public class Demo1 {

public static void main(String [] a) {

Shape object = new Circle(7.01);

System.out.print("Square: " + object.getSquare());

}

}

Об'єктне посилання на інтерфейсний тип можна пов'язувати з будь-яким об'єктом, чий клас реалізує цей інтерфейс, однак через це посилання можна викликати тільки методи, оголошені в інтерфейсі. Власні методи класу недоступні.

Клас може розширювати тільки один клас і при цьому реалізовувати безліч інтерфейсів.

Якщо в суперкласі і інтерфейсі виявляться методи з однаковою сигнатурою, то реалізований метод інтерфейсу перекриє видимість методу суперкласу. Для виклику методу суперкласу необхідно використовувати звернення **super**.

Клас може реалізувати більш одного інтерфейсу. Об'єкти можуть мати кілька типів: тип свого класу і типи всіх інтерфейсів, які вони реалізують.

#### Змінні інтерфейсу в якості констант

У тому випадку, коли ви реалізуєте в класі який-небудь інтерфейс, всі його змінні можна використовувати в класі як константи.

Для імпорту констант в простір імен класу можна використовувати і змінні з модифікатором *final*.

У наведеному нижче прикладі проілюстровано використання інтерфейсу для визначення констант і їх подальшого використання.

**Приклад**

import java.util.Random;

interface SharedConstants {

int NO = 0;

int YES = 1;

int MAYBE = 2;

int LATER = 3;

int SOON = 4;

int NEVER = 5; }

class Question implements SharedConstants {

Random rand = new Random();

int ask() {

int prob = (int) (100 \* rand.nextDouble());

if (prob < 30) return NO ; // 30%

else if (prob < 60) return YES; // 30%

else if (prob < 75) return LATER; // 15%

else if (prob < 98) return SOON; // 13%

else

return NEVER ; // 2% } }

}

}

class AskMe implements SharedConstants {

static void answer(int result) {

switch(result) {

case NO:

System.out.println("No");

break;

case YES:

System.out.println("Yes");

break;

case MAYBE:

System.out.println("Maybe");

break;

case LATER:

System.out.println("Later");

break;

case SOON:

System.out.println("Soon");

break;

case NEVER:

System.out.println("Never");

break;

}}}

public class Interface {

public static void main(String[] args) {

Question q = new Question();

AskMe.answer(q.ask());

AskMe.answer(q.ask());

AskMe.answer(q.ask());

AskMe.answer(q.ask());

}

}

Нижче наведено результат роботи програми (результати при різних запусках програми відрізняються, оскільки в ній використовується клас генерації випадкових чисел *Random* пакету java.util.):

Later

Sоon

No

Yes

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Які класи називаються абстрактними?
2. Що таке інтерфейс?
3. Чим відрізняються абстрактний клас і інтерфейс?
4. Яке призначення слова implements?
5. Як можна використовувати змінні інтерфейсу?
6. Як реалізується множинне спадкування у мові Java?

## Робота з рядками

### 1.14.1 Клас String

У пакет *java.lang* вбудований клас, який інкапсулює структуру даних, що відповідає символьному рядку. Цей клас, званий *String*, не що інше, як об'єктне представлення символьного масиву. За визначенням, кожен об'єкт класу String не може бути змінений. У цьому класі є методи, які дозволяють порівнювати рядки, здійснювати в них пошук і вибирати певні символи і підрядки. Клас *StringBuilder*, а також його попередник *StringBuffer* використовується тоді, коли рядок після створення потрібно змінювати.

Ці класи оголошені як *final*, і це означає, що ні від одного з них не можна утворювати підкласи. Це було зроблено для того, щоб можна було застосувати деякі види оптимізації, які дозволяють підвищити продуктивність при виконанні операцій обробки рядків.

#### Конструктори

Як і у випадку будь-якого іншого класу, ви можете створювати об'єкти типу *String* за допомогою оператора *new*. Для створення пустого рядка використовується конструктор без параметрів:

String s = new String():

Наведений нижче фрагмент коду створює об'єкт s типу *String*, ініціалізує його рядком з трьох символів, переданих конструктору в якості параметра в символьному масиві.

char chars[] = { 'а', 'b', 'с' }:

String s = new String(chars);

System.out.println(s):

Цей фрагмент коду виводить рядок «abc».

У наступного конструктора — 3 параметри:

String(char chars[], int початковийІндекс, int числоСимволів);

Використаємо такий спосіб ініціалізації в наступному прикладі:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' }:

String s = new String(chars,2,3);

System.out.println(s);

Цей фрагмент виведе «cde».

#### Створення рядків

Java має стандартне скорочення для операції створення рядка — запис у вигляді літерала, в якому вміст рядка знаходиться у парі подвійних лапок. Нижче наводиться фрагмент коду еквівалентний одному з попередніх, в якому рядок ініціалізувався масивом типу *char*.

String s = "abc";

System.out.println(s);

Один із поширених методів, який використовується з об'єктами *String* — метод *length*, що повертає кількість символів в рядку. Черговий фрагмент коду виводить число 3 – кількість символів в рядку.

String s = "abc";

System.out.println(s.length);

В Java зручним є те, що для кожного рядка-літерала створюється свій представник класу *String*, так що ви можете викликати методи цього класу безпосередньо для рядків-літералів, а не тільки для змінних-посилань. Черговий приклад також виводить число 3.

System.out.println("abc".Length());

#### Злиття рядків

В наступному прикладі за допомогою оператора + (конкатенація) три рядки об'єднуються в один.

int age=21;

String s = "He is " + age + " years old.";

System.out.println(s);

Такий вираз прочитати і зрозуміти безумовно легше, ніж його еквівалент, записаний з явними викликами методу *append*:

int age=21;

StringBuilder s = new StringBuilder("He is ").append(age);

s.append(" years old.").toString();

System.out.println(s);

Ще раз звернемося до нашого останнього прикладу:

String s = "He is" + age + " years old.";

У тому випадку, коли age — не String, а, скажімо, типу int, то це ціле значення змінної передається методу *append* класу StringBuilder, який перетворює його у текстовий вигляд і додає в кінець об'єкта-рядка. Треба бути уважним при злитті рядків з числами, інакше результат може вийти зовсім не той, який ви чекали. Погляньте на наступний рядок:

String s = "four:" + 2 + 2;

Виходячи з послідовності виконання операторів, у результаті буде "four: 22".

Для того, щоб спочатку виконати додавання цілих чисел, потрібно використовувати дужки :

String s = "four: " + (2 + 2);

В цьому випадку результатом буде "four: 4".

#### Витяг символів

Для того, щоб витягти один символ з рядка, ви можете посилатися безпосередньо на символ у рядку за допомогою його індексу в методі *charAt*. Нижче показано приклад застосування цього методу для підрахунку кількості слів у рядку.

String s="This is just test string";

int i, Words = 0;

for(i=0;i<s.length();i+=1)

{

if(s.charAt(i) == ' ')

Words += 1;

}

Words += 1;

System.out.println("Вихідний рядок: "+s);

System.out.println("У вихідному рядку "+Words+" слів");

Якщо треба за раз витягти кілька символів, можна скористатися методом *getChars*. У наведеному нижче фрагменті показано, як слід вибрати масив символів з об'єкту типу *String*.

class getCharsDemo {

public static void main(String args[]) {

String s = "This is a demo of the getChars method.";

int start = 10;

int end = 14;

char buf[] = new char[end - start];

s.getChars(start, end, buf, 0);

System.out.println(buf);

} }

Як видно, метод *getChars* не включає у вихідний буфер символ з індексом *end*. У виведенні результату роботи нашого прикладу рядок складається з чотирьох символів. В результаті отримаємо «demo».

Для зручності роботи в *String* є ще одна функція — *toCharArray*, яка повертає у результуючому масиві типу *char* весь рядок. Альтернативна форма того ж самого механізму дозволяє записати вміст рядка в масив типу *byte*, при цьому значення старших байтів 16-бітних символах відкидаються. Відповідний метод називається *getBytes*, і його параметри мають той же сенс, що і параметри *getChars*, але з єдиною різницею — в якості третього параметра треба використовувати масив типу byte.

String s = "This is a demo of the getChars method.";

char buf[];

buf=s.toCharArray();

System.out.println(buf);

В результаті отримаємо повне речення рядка s.

Можна вибрати підрядок з об'єкта String, використовуючи метод *substring*. Цей метод створює нову копію символів з того діапазону індексів оригінальної рядка, який вказано при виклику методу. Можна вказати тільки індекс першого символу потрібного підрядка — тоді будуть скопійовані всі символи, починаючи з зазначеного і до кінця рядка. Також можна вказати початковий і кінцевий індекси — при цьому у новий рядок будуть скопійовані всі символи, починаючи з першого зазначеного, і до (але не включаючи його) символу, заданого кінцевим індексом.

"Hello World".substring(6) -> "World"

"Hello World".substring(3,8) -> "lo Wo"

#### Порівняння рядків

Для порівняння двох рядків, можна скористатися методом *equals* класу *String*. Альтернативна форма цього методу називається *equalsIgnoreCase*, при її використанні відмінність регістрів букв в порівнянні не враховується. Нижче наведено приклад, що ілюструє використання обох методів:

class equalDemo {

public static void main(String args[]) {

String s1 = "Hello";

String s2 = "Hello";

String s3 = "Good-bye";

String s4 = "HELLO";

System.out.println(s1 + "дорівнює" + s2 + "-> " + s1.equals(s2));

System.out.println(s1 + "дорівнює" + s3 + "-> " + s1.equals(s3));

System.out.println(s1 + "дорівнює" + s4 + "-> " + s1.equals(s4));

System.out.println(s1 + " дорівнює без урахування регістру" + s4 + "-> " +

s1.equalsIgnoreCase(s4));

} }

Результат виконання цього прикладу :

Hello дорівнює Hello -> true

Hello дорівнює Good-bye -> false

Hello дорівнює HELLO -> false

Hello дорівнює без урахування регістру HELLO -> true

В класі *String* реалізована група сервісних методів, які є спеціалізованими версіями методу *equals*. Метод *regionMatches* використовується для порівняння підрядка у вихідному рядку з підрядком в рядку-параметрі. Метод *startsWith* перевіряє, чи починається даний підрядок фрагментом, переданим методу в якості параметра. Метод *endsWith* перевіряє, чи збігається з параметром кінець рядка.

Метод equals і оператор == виконують різні перевірки. Якщо метод *equal* порівнює символи всередині рядків, то оператор == порівнює дві змінні-посилання на об'єкти і перевіряє, чи вказують вони на різні об'єкти або на один і той самий. В черговому прикладі це добре видно — вміст двох рядків однаковий, але, тим не менш, це — різні об'єкти, так що *equals* і == дають різні результати.

class EqualsNotEqualTo {

public static void main(String args[]) {

String s1 = "Hello";

String s2 = new String(s1);

System.out.println(s1 + "дорівнює" + s2 + "-> " + s1.equals(s2));

System.out.println(s1 + "== " + s2 + ", -> " + (s1 == s2));

} }

Ось результат запуску цього прикладу:

Hello equals Hello -> true

Hello == Hello -> false

#### Впорядкування рядків

Часто буває недостатньо просто знати, що два рядки є ідентичними. Для застосунків, в яких потрібно виконати сортування, треба знати, який з двох рядків менше по довжині. Для відповіді на це питання потрібно скористатися методом *compareTo* класу String. Якщо ціле значення, повернуте методом, від’ємне, то рядок, з якого був викликаний метод, менше рядка-параметра, якщо додатне, то навпаки. Якщо ж метод compareTo повернув значення нуль, рядки ідентичні.

#### Пошук у рядках

В клас String включена підтримка пошуку певного символу або підрядка, для цього в ньому є два методи — *indexOf* і *lastIndexOf*. Кожен з цих методів повертає індекс того символу, який треба знайти, або індекс початку шуканого підрядка. У будь-якому випадку, якщо пошук виявився невдалим, методи повертають значення -1. У наступному прикладі показано, як користуватися різними варіантами цих методів пошуку.

class indexOfDemo {

public static void main(String args[]) {

String s = "Now is the time for all good men" +

"to come to the aid of their country" +

"and pay their due taxes.";

System.out.println(s);

System.out.println("indexOf(t) =" + s.indexOf('f'));

System.out.println("lastlndexOf(t) =" + s.lastlndexOf('f'));

System.out.println("indexOf(the) =" + s.indexOf("the"));

System.out.println("lastlndexOf(the) =" + s.lastlndexOf("the"));

System.out.println("indexOf(t, 10) =" + s.indexOf('f' , 10));

System.out.println("lastlndexOf(t, 50) =" + s.lastlndexOf('f' , 50));

System.out.println("indexOf(the, 10) =" + s.indexOf("the", 10));

System.out.println("lastlndexOf(the, 50) =" + s.lastlndexOf("the", 50));

} }

Нижче наведено результат роботи цієї програми. Зверніть увагу на те, що індекси в рядках починаються з нуля.

Now is the time for all good men to come to the aid of their country and pay their due taxes.

indexOf(t) = 7

lastlndexOf(t) = 87

indexOf(the) = 7

lastlndexOf(the) = 77

indexOf(t, 10) = 11

lastlndexOf(t, 50) = 44

indexOf(the, 10) = 44

lastlndexOf(the, 50) = 44

#### Модифікація рядків при копіюванні

Оскільки об'єкти класу *String* не можна змінювати, то кожний раз, коли треба модифікувати рядок, доведеться або копіювати його в об'єкт типу *StringBuilder*, або використовувати один з методів класу *String*, які описані нижче і створюють нову копію рядка, вносячи в неї зміни.

Злиття, або конкатенація рядків виконується за допомогою методу *concat*. Цей метод створює новий об'єкт *String*, копіюючи в нього вміст вихідного рядка і додаючи в його кінець рядок, зазначений в якості параметра методу.

"Hello".concat(" World") -> "Hello World"

У методі *replace* в якості параметрів задаються два символи. Усі символи, що збігаються з першим, замінюються в новій копії рядка на другий символ.

"Hello".replace('l' , 'w') -> "Hewwo"

Пара методів *toLowerCase* і *toUpperCase* перетворює всі символи вихідної рядка в нижній і верхній регістр, відповідно.

"Hello".toLowerCase() -> "hello"

"Hello".toUpperCase() -> "HELLO"

Метод *trim* вилучає з вихідної рядка всі лідируючи та кінцеві пробіли.

" Hello World ".trim() -> "Hello World"

Для перетворення значення якогось іншого типу даних в рядок існує метод *valueOf*. Цей статичний метод визначений для будь-якого існуючого в Java типу даних (тобто маємо справу з суміщеними методами).

### 1.14.2 Класи StringBuilder та StringBuffer

Клас *StringBuilder* з'явився в Java SE5, а цього використовувався клас *StringBuffer*. Клас *StringBuilder* містить повний набір методів класу *StringBuffer* і ще має додаткові методи, наприклад, *reverse*. У програмах, які не створюють окремих потоків управління, функції класу *StringBuilder* виконуються більш ефективно.

Об'єкти типу *StringBuilder* та *StringBuffer* являють собою послідовності символів, які можуть розширюватися і модифікуватися. Java активно використовує цей клас, але деякі програмісти вважають за краще працювати тільки з об'єктами типу *String*.

#### Конструктори

Об'єкт *StringBuilder* можна створити без параметрів, при цьому в ньому буде зарезервовано місце для розміщення 16 символів без можливості зміни довжини рядка. Ви також можете передати конструктору ціле число, для того щоб чітко визначити необхідний розмір буфера. І, нарешті, ви можете передати конструктору рядок, при цьому він буде скопійований в об'єкт і додатково до цього в ньому буде зарезервовано місце ще для 16 символів. Поточну довжину *StringBuilder* можна визначити, викликавши метод *length*, а для визначення всього місця, зарезервовані під рядок в об'єкті *StringBuilder* потрібно скористатися методом *capacity*. Нижче наведено приклад, що пояснює це:

class StringBuilderDemo {

public static void main(String args[]) {

StringBuilder sb = new StringBuilder("Hello");

System.out.println("buffer =" + sb);

System.out.println("length =" + sb.length());

System.out. println("capacity =" + sb.capacity());

} }

Ось результат роботи цієї програми, з якого видно, що в об'єкті *StringBuilder* для маніпуляцій з рядком зарезервовано додаткове місце.

buffer = Hello

length = 5

capacity = 21

#### Методи

Якщо після створення об'єкта *StringBuilder* треба зарезервувати місце для певної кількості символів, ви можете для встановлення розміру буфера скористатися методом *ensureCapacity*. Це буває корисно, коли ви заздалегідь знаєте, що вам доведеться додавати до буферу багато невеликих рядків.

Якщо вам знадобиться в явному вигляді встановити довжину рядка у буфері, скористайтеся методом *setLength*. Якщо ви задасте значення більше ніж довжина рядка, який міститься в об'єкті, цей метод заповнить кінець нового рядка символами з кодом нуль.

Один символ може бути вибраний з об'єкта *StringBuilder* за допомогою методу *charAt*. Метод *setCharAt* дозволяє записати в задану позицію рядка потрібний символ.

Метод *append* класу *StringBuilder* зазвичай викликається неявно при використанні оператора + у виразах з рядками. Для кожного параметра викликається метод *String.valueOf* і його результат додається до поточного об'єкту *StringBuilder*. До того ж при кожному виклику метод append повертає посилання на об'єкт *StringBuilder*, з яким він був викликаний. Це дозволяє вибудовувати в ланцюжок послідовні виклики методу, як це показано в наступному прикладі.

class appendDemo {

public static void main(String args[]) {

String s;

int a = 42;

StringBuilder sb = new StringBuilder (40);

s = sb.append("a = ").append(a).append("!").toString();

System.out.println(s);

} }

Результат роботи цього прикладу:

а = 42!

Метод *insert* ідентичний методу *append* в тому сенсі, що для кожного можливого типу даних існує своя суміщена версія цього методу. На відміну від *append*, він не додає символи, які повертаються методом *String.valueOf*, в кінець об'єкта *StringBuilder*, а вставляє їх у певне місце в буфері, яке задається першим параметром. В черговому прикладі рядок "there" вставляється між "hello" і "world!".

class insertDemo {

public static void main(String args[]) {

StringBuilder sb = new StringBuilder("hello world !");

sb.insert(6,"there ");

System.out.println(sb);

} }

Після виконання ця програма виводить наступний рядок:

hello there world!

**Приклад**

За допомогою генератора випадкових чисел створити послідовність цілих чотирьохзначних чисел у кількості N (ціле двозначне випадкове число). Сформувати нову послідовність, помінявши місцями першу та другу цифри у кожному з чотирьохзначних чисел. Обидві послідовності вивести у дві колонки.

int N = (int)(Math.random()\*59+10.0); //випадкове двозначне число

System.out.println("Число1 Число2");

int a=0;

for (int I=2; I<N; I++)

{

a = (int)(Math.random()\*8999+1000);// випадкове чотиризначне число

String s = Integer.toString(a); // перетворення числа в рядок

String t1 = (s.substring(0,1)); // перша цифра числа

String t2 = (s.substring(1,1)); // друга цифра числа

StringBuilder sb = new StringBuilder(s);

System.out.print(sb+" ");

sb.replace(0, 1, t2); // заміна першої цифри

sb.replace(1, 1, t1); // заміна другої цифри

System.out.println(sb);

}

Результат:

Число1 Число2

7718 7718

5854 8554

6741 7641

…

### 1.14.3 Клас StringTokenizer

Обробка тексту часто передбачає розбиття тексту на послідовність лексем - слів (tokens). Клас-утиліта *StringTokenizer* призначений для такого розбиття, часто званого лексичним аналізом або скануванням. Для роботи *StringTokenizer* вимагає вхідний рядок і рядок символів-розділювачів. За замовчуванням в якості набору розділювачів використовуються звичайні символи-роздільники: пробіл, табуляція, перехід на наступний рядок і повернення каретки. Після того, як об'єкт *StringTokenizer* створений, для послідовного вибору лексем з вхідного рядка використовується його метод *nextToken*. Інший метод — *hasMoreTokens* — повертає *true* в тому випадку, якщо в рядку ще залишилися невибрані лексеми. *StringTokenizer* також реалізує інтерфейс *Enumeration*, а це означає, що замість методів *hasMoreTokens* і *nextToken* ви можете використовувати методи *hasMoreElements* і *nextElement*, відповідно.

**Приклад**

String in = "Назва=Философия Java:" + "Автор=Брюс Эккель:" + "isbn=978-5-496-01127-3:" + "Рік=2015:" + "Сайт=www.mindviewinc.com";

StringTokenizer st = new StringTokenizer(in, "=:");

while (st.hasMoreTokens()) {

String key = st.nextToken();

String val = st.nextToken();

System.out.println(key + "\t" + val);

В результаті виконання програми будемо мати таке:

Назва Философия Java

Автор Брюс Эккель

isbn 978-5-496-01127-3

Рік 2015

Сайт [www.mindviewinc.com](http://www.mindviewinc.com)

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Які існують засоби обробки символьних рядків?
2. Які існують методи класу String?
3. Які є способи створення і ініціалізації об’єктів класу String?
4. Які існують методи класу StringBuilder?
5. Чим відрізняються класи StringBuffer і StringBuilder?
6. Коли використовують клас StringTokenizer?
7. Які оператори або методи використовують для конкатенації рядків?
8. Наведіть приклад використання якогось методу для оброблення символьної інформації.
9. Як можна перетворити числове значення змінної в символьний рядок?

## Потоки виконання

### 1.15.1 Поняття потоку виконання

Паралельне програмування, пов'язане з виконанням декількох задач - концептуальна парадигма, в якій ви поділяєте свою програму на два або кілька потоків, які можуть виконуватися одночасно.

У багатьох середовищах паралельне виконання завдань представлено в тому вигляді, який в операційних системах називається багатозадачністю. Це не те ж саме, що паралельне виконання потоків. У багатозадачних операційних системах ви маєте справу з повноваговими процесами, в системах з паралельним виконанням задач окремі завдання частіше за все називають *потоками виконання* (*light-weight processes, threads*).

У системах без паралельних потоків використовується підхід, який називають циклом обробки подій. У цій моделі єдиний потік виконує нескінченний цикл, який чекає виникнення події, яку потім перевіряє і обробляє.

*Процес* **-** це програма, запущена на виконання**.**

*Потоки (threads***) -** це завдання, на які розбивається процес і які можуть виконуватися паралельно.

Однопотокова програма має єдину точку входу (метод main ()) і єдину точку виходу.

Багатопотокова програма має початкову точку входу (метод main ()), а потім може бути багато точок входу і виходу, які можуть працювати спільно з main ().

Потік не є самостійною програмою, тому що він не може працювати сам по собі.   Він працює в рамках процесу (програми) і розділяє доступ до ресурсів процесу.

Якщо ви можете розділити свою задачу на потоки, які незалежно виконуються і можете автоматично перемикатися з одного потоку, який чекає настання події, на інший, якому є чим зайнятися, то за той самий проміжок часу ви виконаєте більше роботи. Імовірність того, що більше ніж одному з потоків одночасно надовго потрібен процесор, мала.

Виконуюча система Java залежить від використання потоків, і всі її бібліотеки класів написані з урахуванням особливостей програмування в умовах паралельного виконання. Java використовує потоки для того, щоб зробити середовище програмування асинхронним.

### 1.15.2 Створення потоку - клас Thread

Існує три способи створення потоку:

1. Розширити клас Thread
2. Реалізувати інтерфейс Runnable
3. Реалізувати інтерфейс Callable з бібліотеки java.util.concurrent.Callable

Перші два способи були ще в Java 1.0, останній зявився в Java 5.0

Перший спосіб створення потоку- розширити клас Thread**:**

* Описати підклас класу Thread, в якому перевизначити метод **run** ();
* Створити екземпляр описаного підкласу;
* Викликати метод **start** () на створеному об'єкті.

Тіло методу *run()* і є тіло потоку. Потік може бути запущено на виконання тільки викликом методу *start()* класу Thread або його підкласу.

Клас *Thread* інкапсулює всі засоби, які можуть вам знадобитися при роботі з потоками. При запуску Java-програми в ній вже є один потік, який виконується. Ви завжди можете з'ясувати, який саме потік виконується в даний момент, за допомогою виклику статичного методу *Thread.currentThread*. Після того, як ви отримаєте дескриптор потоку, ви можете виконувати над цим потоком різні операції навіть в тому випадку, коли паралельні потоки відсутні. У прикладі показано, як можна управляти потоком, який виконуються в даний момент.

class CurrentThreadDemo {

public static void main(String args[]) {

Thread t = Thread.currentThread();

t.setName("Мій потік");

System.out. println("current thread: " + t);

try {

for (int n = 5; n > 0; n--) {

System.out.println(" " + n);

Thread.sleep(1000);

} }

У цьому прикладі поточний потік зберігається в локальній змінній *t.* Ми використовуємо цю змінну для виклику методу *setName*, який змінює внутрішнє ім'я потоку на "*Мій потік*", з тим, щоб результат роботи програми був легким для читання. На наступному кроці ми входимо в цикл, в якому ведеться зворотний відлік рахівника з 5 до 1, причому на кожній ітерації за допомогою виклику методу *Thread.sleep()* робиться пауза тривалістю в 1 секунду. Аргументом для цього методу є значення тимчасового інтервалу в мілісекундах. Цикл укладений в *try/catch* блок тому, що метод *Thread.sleep* () може викинути виключення *InterruptedException*. Це виключення збуджується в тому випадку, якщо будь-якому іншому потоку знадобиться перервати даний потік. В даному прикладі ми в такій ситуації просто виводимо повідомлення про перехоплення виключення. Нижче наведено результат роботи цієї програми:

current thread: Thread[Мій потік,5,main]

5

4

3

2

1

### 1.15.3 Створення потоку - інтерфейс Runnable

Другий спосіб створення потоку- реалізувати інтерфейс *Runnable*:

* Описати клас, який реалізує інтерфейс Runnable;
* Створити об'єкт класу Thread, якому передати посилання на об'єкт описаного класу;
* Викликати метод start () на об'єкті класу Thread.

Для того, щоб реалізувати цей інтерфейс, клас повинен надати визначення методу *run*. Нижче наведено приклад, в якому створюється новий потік.

class ThreadDemo implements Runnable {

ThreadDemo() {

Thread ct = Thread.currentThread();

System.out.println("currentThread: " + ct);

Thread t = new Thread(this, "Demo Thread");

System.out.println("Thread created: " + t);

t.start();

try {

Thread.sleep(3000);

}

catch (InterruptedException e) {

System.out.println("interrupted");

}

System.out.println("exiting main thread");

}

public void run() {

try {

for (int i = 5; i > 0; i--) {

System.out.println("" + i);

Thread.sleep(1000);

} }

catch (InterruptedException e) {

System.out.println("child interrupted");

}

System.out.println("exiting child thread");

}

public static void main(String args[]) {

new ThreadDemo();

} }

Тут створено другий потік за допомогою іншого екземпляру класу *Thread*, при цьому перший параметр конструктора *this* говорить про те, що ми хочемо викликати метод *run* поточного об’єкта. Метод *start* починає виконання нового потоку, після чого основний потік переходить в стан чекання на три секунди, а потім виводе повідомлення і завершує роботу. Другий потік продовжує свою роботи до тих пір, поки значення рахівника циклу не зменшиться до нуля. Нижче показано результат роботи програми через п’ять секунд.

Thread created: Thread[Demo Thread,5,main]

5

4

3

exiting main thread

2

1

exiting child thread

Який з описаних спосібів використати?

Через спадкування Thread:

* Легше використовувати в простих застосунках;
* Однак потік не може бути спадкоємцем ніякого іншого класу;
* Є можливість перевизначити поведінку управлінням потоком (різні методи класу Thread);

Через реалізацію Runnable:

* Більш гнучкий, тому що потік може бути спадкоємцем будь-якого класу;
* Відокремлено тіло потоку від управління потоком;

Використовується в високорівневому управлінні потокам

### 1.15.4 Пріоритети потоків

*Пріоритети* потоків - це просто цілі числа в діапазоні від 1 до 10 і має сенс тільки співвідношення пріоритетів різних потоків. Пріоритети ж використовуються для того, щоб вирішити, коли потрібно зупинити один потік і почати виконання іншого. Це називається перемиканням контексту. Потік може добровільно віддати управління - за допомогою явного системного виклику або при блокуванні на операціях введення-виведення, або він може бути припинений примусово. У першому випадку перевіряються всі інші потоки, і управління передається тому з них, який готовий до виконання і має найвищий пріоритет. У другому випадку, фоновий потік, незалежно від того, чим він зайнятий, призупиняється примусово для того, щоб почав виконуватися потік з більш високим пріоритетом.

Якщо ви хочете добитися від Java передбачуваної незалежної від платформи поведінки, вам слід проектувати свої потоки таким чином, щоб вони самостійно звільняли процесор. Нижче наведено приклад з двома потоками з різними пріоритетами, які не ведуть себе однаково на різних платформах. Пріоритет одного з потоків за допомогою виклику *setPriority* встановлюється на два рівні вище Thread.NORM\_PRIORITY, тобто пріоритету по замовчуванню. У іншого потоку пріоритет, навпаки, на два рівня нижче. Обидва цих потоки запускаються і працюють протягом десяти секунд. Кожен з них виконує цикл, в якому збільшується значення змінної-лічильника. Через десять секунд після їх запуску основний потік зупиняє роботу і виводить значення лічильників, які показують, скільки ітерацій циклу встиг виконати кожен з потоків.

class Clicker implements Runnable {

int click = 0;

private Thread t;

private boolean running = true;

public clicker(int p) {

t = new Thread(this);

t.setPriority(p);

}

public void run() {

while (running) {

click++;

} }

public void stop() {

running = false; }

public void start() {

t.start();

} }

class HiLoPri {

public static void main(String args[]) {

Thread.currentThread().setPriority(Thread.MAX\_PRIORITY);

clicker hi = new clicker(Thread.NORM\_PRIORITY + 2);

clicker lo = new clicker(Thread.NORM\_PRIORITY - 2);

lo.start();

hi.start();

try {

Thread.sleep(10000);

}

catch (Exception e) {

}

lo.stop();

hi.stop();

System.out.println(lo.click + " vs. " + hi.click);

} }

По результату роботи програми можна побачити, що потоку з більш низьким пріоритетом надано менше процесорного часу.

### 1.15.5 Синхронізація потоків

Коли двом або більше потокам потрібен паралельний доступ до одних і тих самих даних (інакше кажучи, до спільного ресурсу), потрібно подбати про те, щоб в кожен конкретний момент часу доступ до цих даних надавався тільки одному з потоків. Java для такої синхронізації надає вбудовану в мову програмування підтримку за допомогою так званого монітора. *Монітор* - це об'єкт, який використовується як засувка. Тільки один з потоків може в даний момент часу володіти монітором. Коли потік отримує цю засувку, кажуть, що він увійшов в монітор. Всі інші потоки, які намагаються увійти в той самий монітор, будуть заморожені доти, поки потік-власник не вийде з монітора.

У кожного Java-об'єкта є пов'язаний з ним монітор, а для того, щоб увійти в нього, треба викликати метод цього об'єкта, зазначений ключовим словом *synchronized*. Для того, щоб вийти з монітора і тим самим передати управління об'єктом іншому потоку, власник монітора повинен всього лише повернутися з синхронізованого методу.

*Синхронізація* **-** це процес упорядкування (поділу) доступу до ресурсів між декількома потоками.Коли потік отримує блокування, то кажуть, що він *«увійшов в монітор».* Всі інші потоки, які намагаються ввести монітор на цьому ж об'єкті, будуть призупинені (кажуть, що вони *«чекають монітор»*).

Способи отримання блокування:

* Використання синхронізованих методів;
* Використання синхронізованих блоків;
* Використання API високого рівня з пакета **java.util.concurrent.locks**

**Синхронізовані методи**

В опис методу додається ключове слово **synchronized**.

<доступ> **synchronized** <опис методу >

Коли JVM виконує **synchronized-**метод, то виконуючий потік визначає, що в method\_info встановлено прапор ACC\_SYNCHRONIZED. Тоді потік автоматично ставить блокування на об'єкт, викликає метод і знімає блокування при виході з методу. Якщо метод кидає виняток, то потік автоматично знімає блокування.

Коли один потік виконує **synchronized-**метод для об'єкта, всі інші потоки, які викликають **synchronized-**методи для того ж об'єкта блокуються (припиняють виконання) поки перший потік не залишить об'єкт;

Коли **synchronized-**метод завершується, то він автоматично встановлює відношення "відбувається до" з будь-яким подальшим викликом **synchronized-**методу на тому ж об'єкті (при цьому гарантується те, що зміни в стані об'єкту видно всім потокам).

**Приклад роботи методу без синхронізації**

class Synchro {

private File f = new File("d:\\data.txt");

public Synchro() {

try { f.delete();

f.createNewFile();

} catch (IOException e) { e.printStackTrace(); }

}

public void writing(String str, int i) {

try {

RandomAccessFile raf = new RandomAccessFile(f, "rw");

raf.seek(raf.length());

System.out.print(str);

raf.writeBytes(str);

Thread.sleep((long) (Math.random() \* 15));

System.out.println("->" + i );

raf.writeBytes("->" + i + " ");

} catch (IOException| InterruptedException e) {

e.printStackTrace();

}

}

}

public class SynchroThreads {

public static void main(String[] args) {

Synchro s = new Synchro();

T t1 = new T("First", s);

T t2 = new T("Second", s);

t1.start();

t2.start();

}

}

class T extends Thread {

private String str;

private Synchro s;

public T(String str, Synchro s) {

this.str = str;

this.s = s;

}

public void run() {

for (int i = 0; i < 5; i++)

s.writing(str, i);

}

}

**Результат**

SecondFirst->0

Second->0

First->1

Second->1

First->2

First->3

First->2

Second->4

->3

Second->4

**Приклад методу з синхронізацією:**

class Synchro {

private File f = new File("d:\\data.txt");

public Synchro() {

try { f.delete();

f.createNewFile();

} catch (IOException e) { e.printStackTrace(); }

}

public synchronized void writing(String str, int i) {

try {

RandomAccessFile raf = new RandomAccessFile(f, "rw");

raf.seek(raf.length());

System.out.print(str);

raf.writeBytes(str);

Thread.sleep((long) (Math.random() \* 15));

System.out.println("->" + i );

raf.writeBytes("->" + i + " ");

} catch (IOException| InterruptedException e) {

e.printStackTrace();

}

}

}

**Результат**

First->0

First->1

First->2

First->3

First->4

Second->0

Second->1

Second->2

Second->3

Second->4

**Синхронізовані блоки**

*Синхронізований блок* – це блок коду, позначений ключовим словом **synchronized**. Синхронізований блок завжди синхронізується на якомусь об'єкті. Всі синхронізовані блоки, які синхронізовані на одному і тому ж об'єкті, можуть мати тільки один виконуваний потік всередині нього в даний момент часу. Всі інші потоки, які намагаються увійти в синхронізований блок, блокуються, поки потік всередині синхронізованого блоку не вийде з нього.

**Приклад блоку без синхронизації**

public class TwoThread {

public static void main(String args[]) {

final StringBuffer s = new StringBuffer();

new Thread() {

public void run() {

int i = 0;

while (i++ < 3) {

s.append("A");

try { sleep(15);

} catch (InterruptedException e) {

e.printStackTrace(); }

System.out.println(s);

}}}.start();

// …

new Thread() {

public void run() {

int j = 0;

while (j++ < 3) {

s.append("B");

try { sleep(10);

} catch (InterruptedException e) {

e.printStackTrace(); }

System.out.println(s);

}}}.start();

}

}

**Результат**

BA

BAB

BABB

BABBA

BABBA

BABBAA

**Приклад синхронізованого блоку**

public class TwoThread {

public static void main(String args[]) {

final StringBuffer s = new StringBuffer();

new Thread() {

public void run() {

int i = 0;

synchronized (s) { // Синхронізований блок на об’єкті s

while (i++ < 3) {

s.append("A");

try { sleep(15);

} catch (InterruptedException e) {

e.printStackTrace(); }

System.out.println(s);

}}}}.start();

// …

new Thread() {

public void run() {

int j = 0;

synchronized (s) {

while (j++ < 3) {

s.append("B");

try { sleep(10);

} catch (InterruptedException e) {

e.printStackTrace(); }

System.out.println(s);

}}}}.start();

}

}

**Результат**

A

AA

AAA

AAAB

AAABB

AAABBB

Синхронізація призводить до конкуренції потоків і змушує виконавчу систему Java виконувати один або кілька потоків більш повільно або навіть припинити їх виконання

**Висновки по використанню ключового слова** **synchronized.**

Ключове слово **synchronized** використовується для синхронізації:

* блоків – по монітору вказаного об’єкта
* методів – по монітору поточного об’єкта (this)
* статичних методів – по монітору класу

Потік отримує блокування рівня *об'єкта*, коли він виконує синхронізований метод екземпляра класу, але він отримує блокування рівня *класу* при вході в *статичний* синхронізований метод. *Статичний* і *нестатичний* синхронізовані методи можуть працювати одночасно або конкурентно, оскільки вони блокуються на різних об'єктах.

Коли *synchronized*-метод статичний, то його монітор належить всьому класу і цей монітор загальний для всіх. Тому перший потік працює з об’єктом статичного класу, а другий потік буде чекати доступу до цього об’єкту. Якщо методи не статичні, але вони належать одному статичному класу, то у них монітор також буде один.

Коли один потік виконує *synchronized***-**метод для одного об'єкта класу, а інший потік, викликає інший *synchronized***-**метод для **іншого об'єкта цього ж класу**, то вони чекати один одного не будуть. Ніякого блокування не буде і слово *synchronized*значення не має (бо у кожного об`єкта є свій монітор і вони заблокуються незалежно).

В потоці в середині одного **synchronized можна викликати інший synchronized метод** (оскільки монітор уже захвачений).

Рекомендується синхронизувати **блок, а не метод**, якщо це можливо.

### 1.15.6 Взаємодія потоків

Потокам часто доводиться координувати свої дії. Найбільш поширена процедура координації - це обачний (обережний) блок. Такий блок починається з перевірки умови продовження своєї роботи:

* якщо результат **true** - потік виконується;
* якщо результат **false** - потік поступається виконанням іншому потоку, поки не отримає повідомлення, що він може відновити свою роботу;

У Java є механізм «спілкування» між потоками, заснований на методах *wait, notify і notifyAll.* Ці методи реалізовані, як *final*-методи класу *Object*, так що вони є в будь-якому Java-класі. Всі ці методи повинні викликатися тільки з *синхронізованих* методів. Правила використання цих методів прості:

* *wait* - призводить до того, що активний потік віддає управління і переходить в режим очікування - до тих пір поки інший потік не викличе метод *notify* з тим же об'єктом.
* *notify* - виводить зі стану очікування перший з потоків, що викликали *wait* з даним об'єктом.
* *notifyAll* - виводить зі стану очікування всі потоки, що викликали *wait* з даним об'єктом.

**Приклад**

Постачальник (OOO) привозить на склад (Q) товар, а склад сповіщає споживача (PPP), що його можна забрати. Після вивезення товару зі складу постачальник сповіщається про те, що можна завозити нову партію товару.

class PC {

public static void main(String args[]) {

Q q = new Q();

new OOO (q);

new PPP (q);

} }

Склад

class Q { //склад - черга

int amount;

synchronized int get() {

System.out.println(«Отримано: " + amount);

return amount;

}

synchronized void set(int n) {

amount = n;

System.out.println(«Відправлено: " + amount);

value = true; notify();

}

}

//Постачальник

class OOO implements Runnable { //постачальник

Q stock;

OOO(Q stock) {

this.stock = stock;

new Thread(this, "Постачальник").start();

}

public void run() {

int number = 0;

while (true)

stock.set(++number);// Постачання товару

}

}

//Споживач

class PPP implements Runnable { //споживач

Q stock;

PPP(Q stock) {

this.stock = stock;

new Thread(this, «Споживач").start();

}

public void run() {

while (true)

stock.get();//Отримати товар

}

}

Хоча методи **put** і **get** класу Q синхронізовані, в нашому прикладі немає нічого, що б могло завадити постачальнику постачати товар до того, як їх отримає споживач, і навпаки, споживачеві нічого не заважає багаторазово забирати один і той же товар зі складу. Так що результат виконання програми містить зовсім не ту послідовність повідомлень, яку нам би хотілося мати:

**Результат**

Для зупинки натисніть Ctrl+C

Отправлено: 1

Отримано: 1

Отримано : 1

Отримано : 1

Отримано : 1

Отримано : 1

Отправлено: 2

Отправлено: 3

Отправлено: 4

Отправлено: 5

Отримано : 5

Як видно, після того, як постачальник поміщає в змінну n значення 1, споживач починає працювати і вибирає це значення 5 разів поспіль. Положення можна виправити, якщо постачальник буде при введенні нового значення встановлювати прапор, наприклад, заносити в логічну змінну значення *true*, після чого буде в циклі перевіряти її значення до тих пір поки споживач не обробить дані і не скине прапор в *false*.

Правильним шляхом для отримання такого результату в Java є використання викликів **wait** і **notify** для передачі сигналів в обох напрямках. Усередині методу **get** Споживач чекає (виклик **wait**), поки Постачальник не повідомить (**notify**), що для Споживача готова чергова порція даних (товару). Після того, як Споживач опрацює ці дані в методі **get**, він сповіщає об'єкт класу Постачальник (знову виклик **notify**) про те, що він може передавати наступну порцію даних. Відповідно, всередині методу **put** Постачальник чекає (**wait**), поки Споживач не обробе дані, потім він передає нові дані і сповіщає (**notify**) про це Споживача. Нижче наведено переписаний відповідним чином клас Q.

class Q { //склад - черга

int amount;

boolean value = false;// true – товар на складі є, false – немає

synchronized int get() {

while ( !value)

try { wait(); // Чекати поки на складі не з'явиться товар

} catch (InterruptedException e) { System.out.println(e); }

System.out.println(«Отримано: " + amount);

value = false; notify();//Забравши товар, інформувати постачальника

return amount;

}

synchronized void set(int n) {

while (value)

try { wait(); // Чекати поки на складі не з'явиться місце

} catch (InterruptedException e) { System.out.println(e); }

amount = n;

System.out.println(«Відправлено: " + amount);

value = true; notify();//Поставивши товар, інформувати споживача

}

}

public class Main {

public static void main(String[] args) {

Q obj = new Q();

new OOO(obj); //постачальник

new PPP(obj); //споживач

System.out.println("Для зупинки натисніть Ctrl+C");

}

}

**Результат**

Для зупинки натисніть Ctrl+C

Отправлено: 1

Отримано: 1

Отправлено: 2

Отримано : 2

Отправлено: 3

Отримано : 3

Отправлено: 4

Отримано : 4

Отправлено: 5

Отримано : 5

#### Методи класу Thread

Перші три методи - статичні, які можна викликати безпосередньо з ім'ям класу Thread.

* *currentThread* - повертає об'єкт Thread, що виконується в даний момент.
* *yield* – виклик його призводить до того, що виконуюча система перемикає контекст з поточного на наступний доступний потік. Це один із способів гарантувати те, що фонові потоки коли-небудь отримають управління.
* *sleep (int n)* - при виклику цього методу виконуюча система блокує потік на *n* мілісекунд. Після того, як цей інтервал часу закінчиться, потік знову буде здатний виконуватися. Час зазвичай вказується в мілісекундах (перша форма, з одним параметром). Час можна вказати додатково до мілісекунд ще і в наносекундах (друга форма, з двома параметрами). Кидає виняток типу InterruptedException якщо деякий потік перериває той, що викликав метод sleep ().
* *isAlive()* – з’ясовує, чи виконується потік.
* *join()* - поточний потік чекає завершення вказаного потоку. Дозволяє одному потоку "приєднатися в кінець" іншого потоку (якщо потік B не може виконувати свою роботу, поки потік А не завершить свою, тоді необхідно, щоб потік В "приєднався" до потоку А. Це означає, що потік не стане актуальним працездатним, поки не закінчиться потік A).

**Відмінність методів *sleep та wait***

***Wait*** використовується при синхронізації потоків для зняття блокування, а ***sleep*** – при часовій синхронізації, тобто для простої призупинки потоку. *Sleep – призупиняє поточний потік, але не знімає блокування,* а *wait –знімає блокування.*

*Thread.sleep ()*є статичним методом, який можна *викликати з будь-якого контекста*, а  *wait* визначений в *java.lang.Object, але він* може *викликатися тільки з синхронізированого блока.*

Наприклад, якщо у нас працює два потоки і ми призупинили один з них на 1 сек, щоб за цей час щось міг зробити інший потік, то цього проміжку часу може бути занадто багато або недостатньо для другого потоку виконати свої дії. Тобто другий потік або буде простоювати, поки не скінчиться 1 сек (хоча міг би уже працювати), або почне працювати до того, коли перший потік не завершить свою роботу і може завадити йому, або буде чекати завершення роботи першого.

Наступний приклад присвячений використанню метода *join*, який теж не знімає блокування, а забезпечує певну послідовність роботи потоків і означає: "приєднати поточний потік до кінця потоку t, так щоб t закінчився раніше, ніж поточний потік зможе працювати знову".

Thread t = new Thread();

t.start();

t.join();

**Приклад**

public class MyThread1 extends Thread {

public void run() {

for (int i = 0; i < 4; i++) {

System.out.println("Thread:" + getName() + " i=" + i);

try {

Thread.sleep(500);

} catch (InterruptedException e) { }

}

}

}

// …

public class DemoJoin {

public static void main(String[] args) {

System.out.println("main method start");

MyThread1 thr1 = new Thread();

thr1.start();

System.out.println("thread started");

try {

thr1.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("main method end");

}

}

**Результат**

main method start

thread started

Thread:Thread-0 i=0

Thread:Thread-0 i=1

Thread:Thread-0 i=2

Thread:Thread-0 i=3

main method end

Крім того існують методи об'єкта Thread

* *start* - говорить виконуючій системі Java, що необхідно створити системний контекст потоку і запустити цей потік. Після виклику цього методу в новому контексті буде викликаний метод *run* новоствореного потоку. Метод *start* з даним об'єктом можна викликати тільки один раз.
* *run* - єдиний метод інтерфейсу Runnable. Він викликається з методу *start* після того, як середовище виконає необхідні операції з ініціалізації нового потоку. Якщо відбувається повернення з методу *run,* поточний потік зупиняється.
* *stop - в*иклик цього методу призводить до негайної зупинки потоку. Більш акуратний спосіб зупинити виконання потоку - встановити значення будь-якої змінної-прапора, передбачивши в методі *run* код, який, перевіривши стан прапора, завершив би виконання потоку.
* *suspend* - відрізняється від методу stop тим, що метод призупиняє виконання потоку, не руйнуючи при цьому його системний контекст. Якщо виконання потоку призупинено викликом *suspend*, ви можете знову активізувати цей потік, викликавши метод resume.
* *resume* - використовується для активізації потоку, припиненого викликом *suspend*. При цьому не гарантується, що після виклику *resume* потік негайно почне виконуватися, оскільки в цей момент може виконуватися інший потік з вищим пріоритетом. Виклик *resume* лише робить потік здатним виконуватися, а коли йому буде передано управління, вирішить планувальник.
* *setPriority (int p)* - встановлює пріоритет потоку, що задається цілим значенням переданого методу параметра. У класі Thread є кілька визначених пріоритетів-констант: MIN\_PRIORITY, NORM\_PRIORITY і MAX\_PRIORITY, які відповідають значенням 1, 5 і 10. Більшість застосунків користувача повинно виконуватися на рівні NORM\_PRIORITY плюс-мінус 1. Пріоритет фонових завдань, наприклад, мережевого введення-виведення або перемальовування екрану, слід встановлювати в MIN\_PRIORITY. Запуск потоків на рівні MAX\_PRIORITY вимагає обережності. Якщо у потоків з таким рівнем пріоритету відсутні виклики sleep або *yield*, може виявитися, що вся виконуюча система Java перестане реагувати на інші процеси.
* *setPriority* – встановлює поточний пріоритет потоку - ціле значення в діапазоні від 1 до 10.
* *setName (String name)* - привласнює потоку вказане в параметрі ім'я. Це допомагає при налагодженні програм з паралельними потоками. Присвоєне за допомогою *setName* ім'я буде з'являтися у всіх трасуваннях стеку, які виводяться при отриманні інтерпретатором не перехопленого виключення.
* *getName* - повертає рядок з ім'ям потоку, встановленим за допомогою виклику setName.

### 1.15.7 Стан потоку

**Створений** (новий) - це стан потоку після створення екземпляра Thread, але метод **start** () для потоку не був викликаний (потік не вважається живим);

**Працездатний** (запущений) - це стан потоку, коли він може виконуватися, але планувальник потоків не вибрав його, щоб виконувати (на потоці викликаний метод **start** () і тепер потік вважається живим);

**Працюючий** (виконується) - це стан потоку коли планувальник потоків вибирає його з пулу працездатних, щоб потік в даний момент став виконуваним;

**Зупинений** («мертвий») - потік вважається «мертвим», коли його метод *run* () завершено (якщо викликати метод *start* () на мертвому екземплярі *Thread*, то станеться виключення часу виконання).

**Очикующий / заблокований / сплячий** - це стан потоку, коли він не може бути обраним для виконання (потік все ще живий, але в даний час не має права працювати, тобто він може повернутися до стану працездатного пізніше, якщо відбудеться певна подія) :

* **Заблокований** - потік може бути заблокований в очікуванні ресурсу (в цьому випадку подія, яка повертає його в стан «працездатний» - цє наявність ресурсів);
* С**плячий** - для потоку запущений *sleep* () на деякий період часу (в цьому випадку, подія, яка повертає його в стан «працездатний» - це закінчення його часу у сплячому стані);
* О**чікуючий** - потік викликає метод **wait** () (в цьому випадку подія, яка повертає його в стан «працездатний» - це відправка повідомлення про те, що він більше може не чекати).

### 1.15.8 Використання загальних ресурсів – змінних

Потоки спілкуються при спільному доступі до ресурсів. Ця форма спілкування виробляє два види можливих помилок:

* зіткнення (втручання) потоків;
* узгодженість пам'яті.

Необхідний інструмент для запобігання цих помилок - це синхронізація. Однак, синхронізація призводить до конкуренції потоків (коли два або більше потоків намагаються отримати доступ до одного ресурсу одночасно) і змушує виконавчу систему Java виконувати один або кілька потоків більш повільно або навіть припинити їх виконання.

Потоки можуть використовувати загальні ресурси – *змінні* у пам’яті:

* будь-яка змінна перед використанням потоком завжди зчитується з основної пам'яті і будь-яка змінна, яка записується потоком, завжди записується в основну пам'ять;
* з метою підвищення ефективності роботи JRE зберігає локальну копію змінної у кожному потоці, який посилається на неї (ці «внутрішньо-потокові» копії змінних допомагають уникнути перевірки основної пам'яті кожного разу, коли потрібен доступ до значення змінної);
* немає ніяких гарантій щодо того, коли JVM зчитує / записує дані внутрішньої пам'яті з / в основну пам'ять (тобто не обов'язково , що зміни, які проводяться з «внутрішньо-потоковою» копією змінної, відразу будуть «бачити» інші потоки );

Локальний кеш потоку має сенс в тому випадку, якщо змінні програми не будуть змінюватися ззовні. Якщо це не так, то корисно використовувати зміннй, яка позначається ключовим словом *volatile*.

Ключове слово *volatile*(ставитьсяперед змінною) гарантує, що всі потоки можуть змінювати і бачити зміни, які відбуваються з цією змінною*:*

* *volatile***-**змінна завжди зчитується з основної пам'яті, і ніколи не зберігається в пам'яті потоку, а значить, завжди доступна будь-якому потоку;
* при запитах на читання і запис *volatile***-**змінної від декількох потоків, системою гарантується виконання спочатку запитів на запис;
* гарантується атомарность операцій читання / запису *volatile***-**змінної; ля всіх інших операцій, таких як ++, повинна виконуватися синхронізація;
* потоки не блокуються в очікуванні звільнення ресурсу***.***

Оскільки змінна оголошена як *volatile* може змінюватися різними потоками, то природно очікувати, що JRE забезпечить ту чи іншу форму синхронізації таких *volatile***-**змінних. JRE дійсно неявно забезпечує синхронізацію при доступі до *volatile*-змінних, але з одним виключенням: читання *volatile*-змінної і запис в *volatile*-змінну синхронізовані, але не для *неатомарних* операцій.

Це означає, що наступний код не є потокобезпечним:

*myVolatileVar ++;*

В наступному сценарії: запускаються два потоки, і один з них зчитує змінну A як 5, а другий - як 10. Якщо значення змінної А змінилося з 5 на 10, то перший потік не дізнається про зміну і буде зберігати неправильне значення A. Однак якщо змінна А позначена як **volatile**, то коли б потік не зчитував значення A, він буде звертатися до головної копії A і зчитувати її поточне значення.

public volatile int value = 0;

first-thread { value++ }

second-thread{ value++ }

Інкрементування змінної не є атомарної операцією: воно вимагає, як мінімум, три інструкції:

* прочитати дані
* збільшити на одиницю
* записати дані

Відповідно, два потоки повинні виконати цю послідовність, але порядок їх виконання між ними не визначений. Через це і можуть виникнути ситуації на кшталт наступної:

* Перший потік прочитав дані
* Другий потік прочитав дані
* Перший потік збільшив значення на 1
* Другий потік збільшив значення на 1
* Другий потік записав значення
* Перший потік записав значення

В результаті маємо результат 1, а не 2 як очікувалося.

Щоб такого не відбувалося, використовують або *синхронізацію*, яку можна реалізувати самостійно або використати *атомарні примітиви*з пакета **java.util.concurrent.**

**Приклад**

public class VolatileTest {

private static volatile int varVolat = 0;

private static int varNonVolat = 0;

public static void main(String[] args) {

ChangeListener thread1 = new ChangeListener();

ChangeMaker thread2 = new ChangeMaker();

thread1.start();

thread2.start();

try { Thread.sleep(500);

} catch (InterruptedException e) { }

thread1.interrupt();

thread2.interrupt();

}

// …

static class ChangeMaker extends Thread {

@Override

public void run() {

int local\_value = 0;

while ( !isInterrupted() ) {

varVolat = varNonVolat = ++local\_value;

}

}

}

// …

static class ChangeListener extends Thread {

@Override

public void run() {

while ( !isInterrupted() ) {

if (varVolat != varNonVolat) {

System.out.println("Error: " + varVolat +

" != " + varNonVolat);

}

}

}

}

}

**Результат**

Error:99! = 148

Error:117887! = 117900

Error:119837! = 119843

Error:121639! = 121645

Error:123476! = 123482

Error:125276! = 125284

Таким чином, можна зробити висновки:

* Використання змінних *volatile* знижує ризик *помилок узгодженості пам'яті*, тому що будь-який запис в змінну *volatile* встановлює відношення "відбувається до" з подальшим читанням цієї ж змінної;
* Всі зміни в змінній *volatile* завжди можуть бачитися іншими потоками;
* Коли потік читає змінну *volatile*, він бачить не тільки останні зміни в *volatile*, але і побічні ефекти коду, який призвів до змін.

### 1.15.9 Фонові потоки

*Фоновий потік* або *потік-демон* - це потік, який може виконуватися на тлі основних потоків і використовується тільки для їх обслуговування (наприклад, як таймер, який посилає сигнали іншим потокам через певні інтервали часу).

Якщо в програмі працездатними залишаються тільки фонові потоки, то програма завершує роботу;

Визначити потік як фоновий потік можна до його запуску на виконання викликом методу *setDaemon (boolean isDaemon*).

Метод *isDaemon* () дозволяє визначити, чи є поточний потік фоновим.

В наступному прикладі фонових потоків не існує і цикл відпрацьовує до кінця.

public class MyThread4 extends Thread {

public void run() {

for (int i = 0; i < 6; i++) {

System.out.println(getName() + ", i=" + i);

try { Thread.sleep(1000);

} catch (InterruptedException e) { }

}

}

}

public class Demo {

public static void main(String[] args) {

new MyThread4().start();

try { Thread.sleep(2000);

} catch (InterruptedException e) { }

System.out.println("method main() finished");

}

}

**Результат**

Thread-0, i=0

Thread-0, i=1

method main() finished

Thread-0, i=2

Thread-0, i=3

Thread-0, i=4

Thread-0, i=5

В наступному прикладі в програмі, коли працездатними залишаються тільки фонові потоки, програма завершує роботу.

public class MyThread4 extends Thread {

public void run() {

for (int i = 0; i < 6; i++) {

System.out.println(getName() + ", i=" + i);

try { Thread.sleep(1000);

} catch (InterruptedException e) { }

}

} }

public class DemoDaemon {

public static void main(String[] args) {

MyThread4 tt = new MyThread4();  
 tt.setDaemon(true);  
 tt.start();

try { Thread.sleep(2000);

} catch (InterruptedException e) { }

System.out.println("method main() finished");

} }

**Результат**

Thread-0, i=0

Thread-0, i=1

method main() finished

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. В чому суть поняття потоку?
2. Які існують способи створення потоку?
3. Яке призначення інтерфейсу Runnable?
4. Як встановлювати пріоритети потоків?
5. Які існують засоби синхронізації потоків?
6. Перерахуйте відомі вам методи класу Thread.
7. Які існують стани потоку?
8. Як потоки використовують загальні ресурси-змінні у в пам’яті?
9. Яке призначення ключового слова *volatile?*

## Робота с базами даних

### 1.16.1 ODBC і JDBC

*ODBC* (Open Database Connectivity - відкритий механізм взаємодії з базами даних) — це програмний інтерфейс доступу до баз даних, розроблений фірмою Microsoft у співпраці з Simba Technologies, покликаний уніфікувати програмну взаємодію з системою управління базою даних (СУБД), зробити його незалежним від постачальника СУБД і програмно-апаратної платформи.

На початку 1990 р. існувало кілька постачальників баз даних, кожен з яких мав власний інтерфейс. Якщо застосункам було необхідно спілкуватися з декількома джерелами даних, для взаємодії з кожною з баз даних було необхідно написати свій код. Для вирішення виниклої проблеми створили стандартний інтерфейс для отримання і відправки даних різних типів. Цей інтерфейс був названий ODBC. C допомогою ODBC прикладні програмісти могли розробляти застосунки для використання одного інтерфейсу доступу до даних, не турбуючись про тонкощі взаємодії з декількома джерелами. Це досягається завдяки тому, що постачальники різних баз даних створюють драйвери, що реалізують конкретне наповнення стандартних функцій ODBC з урахуванням особливостей їхнього продукту.

*JDBC* (Java Database Connectivity — з'єднання з базами даних на Java) — дозволяє Java-програмам отримати доступ до даних різних СУБД. ODBC був узятий в якості основи для JDBC через його популярність серед незалежних постачальників програмного забезпечення і користувачів.

Вже у версії JDK1.1 з'явився пакет класів java.sql, що забезпечував більшість функцій, відомих на той час розробникам ODBC-застосунків, наприклад, *java.sql.CallableStatement*, який забезпечує виконання на Java збережених процедур; *java.sql.DatabaseMetaData*, який досліджує базу даних на предмет її реляційної повноти і цілісності з отриманням найрізноманітніших даних про типи і вміст таблиць, полів, індексів, ключів і т. і.; *java.sql.ResultSetMetaData*, за допомогою якого можна виводити в зручному вигляді всю необхідну інформацію з таблиць бази даних або друкувати самі метадані у вигляді назв таблиць і полів.

JDBC дозволяє виконувати три основні функції:

* Встановити з'єднання з базою даних
* Надсилати запити та змінювати стан бази даних
* Обробляти результати запитів

### 1.16.2 Встановлення з'єднання з БД

Оскільки кожна СУБД є окремим програмним продуктом, для підключення до неї Java використовує спеціальний драйвер, який пишеться розробниками цієї СУБД. На офіційному сайті, як правило, доступні для завантаження відповідні драйвери під кожну з версій СУБД.

Кожен JDBC-драйвер належить одному з нижче вказаних типів:

Транслює JDBC в ODBC і для взаємодії з базою даних використовує драйвер ODBC. Компанія Sun включила до складу JDK один такий драйвер - міст JDBC/ODBC. Однак для його використання потрібно відповідним чином встановити і налаштувати ODBC-драйвер.

Створюється переважно на мові Java і частково на власній мові програмування, яка використовується для взаємодії з клієнтським API бази даних. Для використання такого драйвера потрібно крім бібліотеки Java встановити специфічний для даної платформи код.

Створюється тільки на основі бібліотеки Java, в якій використовується незалежний від бази даних протокол взаємодії сервера і бази. Цей протокол дозволяє транслювати запити у відповідності зі специфікою конкретної бази. Якщо код, що залежить від бази даних, знаходиться тільки на сервері, доставка програм істотно спрощується.

Являє собою бібліотеку Java, яка транслює JDBC-запити безпосередньо в протокол конкретної бази даних.

Для того, щоб почати роботу з БД нам потрібно мати драйвер. У наведених нижче прикладах буде розглядатися робота із сервером баз даних MySQL. Сам сервер і драйвер до нього можна взяти на сайті: [*http://dev.mysql.com/downloads*](http://dev.mysql.com/downloads) *(*драйвер- *com.mysql.jdbc* Driver). У розділі MySQL Connector/J вибираємо останню версію так званого коннектора -це jar-архів *mysql-connector-java-5.1.5-bin.jar*. Він має бути зазначений у *classpath*, лежати в папці з проектом або доданий в проект за допомогою можливостей IDE (у Eclipse, наприклад, додавати бібліотеки можна таким чином: : *Import->General->Archive File)*.

Драйвер треба додати до проекту. В Eclipse це можна зробити так: натиснути правою кнопкою миші на назві проекту в оглядачі проектів і вибрати пунктBuild Path -> Configure Build Path. Далі натиснути на вкладці Library і Add External JARs, вибрати потрібний драйвер і натиснути ОК.

Ім'я драйвера до іншої СУБД можна знайти на сайті розробника. Наприклад для Oracle ім'я драйвера буде *oracle.jdbc.driver.OracleDriver*, для MS Access - *sun.jdbc.odbc.JdbcOdbcDriver*.

Перш ніж використовувати драйвер, його необхідно спочатку зареєструвати:

Class.forName("DriverName");

Клас *DriverManager* використовується для встановлення з'єднання з базою даних. Для цього необхідно вказати йому спеціальну URL-адресу, а також логін і пароль користувача, зареєстрованого в СУБД. *URL* — це спеціальний рядок, що має наступний формат:

jdbc:<subprotocol>:<subname>

де <subprotocol> — ім'я драйвера або ім'я механізму підключення,

<subname> — це рядок, в якому вказується хост, порт, ім'я бази даних.

Наприклад, для MySQL URL може бути таким:

jdbc:mysql://localhost:3306/MyDataBaseName

У цьому випадку частина URL *//localhost:3306/MyDataBaseName* описує ім'я хоста, порт і відповідний ідентифікатор для доступу до відповідної бази даних.

Для Oracle: jdbc:oracle:thin:@localhost:1521/MyDataBaseName

Для MS Access: jdbc:odbc:db1\_SQL

В останньому випадку використовується механізм ODBC.

На сайті розробників можна знайти URL для їхнього продукту.

Знаючи URL, логін і пароль користувача, а також маючи зареєстрований в системі драйвер, встановити підключення можна так:

Connection c = DriverManager.getConnection("URL", "User\_Login", "User\_Password");

Об'єкт класу *Connection* являє собою з'єднання з базою даних.

### 1.16.3 Створення та виконання запитів

Після того, як з'єднання з базою даних встановлено, ми можемо відправляти запити. Для цього нам знадобитися клас *Statement*, який призначений для зберігання SQL команд і може бути створений таким чином:

Statement st = c.createStatement ();

Після цього ми можемо виконувати запити. Результати запиту будуть зберігатися в об'єкті класу *ResultSet*.

ResultSet rs = st.executeQuery ( "select \* from Table\_Name");

Цей запит вибере всі дані з таблиці *Table\_Name* і поверне об'єкт *ResultSet*. Також може бути корисним метод *st.execute ( «Запит»)*, але тоді об'єкт *ResultSet* треба отримувати окремим методом *st.getResultSet ()*.

Загальна форма методів для виконання запитів - *executeXXX().* За допомогою методу *executeUpdate ()* можна виконувати команди типу *update, insert, delete, create table, drop table*. Для виконання команди *select* потрібно використовувати метод *executeQuery ().* Існує також універсальний метод *execute ()*, який може застосовуватися для виконання довільних SQL-команд, але він використовується в основному для інтерактивного створення запитів, а також, якщо результатів виконання запитів може бути декілька.

Методи *getString (), getDouble (), getInt ()* повертають відповідно рядкові, дробові і цілочисельні змінні, що знаходяться в полях таблиці. Ці методи можуть приймати номер стовпчика *ResultSet* як аргумент, причому нумерація починається з 1.

Для параметризрваного SQL запиту використовується клас *PreparedStatement*. Він може бути використаний, наприклад, так:

PreparedStatement pst = c.prepareStatement ( "select \* from MoCoUser where login =?");

//? - Це параметр

pst.setString (1, "user"); // встановлення значення параметра.

//Зверніть увагу: нумерація параметрів починається не з 0, а з 1!

// Виконання запиту

ResultSet prs = pst.executeQuery ();

Для виклику функції або процедури використовується клас *CallableStatement*:

CallableStatement cst = c.prepareCall ( "CALL proc\_name (?,?)");

// У процедуру також можна передавати параметри

cst.setInt (1, 100);

cst.setString (2, "String");

ResultSet rs = cst.executeQuery ();

### 1.16.4 Обробка результатів запиту

Результатом виконання SQL запиту до БД буде таблиця. У java результат зберігається в об'єкті класу *ResultSet*. Для виведення рядків цієї таблиці на екран використовується декілька методів.

Для переходу по рядках вперед і назад в класі *ResultSet* використовуються методи *next () і previous ().* Щоб перейти до першого або останнього рядку - *first () і last ()* відповідно. Обробка результатів в циклі може виглядати так:

while (rs.next ()) {

//обробка результатів

}

де *rs* - це об'єкт класу *ResultSet*. Метод *next* () повертає *true*, якщо є такий рядок, *false* - більше рядків немає.

Для отримання значень з певної колонки поточного рядка можна скористатися наведеними нижче або іншими аналогічними методами:

getInteger (<param>), getString (<param>), getDouble (<param>), getDate (<param>),

*де* <param> - це номер колонки ( якщо типу int) або назва колонки ( якщо типу String).

**Приклад**

rs.getString (2); // Поверне рядок, що знаходиться

//у другій колонці поточного рядка

rs.getDouble ( "average\_score"); // Поверне значення типу double, що //знаходиться у колонці з назвою "average\_score".

Оскільки дані в БД можуть мати значення *null*, має сенс перед їх вибіркою перевірити це, щоб не отримати виключення. Така перевірка виконується методом *isNull (<param>),* який поверне *true* або *false*.

Нижче наведено приклад роботи програми з JDBC:

package ru.javaxblog.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class JDBCExample {

     public static void main (String [] args) {

     String user = "User\_Login"; // Логін користувача

     String password = "User\_Password"; // Пароль користувача

     String url = "jdbc: oracle: thin: @localhost: 1521 / MyDBName"; // URL адреса

     String driver = "oracle.jdbc.driver.OracleDriver"; // Ім'я драйвера

     try {

                Class.forName (driver); // Реєструємо драйвер

           }

сatch (ClassNotFoundException e) {

                e.printStackTrace ();

           }

           Connection c = null; // З'єднання з БД

           try {

             c = DriverManager.getConnection (url, user, password); // Установлення з'єднання з БД

             Statement st = c.createStatement (); // Готуємо запит

ResultSet rs = st.executeQuery ( "select \* from Table\_Name"); // Виконуємо запит до //БД, результат - у змінній rs

             while (rs.next ()) {

      System.out.println (rs.getString ( "Login")); // Послідовно для кожного рядка //виводимо значення з колонки ColumnName

                }

           } сatch (Exception e) {

                e.printStackTrace ();

           }

           finally {

                // Закрити з'єднання з БД

                try {

                     if (c! = null)

                     c.close ();

                } сatch (SQLException e) {

                    e.printStackTrace ();

                }

           }

      }

}

### 1.16.5 Створення бази даних

Створимо БД для зберігання книг, яка буде називатися *bookstore*.

**Приклад**

package ru.javatalks.faq.db;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class CreatingDatabase {

// створюємо базу даних:

private final static String createDatabaseQyery =

"CREATE DATABASE bookstore CHARACTER SET utf8 COLLATE utf8\_general\_ci";

public static void main (String [] args) {

Connection connection = null;

Statement statement = null;

try {

// Завантажуємо драйвер

Class.forName ( "com.mysql.jdbc.Driver");

// Потрібно створити підключення до БД. У MySQL обов'язково є системна база,

// до неї і будемо створювати з'єднання.

String url = "jdbc: mysql: // localhost / mysql";

// За замовчуванням користувач - root, паролю немає

connection = DriverManager.getConnection (url, "root", "");

statement = connection.createStatement ();

// створюємо базу за допомогою executeUpdate ().

statement.executeUpdate (createDatabaseQuery);

} catch (Exception e) {

e.printStackTrace ();

} finally {

// позакриваємо все

if (statement! = Null) {

try {

statement.close ();

} catch (SQLException e) {

e.printStackTrace ();

}

}

if (connection! = null) {

try {

connection.close ();

} catch (SQLException e) {

e.printStackTrace ();

}

}

}

}

}

1.16.6 Створення таблиці

Створимо таблицю *books* в новій БД, яка буде вміщувати таку інформацію про книги: назва, автор, ціна, коментарі.

**Приклад**

package ru.javatalks.faq.db;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class CreatingTable {

private final static String createTableQuery = "CREATE TABLE `books` (" +

" `id` int(11) NOT NULL auto\_increment," +

" `title` varchar(50) default NULL," +

" `comment` varchar(100) default NULL," +

" `price` double default NULL," +

" `author` varchar(50) default NULL," +

" PRIMARY KEY (`id`)" +

") ENGINE=InnoDB DEFAULT CHARSET=utf8;";

public static void main(String[] args) {

Connection connection = null;

Statement statement = null;

try {

Class.forName("com.mysql.jdbc.Driver");

//Підключаємося до створеної БД.

//Значення параметрів після "?" зрозуміло з їх імен.

String url = "jdbc:mysql://localhost/bookstore" +

"?characterEncoding=utf8";

connection = DriverManager.getConnection(url, "root", "");

statement = connection.createStatement();

statement.executeUpdate(createTableQuery);

} catch (Exception e) {

e.printStackTrace();

} finally {

//закриваємо все

if (statement != null) {

try {

statement.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (connection != null) {

try {

connection.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

}

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що означають терміни ODBC і JDBC?
2. Як встановити з'єднання з існуючою базою даних?
3. Як створювати та виконувати запити до бази даних?
4. Яке призначення класу ResultSet і його методів?
5. Як створити власну базу даних?
6. Як створити таблицю в базі даних?

## 1.17 Реалізація принципів об’єктно-орієнтованого програмування

### 1.17.1 Спадкування

*Спадкування (inheritance)* – один з принципів *об’єктно-орієнтованого програмування* (ООП). Це механізм, за допомогою якого один об’єкт може набувати властивості іншого. Точніше, об’єкт може успадковувати основні властивості іншого об’єкта та набувати нових рис, які характерні лише для нього. Спадкування є дуже важливим, оскільки воно дозволяє підтримувати принцип *ієрархії класів* (hierarchical classification). Наприклад, клас мобільних телефонів є підкласом класу “Телефон”, який, у свою чергу, входить до ще більшого класу “Електрозв’язок”. Разом з тим, клас “Електрозв’язок” є підкласом класу “Способи зв’язку”, до складу якого, крім електрозв’язку, входять супутниковий зв’язок, радіозв’язок, поштовий зв’язок тощо. Застосування ієрархії класів дозволяє керувати великими потоками інформації. Прикладом подібної ієрархії є системи класифікації в ботаніці, зоології тощо.

В об’єктно-орієнтованому програмуванні спадкування – це процес створювання нових класів, які називають *похідними* класами (*нащадками*) на базі вже існуючих *батьківських* (*базових*) класів. Похідний клас успадковує всі можливості батьківського (базового) класу, але може бути удосконаленим за рахунок змінювання існуючих методів і введення нових власних полів та методів. Батьківський клас (чи клас вищого рівня) при цьому залишається незмінним. Похідний клас (клас нижчого рівня), у свою чергу, сам може слугувати за батьківський.

Найважливішою позитивною якістю спадкування в ООП є те, що воно дає можливість уникати повторювань програмного коду для кожного об’єкта, адже спільний для множини подібних класів код може бути винесено до методів їхнього спільного батьківського класу. Це є доволі зручний спосіб, оскільки програміст може використовувати класи, створені будь-ким іншим, без модифікації коду, лише створюючи похідні класи.

Спадкування буває простим і множинним. Простим називається спадкування, за якого похідний клас має один батьківський клас. *Множинне* *спадкування* означає, що клас має кілька батьківських класів, і застосовується для того, щоб забезпечити похідний клас їхніми властивостями. При застосовуванні множинного спадкування треба ретельно стежити за тим, щоби похідний клас не успадкував поля чи методи з однаковими іменами, але різні за змістом.

Реализація спадкування реалізується клбчовим словом **extends**

{<доступ>}**class** <імя підкласу> **extends** <ім’я суперкласу> {

<тіло підкласу> }

Залежно від прав доступу до елементів суперкласу підклас:

* успадковує відкриті (**public**) і захищені (**protected**) елементи незалежно в одному або різних пакетах вони визначені;
* успадковує елементи **default**, якщо вони визначені в одному пакеті;
* закриті елементи суперкласу (**private**) і конструктори не успадковуються.

Таким чином, для створювання похідного класу використовують ключове слово *class*, після якого записують ім’я нового класу, за яким стоїть слово *extends*, а потім зазначають ім’я батьківського класу:

class Point3D extends Point{ int z;

Point3D(int x, int y, int z) {

this.x = x;

this.у = у;

this.z = z; }

Point3D() {

this(-1,-1,-1);

} }

В цьому прикладі ключове слово *extends* використовується для того, щоб сповістити транслятору про намір створити підклас *Point3D* (точка у просторі) класу *Point* (точка на площині). В цьому підкласі не треба створювати змінні х та у, оскільки *Point3D* успадковує їх від свого суперкласу *Point*.

Слово *super* дозволяє звернутися безпосередньо до конструктора суперкласу.

class Point3D extends Point { int z;

Point3D(int x, int у, int z) {

super(x, y); // ми викликаємо конструктор суперкласу

this.z=z;

public static void main(String args[]) {

Point3D p = new Point3D(10, 20, 30);

System.out.println( " x = " + p.x + " y = " + p.y + " z = " + p.z);

} }

Переваги спадкування реалізуються у багаторазовому використанні полів та методів:

* Після того, як поведінка (метод) визначається в суперкласі, ця поведінка автоматично успадковується всіма підкласами (таким чином, ви пишете метод тільки один раз і він може бути використаний усіма підкласами);
* Після того, як набір властивостей (полів) визначено в суперкласі, цей же набір властивостей успадковується всіма підкласами (клас і його дочірні класи поділяють загальний набір властивостей);
* Підкласу потрібно тільки реалізувати відмінності між собою та суперкласом.

*Відносно полів при спадкуванні:*

* Успадковані поля можуть бути використані безпосередньо (прямо), як і будь-які інші поля;
* Ви можете оголосити поле в підкласі з тим же ім'ям, що і в суперкласі, приховуючи його таким чином не рекомендуется);
* Ви можете оголосити нові поля в підкласі, яких немає в суперкласі.

*Відносно методів при спадкуванні:*

* Успадковані методи можуть бути використані безпосередньо;
* Ви можете написати в підкласі новий метод, який буде мати ту ж саму сигнатуру, що і один з методів в суперкласі, таким чином, перевизначаючи його;
* Ви можете написати в підкласі новий статичний метод, який має ту саму сигнатуру, що і один з методів в суперкласі, таким чином, приховуючи його;
* Ви можете оголосити нові методи в підкласі, яких немає суперкласі.

*Відносно конструкторів у підкласі :*

* Ви можете написати конструктор підкласу, який викликає конструктор суперкласу за допомогою ключового слова super. Зазвичай явний виклик конструктора суперкласу в тілі конструктора підкласу використовується для ініціалізації закритих елементів суперкласу:

**super(<список аргументів>);**

При створенні об'єкта підкласу конструктори всіх класів викликаються в порядку їх підпорядкованості - від суперкласу до підкласу. Такі виклики називаються **ланцюжками конструкторів.**

Тому явне звернення до конструктору суперкласу має бути **першим** оператором конструктора підкласу.

Якщо конструктор підкласу не викликає явно жоден з конструкторів суперкласу, то автоматично викликається конструктор суперкласу **за замовчуванням.**

Однак, якщо в суперкласі немає конструктора за замовчуванням, а конструктор підкласу явно не викликає ніякого іншого конструктора суперкласу, то компілятор Java видає **повідомлення про помилку.**

**Перевизначення методів**

* Метод примірника в підкласі з тієї ж сигнатурою (ім'я, кількість і типи параметрів) і типом значення, що повертається, що і метод примірника в суперкласі, перевизначає (заміщує) цей метод суперкласу.
* Здатність підкласу перевизначати методи дозволяє класу успадковуватися від якогось суперкласу, чия поведінка "досить близько" до нього, щоб змінити поведінку в міру необхідності.
* При перевизначенні методу можна використовувати анотацію @**Override**, яка вказує компілятору, що це перевизначений метод суперкласу (якщо компілятор виявляє, що цей метод не існує ні в одному з суперкласів, то він буде генерувати помилку).
* У підкласі: якщо поле має теж ім'я, що і поле в суперкласі, то воно приховує поле суперкласу, навіть якщо їх типи розрізняються.
* Для звернення до поля суперкласу використовується ключове слово **super.**
* При використанні ключового слова **super** звернення завжди відбувається до найближчого суперкласу.
* Для скасування можливості **перевизначення** методу перед його описом використовується специфікатор **final.**

**Приклад**

Створити клас “два дійсних числа”, які можуть відображати значення висоти та ширини геометричної фігури. На базі цього класу створити два похідні класи: “прямокутник”, з методом, який обчислює площу прямокутника, та “циліндр”, який обчислює площу бокової поверхні циліндра. Для класу “прямокутник” створити похідний клас “паралелепіпед”, який обчислює об’єм паралелепіпеда з використання параметра – значення висоти.

Нижче наводиться текст програми:

package inher;

public class Inher {

public static void main(String[] args) {

areal\_r\_c r\_c=new areal\_r\_c(1,2);

rectangle r = new rectangle(3,4);

cylinder c = new cylinder(2.0f,3.0f);

paral p = new paral(1,2,3);

System.out.println("p1 = " + r\_c.heigth + ", " + r\_c.width);

System.out.println("Площа прямокутника = " + r.arear());

System.out.println("Площа циліндра = " + c.areac(1,3));

System.out.println("Об'єм паралелепіпеда = " + p.areap(1,2,3));

}

}

class areal\_r\_c // Оголошення батьківського класу «два дійсних числа»

{ protected float heigth, width ; // Забезпечення доступу до даних похідного класу

areal\_r\_c (float heigth, float width)

{this.heigth=heigth;

this.width=width;

}

}

// Оголошення похідного класу “прямокутник”

class rectangle extends areal\_r\_c

{

//Реалізація методів похідного класу “прямокутник”

rectangle(float h, float w) // Конструктор

{ super(h,w);

}

double arear() // Обчислення площі

{ return heigth\*width; }

}

// Оголошення похідного класу “циліндр”

class cylinder extends areal\_r\_c

{

float heigth, width ;

// Реалізація методів похідного класу “циліндр”

cylinder(float heigth, float width) // Конструктор

{ super(heigth, width );

}

// Обчислення площі бокової поверхні циліндра

double areac(float heigth, float width) { return 3.14\*width\*heigth; }

}

// Оголошення класу “паралелепіпед”, похідного від класу “прямокутник”

class paral extends rectangle

{

float hpp;

// Конструктор

paral( float h, float w, float hp1)

{

super(h,w);

this.hpp=hp1;

}

// Реалізація методів похідного класу “паралелепіпед”

// Обчислення об'єму паралелепіпеда з використанням

// методу arear батьківського класу

double areap(float heigth, float width,float hpp) { return arear()\*hpp; }

}

### 1.17.2 Інкапсуляція

*Інкапсуляція* - це механізм, який пов'язує код разом з даними і зберігає їх у безпеці, як від зовнішнього впливу, так і від помилкового використання. Основою інкапсуляції в мові Java є клас.

Інкапсуляція дозволяє зберігати в середині класу як властивості, так і методи їх обробки. Використання інкапсуляції дозволяє обмежити доступ до частини властивостей та/або методів класу з інших об’єктів, а також заборонити пряме звернення до властивостей.

**Модифікатори рівня доступу**

Java надає кілька рівнів захисту, що забезпечують можливість тонкого налаштування області видимості даних і методів.

В загальному описі класу замість слова {<доступ>} можуть стояти так звані *модифікатори* та *специфікатори.*

У мові Java є три рівня доступу, що визначаються ключовими словами-*модифікаторами*: *private* (закритий), *public* (відкритий) і *protected* (захищений). Вміст елементів табл.1.17.1 визначає досяжність змінної з вказаним модифікатоом (стовпець) із зазначеного місця (рядок).

Таблиця 1.17.1. Модифікатори рівня доступу

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | private | Модифікатор відсутній | private protected | protected | public |
| той же клас | так | так | так | так | так |
| підклас у тому ж пакеті | ні | так | так | так | так |
| незалежний клас у тому ж пакеті | ні | так | ні | так | так |
| підклас у другому пакеті | ні | ні | так | так | так |
| незалежний клас у другому пакеті | ні | ні | ні | ні | так |

З таблиці видно, що елемент, оголошений *public*, доступний з будь-якого місця. Все, що оголошено *private*, є доступним тільки всередині класу, і ніде більше. Якщо у елемента взагалі не вказано модифікатор рівня доступу, то такий елемент буде видно з підкласів і класів того ж пакета. Саме такий рівень доступу використовується в мові Java за замовчуванням. Якщо ж ви хочете, щоб елемент був доступний ззовні пакета, але тільки підкласам того класу, якому він належить, вам потрібно оголосити такий елемент *protected*. І нарешті, якщо ви хочете, щоб елемент був доступний тільки підкласам, причому незалежно від того, чи знаходяться вони в даному пакеті чи ні - використовуйте комбінацію *private protected*.

Нижче наведено приклад, в якому представлені всі допустимі комбінації модифікаторів рівня доступу. У вихідному коді визначається три класи - Protection, Derived і SamePackage, які належать пакету *р1*. У першому з цих класів визначено п'ять цілих змінних - по одній на кожну з можливих комбінацій рівня доступу. Змінної *n* приписаний рівень доступу за замовчуванням, *n\_pri* - рівень *private*, *n\_pro - protected*, *n\_pripro - private protected* і *n\_pub - public*. У всіх інших класах ми намагаємося використовувати змінні першого класу. Ті рядки коду, які через обмеження доступу привели б до помилок при трансляції, закоментовані за допомогою однорядкових коментарів (//) - зазначено, звідки доступ при такій комбінації модифікаторів був би можливий. Другий клас - *Derived* - є підкласом класу *Protection* і розташований в тому ж пакеті р1. Тому йому доступні всі перераховані змінні за винятком *n\_pri*. Третій клас, *SamePackage*, розташований в тому ж пакеті, але при цьому не є підкласом *Protection*. З цієї причини для нього недоступна не тільки змінна *n\_pri*, але і *n\_pripro*, рівень доступу якої - *private protected*.

**Приклад**

package р1;

public class Protection {

int n = 1;

private int n\_pri = 2;

protected int n\_pro = 3;

private protected int n\_pripro = 4;

public int n\_pub = 5;

public Protection() {

System.out.println("конструктор базового класу");

System.out.println("n = " + n);

System.out.println("n\_pri = " + n\_pri);

System.out.println("n\_pro = " + n\_pro);

System.out.println("n\_pripro = " + n\_pripro);

System.out.println("n\_pub = " + n\_pub);

} }

class Derived extends Protection {

Derived() {

System.out.println("конструктор похідного класуr");

System.out.println("n = " + n);

// тільки в класі

// System.out.println("n\_pri = " + n\_pri);

System.out.println("n\_pro = " + n\_pro);

System.out.println("n\_pripro = " + n\_pripro);

System.out.println("n\_pub = " + n\_pub);

} }

class SamePackage {

SamePackage() {

Protection p = new Protection();

System.out.println("конструктор класу того ж пакету");

System.out.println("n = " + p.n);

// тільки в класі

// System.out.println("n\_pri = " + p.n\_pri);

System.out.println("n\_pro = " + p.n\_pro);

// тільки в класі і підкласі

// System.out.println("n\_pripro = " + p.n\_pripro):

System.out.println("n\_pub = " + p.n\_pub):

} }

**Специфікатори**

Властивості елементів класу і доступ до них також визначається *специфікаторами.*

Специфікатор **final:**

* Якщо використовується перед описом *поля*, то його не можна змінити .
* Якщо використовується перед описом *методу*, то його не можна перевизначити: компілятор Java може вбудувати код *final-метода* в те місце, де він використовується.
* Якщо використовується перед описом *класу*, то він не може бути суперкласом.

Специфікатор **static**:

* Визначає загальні члени класу для всіх примірників (об'єктів) цього класу, тобто опис поля або методу з static каже, що вони належать класу, а не екземпляру класу.
* Кожен екземпляр класу розділяє *static*-змінну (поле) класу, яка знаходиться в одному певному місці в пам'яті.

Обмеження для статичних методів:

* статичні методи можуть викликати безпосередньо тільки статичні методи;
* статичні методи можуть звертатися безпосередньо тільки до статичних даних;
* на статичні методи не можна посилатися через посилання *this* і *super*;
* статичні методи можуть перевантажуватися у нестатичні і навпаки.

**Різниця між *final* та *static* змінними**

Якщо ми об’явили змінну наступним чином:

static final String color = "Black";

це буде гарантувати, кожен раз раз, коли цей клас створюється, це значення поля колору буде "чорним", якщо воно не буде змінене.

public class Speaker {

static String color = "Black";

}

public class Sample {

public static void main(String args[]) {

System.out.println(Speaker.color); //буде виведене як "Black"

Speaker.color = "white";

System.out.println(Speaker.color); // буде виведене як "White"

}}

Змінимо об’явлення на *static final String color = "Black";*

Теперь, якщо змінити кольор, то буде помилка, тому що ключове слово *final* гарантує, что значення цього поля ніколи не зміниться.

public class Speaker {

static final String color = "Black";

}

public class Sample {

public static void main(String args[]) {

System.out.println(Speaker.color); //повинно буде виведено як "Black"

Speaker.color = "white"; //помилка.

System.out.println(Speaker.color); //не буде виконано

}}

### 1.17.3 Поліморфізм

*Поліморфізм* – це подання різних форм одного явища. *Поліморфізм* - це властивість класів, згідно з якою поведінка об'єктів класу може визначатися не на етапі компіляції, а на етапі виконання. Класи, що надають ідентичний інтерфейс, але реалізовані під конкретні специфічні вимоги, мають назву *поліморфних класів*.

В Java *поліморфізм* можна реалізувати через:

* **спадкування** - з перевизначенням параметрів і методів базового класу;
* **абстрактні** класи - шаблони для відмінних реалізацій в різних класах;
* **інтерфейси** - для імплементації класами.

**Перевантаження методів**

* Java підтримує перевантаження методів на базі методів з різною сигнатурою.
* Перевантажені методи мають однакові імена, але різні за кількістю, типом та порядком проходження аргументи, переданих в метод.
* Компілятор не враховує тип значення, що повертається при розрізненні перевантажених методів.

Підключення тіла функції до точки її виклику має назву *зв'язування*. Якщо воно відбувається до початку виконання програми, мова йде про *раннє зв'язування*.

*Пізнє зв'язування* означає, що підключення відбувається під час виконання програми та в об'єктно-орієнтованих мовах залежить від типів об'єктів. Пізнє зв'язування ще називають *динамічним*, або *зв'язуванням часу виконання*. Для реалізації поліморфізму використовується механізм пізнього зв'язування. Воно виявляється в можливості виклику перевизначеного методу підкласу через посилання на суперклас (тобто визначення варіанту методу відбувається під час виконання і грунтується на типі об'єкта).

Через посилання на суперклас можна викликати тільки успадковані методи суперкласу (тобто методи, визначені тільки в підкласі, не можна викликати через об'єктне посилання на суперклас).

У мовах об'єктно-орієнтованого програмування пізнє зв'язування реалізується через механізм віртуальних функцій. *Віртуальна функція* (віртуальний метод, virtual method) - це функція, визначена в базовому класі, та перевизначена у похідних, так, що конкретна реалізація функції для виклику буде визначатися під час виконання програми. Вибір реалізації віртуальної функції залежить від реального (а не оголошеного під час опису) типу об'єкта.

У C++ для позначення віртуальної функції використовують модифікатор *virtual*. У Java усі функції є віртуальними, за винятком конструкторів, статичних (static), фінальних (final) і закритих (private) методів. На відміну від C++, слово virtual не використовується.

Починаючи з Java 5, перед перевизначеними віртуальними методами розміщують директиву @Override, яка дозволяє компілятору здійснити додаткову перевірку синтаксису - відповідність сигнатури нової функції сигнатурі перевизначеної функції базового класу. Використання @Override є бажаним, але не обов'язковим.

Наприклад, завдяки поліморфізму кожен клас Java може визначити свою віртуальну функцію **toString**(), яка буде викликана для автоматичного отримання даних про об'єкт у вигляді рядку.

### 1.17.4 Приведення типів об’єктів

В Java приведення типів об'єктів - це коли посилання на об'єкт може бути приведено до типу іншого посилання на об'єкт. Приведення можливо до свого типу класу або до одного з його підкласів, до суперкласів, або до інтерфейсів:

( ReferenceType )*RelationalExpression*

*Наприклад*,

Object obj = "abracadabra";

String str = (String) obj;

Якщо ви намагаєтеся виконати приведення класу, але об'єкт не цього типу, то має місце помилка типу *ClassCastException*.

Щоб уникнути такої помилки можна перевірити коректність приведення за допомогою оператора *instanceof*. Оператор *instanceof*виробляє *true*, якщо значення *RelationalExpression* не нуль и може бути приведено до *ReferenceType****,*** не викидуючи *ClassCastException*.

Інакше результат буде - *false*

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що таке спадкування? Наведіть приклади.
2. Що таке ієрархія класів?
3. Як оголошується похідний клас?
4. Яка різниця між простим і множинним спадкуванням?
5. Які модифікатори доступу ви знаєте?
6. Який рівен доступу визначається по замовчуванню?
7. Які специфікатори ви знаєте?
8. Що таке поліморфізм? Наведіть приклади.
9. Які способи реалізації поліморфізму ви знаєте?
10. Що таке статичне та динамічне зв'язування?

# Частина 2. Розширені можливості мови Java

## 2.1. Потокове введення / виведення даних

### 2.1.1 Потоки для різних типів даних

*Потоки* є абстракцією, яка пов'язує програму з фізичним пристроєм за допомогою системи введення / виведення Java. Потік являє собою послідовність даних. Джерелом даних і пунктом призначення даних можуть бути все, що містить, генерує або споживає дані (файли на диску, інша программа, периферійний пристрій, мережевий роз'єм, масив і т.д.) Потоки підтримують різні види даних: байти, примітивні типи даних, локалізовані символи і об'єкти. Деякі потоки просто передають дані, інші управляють і перетворюють дані до зручного вигляду.

Java всередині зберігає символи (тип *char*) в 16-бітному наборі символів UCS-2. Зовнішнє джерело даних може зберігати символи в іншому наборі символів (наприклад, US-ASCII, ISO-8859-х, UTF-8, UTF-16 і багато інших), фіксованої довжини 8 біт або 16-біт, або змінної довжини від 1 до 4 байт. Як наслідок, Java потрібно розрізняти введення / виведення на основі байтів для обробки двійкових даних, і введення / виведення на основі символів для обробки тексту.

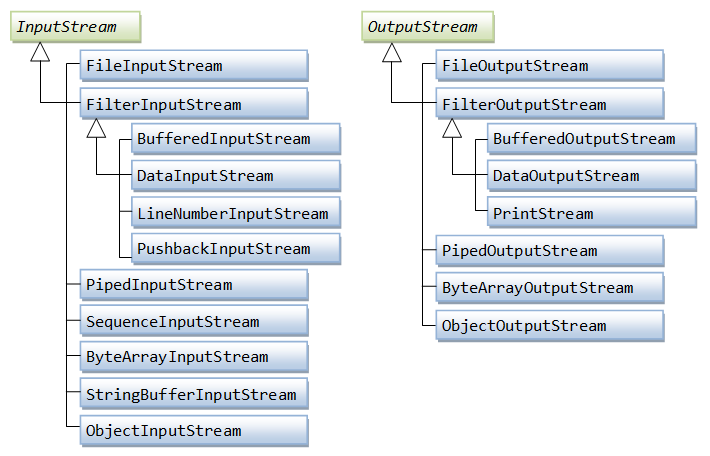
### 2.1.2 Потоки байтового введення/виведення

Таблиця 2.1.1. Методи класу InputStream

|  |  |
| --- | --- |
| Модификатор і тип | Метод і опис |
| abstract int | **read()**  Прочитати наступний байт з потоку введення |
| int | **read(byte[]b)**  Прочитати деяку кількість байтів з потоку вводу та зберегти їх в масиві ***b*** |
| int | **read(byte[]b, int off, int len)**  Прочитати ***len*** байтів з потоку вводу і зберегти їх у масиві ***b***, починаючі зі зміщення ***off*** |
| void | **reset()**  Повертає потік в стан на момент виклику методу *mark()* для даного потоку вводу |
| Long | **skip(long n)**  Пропустити і відкинути ***n*** байтів з потоку вводу |
| void | **close()**  Закрити потік введення і звільнити системні ресурси, звязані з цим потоком |
| void | **mark(int readlimit)**  Позначити (маркувати) поточну позицію в потоці вводу |
| boolean | **markSupported()**  Перевірити, чи підтримує потік введення маркування і скидання |

Таблиця 2.1.2. Методи класу OutputStream

|  |  |
| --- | --- |
| Модификатор і тип | Метод і опис |
| void | **close**()  Закрити потік виводу і звільнити системні ресурси, зв’язані з цим потоком |
| void | **flush**()  Почистити потік виводу і змусити записати дані |
| void | **write**(byte[]b)  Записати b.length байт з масиву b в потік виводу |
| void | **write**(byte[]b, int off, int len)  Записати len байтів з масиву b, починаючи з позиції off , в потік виводу |
| abstract void | **write(int b)**  Записати вказаний байт в потік виводу |



**Стандартні потоки введення/виведення:**

* стандартне введення, доступ через *System.in* (об'єкт типу *InputStream*);
* стандартне виведення, доступ через *System.out* (об'єкт типу *PrintStream*);
* стандартние виведення помилки, доступ через *System.err* (об'єкт типу *PrintStream*).

Клас *PrintStream* використовує внутрішній об'єкт символьного потоку для наслідування багатьох особливостей символьних потоків.

*Приклад*: використання стандартного потоку введення:

public class Main {

public static void main(String[] args) {

InputStream stdIn = System.in;

try {

int value = stdIn.read();

System.out.println(value);

} catch ( IOException e ) {

//...

}

}

}

При виконанні будь-якої операції читання (вводу) може бути викинуто **виключення типу IOException:**

* *IOException* - це загальний клас винятків, вироблених невдалими або перерваними операціями введення / виводу;
* *IOException* сигналізує, що стався якийсь вид виключення введення / виведення.

Exception

IOException

CharConversionException

EOFException

FileNotFoundException

InterruptedIOException

ObjectStreamException

**Зчеплені потоки**

У мові Java передбачений механізм розподілу відповідальності - одні потоки витягають байти з файлів, інші об'єднують байти в дані.

Для використання можливостей різних потоків їх необхідно об'єднувати в зчеплені потоки.

*Наприклад*,

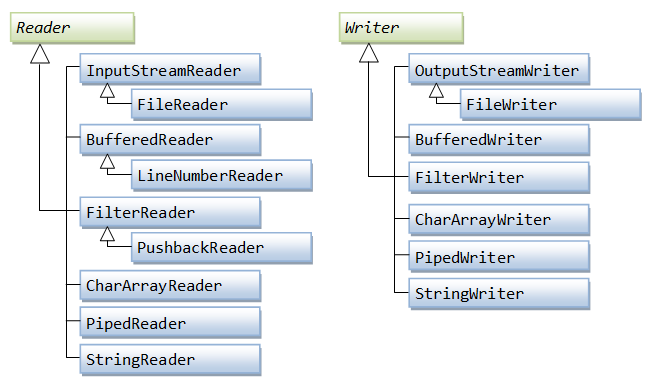
DataInputStream din = new DataInputStream(

new BufferedInputStream(

new FileInputStream("data.txt")));

### 2.1.3 Потоки символьного введення/виведення

Клас *Reader* і *Writer* є абстрактними классами.   Клас Reader визначає методи read () і readLine () для читання символу і рядка символів відповідно. Клас PrintWriter визначає методи print () і println () .



### 2.1.4 Файлові потоки введення/виведення

**Послідовний** **доступ до файлу**

Для роботи з файлами необхідно імпортувати пакет *java.io*. Для створення потоку і відкриття файлу необхідно створити об'єкт одного з класів:

* FileInputStream(String name);
* FileReader(String name);
* FileOutputStream(String name);
* FileWriter(String name).

Класи *FileInputStream, FileOutputStream, FileReader, FileWriter* використовують **послідовний** доступ до вмісту файлу.

*FileInputStream і FileOutputStream* в Java використовують для здійснення вводу та виводу 8-бітних байтів, у той час як потоки символів використовуються для введення та виведення 16-бітного юнікода і найчастіше для цього використовуються класи *FileReader та FileWriter*.

**Довільний доступ до файлу**

Для забезпечення читання/запису даних в довільну позицію файлу використовується клас ***RandomAccessFile*.**

Довільний доступ до файлів працює подібно до великого масиву байтів, що зберігаються в файловій системі.

Потік з довільним доступом до вмісту файла завжди створюється з **режимом доступу**:

* "r" - тільки для читання;
* "rw" - для читання і запису;
* "rws" - для читання і запису с записом на пристрій.

Довільний доступ має курсор, який називається *вказівником* файлу. Операції введення / виведення починаються від вказівника файлу і просувають його до кінця файлу. Поточна позиція вказівника файлу може бути отримана методом *getFilePointer* () і встановлена ​​методом *seek (long pos).*

*Наприклад,*

RandomAccessFile raf = null;

try {

//…

raf = new RandomAccessFile("C:\\test.txt", "rw");

raf.write(new byte[] {0,1,2,3,4,5,6,7,8,9});

raf.seek(5);

raf.write(new byte[] {66,77,88});

raf.seek(0);

byte[] buf = new byte[10];

int n = raf.read(buf,0,10);

System.out.println(Arrays.toString(buf));

raf.close();

//…

Результат в консолі:

[0, 1, 2, 3, 4, 66, 77, 88, 8, 9]

**Клас File**

Клас *File* - це клас, який не призначений для роботи з потоками введення-виведення. Він безпосередньо спілкується з файлами і файловою системою і описує властивості файлу або директорії.

Клас *File* можна використовувати для:

* визначення, що файл або директорія існує;
* створення директорії, якщо вона не існує;
* визначення довжини файлу в байтах;
* перейменування або переміщення файлу;
* видалення файлу;
* визначення, що є шлях до файлу або директорії;
* читання списку файлів в директорії.

*Наприклад,*

File file = new File("c:\\testfile.txt");

boolean isDirectory = file.isDirectory();//*Перевірка чи є директорія*

boolean fileExists = file.exists();//*Перевірка – чи існує файл*

long length = file.length();//*Отримати розмір файлу*

boolean sucMov =

file.renameTo(new File("c:\\newfile.txt"));// *Перейменувати*

boolean successDel = file.delete(); // *Видалити файл*

Об’єкт типу **File** можна використовувати при створенні потоку читання / запису в файл за допомогою *FileInputStream(****file****) або FileReader(****file****):*

File a = new File("D:/myprogramm/java/test");

InputStream a = new FileInputStream(a);

//далі використовується один з методів **read**

У Java каталог представлений файлом, який може містити список інших файлів і каталогів. Використовуючи об'єкт *File*, ви можете створити каталог, продивитися список файлів, представлених у каталозі.

File pathDir = null;

String[] pathsFilesAndDir;

// Створення нового об’єкта file

pathDir = new File("/NetBeans 8.2/Projects/");

// Масив файлів і каталогів

pathsFilesAndDir = pathDir.list();

for(String path:pathsFilesAndDir) { // Виведення списку файлів і каталогів System.out.println(path);

}

Коли ви специфікуєте відносне ім'я файлу, таке як "myfile.txt" або "mydirectory / myfile.txt", то файл шукається відносно каталогу, в якому була запущена віртуальна машина Java.

Ви можете визначити розташування стартового каталогу наступним викликом:

String dir = System.getProperty ( "user.dir");

**Особливості работи з файлами**

* Якщо при створенні потоку введення (читання) файл не існує, то генерується помилка відкриття файлу;
* Якщо при створенні потоку виведення файл не вдається створити (неіснуючий) або відкрити, то генерується помилка відкриття файлу;
* По закінченню роботи з файлом його потрібно закрити для підтвердження всіх проведених змін, а також для звільнення виділених для нього системних ресурсів. Традиційний підхід - виклик методу *close* ().

### 2.1.5 Буферезоване введення/виведення символів та рядків

Клас *BufferedWriter* записує текст у потік, але попередньо накопичує символи у буфері, тим самим зменшуючи кількість звернень до фізичного носія при записуванні даних.

Клас *BufferedWriter* має такі конструктори:

BufferedWriter(Writer out)

BufferedWriter(Writer out, int sz)

Як параметр він приймає потік виведення, в який треба здійснити запис. Другий параметр вказує на розмір буфера.

Приклад.

BufferedWriter bw = new BufferedWriter(new FileWriter("notes4.txt")))

String text = "Hello  World!\nHey! Teachers!.";

            bw.write(text);

*BufferedReader* може читати дані не тільки по одному символу (хоча метод *read* () для цих цілей у нього теж є), а ще і цілими рядками за допомогою методу *readLine* ().

Приклад читання файлу посимвольно:

BufferedReader br = new BufferedReader (new FileReader("notes4.txt")))

        {

           // читання посимвольно

            int c;

            while((c=br.read())!=-1){

                System.out.print((char)c);

            }

        }

Приклад читання файлу по рядкам:

BufferedReader br = new BufferedReader(new FileReader("notes4.txt")))

{

    //читання по рядкам

    String s;

    while((s=br.readLine())!=null){

        System.out.println(s);

    }

### 2.1.6 Потоки для роботи з примітивними типами

Потоки даних підтримують бінарне введення / виведення значень примітивних типів даних. Ці потоки даних реалізують або інтерфейс *DataInput* або інтерфейс *DataOutput*. Найбільш широко використовуються реалізаціїї цих інтерфейсів: *DataInputStream і DataOutputStream*.

Приклад записування в файл даних різних типів:

try {

DataOutputStream out = new DataOutputStream(

new BufferedOutputStream(

new FileOutputStream("dataout.dat")));

out.writeShort(1200);

out.writeInt(50000);

out.writeLong(12345678L);

out.writeDouble(55.66);

out.writeBoolean(true);

out.writeUTF("Hello!!!");

out.flush();

} catch (IOException ex) {

//...

}  
//...

Тут **out.flush() -** примусовий запис в файл (з потоку виведення в пункт призначення) .

Приклад, читання з файлу даних різних типів:

try {

DataInputStream in = new DataInputStream(

new BufferedInputStream(

new FileInputStream("dataout.dat")));

System.out.println("short: " + in.readShort());

System.out.println("int: " + in.readInt());

System.out.println("long: " + in.readLong());

System.out.println("double: " + in.readDouble());

System.out.println("boolean: " + in.readBoolean());

System.out.println("String UTF: " + in.readUTF());

System.out.println();

} catch (IOException ex) {

//...

}

//...

Результат:

*short: 1200*

*int: 50000*

*long: 12345678*

*double: 55.66*

*boolean: true*

*String UTF: Hello!!!*

**Читання файлу з клавіатури за допомогою** InputStreamReader

*System.in* - це об'єкт класу *InputStream*, тобто вхідний потік, і він прив'язаний до системного пристрою введення даних - клавіатури.

Треба передати потік *System.in* об'єкту *InputStreamReader,* який і читає байти з клавіатури.

InputStreamReader reader = new InputStreamReader(System.in);

while (true) {

int x = reader.read();

System.out.println(x);

}

В буферизованому варіанті це буде іиглядати таким чином:

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

**Читання за допомогою** *Scanner*

Незважаючи на те, що і *BufferedReader* і *Scanner* (з'явився в 5-й версії) можуть читати файл або введення користувача з командного рядка, між ними є деякі суттєві відмінності. Однією з головних відмінностей класу *BufferedReader* від *Scanner* є те, що *BufferedReader* призначений для простого зчитування текстових даних, тоді як клас *Scanner* призначений для читання та парсингу (розкладання) текстових даних на примітивні типи Java, такі як *int, short, float, double та long*.

Ось 5 ключових відмінностей між класом Scanner та BufferedReader Java API:

1. Сканер набагато потужніша утиліта, ніж BufferedReader. Він може крім читання String виконувати **парсінг** введених користувачем даних та прочитати int, short, float, double та long. BufferedReader може читати String-дані.

2. BuffredReader має значно **більший буфер** (8 КБ), ніж сканер (1 КБ), а це означає, що якщо ви читаєте довгі рядки з файлу, ви повинні використовувати BufferedReader, але для коротких введень та введення інших даних, ніж String, ви можете використовувати клас Scanner.

3. BufferedReader старший за сканер. Він присутній у Java ще в JDK 1.1, але Scanner представлений лише у версії JDK 1.5.

4. Сканер використовує **регулярні вирази** для читання та розбору тексту. Він може приймати спеціальні користувальницькі роздільники і розкладати текст на примітивний типи даних, наприклад int, long, short, float або double з використанням методів nextInt (), nextLong (), nextShort (), nextFloat () та nextDouble (), тоді як BufferedReader може читати та зберігати тільки рядки String методом readLine ().

5. Інша основна відмінність класу BufferedReader від Scanner полягає в тому, що BufferedReader **синхронізований**, а Scanner - ні. Це означає, що ви не можете розподілити Scanner між декількома потоками, але ви можете це зробити з об'єктом BufferedReader.

Ця синхронізація також робить BufferedReader трохи **повільнішим** в одно-потоковому середовищі порівняно зі сканером, але різниця у швидкості компенсується, за рахунок того, що Scanner працює з регулярними виразами, і це в підсумку робить BufferedReader швидшим для читання String.

Приклад.

Scanner userFile = new Scanner (System.in);

System.out.println("What is the file path ? ");

if (userFile.hasNextLine()){

try{

ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream(userFile.nextLine()));

FileOutputStream fos = new FileOutputStream(userFile);

ObjectOutputStream os = new ObjectOutputStream(fos);

os.writeObject(applicant);//запис у файл об’єкта

os.flush();

os.close();

}

catch (FileNotFoundException e)

{

e.printStackTrace();

}

catch (IOException e)

{

e.printStackTrace();

}

//далі читаємо об’єкт з тільки но записаного файлу

try {

System.out.println("Enter the filename for object reading :");

ObjectInputStream is = new ObjectInputStream(new FileInputStream(userFile.nextLine()));

Applicant a = (Applicant) is.readObject();

System.out.println("read surname " + a.Surname);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

### 2.1.7 Об’єктні потоки

Java для передачі між програмою і джерелом / пунктом призначення даних у вигляді об'єктів використовує спеціальні об'єктні потоки.

Для запису в пункт призначення необхідно:

а) створити об'єкт класу *ObjectOutputStream*;

б) викликати метод *writeObject* ().

Для читання з джерела даних необхідно:

а) створити об'єкт класу *ObjectInputStream*;

б) викликати метод *readObject* ().

*Наприклад,*

try (ObjectOutputStream out = new ObjectOutputStream(  
 new FileOutputStream("**data.ser**"))) {  
 Integer A = new Integer(55);  
 Float F = new Float(5.5f);  
 out.writeObject(A);  
 out.writeObject(F);

} catch (FileNotFoundException e) {  
 System.out.println("File not found!");

} catch (IOException e) {  
 e.printStackTrace();

}

Розширення у назві файлу data.**ser** – у відповідності до угоди про файли, що містить об'єкти (від слова *“serialization”).*

### 2.1.8 Звільнення ресурсів

По завершенні роботи з ресурсом при введені-виведенні його треба звільняти, а в разі файлу - закривати. У пакеті java.lang для цього визначений метод *close* (). Перед ним можно також використовувати метод flush() для остаточного звільнення буферу даних.

Ресурсами можуть бути екземпляри класів, які реалізують інтерфейс *AutoCloseable*, визначений у пакеті java.lang (інтерфейс містить тільки один метод *close* (),

З Java 7 введений новий механізм обробки виключень під назвою *«try з ресурсами» (try-with-resources)*. Цей механізм обробки виключень орієнтований на обробку винятків, коли ви використовуєте ресурси, які повинні бути закриті після використання, наприклад, InputStream, OutputStream і т.д.

Синтаксис оператора *try-with-resources*:

**try** (<специфікация ресурса>) {

<код, який використовує ресурс>

}

<специфікація ресурса> - це оператор, який оголошує і ініціалізує ресурс, такий як файловий потік даних.

По завершенні блоку *try* ресурс автоматично звільняється, а в разі файлу - він закривається.

* Ресурсами можуть бути екземпляри класів, які реалізують інтерфейс *AutoCloseable*, визначений у пакеті java.lang (інтерфейс містить тільки один метод *close* (), який і викликається по завершенню блоку try);
* Ресурс, оголошений в блоці try, неявно є **final** (тобто не можна змінити ресурс після того, як його було створено; область видимості ресурсу обмежується блоком **try**);
* Можна керувати кількома ресурсами, перерахувавши їх в оголошенні через крапку з комою (тобто один і той блок try може використовуватися для контролю декілька ресурсів);
* Вторинні виключення приховуються і додаються в список пригнічених (можна переглянути методом getSuppressed (), визначеному в класі Throwable).

**Приклад**закриття файлу до появи оператора ***try-з-ресурсами***

static String readFirstLineFromFileWithFinallyBlock(String path)                                        throws  IOException {   
    BufferedReader br = new BufferedReader( new FileReader(path));   
    try {

return  br.readLine();

}

finally {

     if (br != null)  br.close();

      }

}

**Приклад** закриття файлу з оператором ***try-с-ресурсами***

static String readFirstLineFromFile(String path)

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

}

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Які види даних підтримують потоки введення/виведення?
2. Які існують методи для байтового введення/виведення?
3. Які існують методи для символьного введення/виведення?
4. Які існують методи для файлового введення/виведення?
5. Які існують об'єктні потоки введення/виведення?
6. Які існують засоби новий обробки виключень при введенні/виведенні?

## 2.2. Вкладені класи

### 2.2.1 Типи вкладених класів

*Вкладений клас (Nested Class)* – це клас, визначений в середині іншого класу. Основні типии вкладених класів:

* *Статичний* вкладений класс;
* *Внутрішній* вкладений класс, який в свою чергу може бути *локальним* та *анонімним*.

Властивості вкладеного класу:

* вкладений клас є членом свого зовнішнього класу;
* вкладений клас може бути - **private**, **public**, **protected** або **package-private**;
* **Внутрішній** клас має прямий доступ до члена свого зовнішнього класу, навіть якщо він був оголошений як закритий;
* **Статичний** вкладений клас не має прямого доступу до членів свого зовнішнього класу, крім статичних.

### 2.2.2 Статичний клас

*Статичний клас* – клас, зі специфікатором *static*.

**public class** OuterClass {

*//...*

**static class** StaticNestedClass {

*// ...*

}

}

Статичний вкладений клас взаємодіє з членами об’єкту свого зовнішнього класу (і інших класів) так само, як і будь-який інший клас верхнього рівня, тобто це клас верхнього рівня, який був вкладений в інший клас верхнього рівня для зручності.

Як і звичайні статичні методи класу, статичний вкладений клас не може звернутися безпосередньо до змінних або методів класу, в який він вкладений.

Статичний вкладений клас може використовувати змінні і методи екземпляра включаючого класу тільки через посилання на об'єкт. Статичний вкладений клас може звернутися безпосередньо тільки до статичних змінних і методів свого включаючого класу.

**Приклад**

class Ship {

public int x=10;

private static int y =10;

public static class Boat {

public void test() {

x = 20; // Звернення до поля екземпляру - помилка

Ship sh = new Ship();

sh.x = 20;// Звернення до поля екземпляру - вірно

y = 20;// Звернення до поля класу - вірно

}

}

}

Статичні вкладені класи, як класи верхнього рівня, доступні з використанням імені «верхнього» класу:

**OuterClass.StaticNestedClass**

Наприклад, щоб створити об'єкт статичного вкладеного класу, використовуйте наступний синтаксис:

***OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();***

### 2.2.2 Внутрішній клас

*Внутрішній клас* – це не статичний вкладений клас.

public class OuterClass {

//...

class InnerClass {

// ...

}

}

*Внутрішні класи* використовуються тоді, коли їх об'єкти не можуть існувати або не потрібні поза тілом зовнішнього класу.

Наприклад, зовнішній об'єкт - це екран, який представляє собою матрицю пікселів (точок), внутрішній клас - це клас, що описує один піксель.

Як і методи і змінні екземпляра, внутрішній клас пов'язаний з екземпляром свого «верхнього» класу. Оскільки внутрішній клас пов'язаний з екземпляром, то він не може визначати в собі статичні члени. Примірник внутрішнього класу має прямий доступ до полів і методів екземпляра свого зовнішнього классу. Зовнішній клас для використання внутрішнього класу повинен створювати його примірник. Примірник зовнішнього класу може бути пов'язаний з декількома екземплярами внутрішнього классу. Примірник внутрішнього класу пов'язаний тільки з одним екземпляром зовнішнього класу.

**Приклад**

class Ship {

private int x=10;

Engine eng = new Engine();

protected class Engine {

public void test() {

// Звернення до поля екземпляра зовнішнього класу безпосередньо

x = 20;

}

}

public void testing() {

// Звернення до методу примірника внутрішнього класу через об'єкт

eng.test();

}

}

Якщо у внутрішньому класі є члени або локальні змінні з такими ж назвами, що і в його зовнішньому класі, тоді вони перекривають область видимості членів зовнішнього класу.

Для доступу до прихованих членів зовнішнього класу внутрішній клас використовує специфікацію:

**OuterClass.this**

Для створення екземпляра внутрішнього класу поза тілом зовнішнього класу, потрібно спочатку створити екземпляр зовнішнього класу, а потім створити екземпляр внутрішнього класу в межах об'єкта зовнішнього по такому синтаксису:

***OuterClass outerObject = new OuterClass();***

***OuterClass.InnerClass innerObject = outerObject.new InnerClass ();***

Наприклад,

Ship ship = new Ship();

Ship.Engine engine = ship.new Engine();

### 2.2.3 Локальний клас

*Локальні класи*– це класи, визначені в блоці, де типовим блоком є ​​тіло методу.

class Ship {

// …..

void work() {

class LocalClass {

}

// ….

}

}

**Обмеження локальних класів**:

* Їх видно тільки в межах блоку, в якому оголошено;
* Вони не можуть бути оголошені як *private, public, protected або static*;
* Вони не можуть мати в собі статичних оголошень (полів, методів, класів), винятком є ​​поля (static final);
* Локальний клас має доступ до членів свого зовнішнього класу;
* Локальний клас має доступ до локальних змінних зовнішнього блоку, які оголошені як *final* (Java 7 і нижче);

Локальний клас може отримати доступ до локальних змінних або параметрів зовнішнього блоку, які є ефективним *final* (змінна або параметр, значення якого ніколи не змінюється після ініціалізації).

### 2.2.4 Анонімний локальний клас

*Анонімний* клас - це клас без імені (якщо необхідно створити тільки один об'єкт класу, то такому класу немає сенсу привласнювати ім’я).

Анонімні класи є частиною виразу - оголошення такого класу виконується одночасно зі створенням його об'єкту.

**new** <parentClass/interface> (<arguments>) {

<тіло анонімного класу>};

**Приклад**

interface MyTest {

void test();

}

class Ship {

void doJob() {

MyTest tst = new MyTest() {

public void test() {

System.out.print("TEST");

}

};

tst.test();

}

}

**Обмеження анонімних класів**:

* Їх видно тільки в межах виразу, в якому оголошено;
* Вони не можуть бути оголошені як *private*, *public*, *protected* або *static*;
* Вони не можуть мати в собі статичних оголошень (полів, методів, блоків ініціалізації), винятком є ​​поля (*static final*);
* Вони можуть мати додаткові поля, методи і блоки ініціалізації примірника;
* Вони не можуть містити оголошення конструкторів.

Анонімний клас має доступ до членів свого зовнішнього класу безпосередньо.

**Приклад**

class Ship {

private int x = 10;

void doJob() {

final int y = 20;

MyTest tst = new MyTest() {

private int z = 10;

{

System.out.print("Init block");

}

public void test(){

System.out.print(x + " " + z + " " + y);

}

};

tst.test();

}

}

**Інтерфейс Comparator і анонімний клас**

Інтерфейс Comparator дає можливість визначити правило зіставлення даних для їх упорядкування. Цей інтерфейс має один абстрактний метод *compare* () для порівняння двох об'єктів.

Для реалізації інтерфейсу Comparator зручно використовувати анонімний клас:

* Реалізації підлягає тільки один метод;
* Необхідно мати один об'єкт цього типу;
* Код реалізації буде розміщуватися поруч з його використанням.

**Приклад**

String[] arr = {"java", "с#", "fortran", "ada", "modula"};

Arrays.sort(arr, new Comparator() {

@Override

public int compare(Object arg0, Object arg1) {

String str1 = (Srting) arg0;

String str2 = (Srting) arg1;

return str1.compareTo(str2) \* (-1);

}

});

System.out.println(Arrays.toString(arr));

**Результат**

[modula, java, fortran, c#, ada]

### 2.2.5 Використання вкладених класів

Причини використання вкладених класів такі:

* Логічне групування класів, які використовуються тільки в одному місці ( Якщо клас корисний тільки для одного іншого класу, то логічно впровадити його в цьому класі і тримати їх разом).
* Посилення інкапсуляції ( Припустимо є два класи верхнього рівня: A і B, де класу B необхідний доступ до членів класу A і які, в іншому випадку, були б оголошені закритими. Приховуючи клас B в межах класу A маємо, що члени класу A можуть бути оголошені private і клас B може отримати до них доступ. Крім того, сам клас B може бути прихований від зовнішнього світу.)
* Спрощення сприйняття коду (коди вкладених класів розміщують ближче до місця, де вони використовуються).

Таблиця 2.2.5. Використання вкладених класів

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Звязок з обєктом Outer** | **Місце обявлення** | **Видимість** |
| Статичний вкладений клас | Ні | Як член іншого класу | Залежить від модифікатора доступу |
| Внутрішній клас | Так | Як член іншого класу | Залежить від модифікатора доступу |
| Локальний клас | Так (якщо визначений не в статичному методі) | В середені методу | Від точки визначення і до кінця методу |
| Анонімний клас | Так (якщо визначений не в статичному методі) | В виразі при визначенні об’єкту | Ні |

### 2.2.6 Вкладені класи і віртуальна машина

*У JVM* немає поняття вкладений клас. Компілятор генерує окремий файл .**class** для кожного класу (як зовнішнього так і внутрішніх). В кінцевому підсумку, буде створено більшу кількість файлів типу **class** з відносно меншою кількістю файлів вихідного коду.

Якщо вкладений клас глобально доступний, то компілятор перетворює його ім'я таким чином:

**nameOuterClass $ nameNestedClass**

Якщо вкладений клас глобально не доступний (закритий), то компілятор може перетворити його ім'я таким чином:

**nameOuterClass $ digit**

Наприклад, внутрішній клас **pkg.Foo.Bar** отримує під час виконання імя **pkg.Foo$Bar** або, можливо, **pkg.Foo$23**, якщо **Bar** є закритим.

Внутрішні класи прив'язані до конкретного об'єкта зовнішнього класу. Для досягнення цього компілятор додає у внутрішній клас поле типу **final** (зазвичай з ім'ям **this$0**), яке містить посилання на зовнішній клас. При цьому в кожен конструктор додається відповідний параметр.

Внутрішній клас і його осяжний клас можуть містити закриті члени, до яких вони мають прямий доступ.

З точки зору Java-машини звертатися до закритих членам іншого класу неприпустимо. Щоб обійти це обмеження, компілятори створюють спеціальні методи доступу:

* Всі методи доступу статичні;
* Мають модифікатор package-private;
* Називаються, починаючи з «access $» (наприклад, access $ 0, access $ 1 і т.д.);
* Методи доступу для читання полів мають один параметр - об'єкт;
* Методи доступу для запису полів - два параметра (об'єкт і нове значення).

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Які існують типи вкладених класів? Які причини їх використання?
2. Які особливості статичного вкладеного класу?
3. Коли використовуються внутрішні класи?
4. Де визначаються *локальні класи?* Які вони мають обмеження?
5. Коли доцільно використовувати анонімні класи? Які вони мають обмеження?

## 2.3. Узагальнення

### 2.3.1 Поняття узагальнення

*Узагальнення (generic)* - синтаксична конструкція Java 5, що являє собою параметр класу або функції, який містить додаткову інформацію про тип елементів та інших даних. Цей параметр беруть у кутові дужки.

Узагальнення надають можливість створення та використання структур даних, безпечних з точки зору типів. Класи, опис яких містить такий параметр, мають назву *узагальнених*.

Наступні два приклади показують різницю у роботі з ArrayList() без використання узагальнення і з його використанням.

Приклад 1.

//До Java 5

public static void main(String[] arg) {

List animals= new ArrayList();

animals.add(“cat”);

animals.add(“dog”); // 1

animals.add(“frog”);

String animal = (String) animals.get(1);

}

Якщо ми не зробимо доункастинг, тобто не вставимо *(String)*, то отримаємо помилку:

Exception in thread "main" java.lang.ClassCastException:

java.lang.String cannot be cast to java.lang.Integer

Це тому що у колекції *ArrayList* зберігаються дані типу *Object* і нам треба виконати приведення типів до нижчого типу по ієрархії, тобто до *String*, що робити не зручно.

Крім об’єктів типу *String* в *ArrayList* можна вставити і об’єкти інших типів (наприклад, якийсь об`єкт свого створеного класу). Завдяки тому, що в *ArrayList* зберігаються об’єкти верхнього класу *Object*, то помилки не буде, але при операції *get* без приведення ми знову отримаємо помилку.

Щоб кожен раз уникати такого незручного приведення типів, починаючі **з** Java 5було введено *параметризацію (generics)*. Приклад її показано на наступному прикладі, який є модифікацією попереднього.

Приклад 2.

//Після Java 5

**List<String> animals2= new ArrayList <String>();**

animals2.add(“cat”);

animals2.add(“dog”); // 1

animals2.add(“frog”);

String animal = animals2.get(1);

Тепер не треба робити доункастинг, тобто не треба вставляти *(String)* і помилки не буде, бо ми явно вказали, що в цій колекції зберігаються тільки дані типу *String*.

Тут ми бачимо певну перевагу **Generics – не треба слідкувати за перетвореннями типів**

А з появоюJava **7** тип параметрізації можна вказувати тільки в одному місці (ліворуч):

List<String> animals2= new ArrayList <>();

Тепер уже, на відміну від Прикладу 1, ми не зможемо в *ArrayList*вставити ще щось, крім *String*. Наприклад, наступний рядок зі вставленням свого об’єкта ourAnimal дасть нам помилку

animals2.add(ourAnimal)

Тобто в цьому - ще одна перевага *Generics – контроль за тим, щоб ми не вводили “неправильні” дані***.** Компілятор перевіряє колекції об’єктів, щоб всі її елементи були одного типу.

Таким чином, переваги узагальнень в тому, що не треба явно виконувати приведення типів; також забезпечується можливість створення однорідних колекцій, які перевіряються компілятором.

Вище ми використовували існуючий у Java клас-колекцію *ArrayList,* який після Java 5 дозволяє застосовували узагальнення (тобто він став узагальненим).

При створенні свого узагальненого класу треба використовувати наступний формат:

class nameClass <T1, T2, ..., Tn> { ... }

Під час створення об'єктаузагальненого класу замість **Т у кутових дужках** вказують імена реальних типів. При цьому можна використовувати **тільки типи-посилання.**

Приклад опису класу з застосуванням узагальнень.

public class ArrayList1<T> {

public T getElem(int i) { ……. }

public void addElem(T ob) { ……. }

………………….

private T[] elem;

}

Тут **T** можна використовувати **будь-де в класі**. Тип змінної T може бути **будь-яким не примітивним типом**: типом будь-якого класу або інтерфейсу.

Нижче показано приклад створення об’єкту узагальненого класу:

nameClass **<Type>** nameVar **= new** nameClass **<Type> ();**

де **nameClass** - і’мя узагальненого класу;

**Type** - конкретний тип, що називається “Аргумент типу".

Приклад.

У попередніх прикладах ми працювали зі стандартною колекцією – динамічним масивом. Тепер створимо свій клас *Pair* (пара), який буде зберігати пару об'єктів будь-якого одного типу. Нехай він містить два посилання на клас Object, щоб досягти універсальності:

public class Pair {

Object first, second;

 //конструктор

public Pair(Object first, Object second) {

this.first = first;

this.second = second;

}

}

Оскільки клас *Object* є базовим для усіх типів-посилань, то наш клас *Pair* можна, наприклад, застосувати для зберігання пари рядків:

Pair p = new Pair("Прізвище", "Ім\'я");

Але, як показав нам приклад 1 тепер при отриманні значення поля з класу треба слідкувати за приведенням типів.

А якщо тепер замість*Object* ми поставимо Pair<T>, то це дає змогу не приводити типи**:**

public class Pair<T> {

T first, second;

  public Pair(T first, T second) { //це конструктор

this.first = first;

this.second = second;

}

public static void main(String[] args) {

//Створення об’єкту узагальненого класу з типом String

Pair<String> p = new Pair<String>("Прізвище", "Ім\'я");

String s = p.first; // Отримуємо рядок без приведення типів

//Створення ще одного об’єкту узагальненого класу з типом Integer

Pair<Integer> p1 = new Pair<Integer>(1, 2);

int i = p1.second; // Отримуємо ціле значення без приведення типів

}

}

Тобто ми створили класи з параметрами різних типів

Pair<String> p = new Pair<String>("Прізвище", "Ім\'я");

Pair<Integer> p1 = new Pair<>(1, 2);

Як видно з прикладу, Java 7 дозволяє не повторювати фактичний параметр узагальнення після імені конструктора.

Тип даних з параметром у кутових дужках (наприклад, Pair<String>) має назву  ***параметризованого типу*.**

Створимо об’єкт класу *ArrayList1* (див.попередній приклад) і скористаємось його методами:

public class ArrayList1 <T> {

public T getElem(int i) { ……. }

public void addElem(T ob) { ……. }

……………….

private T[] elem;

}

public static void main(String[] arg) {

ArrayList1<Integer> set = new ArrayList1<Integer>();

set.addElem(Integer.valueOf(10));

Integer xx = set.getElem(0);

**set.addElem("Java"); //Помилка компіляції**

Integer yy = set.getElem(1);

}

Тут об’єкт set повинен вміщувати тільки змінні типу Integer.

Узагальнення по зовнішньому представленню і використанню аналогічні шаблонам C++.

Існують домовленності по позначенню "Type Parameter“ (для позначення параметра використовується одна велика літера):

<Е> - для елемента набору даних;

<T> - для типу;

<K, V> - для ключа і значення;

<N> - для числа;

S, U, V, etc. - для 2, 3, 4 параметрів типу.

Окрім узагальнених класів, можна створювати *узагальнені інтерфейси.* Узагальнений клас або інтерфейc може мати декілька параметрів ("Type Parameter“):

interface DataAccess<K, V> {

public K getKey();

public V getValue();

}

Приклад реалізації узагальненого інтерфейсу:

class Data<K, V> implements DataAccess <K, V> {

private K key;

private V value;

public Data(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() { return key; }

public V getValue() { return value; }

}

Параметр може бути використаний в описах методів, оголошених в інтерфейсі. Java також дозволяє створювати *узагальнені методи* всередині як узагальнених, так і звичайних (неузагальнених) класів.

### 2.3.2 Узагальнений метод

Узагальненими можуть бути:

* Методи класу (статичні);
* Методи екземпляру (не статичні);
* Конструктори.

В цьому синтаксису методу (на відміну від синтаксису класу*)* ***<T>*** *ми* повинні ставити***перед***змінною, яку метод повертає

*Синтаксис*:

***access <T > returnType*** *methodName* ***(T arg) { }***

де **access –** модифікатор доступу, **Т** - параметр типу

Синтаксис виклику статичного методу класу:

***nameClass.<Type>*** *methodName* ***(listArguments);***

Синтаксис виклику методу об’єкта:

***instance.<Type>*** *methodName* ***(listArguments);***

**Приклад** 1 узагальненого методу

public class ArrayPrinter {

  public static<T> void printArray(T[] a) {

for (T x : a)

System.out.print(x + "\t");

System.out.println();

}

//Двічі викликається метод printArray з параметрами різних типів

public static void main(String[] args) {

String[] as = {"First", "Second", "Third"};

printArray(as);

Integer[] ai = {1, 2, 4, 8};

printArray(ai);

}

}

**Приклад** 2 узагальненого статичного методу

class GenericMethod {

public static <T> byte asByte(T num) {

if (num instanceof Number) {  
 return ((Number)num).byteValue();  
 } else {  
 return 0;

}

}

}

Вбудований у Java метод *byteValue*() існує тільки для типу **Number**, тому для запобігання помилки використовується оператор **instanceof.**

Приклад виклику статичного методу з попереднього класу:

class Main {

public static void main(String [] args) {

System.out.println(

GenericMethod.<Integer>asByte(new Integer(7)));

System.out.println(

GenericMethod.<Float> asByte(new Float(7.F)));

System.out.println(

GenericMethod.<Character> asByte(new Character('7')));

**Результат на консолі:**

**7**

**7**

**0**

### 2.3.3 Обмеження на параметр типу

Для запобігання помилок використовуються обмеження (границі) і використовуються ключові слова **extends** і **super** :

* *Обмеження зверху* (<**T** **extends Type**>) – тут **T** може бути тільки типу **Туре** або його підтипом;
* *Обмеження знизу*(<**T** **super Type**>) – тут **T** може бути тільки типу **Туре** або його супертипом;

Приклад:

//в цьому класі можна використовувати всі числові типи

class Div <T extends Number> {  
 private T x, y;  
 Div(T a, T b) {  
 x = a;  
 y = b;  
 }  
 public int ddiv() {  
 return (int)(x.doubleValue() / y.doubleValue());  
 }  
}

//Немає неообхідності явно перевіряти тип **Number**, компілятор зробе це сам.

class Main {

public static void main(String [] args) {

double a = Math.random()\*100,  
b = Math.random()\*100;  
short aa = (short)(Math.random()\*100),  
bb = (short)(Math.random()\*100);  
Div <Double> obj1 = new Div <>(a, b);//використовувати Double дозволено  
 System.out.println(a + " : " + b + " = " + obj1.ddiv());  
Div <Short> obj2 = new Div <>(aa, bb); // використовувати Short дозволено   
 System.out.println(aa + " : " + bb + " = " + obj2.ddiv());

Div <Character> obj3 = new Div<>('9', '2'); // використовуватиChar не дозволено

// *Тут повинен бути тип Number. Буде помилка компіляції*

}

}

На параметр типу може бути накладено декілька обмежень (розділених за допомогою **&**):

**<T extends Type1 & Type2 & Type3 & …>**

де *Type1* – суперклас або інтерфейс; *Type2* і *Type3* – інтерфейси.

**Приклад**

class A { …. }

interface B { …. }

interface C { …. }

class D <T extends A & B & C> { …. }

Т – це підтип всіх типів, які перераховані у границх.

### 2.3.4 Обмеження на узагальнення

Java накладає певні обмеження на узагальнення:

* Не можна створити узагальнений тип з примітивним типом;

DataAccess <int, double> data = new Data<>(10, 5.5); //помилка

* Не можна створити екземпляр параметра типу;

public static <T> T testInst() {

T elem = new T(); //помилка

return elem;

* Не можна оголосити статичне поле, тип якого параметр типу;

public class MobileDevice<T> {

private static T os; //помилка

// ...

}

* Не можна використовувати приведення або *instanceof* з параметром типу;

public static <E> void test(ArrayList<E> array) {

if (array instanceof ArrayList<Integer>) { //помилка

// ...

}

}

* Не можна створити масив узагальненого типу;

DataAccess <Integer, Double>[] data = new Data<Integer, Double>[5]; //помилка

* Не можна перевантажити методи, де параметризовані типи кожного перевантаження підчищаються до одного і того ж не обробленого типу;

class Example {

public void test(Data<Integer, String> a) { //… }//помилка

public void test(Data<String, Double> a) { //... }//помилка

}

* Не можна створити, спіймати або кинути об'єкти виключень узагальненого типу. Узагальнений клас не може розширювати клас **Throwable**  напряму чи не напряму (через своїх нащадків).

class MathException<T> extends Exception { ... }//помилка

### 2.3.5 Узагальнення і спадкування

**ArrayList<Integer>** не є підтипом **ArrayList<Number>**, навіть якщо **Integer** є підтипом **Number.**

*Приклад*,

ArrayList <Number> mb1 = new ArrayList <>();

ArrayList <Integer> mb2 = new ArrayList <>();

**mb1 = mb2; //Помилка компіляції**

### 2.3.6 Підстановлювальний тип

*Підстановлювальний тип*(*wildcard character)* - позначається через <?> і означає невідомий тип (тобто використовується для опису узагальненого класу, інтерфейсу або методу з невідомим параметром типу). Підстановлювальний тип корисний *для методу, який може бути реалізований за допомогою функціональних можливостей, що надаються в класі Object, а також коли код, використовує методи узагальненого класу, що не залежать від параметра* ***типу.***

Різниця між Т і ? в наступному:

* **Символ** **підстановки** ? (*wildcard character)*: замість нього можна підставляти **будь-який тип**
* **<T>** можна використовувати в різних місцях коду, але він буде відповідати **одному** типу.

*Наприклад,* доповнимо опис класу Div з попереднього прикладу методом:

boolean equalsDiv(Div<T> data) {

if (this.ddiv() == data.ddiv())

return true;

else return false;

}

//Цей метод може викликатися без помилки тільки за умови збігу типів //аргументів для обох об`єктів класу Div.

Div<Integer> d\_1 = new Div<>(10, 20);

Div<Double> d\_2 = new Div<>(5.5, 1.1);

boolean res = d\_1.equalsDiv(**d\_2); //Помилка компіляції**

Для зняття цього обмеження треба використовувати **символ** **підстановки** ? (*wildcard character)*: за мість нього можна підставляти будь-який тип

boolean equalsDiv(Div <?> data) { // data - невідомого типу

if (this.ddiv() == data.ddiv())

return true;

else return false;

}

//……

Div<Integer> d\_1 = new Div<>(10, 20);

Div<Double> d\_2 = new Div<>(5.5, 1.1);

boolean res = d\_1.equalsDiv(d\_2);

Виведення результату на консолі:

61.95198795487858 :12.204340305758842 = 2

26 : 14 = 1

false

**Обмеження для підстановлювального типу**

Підстановлювальний тип можна використовувати з межами, щоб послабити обмеження на змінну:

* **<? extends SuperClass> -** верхняя межа;
* **<? super SubClass>** - нижня межа.

*Наприклад,* **List <Number>** є більш обмеженим, ніж **List <? extends Number>**, оскільки перший збігається тільки зі списком типу Number, в той час як другий - збігається зі списком типу Number або будь-якого з його підкласів (Integer, Double, Float і т.д.).

**Верхня межа:**

DataAccess <? extends Number, ? > ddd = new Data<Integer, Character>(12, 'k');

ddd = new Data<Number, String>(99, "bbb");

ddd = new Data<Double, String>(9.99, "bbb");

ddd = new **Data<int[], Long>(**new int[] {1,2}, 100L);

В останньому рядку буде помилка: тип першого параметру повинен бути Number або його підклас, а другий параметр може бути довільним**.**

**Нижня межа:**

DataAccess<? super Integer, ? > ddd = new Data<Integer, Character>(12, 'k');

ddd = new Data<Number, String>(99, "bbb");

ddd = new Data<Object, String>(9.99, "bbb");

ddd = new **DataAccess<int[], Long>(**new int[] {1,2}, 100L);

В останньому рядку буде помилка: тип першого параметру повинен бути Integer, Number або Object, а тип другого параметру може бути довільним.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Які переваги використання синтаксичної конструкції *узагальнення?*
2. Що називаютьузагальненим класом та методом?
3. Які існують обмеження на *узагальнення*?
4. Які межі можна встановити для підстановлювального типу?

## Серіалізація

### 2.4.1 Серіалізація і десеріалізація

*Серіалізація* - це процес перетворення стану об'єкта і його метаданих (наприклад, ім'я класу і імена атрибутів) в формат, в якому вони можуть бути збережені (наприклад, у файлі, або буфері пам'яті, або передані через мережеве з'єднання) і відновлені в цьому ж або в іншому комп'ютерному середовищі.

Серіалізация використовується для полегшення переносимості і зв'язку через сокети або віддалені виклики методів Java. *Серіалізация* дозволяє записувати і читати з потоку дані за допомогою існуючого протоколу, щоб забезпечити сумісність з механізмами запису і читання за замовчуванням.

*Десеріалізація* - це процес відновлення (відтворення копії) об'єкта за інформацією з двійкового потоку.

Для того, щоб об'єкти класу були здатні до серіалізації, необхідно класу реалізовувати інтерфейс *java.io.Serializable*.

Серіалізация і десеріалізація виконуються з урахуванням версії об'єкта. Java використовує властивість serialVersionUID, яке не потрібно оголошувати і яке обчислюється на основі атрибутів класу, його імені і положення в локальному кластері. Якщо у джерела і приймача різні serialVersionUID, то середовище виконання вважає, що це різні класи і викине InvalidClassException. Властивість serialVersionUID змінюється щоразу, коли додається або видаляється елемент класу; змінюється порядок опису елементів класу. Можна зафіксувати значення властивості serialVersionUID, оголосивши в класі його як поле типу *private static final long.*

При спробі серіалізовати об'єкт класу, який реалізує інтерфейс *Serializable,* але об'єкт є посиланням на несереалізуемий клас, буде згенеровано *NotSerializableException.*

Процес десеріалізації відбувається з використанням *рефлексії*, тобто при відновленні об'єкта конструктор на пряму не викликається.

**Приклад**

*//Приклад, серіалізації об’єкта*

public class Student implements Serializable {

private static final long serialVersionUID = 10L; //Фіксація версії об’єкта

private String firstName;

private int group;

private int age;

public Student(String firstName, int group, int age) {

this.firstName = firstName;

this.group = group;

this.age = age;

System.out.println("Constructor");

}

public String toString() {  
 return "FirstName = " + firstName +  
 ", Group = " + group + ", Age = " + age;  
}

Student stud = new Student("Alex", 217, 20);

try (ObjectOutputStream oos = new ObjectOutputStream(

new FileOutputStream("C:\\student.ser")) ;  
 ObjectInputStream ois = new ObjectInputStream(  
 new FileInputStream("C:\\student.ser")) ) {

oos.writeObject(stud);

stud = null;  
Student ss = (Student)ois.readObject();  
System.out.println(ss);

} catch(IOException | ClassNotFoundException e) {

e.printStackTrace();

}

**Результат**

Constructor - Виклик конструктора тільки при створенні об’єкта

FirstName = Alex, Group = 217, Age = 20

### 2.4.2 Серіалізація складних об’єктів

*Складний (складений) об'єкт -* це об'єкт, який має в своєму складі посилання на інші об'єкти.

*Наприклад,* додамо в опис класу Student посилання на об'єкт класу Elective і методи getElective () і setElective ().

public class Elective {  
 private long id;  
 private String name;  
 public Elective(long id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public String toString() {  
 return "id = " + id + ", name = " + name;  
 }

}

Для правильного запису об'єкту серіалізуемого класу необхідно, щоб він містив посилання на об'єкти класів, які серіалізуються та посилання, які не підлягають серіалізації.

Для відмови від серіалізації поля класу можна використовувати ключове слово **transient**.

**Приклад**

public class Student implements Serializable {

private static final long serialVersionUID = 10L;

private String firstName;

private int group;

private int age;

private transient Elective elective;

// …….

**Результат**

Constructor

FirstName = Alex, Group = 217, Age = 20

null - Інформація про факультатив не була записана в потік.

В наступних випадках не має сенсу застосування ключового слова **transient:**

* Коли значення поля *обчислюється* на основі значень інших полів, то з міркувань економії часу і трафіку має сенс *утриматися* від серіализації поля;
* Коли значення поля коректно тільки в рамках поточного контексту (поле, зберігає посилання на ресурс системи);
* З міркувань безпеки (поле зберігає пароль).

Оскільки статичні поля класу не серіалізуются, то не має сенсу одночасне використання модифікаторів *static* і *transient*.

*//Приклад серіалізації зі статичним полем класу*,

class A implements Serializable {  
 private static int n = 0;  
 private int i;  
 A(int i) {  
 this.i = i;  
 n = i;  
 }  
 public String toString(){  
 return i + " " + n;  
 }

}

**Серіалізація і спадкування**

При десеріалізації використовується механізм *рефлексії*, за допомогою якого відтворюється інформація про клас і під об'єкт виділяється пам'ять, після чого його поля заповнюються значеннями з потоку (конструктор об'єкта при цьому не викликається).

При десеріалізациі об'єкта підкласу, якщо батьківський клас не було серіалізовано, викликається конструктор без параметрів для батьківського класу.

При відсутності конструктора без параметрів у батьківського класу при десеріалізациі об'єкта виникне помилка типу InvalidClassException.

### 2.4.3 Користувальницька серіалізація

Модифікація серіалізації за замовчуванням:

* Для реалізації користувальницької серіалізації необхідно визначити методи *writeObject* () і / або *readObject* ().
* Це не перевизначення методів потоків *ObjectOutputStream* і *ObjectInputStream*: віртуальна машина перевіряє і викликає ці методи від засобів рефлексії.
* Ці методи повинні бути описані як закриті, щоб гарантувати не можливість їх перевизначення або перевантаження.

Власна Серіалізація

Для цього використовуємо інтерфейс ***Externalizable***, який має два методи, які треба перевизначити:

public void *writeExternal*(ObjectOutput out) throws

IOException;

publicvoid*readExternal*(ObjectInput in)throws

IOException*, ClassNotFoundException****;***

Необхідно, щоб серіалізуємий клас мав конструктор без параметрів, тому що при десеріалізациі викликається спочатку цей конструктор, а потім метод readExternal (). Всі спадкоємці такого класу теж будуть вважатися такими, що не реалізують інтерфейс Externalizable, і у них теж повинен бути конструктор без параметрів.

Використання інтерфейсу *Externalizable* надає повний контроль над процесом серіалізації: серіалізація по замовчуванню (потік *ObjectOutputStream*) зберігає посилання на об'єкти, які в нього записуються (тобто якщо стан об'єкта, який вже був записаний, буде записуватися знову, то новий стан не зберігається):

MyObject obj = new MyObject();

obj.setState(100); //встановлення стану

out.writeObject(obj);

obj.setState(200); //зміна стану

out.writeObject(obj); //новий стан не буде збережений

Контроль версій добре працює до тих пір, поки вносяться зміни сумісні (тобто при додаванні і / або видаленні методів і / або полів можна зафіксувати значення властивості serialVersionUID, щоб не отримати помилку InvalidClassException; при несумісних змінах, наприклад, зміна ієрархії об'єктів або припинення реалізації інтерфейсу Serializable - це робити не рекомендується). *Власна* серіалізація не працює з властивістю serialVersionUID. Серіалізація за *замовчуванням* є простою, але менш продуктивною. *Власна* серіалізація може підвищити продуктивність до 50%.

Особливості власної серіалізації:

* Якщо поле в класі оголошено з ключовим словом *transient*, то при використанні інтерфейсу *Externalizable* таке поле все одно можна і записувати і читати з потоку;
* Якщо поле в класі оголошено як статичне, то при використанні інтерфейсу *Externalizable* таке поле все одно можна і записувати і читати з потоку;
* Якщо поле в класі оголошено з модіфіктором *final*, то десеріалізовать його НЕ можна (*final*-поля повинні бути ініційовані в конструкторі, а після цього змінити значення цього поля в методі readExternal () буде неможливо).

**Приклад**

//Приклад застосування інтерфейсу Externalizable:

public class User implements Externalizable {

private int id; private String username;

public User() { }

public User(int id, String username) {

this.id = id; this.username = username;

}

@Override

public void writeExternal(ObjectOutput out) throws IOException {

out.writeInt(id); out.writeObject(username);

}

@Override

public void readExternal(ObjectInput in) throws IOException,  
 ClassNotFoundException {

id = in.readInt(); username = (String) in.readObject();

}

}

User userWrite = new User(1, "AlexUser");

try (ObjectOutputStream oos = new

FileOutputStream("userfile.ser"));  
 ObjectInputStream ois = new ObjectInputStream(

new FileInputStream("userfile.ser"))) {  
oos.writeObject(userWrite);  
User userRead = (User) ois.readObject();  
System.out.println("User: " + userRead);

} catch (IOException | ClassNotFoundException e) {  
 e.printStackTrace();

}

**Результат**

User: id = 1, username = AlexUser

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

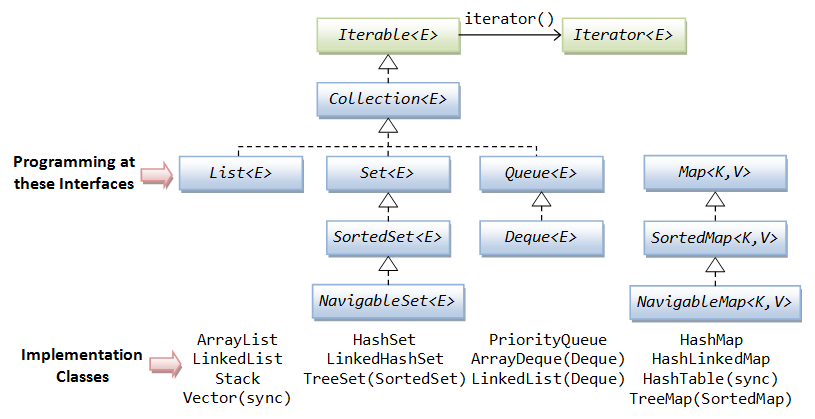
1. Назвіть причини використання серіалізації.
2. Які особливості процесу десеріалізації?
3. Як зробити об'єкт, здатним до серіалізації?
4. Як відмовитися від серіалізації поля класу?
5. В яких випадках не має сенсу застосовувати ключове слово transient?
6. Як при десеріалізації використовується механізм рефлексії?
7. Які особливості і переваги користувальницької серіалізації?

## Колекції

### 2.5.1 Інтерфейс Collection та клас Collections

Масиви мають значні недоліки - розмір масиву, індексна адресація, що не завжди зручно, тому що вона обмежує можливості додавання і видалення об'єктів. Щоб позбутися цих недоліків використовують рекурсивні типи даних, такі як *списки* і *дерева*. Стандартний набір *колекцій* Java призначений для позбавлення програміста необхідності самостійно реалізовувати ці типи даних і постачає його додатковими можливостями.

В колекціях можуть зберігатися будь які типи за посиланням.



Список методів інтерфейсу Collection, які реалізуються всіма колекціями:

1. boolean ***add***(Object obj); - додати елемент до колекції

2. void ***clear***(); - видалити всі елементи колекції

3. boolean ***contains***(Object obj); - чи належить елемент до колекції?

4. boolean ***remove***(Object obj); - видалити елемент колекції

5. int ***size***(); - розмір колекції

6. Iterator ***iterator***(Object obj); - додати до колекції ітератор

7. Object [] ***toArray***(); - отримати масив з колекції

8. boolean ***isEmpty***(); - чи є колекція пустою?

**Приклад**

Collection<String> myColl = new ArrayList<>();

myColl.add("data");

myColl.add("text");

myColl.add("hello");

myColl.add("java");

System.out.println(myColl);

System.out.println(myColl.size());

myColl.clear();

System.out.println(myColl);

**Результат**

[data, text, hello, java]

4

[]

Клас *java.util.Collections* складається тільки з статичних методів, які працюють з колекціями або повертають колекції. Він містить алгоритми, які працюють з колекціями як "обгортки" і повертають нову колекцію на базі зазначеної колекції. Всі методи цього класу кидають виняток типу NullPointerException, якщо колекції або об'єкти класу, що надаються їм, мають значення null.

**Методи класу Collections**

***addAll (Collection <? super T> c, T ... elements) -*** додає всі вказані елементи до зазначеної колекції;

***sort (List <T> list) -*** впорядковує список по зростанню;

***sort (List <T> list, Comparator <? super T> c) -*** впорядковує список в зазначеному порядку;

***swap (List <?> list, int i, int j) -*** міняє місцями елементи в зазначених позиціях зазначеного списку;

***reverse (List <?> list) –*** розташовує елементи списку в зворотному  порядку;

***shuffle (List <?> list) -*** перемішує елементи списку з використанням генератора випадкових чисел по замовчуванню

**Приклад**

List<String> list = new ArrayList<String>();

Collections.addAll(list,"bb","a","ff","cc","b","d");

System.out.println(list);

Collections.sort(list);

System.out.println(list);

Collections.swap(list,0,2);

System.out.println(list);

Collections.reverse(list);

System.out.println(list);

Collections.shuffle(list);

System.out.println(list);

**Результат**

[bb, a, ff, cc, b, d]

[a, b, bb, cc, d, ff]

[bb, b, a, cc, d, ff]

[ff, d, cc, a, b, bb]

[d, bb, a, cc, b, ff]

### 2.5.2 Інтерфейс Iterator

*Iterator* – це об’єкт, який дозволяє послідовно переміщуватися по колекції і видаляти певні елементи колекції.

Крім *Iterator* існує також *ListIterator*. Різниця між ними в наступному:

1. *Iterator* може використовуватися для перебору елементів *Set*, *List* та *Map*. На відміну від нього, *ListIterator* може бути використаний тільки для перебору елементів колекції List.
2. *Iterator* дозволяє перебирати елементи тільки в одному напрямку, за допомогою методу *next* (). Тоді як *ListIterator* дозволяє перебирати список в обох напрямках, за допомогою методів *next* () і *previous* ().
3. За допомогою *ListIterator* ви можете модифікувати список, додаючи / видаляючи елементи за допомогою методів *add* () і *remove* (). *Iterator* не підтримує даного функціоналу .

public interface Iterator<E> {

boolean hasNext(); // to check existing of next element

E next(); // to get current element

void remove(); // to delete current element

}

**Iterator і Enumeration**

*Enumeration* в два рази швидше *Iterator* і використовує менше пам'яті. Iterator потокобезпечний, тому що не дозволяє іншим потокам модифікувати колекцію при переборі.

*Enumeration* можна використовувати тільки для *read-only* колекцій. У нього відсутній метод *remove* (). Enumeration має такі методи: *hasMoreElement* (), *nextElement* ().

### 2.5.3 Спискові інтерфейси: List, Queue, Deque

**Список (List)**

*List* - це впорядкований список. Об'єкти зберігаються в порядку їх додавання до списку. Доступ до елементів списку здійснюється за індексом.

Реалізації: *ArrayList*, *LinkedList*.

**Методи List**

1. ListIterator *listIterator*(); - отримати ітератор для колекції

2. ListIterator *listIterator*(int index); - отримати ітератор, для якого перший виклик методу *next()*  повертає елемент з позиції *index*

3. void *add*(int index, Object obj); - додати елемент в позицію *index*

void *addAll*(int index, Collection c); - додати колекцію с в позицію *index*

5. Object *remove*(int index); - видалити елемент з позиції *index*

6. Object *get*(int index); - отримати елемент з позиції *index*

7. Object *set*(int index, Object obj); - замінити елемент в позиции *index і* повернути старий

8. int *indexOf*(Object obj);- отримати позицію першого входження елемента

9. int *lastIndexOf*(Object obj); - отримати позицію останнього вхождення елемента

**Списки ArrayList та LinkedList**

*ArrayList* - це список, реализований на основі масиву, а *LinkedList* — це класичний зв’язний список, заснований на об’єктах з посиланнями між ними. У *LinkedList* **-**  кожен вузол має посилання на наступний вузол у списку.

Переваги і недоліки.

* *LinkedList* в абсолютних значеннях проиграє *ArrayList* і по памяті і по швидкості виконання операцій.
* Недоліки *ArrayList* проявляються при вставці / видаленні елемента в середині списку – це потребує перезапис всіх елементів розміщених «правіше» в списку на одну позицію вліво, крім того, при видаленні елементів розмір масиву не зменшується, до явного виклику методу trimToSize ().
* *LinkedList* переважно застосовувати, коли відбувається активна робота (вставка / видалення) з серединою списку або у випадках, коли необхідно гарантований час додавання елемента в список.
* При видаленні довільного елемента зі списку, всі елементи, що знаходяться «правіше» зміщуються на одну клітинку вліво і реальний розмір масиву (його ємність, capacity) не змінюється ніяк. Механізм автоматичного «розширення» масиву існує, а ось автоматичного «стиснення» - ні, можна тільки явно виконати «стиснення» командою *trimToSize* ().
* Недоліки *ArrayList* проявляються при вставці / видаленні елемента в середині списку - це призводить до перезапису всіх елементів розміщених «правіше» в списку на одну позицію вліво, крім того, при видаленні елементів розмір масиву не зменшується, до явного виклику методу *trimToSize* ().
* Масиви можуть працювати швидше і брати менше пам'яті. Списки втрачають в продуктивності через можливість автоматичного збільшення розміру та супутніх перевірок. Плюс до цього, що розмір списку збільшується не на 1, а на більшу кількість елементів (у 1.5 рази).

**Черга (Queue)**

*Queue (черга)* – інтерфейс-колекція, призначена для зберігання елементів в порядку перший зайшов-перший вийшов (FIFO). На додаток до базових операцій інтерфейсу Collection, черга надає додаткові операції вставки, отримання і контролю.

**Методи інтерфейсу Queue<E>**

1. E ***element***(); – отримати перший елемент черги
2. E ***peek***(); – отримати перший елемент черги (повертає **null**, якщо черга пуста)
3. boolean ***offer***(E o); – додати елемент в чергу, якщо можна
4. boolean ***add***(E o); – додати елемент в чергу, якщо можна
5. E ***remove***(); – отримати і видалити перший елемент черги
6. E ***poll***(); – отримати і видалити перший елемент черги(повертає **null**, якщо черга пуста) .

**Двостороння черга (Deque)**

*Deque (двостороння черга)* – інтерфейс-колекція, призначена для зберігання елементів в порядку «перший зайшов-перший вийшов» (FIFO), а також «останнім зайшов –першим вийшов» (LIFO), тобто використовуватись, як стек.

### 2.5.4 Множина Set

*Set* — множина об’єктів , які не повторюються. Реалізація: *HashSet* – порядок визначає Hash-функція; *TreeSet* – порядок забезпечується бінарним пошуковим деревом (BST).

*SortedSet* - порядок між об'єктами може бути визначено, як за допомогою методу *compareTo* інтерфейсу java.lang.*Comparable* <T>, так і за допомогою спеціального класу-компаратора, що спадкує інтерфейс java.util.Comparator <T>.

**Множини TreeSet і HashSet**

*HashSet* реалізовано на основі хеш-таблиці, а *TreeSet* — на основі бинарного дерева.

Клас *Object* має метод *hashCode* (), який використовується класом *HashSet* для ефективного розміщення об'єктів, що заносяться в колекцію. У класах об'єктів, що заносяться в *HashSet*, цей метод повинен бути перевизначений (override).

*TreeSet* – вміщує *TreeMap* який в свою чергу використовує збалансоване *бінарне червоно-чорне дерево* для зберігання елементів. *TreeSet* хороший тим, що для операцій *add*, *remove* і *contains* потрібно гарантований час *log (n).*

Всі елементи після додавання в звичайне бінарне дерево будуть знаходиться в одній гілці довжиною N елементів, що зводить нанівець всі переваги такої структури, як дерево (фактично виходить список). В основі *TreeSet* лежить червоно-чорне дерево, яке вміє само себе балансувати. У результаті, в якому б порядку ви не додавали елементи у *TreeSet* , переваги цієї структури даних будуть зберігатися.

*У HashSet* додавання, пошук і видалення елементів виконується за постійний час. У *HashSet* в якості ключа (на відміну від *HashMap*) береться сам элемент. Порядок перебору (ітерування) елементів не відповідає порядку вставки.

*У TreeSet* час виконання основних операцій відповідає *lоgN* (N - поточна кількість вузлів в дереві). Він забезпечує впорядковане зберігання елементів у вигляді червоно-чорного дерева (щоб гарантувати не виродження дерева в список при вставці елементів в порядку зростання значень).

*LinkedHashSet* – це хеш-таблиця зі зв'язаним двонаправленим списком, в якому зберігається порядок, в якому елементи були вставлені в набір.

**Приклад**

Set<String> mySet = new LinkedHashSet<>();

mySet.add("data");

mySet.add("java");

mySet.add("data");

mySet.add("sorting");

mySet.add("hi");

mySet.add("hello");

System.out.println(mySet);

**Результат**: порядок ітерування співпадає з порядком вставки

[data, java, sorting, hi, hello]

### 2.5.5 Карти відображень Map

*Інтерфейс* *Map <K, V>* використовується для відображення кожного елемента з однієї множини об'єктів (ключів) на інше (значень). *Map* - надає методи для роботи з даними виду «ключ / значення» (ключ - це об'єкт, який використовується для подальшого видалення даних - значень). *Map* - не може містити дублікати ключів. У *Map* - кожен ключ може відповідати не більше ніж одному значенню.

Реализації:

* java.util.HashMap<K,V>,
* java.util.LinkedHashMap<K,V>,
* java.util.TreeMap<K,V>,
* java.util.WeakHashMap<K,V>.

**Методи інтерфейсу Map<K,V>**

V get(K key) - отримати об’єкт, по вказаному ключу

V put(K key, V value) - додати об’єкт в карту

boolean containsKey(Object key) - перевірити чи є в карті заданий ключ

boolean containsValue(Object value) - перевірити чи є в карті заданий об’єкт

Set <K> keySet() - отримати множину ключів

Collection<V> values() - отримати набір об’єктів

Set<Map.Entry<K, V>> entrySet() - отримати множину пар «ключ-значення»

K getKey() - отримати ключ поточного елементу

V getValue() - отримати значення поточного елементу

V setValue(V newValue) - замінити поточний об’єкт

Інтерфейс *Map* має дві найбільш популярні реалізації: класи *TreeMap* і *HashMap*.

*TreeMap* реалізований на червоно-чорному дереві, час додавання / пошуку / видалення елемента - O (log N)

У *HashMap* час доступу до окремого елементу - O (1) за умови, що хеш-функція (Object.hashCode ()) визначена нормально (в разі Integer).

Загальна рекомендація - якщо не потрібна впорядкованість, використовувати HashMap. Виняток - ситуація з числами, які в якості ключів майже завжди дуже погані. Для них потрібно використовувати TreeMap.

Деякі методи HashTable синхронізовані, тому вона повільніше HashMap. Загальне зауваження - не рекомендується використовувати HashTable в багатопотокових застосуваннях. Для цього є ConcurrentHashMap. Можна отримати синхронізовану колекцію з не синхронізовані, використовуючі наступні методи: Collections.synchronizedList (list), Collections.synchronizedSet (set), Collections.synchronizedMap (map).

**Клас HashMap**

*Клас HashMap* не є упорядкованою колекцією: порядок зберігання елементів залежить від хеш-функції. Властивості цього класу:

* додавання елемента виконується за постійний час;
* час видалення / отримання залежить від розподілу хеш-функції;
* дозволяє використовувати величину **null** як в якості ключа, так і значення.

Конструктори HashMap:

* HashMap () - порожня карта з початковою ємністю 16 і коефіцієнтом завантаження 0,75);
* HashMap (int initialCapacity) - порожня карта із заданою початковою ємністю і коефіцієнтом завантаження 0,75;
* HashMap (int initialCapacity, float loadFactor) - порожня карта з заданими початковою ємністю і коефіцієнтом   завантаження.

*Ємність* - це число осередків в хеш-таблиці.

*Коефіцієнт завантаження* (*loadFactor*) - це міра того, наскільки заповнену хеш-таблицю дозволено отримати, перш ніж її ємність автоматично збільшиться.

*Максимальна кількість Hash-значень* - capacity\*loadFactor.

*Максимальна кількість значень hashCode ()* - 2^32 (оскільки тип цієї змінної - Integer).

**Клас LinkedHashMap**

*LinkedHashMap* - це колекція, яка підтримує двонаправлений зв'язаний список, що проходить через всі його записи:

* LinkedHashMap визначає порядок ітерації, який відповідає порядку, в якому ключі були вставлені в карту;
* повторна вставка ключа, не впливає на порядок додавання;
* метод put (key, value), завжди викликає метод containsKey (key) для перевірки існування ключа (якщо ключ є, то відбувається заміна значення).

**Клас TreeMap**

*TreeMap* є впорядкованою колекцією з наступними властивостями:

* по-замовчуванню колекція сортується по ключам з використанням принципу «природнього порядку»;
* ця поведінка може бути змінена за допомогою об'єкта **Comparator**, який вказується в якості параметра при створенні об'єкта **TreeMap**;
* реалізація заснована на червоно-чорних деревах.

**Приклад**

Map<String, Integer> hm = new TreeMap<>();

hm.put("ee", 5);

hm.put("cc", 3);

hm.put("aa", 1);

hm.put("bb", 2);

hm.put("dd", 4);

hm.put("ff", 6);

System.out.println(hm);

int x = hm.get("dd");

System.out.println(x);

**Результат**

{aa=1, bb=2, cc=3, dd=4, ee=5, ff=6}

4

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Яке призначення колекцій?
2. Які основні методи інтерфейсу Collection та класу Collections?
3. В чому різниця інтерфейсів Iterator і Enumeration?
4. Назвіть особливості спискових інтерфейсів List, Queue та Deque.
5. Назвіть особливості реалізацій колекції Set.
6. Коли використовують інтерфейс Map? В яких класах реалізовані його методи?

## Рефлексія

*Рефлексія* (від лат. – повернення назад, відображення) - це процес, коли програма може відстежувати і модифікувати власну структуру і поведінку під час виконання. API рефлексії мови java знаходиться в пакеті *java.lang.reflect* і використовується для перегляду інформації про класи, інтерфейси, методи, поля, конструктори, анотації під час виконання java програм.

За допомогою рефлексії java можна виконати наступне:

* Визначити клас об'єкта;
* Отримати інформацію про модифікатори класу, поля, методи, конструктори і суперклас;
* З'ясувати, які константи і методи належать інтерфейсу;
* Створити екземпляр класу, ім'я якого невідоме до моменту виконання програми;
* Отримати і встановити значення властивості об'єкта;
* Викликати метод об'єкта;
* Створити новий масив, розмір і тип компонентів якого невідомі до моменту виконання програми.

### 2.6.1 Тип Class

*Клас–тип Class* - представляє метадані, тобто дані, за допомогою яких описується структура іншого типу даних. Всі класи *java* мають пов'язаний з ними об'єкт типу *Class*. Клас *Class* та його підкласи в ієрархії вбудованих класів *java* показано нижче.

* Object
  + Class<T>
  + AccessibleObject
    - Constructor
    - Field
    - Method

Клас з ім'ям *Сlass* представляє характеристики класу, екземпляром якого є об'єкт. Він зберігає інформацію про те, чи не є об'єкт насправді інтерфейсом, масивом або примітивним типом, який суперклас об'єкта, яке ім'я класу, які в ньому конструктори, поля, методи і вкладені класи.

У класі *Class* немає конструкторів, екземпляр цього класу створюється виконуючою системою Java під час завантаження класу і надається методом getClass () класу object, наприклад:

String s = "Це рядок";

Class з = s.getClass ();

Статичний метод forName (string class) повертає об'єкт класу *Class* для класу, зазначеного в аргументі, наприклад:

Class cl = Class.forName ( "Java, lang.String");

Але цей спосіб створення об'єкта класу *Class* вважається застарілим (deprecated). У нових версіях JDK для цієї мети використовується спеціальна конструкція - до імені класу через точку додається слово class:

Class c2 = Java.lang.String.class;

Логічні методи *isArray (), isIntetface (), isPrimitive ()* дозволяють уточнювати, чи не є об'єкт масивом, інтерфейсом або примітивним типом.

Якщо об'єкт посилального типу, то можна отримати відомості про вкладені класи, конструктори, методи і поля методами getDeclaredClasses (), getDeclaredConstructors (), getDeclaredMethods (), getDeclaredFields (), у вигляді масиву класів, відповідно - Class, Constructor, Method, Field. Останні три класи розташовані в пакеті *java.lang.reflect* і містять відомості про конструктори, поля і методи аналогічно тому, як клас Сlass зберігає відомості про класи.

Методи getClasses (), getConstructors (), getІnterfaces (), getMethods (), getFieids () повертають такі ж масиви, але не всіх, а тільки відкритих членів класу.

Метод getSuperclass () повертає суперклас об'єкта посилального типу, getPackage () - пакет, getModifiers () - модифікатори класу у бітовому вигляді. Модифікатори можна потім розшифрувати методами класу Modifier з пакета Java.lang.reflect.

Існує кілька способів отримати об'єкт типу *Class*, в залежності від того, чи є доступ до:

* об'єкту;
* імені класу;
* типу;
* іншого існуючого об'єкту Class.

**Способи отримання екземпляра Class**

I) Через існуючий об’єкт класу: треба викликати метод getClass() на об’єкті.

Наприклад,

Student myStud = new Student("Ivan", 213, 19);

Class<?> clazz1 = myStud.getClass();

double[] arrayDouble = new double[10];

Class<?> clazz2 = arrayDouble.getClass();

Class<?> clazz3 = "foo".getClass();

Цей спосіб працює тільки для типів-посилань.

II) Через доступное ім’я типу (немає доступу до об’єкта): треба до імені типу додати нотацію .class

Наприклад,

Class<?> clazz1 = Student.class;

Class<?> clazz2 = int[].class;

Цей спосіб працює тільки для примітивних типів.

Наприклад,

Class<?> clazz3 = int.class;

Екземпляр типу Class для примітивного типу можна отримати і через поле TYPE відповідного класу-обгортки:

Class<?> clazz4 = Integer.TYPE;

III) Через виклик статичного методу *Class.forName():* потрібно цьому методу передати повне кваліфікаційне ім'я типу як рядок символів.

Наприклад,

Class<?> clazz1 = Class.forName("java.lang.Thread");

Цей спосіб не працює для примітивних типів даних.

Для масивів в якості кваліфікаційного типу можна використовувати наступний синтаксис:

Class<?> cDoubleArray = Class.forName("[D");

Це те саме, що й *double[].class*

IV) Через вже існуючий екземпляр типу *Class* (тобто використовуючи засоби відображення): потрібно викликати метод *getSuperclass* () на об'єкті підкласу для отримання опису суперкласу.

Наприклад,

Class<?> sclazz = Double.class.getSuperclass();

Результат буде типом *java.lang.Number*.

Треба викликати метод *getEnclosingClass*() для отримання обмежуючого класу.

Наприклад,

Class<?> clazz = Map.Entry.class.getEnclosingClass();

Результат буде типом *java.util.Map*

Можна використовувати метод *getEnclosingClass* () для отримання обмежуючого класу для анонімного класу.

Наприклад,

public class MyClass {  
 public static void main(String[] args) {  
 Comparator<Integer> comparator = new Comparator<>() {  
 public int compare(Integer o1, Integer o2) {  
 return o2 - o1;  
 }  
 };  
 Class clazz = comparator.getClass().getEnclosingClass();  
 }  
}

Результат будет типом *MyClass*.

Потрібно викликати метод *getClasses* () для отримання *public* типів, які оголошені як члени класу, включаючі і спадкоємців.

Наприклад,

Class<?>[] clazz1 = Character.class.getClasses();

Результатом будуть типи Character.Subset, Character.UnicodeBlock и Character.UnicodeScript.

Потрібно викликати метод getDeclaredClasses () для отримання всіх типів, які оголошені в цьому класі

Наприклад,

Class<?>[] clazz2 = Map.class.getDeclaredClasses();

Результат будет типом java.util.Map$Entry

### 2.6.2 Отримання інформації про модифікатори

Необхідно викликати метод getModifiers() на екземплярі Class: набір модифікаторів повертається у вигляді цілого числа з різними бітовими позиціями, що відображають різні модифікатори.

Наприклад,

int modifs = clazz.getModifiers();

Використовувати клас Modifier з пакета java.lang.reflect, який містить статичні методи і константи для декодування модифікаторів доступу до класу і членів класу.

Таблиця 2.6.1. Методи і константи для декодування модифікаторів доступу

|  |  |  |
| --- | --- | --- |
| ***Ім’я константи*** | ***Назва методу*** | ***Модифікатор*** |
| ABSTRACT | isAbstract | abstract |
| FINAL | isFinal | final |
| INTERFACE | isInterface | interface |
| NATIVE | isNative | native |
| PRIVATE | isPrivate | private |
| PROTECTED | isProtected | protected |
| PUBLIC | isPublic | public |
| STATIC | isStatic | static |
| STRICT | isStrict | strictfp |
| SYNCHRONIZED | isSynchronized | synchronized |
| TRANSIENT | isTransient | transient |
| VOLATILE | isVolatile | volatile |

### 2.6.3 Отримання інформації про елемент класу

У пакеті *java.lang.reflect* знаходиться інтерфейс *Member*, який відображає ідентифікаційну інформацію про один елемент (поле або метод) або конструктор. Цей інтерфейс реалізується класами *java.lang.reflect.Field*, *java.lang.reflect.Constructor*, *java.lang.reflect.Method*. Ці класи використовуються для надання інформації про сам елемент і про динамічний доступ до нього.

**Отримання інформации про поле**

Необхідно викликати метод *getFields* () на примірнику *Class*: повертає масив об'єктів типу *Field*, що відображають всі public поля, оголошені в класі або інтерфейсі.

Наприклад, Field[] fields = clazz.getFields();

Необхідно викликати метод getDeclaredFields () на примірнику Class: повертає масив об'єктів типу Field, що відображають всі поля, оголошені в класі або інтерфейсі.

Наприклад, Field[] fields = clazz.getDeclaredFields();

Відображене поле може бути полем класу (статичним) або полем об’єкту.

**Отримання інформації про конструктор**

Необхідно викликати метод *getConstructors* () на примірнику *Class*: повертає масив об'єктів типу *Constructor*, що відображають всі public конструктори, оголошені в класі.

Наприклад,

Constructor[] constrs = clazz.getConstructors();

Необхідно викликати метод getDeclaredConstructors () на примірнику Class: повертає масив об'єктів типу Constructor, що відображають всі конструктори, оголошені в класі.

Наприклад,

Constructor[] constrs = clazz.getDeclaredConstructors();

**Створення об’єкта класу через рефлексію**

Необхідно отримати об'єкт типу *Constructor*, який відображає необхідний конструктор для створення об'єкта: застосування методу getConstructor () на примірнику типу Class, передавши йому масив типів параметрів у вигляді об'єктів Class.

Викликати метод newInstance () на примірнику класу Constructor, передавши йому список відповідних аргументів.

Клас Constructor дозволяє розширювати перетворення в разі зіставлення аргументів на методі newInstance () з формальними параметрами базового конструктора, але кидає виняток типу IllegalArgumentException при звуженому перетворенні.

**Отримання інформації про метод**

Необхідно викликати метод getMethods () на примірнику Class: повертає масив об'єктів типу Method, що відображають всі public методи.

Наприклад,

Method[] methods = clazz.getMethods();

Необхідно викликати метод getDeclaredMethods() на примірнику Class: повертає масив об'єктів типу Method, що відображають всі методи.

Наприклад,

Method[] methods = clazz.getDeclaredMethods();

Повертаються методи, оголошені в класі / інтерфейсі, а також успадковані від суперкласу і інтерфейсів. Якщо відкритих методів немає або Class описує примітивний тип, то повертається масив довжиною нуль.

**Динамічний виклик методу**

Клас *Method* має метод

*Object* ***invoke****(Object target, Object ... parameters)*

який використовується для динамічного виклику методу. Він приймає перший параметр - посилання на об'єкт, для якого викликається; другий параметр довільної довжини - перелік аргументів типу Object.

Особливості:

* Аргументи автоматично розпаковуються, якщо формальний параметр примітивного типу;
* при необхідності можуть викликатися методи перетворення;
* якщо метод, що викликається є статичним, то посилання на об'єкт ігнорується;
* при виклику статичного методу замість посилання на об'єкт можна вказувати null;
* якщо метод, що викликається, не має параметрів, то вказується тільки посилання на об'єкт або в якості другого аргументу - значення null;
* якщо метод, що викликається має значення, що повертається і це значення примітивного типу, то воно упаковується в об'єкт відповідного класу-обгортки;
* якщо метод, що викликається не має значення, що повертається, то з методу invoke () повертається значення null.

**Доступ до закритих елементів класу**

Викликати метод setAccessible (true) на об'єктах класів Field, Constructor, Method, які відображають представлення закритих елементів класу.

**Приклад**

MyTestReflect obj = new MyTestReflect();

Field field = clazz.getDeclaredField("index");

field.setAccessible(true);

// Залежно від типу поля викликається відповідний метод

System.out.println("Private field value: " + field.getInt(obj));

field.setInt(obj, 100);

System.out.println("New private field value: " + field.getInt(obj));

**Результат**

Private field value: 2016001

New private field value: 100

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що означає слово *рефлексія* умові java?
2. Яке призначення класу Class?
3. Якими способами можна отримання екземпляр класу Class?
4. Як за отримати інформацію про модифікатори?
5. Як отримати інформацію про елементи класу?

## Графічний інтерфейс користувача

### 2.7.1 AWT

У мові Java для створення інтерфейсу користувача використовується такі пакети, як *AWT, Swing, SWT*, *JavaFX*. Склад компонентів цих пакетів схожий на набір засобів традиційних графічних користувальницьких інтерфейсів (GUI).

AWT (Abstract Window Toolkit) – перша бібліотека для роботи з графікою, з'явилася у версії 1.0.

*Компонент (Component)* AWT - це абстрактний клас, який інкапсулює всі атрибути візуального інтерфейсу: обробка введення з клавіатури, управління фокусом, взаємодія з мишею, повідомлення про вхід / вихід з вікна, зміни розмірів і положення вікон, подання графічного представлення, збереження поточного текстового шрифту, кольорів фону і переднього плану та інше. Об'єкти класу Component будуть далі називатися компонентами.

Кожній компоненті в цій системі ставиться у відповідність компонент операційної системи, в якій виконується програма. Зв'язок здійснюється через так звані peer-об'єкти, що створюються глибоко всередині реалізації графічного інтерфейсу, користувач їх не бачить. Більшість викликів у цих peer-об'єктах насправді *native*, відповідно, потрібна реалізація під кожну операційну систему (ОС). Хоч AWT і вважалась кросплатформною, компоненти AWT були реалізовані платформозалежним способом для кожної реалізації Java машини.

У мові Java існує кілька класів для створення ієрархічної структури елементів інтерфейсу користувача. Більшість класів є дочірніми по відношенню до *Component*.

На наступному рівні ієрархії розташований підклас *Container*, який містить підкласи *Panel* і *Window*.

Клас Panel надає місце для розташування інших компонентів. Клас *Applet* (є підкласом класу Panel) - вікно для виведення в браузері.

Клас *Window* має два підкласи:

* *Frame* - для створення форм застосунків або кадрів у вікні браузера.
* *Dialog* - для створення діалогових вікон.

Компоненти являють собою базові блоки для інтерфейсу користувача, їх можна групувати і об'єднувати в панелі. У той же час панелі самі є компонентами, що дозволяє "вкладати" об'єкти один в інший, створюючи складну ієрархію.

Компоненти створюються так само, як і інші об'єкти мови Java. Новий компонент слід помістити на панель або в інший контейнер, який відповідає за його відображення на екрані. Наприклад, компонент *Label* ( «напис») може бути створений так:

Label label1 = new Label ( "Hi !!!");

У класі *Graphics* існують різноманітні методи роботи з графічними примітивами. Засоби мови Java дозволяють змінювати колір та створювати зображення ліній, прямокутників, еліпсів та багатокутників. Кожна графічна компонента має метод *paint(java.awt.Graphics)*, який викликається для малювання. Цей метод визначено в класі *java.awt.Component* і перевизначений скрізь, де необхідно.

Найчастіше методу *paint* передається об'єкт класу *Graphics*. Для побудови зображень необхідно звертатись до його методів. Також можна створити власний об'єкт цього класу, наприклад, для прискорення виведення зображень під час відтворення складної анімації.

Проблеми AWT:

* Подібні елементи у різних ОС можуть мати деякі відмінності,
* ОС часто мають елементи GUI, які відсутні в інших ОС,
* Використання *native*-методів у AWT призводить до виникнення помилок на конкретних платформах.

### 2.7.2 Swing

Для вирішення зазначених проблем AWT було розроблено бібліотеку *Swing*. Заснований на інших принципах, він дозволяє створювати інтерфейс користувача, який здатний виглядати і поводитися однаково у всіх ОС. З іншого боку, при необхідності можна легко перейти на вигляд і поведінку програми для поточної ОС.

Основна особливість Swing у тому, що майже всі компоненти написані на Java. Тому в принципі вони мали виглядати однаково скрізь, але працювали повідьніше, ніж в AWT.

Частина елементів AWT залишена у Swing, інша частина - замінена, також додані нові елементи. В результаті вийшла повноцінна графічна система, що дозволяє створювати програми з різноманітною графікою.

Розробники бібліотеки SWT вдало поєднали найкраще з AWT та Swing. SWT є не для всіх платформ. Компоненти та функції SWT, які підтримуються графічної підсистемою, як й у AWT працюють через адаптери, а інші функції, як і Swing, дописані на Java.

Основне поняття у Swing, як і у AWT це – *компонент* (component). Компоненти можуть бути видимими (мають графічний образ) та невидимими.

Щоб компонент з'явився на екрані, його треба помістити в контейнер (і компоненти і контейнери - об'єкти деяких класів Java).

Основою для Swing залишається AWT, що видно з наведеної нижче схеми:

**Component(AWT) - > Container (AWT) -> JComponent (Swing) -> інші**

Набір елементів GUI у Swing більший, ніж у AWT, і може бути розширений.

Сильна прив'язка до «нативних» методів відсутня, що знижує ймовірність виникнення помилок. Відображення на різних платформах однакове. Нижче показано ієрархію класів Swing



Клас Graphics2D забезпечує покращений контроль геометрії, роботу з кольором, перетворення координат тощо.

Метод *paint(Graphics)* у Swing зазнав логічних змін порівняно з AWT.

Метод *JComponent.paint(Graphics)* делегує малювання трьом protected-методам – *paintComponent(Graphics), paintBorder(Graphics) та paintChildren(Graphics).*

*paintComponent* – ​​цей метод дозволяє малювати саму компоненту. Тобто. робить те, що у AWT робив *paint(Graphics)*. Саме цей метод необхідно перевизначати для того, щоби намалювати щось на компоненті.

*paintBorder* – малює рамку компоненти. Це нововведення Swing – у AWT рамок компоненти не мали.

*paintChildren* – малює дочірні елементи. Це ще одна зміна поведінки метода *paint (Graphics)* - в AWT останній дочірні елементи не малював.

Робота з вікнами у Swing реалізована таким чином:

* Використовується клас **javax.swing.JFrame,**
* Вміст вікна - на панелі, посилання на яку можна отримати викликом, методу **getContentPane(),**
* Параметрами вікна можна керувати.

*Frame* містить *RootPane* - основну панель, яка керує багатошаровою панеллю *LayeredPane*.

Остання панель містить *ContentPane* та «скло» *GlassPane*. Зазвичай скло невидиме, але воно дозволяє перехоплювати керуючі дії на ті компоненти, які воно накриває.

Багатошарова панель дозволяє відображати компоненти у глибину, визначаючи порядок перекриття одних компонентів іншими. До складу багатошарової панелі входить панель вмісту і може також включатися рядок меню.

Незважаючи на те, що прозора та багатошарова панелі є невід'ємними частинами контейнера верхнього рівня і виконують важливі функції, їх дії здебільшого приховані не тільки від користувачів, а й від розробників прикладних програм.

Прикладна програма взаємодіє переважно з панеллю вмісту, оскільки у неї включаються візуальні компоненти. Іншими словами, додаючи компонент, наприклад кнопку, у контейнер верхнього рівня, ви насправді вводите його на панелі вмісту.

У наступному прикладі програми створюється фрейм, у якому міститься графічний елемент "мітка", що представляє рядок тексту. Фрейм відображається на екрані.

import javax.swing.\*; //приклад з Sun Tutorial

public class HelloWorldSwing{

private static void createAndShowGUI(){

//при створенні фрейма в якості параметра передано заголовок вікна

JFrame frame = new JFrame("HelloWorldSwing");

//при закритті вікна програма завершить роботу

frame.setDefaultCloseOperation (JFrame.EXIT\_ON\_CLOSE);

JLabel label = new JLabel("Hello World"); // поле-підпис

frame.getContentPane().add(label); //виведення підпису на панелі

frame.setSize(400, 200); //розмір фрейма

frame.setVisible(true); } // для відображення фрейму на екрані

#### Розміщення компонентів

Підтримка менеджерів розташування (компоновок або макетів) вбудована в базовий клас java.awt.Container за рахунок реалізації інтерфейсу java.awt.LayoutManager.

Приклади менеджерів компоновок:

* FlowLayout
* BorderLayout
* GridLayout
* GridBagLayout
* CardLayout
* [BoxLayout](https://java-online.ru/swing-layout.xhtml#boxlayout)
* [GroupLayout](https://java-online.ru/swing-layout.xhtml#grouplayout)
* [SpringLayout](https://java-online.ru/swing-layout.xhtml#springlayout)

Менеджер розташування дозволяє не тільки згладити різницю між операційними системами, але до того ж дає можливість з легкістю змінювати зовнішній вигляд програми, не дбаючи про те, як при цьому змінюються розміри та розташування компонентів. Для будь-якого компонента Swing можна встановити менеджер розташування або дізнатися, який менеджер використовується в даний момент. Для цього призначені методи setLayout() та getLayout().

Компонент можна примусово розмістити (методом *move*()) в потрібному місці, задавши координати (х, у). Метод *location*() повертає координати компонента у вигляді об'єкта *Point*. Метод *locate*() за заданими координатами повертає компонент, розташований у цьому місці.

Нижче коротко описані деякі менеджери розташування.

#### FlowLayout

Клас *FlowLayout* реалізує простий спосіб розміщення, при якому компоненти розташовуються, починаючи з лівого верхнього кута, зліва направо і зверху вниз. Якщо у поточний рядок не поміщається черговий компонент, він розташовується в лівій позиції нового рядка. Праворуч, ліворуч, зверху і знизу компоненти відокремлюються один від одного невеликими проміжками. Ширину цього проміжку можна задати в конструкторі *FlowLayout*. Кожен рядок з компонентами вирівнюється по лівому або правому краю, або центрується в залежності від того, яку з констант LEFT, RIGHT або CENTER було передано конструктору. Режим вирівнювання за замовчуванням — CENTER, ширина проміжку за замовчуванням — 5 пікселів.

#### BorderLayout

Клас *BorderLayout* реалізує звичайний стиль розміщення для вікон верхнього рівня, в якому передбачено чотири вузьких компонента фіксованої ширини по краях, і одна велика область у центрі, яка може розширюватися і звужуватися в двох напрямках, займаючи весь вільний простір вікна. У кожній з цих областей є рядки-імена North (північ), South (південь), East(схід) і West (захід), які відповідають чотирьом краям, a Center — центральній області. Нижче наведено приклад *BorderLayout* з компонентом у кожній з названих областей.

#### GridLayout

Клас *GridLayout* розміщує компоненти в простій рівномірній сітці. Конструктор цього класу дозволяє задавати кількість рядків і стовпців. Нижче наведено приклад, в якому *GridLayout* використовується для створення сітки 4х4. П’ятнадцять квадратів з шіснадцяти заповнюються кнопками, позначеними відповідними індексами.

int n = 4;

frame.getContentPane().setLayout(new GridLayout(n, n)); frame.getContentPane().setFont(new Font("Helvetica", Font.BOLD, 24));

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

int k = i \* n + j;

if (k > 0)

frame.getContentPane().add(new JButton("" + k));

}}

#### Обробка подій

Обробка подій реалізована за допомогою класів в пакетах:

* java.awt.event
* javax.swing.event

Модель обробки подій Java базується на концепції слухача подій. Слухачем події є об'єкт, зацікавлений в отриманні даної події. У об'єкті, який породжує подію (у джерелі подій), міститься список слухачів, зацікавлених в одержанні повідомлення про те, що дана подія відбулася, а також методи, які дозволяють слухачам додавати або видаляти себе з цього списку. Коли джерело породжує подію (або коли об'єкт джерела зареєструє подію, пов'язану з введенням інформації користувачем), воно сповіщає всі об'єкти слухачів подій про те, що дана подія відбулася.

Джерело події сповіщає об'єкт слухача шляхом виклику спеціального методу і передачі йому об'єкта події (примірника підкласу *EventObject*). Для того щоб джерело могло викликати даний метод, він повинен бути реалізований для кожного слухача. Це пояснюється тим, що всі слухачі подій в Java визначені у вигляді інтерфейсів для кожного з визначених у ньому типів подій (наприклад, для подій *MouseEvent* визначено два інтерфейси слухачів: *MouseListener* і *MouseMotionListener*, а для подій *ActionEvent* – тільки один *ActionListener*). Всі інтерфейси слухачів подій є розширеннями інтерфейсу *java.util.EventListener*.

В інтерфейсі слухача подій може визначатися декілька методів. Наприклад, клас подій, подібний MouseEvent, описує декілька подій, пов'язаних з мишею, таких як події натискання і відпускання кнопки миші. Ці події викликають різні методи відповідного слухача. За встановленою угодою, методам слухачів подій може бути переданий один аргумент, який є об'єктом того, що відповідає даному слухачеві. Наприклад, mousePressed(MouseEvent e), де MouseEvent e – це об'єкт події, яка відповідає методу mousePressed. В цьому об'єкті повинна міститися вся інформація, необхідна програмі для формування реакції на цю подію. В таблиці 2.7.1 наведені типи подій, відповідні їм слухачи, а також методи, визначені в кожному інтерфейсі слухача.

Для кожного інтерфейсу слухачів подій, що містить кілька методів, визначено простий клас-адаптер, який забезпечує порожнє тіло для кожного з методів відповідного інтерфейсу. Коли потрібен тільки один або два таких методу, іноді простіше отримати підклас класу-адаптера, ніж реалізувати інтерфейс самостійно. При отриманні підкласу адаптера потрібно лише перевизначити ті методи, які потрібні, а при реалізації інтерфейсу необхідно визначити всі методи, в тому числі і непотрібні в даній програмі.

Таблиця 2.7.1. Типи подій, слухачі і методи слухачів в Java

|  |  |  |
| --- | --- | --- |
| **Клас події** | **Інтерфейс слухача** | **Методи слухача** |
| ActionEvent | ActionListener | actionPerformed() |
| AdjustmentEvent | AdjustmentListener | adjustmentValueChanged() |
| ComponentEvent | ComponentListener | componentHidden() componentMoved() componentResized() componentShown() |
| ContainerEvent | ContainerListener | componentAdded() componentRemoved() |
| FocusEvent | FocusListener | focusGained()  focusLost () |
| ItemEvent | ItemListener | itemStateChanged() |
| KeyEvent | KeyListener | keyPressed()  keyReleased()  keyTyped() |
| MouseEvent | MouseListener | mouseClicked()  mouseEntered()  mouseExited()  mousePressed()  mouseReleased() |
| MouseMotionListener | mouseDragged()  mouseMoved() |
| TextEvent | TextListener | textValueChanged() |
| WindowEvent | WindowListener | windowActivated() windowClosed() windowClosing() windowDeactivated() windowDeiconified() windowlconified() windowOpened() |

Заздалегідь визначені класи-адаптери називаються так само, як і інтерфейси, які вони реалізують, але в цих назвах слово Listener замінюється на слово Adapter: наприклад, MouseAdapter, WindowAdapter і таке інше.

Якщо реалізований інтерфейс слухача або отримані підкласи класу-адаптера і якщо створений екземпляр класу - конкретний об'єкт слухача подій, то цей слухач повинен бути підключений до відповідного джерела подій. У програмах джерелом подій завжди є який-небудь компонент. У методах підключення (реєстрації) слухачів подій використовуються стандартні угоди про імена: якщо джерело подій породжує події типу X, в ньому існує метод *addXListener* () для додавання слухача і метод *removeXListener* () для його видалення. У програмі вище, наприклад, це:

b.addActionListener (new MyActionEventListener ());

Тут для кнопки додали слухача; в дужках безіменний об'єкт класу *MyActionEventListener* - реалізація інтерфейсу *ActionListener* ().

this.addMouseListener (new MyMouseAdapter ());

В рядку вище для панелі-апплету додали слухача; в дужках безіменний об'єкт класу MyMouseAdapter - підклас MouseAdapter.

Для визначення типів подій, які можуть породжуватися даними компонентом, можна просто переглянути, які методи зареєстровані для його слухача подій. Наприклад, з опису API для об'єкта класу *Button* випливає, що він породжує події *ActionEvent*. У таблиці 2.7.2 наведено список компонентів і подій, які вони породжують.

Є також інший спосіб обробляти події - з використанням так званих *диспетчерів*. Він полягає в тому, що для всіх кнопок, наприклад, як це показано в прикладі нижче, реалізується інтерфейс *ActionListener*.

class Forwarder implements ActionListener{…}

В ньому в методі *actionPerformed (ActionEvent e)* викликається той метод, який буде відповідати даній кнопці. Таким чином, клас *Forwarder* використовується тільки як передавальне ланка до методів - обробників подій.

public class dispetcher extends JFrame {

public dispetcher () {

super ( "dispetcher");

setDefaultCloseOperation (EXIT\_ON\_CLOSE);

getContentPane (). setLayout (new FlowLayout ());

add (new JLabel ( "Applet!", JLabel.CENTER));

btn1 = new JButton ( "OK!");

getContentPane (). add (btn1);

btn2 = new JButton ( "Cancel!");

getContentPane (). add (btn2);

Таблиця 2.7.2. Компоненти і події, які вони породжують

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Породжена подія** | **Значення** |
| Button | ActionEvent | Користувач натиснув кнопку |
| CheckBox | ItemEvent | Користувач встановив або скинув прапорець |
| CheckBoxMenuItem | ItemEvent | Користувач встановив або скинув прапорець меню |
| Choice | ItemEvent | Користувач вибрав елемент списку або скасував його вибір |
| Component | ComponentEvent | Компонент або переміщений, або він став прихованим, або відображеним |
|  | FocusEvent | Компонент отримав або втратив фокус введення |
|  | KeyEvent | Користувач натиснув або відпустив клавішу |
|  | MouseEvent | Користувач натиснув або відпустив кнопку миші, або курсор миші увійшов або вийшов з області, займаній елементом, або користувач просто пересунув мишу або пересунув миша, утримуючи кнопку миші |
| Container | ContainerEvent | Компонент доданий в контейнер або видалений з нього |
| List | ActionEvent | Користувач виконав подвійне клацання мишею на елементі списку |
|  | ItemEvent | Користувач вибрав елемент списку або скасував вибір |
| MenuItem | ActionEvent | Користувач вибрав пункт меню |
| Scrollbar | AdjustmentEvent | Користувач здійснив прокрутку |
| TextComponent | TextEvent | Користувач змінив текст компоненту |
| TextField | ActionEvent | Користувач закінчив редагування тексту компоненту |
| Window | WindowEvent | Вікно було відкрите, закрите, представлено у вигляді піктограми, відновлено або потребує відновлення |

// Цей об'єкт буде передаватися як аргумент слухачам кнопки, а клас цього //об'єкта

// - це реалізація інтерфейсу, де метод actionPerformed перевизначений так, що

// викликає інші методи, в залежності від натиснутої кнопки

Forwarder forw = new Forwarder ();

btn1.addActionListener (forw);

btn2.addActionListener (forw);

pack ();

setVisible (true);}

JButton btn1, btn2;

class Forwarder implements ActionListener {

public void actionPerformed (ActionEvent e) {

if (e.getSource () == btn1) onOK (e);

if (e.getSource () == btn2) onCancel (e);

}

}

public void onOK (ActionEvent e) {

System.out.println ( "onOK");

}

public void onCancel (ActionEvent e) {

System.out.println ( "onCancel");

}

public static void main (String [] args) {

new dispetcher ();

}

}

Нижче наведено приклад неповної програми для обробки подій клацання на кнопці ОК.

okButton.addActionListener(new ButtonClickListener());

…

private class ButtonClickListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

String command = e.getActionCommand();

if( command.equals( "OK" )) {

statusLabel.setText("Ok Button clicked.");

} else if( command.equals( "Submit" ) ) {

statusLabel.setText("Submit Button clicked.");

} else {

statusLabel.setText("Cancel Button clicked.");

}

}

}

### 2.7.3 JavaFX

JavaFX – це платформа для створення насичених інтернет застосунків RIA - Rich Internet Application.

JavaFX пропонує багатий графічний та медійний API з підтримкою апаратних графічних прискорювачів та великий вибір нових компонентів: елементів керування, графіків, мультимедіа та вбудованого браузера.

JavaFX Runtime складається з набору бібліотек Java, що забезпечують інтерфейси сучасним стандартом, а також дозволяє отримати доступ до певних апаратних ресурсів.

Починаючи з версії Java SE 7 Update 6, JavaFX є частиною реалізації Oracle Java SE.

JavaFX поставляється з великим набором компонентів графічного інтерфейсу, таких як кнопки, текстові поля, таблиці, меню, діаграми і т.д.

JavaFX використовує стилі CSS, також можна використовувати спеціальний формат FXML для створення GUI, щоб, ти самим, відокремити ресурси графічного інтерфейсу від кода. Це полегшує швидке розміщення графічного інтерфейсу користувача або зміну зовнішнього вигляду чи композиції без необхідності внесення змін у коді.

JavaFX додатково працює з інтегрованою 3D-графікою, а також аудіо, відео та вбудованими мережевими застосунками. JavaFX простий у освоєнні та добре оптимізований. Він підтримує такі операційні системи, як Windows, UNIX системи та Mac OS.

Структура типового застосунку JavaFX представлена ​​трьома рівнями: *Stage*, *Scene, Node*.

Stage - це контейнер, що вміщує весь вміст GUI, всередині нього знаходиться щонайменше один елемент Scene. Усі елементи Scene зберігаються у вигляді так званого Scene Graph, тобто у вигляді ієрархії елементів, де кожен елемент - вузол, званий як Node.

Вузли Node - це елементи управління, наприклад, кнопки мітки, або навіть макети (layout), всередині яких може бути кілька вкладених компонентів. Вкладеність може бути багаторівневою, коли макети містять інші макети та звичайні компоненти.

Вузол Node має такі властивості:

* id (типу String),
* Parent (тільки один кореневий елемент),
* Scene,
* Стиль (styleClass, style),
* Видимість, активність, прозорість,
* Розміри (min, max, preferred),
* Границі (boundsInLocal, boundsInParent, layoutBounds),
* Трансформації (зсув, обертання, масштаб, нахил),
* Ефекти,
* Події (mouse, key, drag, touch, rotate, scroll, swipe, zoom).

**Основні структурні складові JavaFX-застосунку:**

* Головний клас JavaFX застосунку успадкований від **javafx.application.Application,**
* Головний метод застосунку   
   **public** **void** **start(javafx.stage.Stage** **primaryStage),**
* Клас **javafx.stage.Stage –** це графічний контейнер головного вікна  
   JavaFX- застосунку,
* Клас **javafx.scene.Scene** - це граф сцени, який складається  
  з кореневого вузла та його дочірніх елементів.

**Життєвий цикл JavaFX-застосунку:**

* Метод **launch()** – точка входу в JavaFX-застосунок,
* Створюється екземпляр класу **javafx.application.Application,**
* Викликається метод **init(),**
* Викликається метод **start(javafx.stageStage)** при створенні потоку застосунку,
* Викликається метод **stop().**

Дочірні вузли графу сцени додаються за допомогою методів **getChildren().add()** або **getChildren().addAll().**

**Властивості JavaFX-компонентів:**

Розрізняють «прості» властивості і властивості як класи-обгортки з простору імен javafx.beans, які реалізують інтерфейси Property і ReadOnlyProperty

Іменування властивостей:

* public *ТипВластивості* get*Ім’яВластивості(*);
* public void set*Ім’яВластивості* (*ТипВластивості* значення);
* public *ТипВластивості* *Ім’яВластивостіProperty*().

Деякі особливості JavaFX:

* Можна вбудовувати JavaFX-компоненти в Swing-форми. Використовується клас **public class JFXPanel extends javax.swing.Jcomponent**
* Механізм зв’язування (**binding**), призначений для зв’язування властивостей об’єкту   
  **public abstract class** LabeledBuilder<B **extends** LabeledBuilder<B>> **extends** ControlBuilder<B>

**Менеджери розташування компонентів на екрані у JavaFX:**

TilePane - елементи в сітці з «плиток» однакового розміру.

GridPane - елементи в гнучкій сітці рядків і стовпців.

StackPane - елементи в стеку в зворотному порядку.

BorderPane - притискає всі вкладені елементи до однієї зі сторін контейнера: зверху, зліва, справа, знизу і по центру.

AnchorPane - якірна панель дозволяє краї дочірніх вузлів прив'язувати зсувами до країв панелі.

FlowPane – розташовує елементи один за однм.

VBox - розташовує всі вкладені елементи в одному вертикальному стовпці.

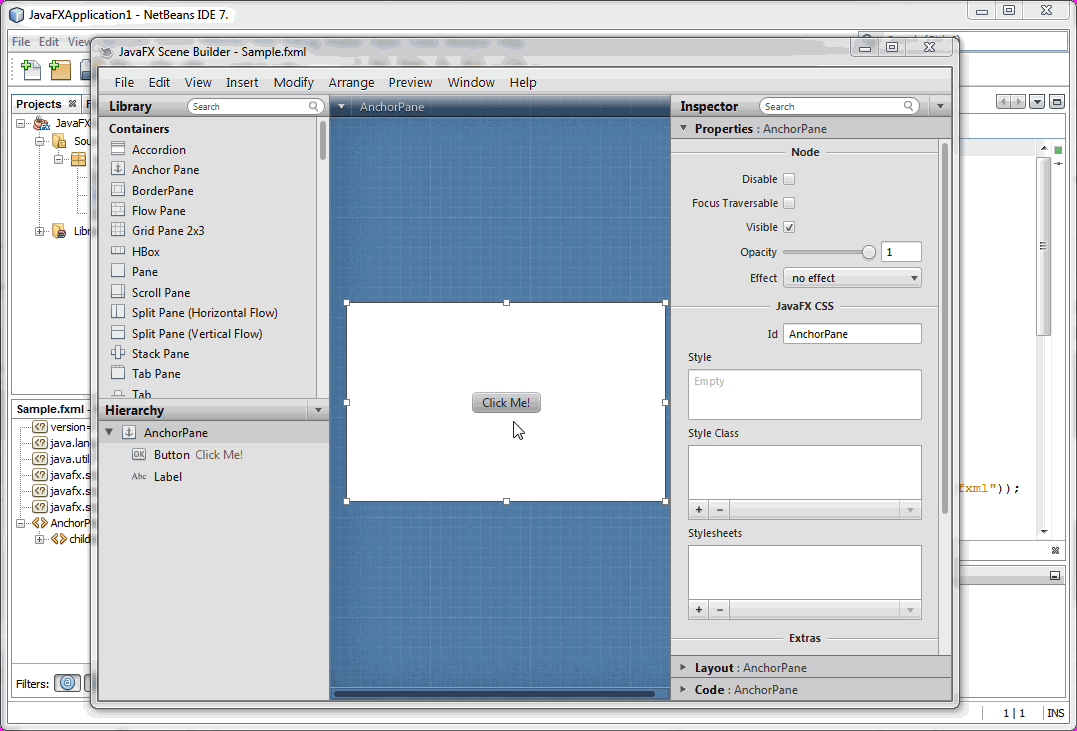
HBox – розташовує всі вкладені елементи в один горизонтальний ряд.

**Приклад застосунку**. У коді створюється кнопка, при натисканні на яку в консоль виводиться напис.

package helloworld;  
   
import javafx.application.Application;  
import javafx.event.ActionEvent;  
import javafx.event.EventHandler;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.layout.StackPane;  
import javafx.stage.Stage;  
   
public class HelloWorld extends Application {  
   
 @Override  
 public void start(Stage primaryStage) {  
 Button btn = new Button();  
 btn.setText("Say 'Hello World'");  
 btn.setOnAction(new EventHandler<ActionEvent>() {  
   
 @Override  
 public void handle(ActionEvent event) {  
 System.out.println("Hello World!");  
 }  
 });

StackPane root = new StackPane();  
 root.getChildren().add(btn);  
  
 Scene scene = new Scene(root, 300, 250);  
  
 primaryStage.setTitle("Hello World!");  
 primaryStage.setScene(scene);  
 primaryStage.show();  
 }  
 public static void main(String[] args) {  
 launch(args);  
 }

**JavaFX Scene Builder**



Метою візуального засобу розробки *SceneBuilder* є зручне та швидке генерування коду візуального інтерфейсу з використанням *fxml*-коду. Засіб візуальної розробки SceneBuilder і середовище розробки програм (наприклад, IntellijIDEA) може працювати єдиним цілим, що в процесі роботи дозволяє легко перемикатися з одного на інше. На виході ми отримаємо десктопну скомпільовану програму (це буде не веб-інтерфейс у браузері).

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що таке графічний інтерфейс користувача і для чого його створюють?
2. Які переваги має пакет Swing у порівнянні з AWT?
3. Які особливрсті пакету JavaFX?
4. Перелічити відомі вам компоненти (елементи керування) та їх властивості.
5. Перелічити відомі вам події, які пов’язані з компонентами.
6. Навести приклади подій, пов’язаних з компонентом «кнопка».
7. Пояснити, як розробити програму обробки події для певного компонента.
8. Які є способи розташування компонентів?

## Багатопотокове програмування - пакет java.util.concurrent

### Переваги пакету java.util.concurrent

Працювати з потоками засобами базової версії Java 1.0 незручно і повязано зі значною кількістю помилок. Недоліки синхронізації з базового пакету:

* неможливо перервати потік, який очікує блокування;
* неможливо опитувати або намагатися отримати блокування, не будучи готовим до довгого очікування;
* блокування повинно бути знято в тому ж стекового фреймі, в якому було розпочато.

Починаючи з версії 5.0, платформа Java має високорівневий API в пакеті *java.util.concurrent*для роботи з потоками виконання. Цей пакет вміщує кілька стандартних розширюваних фреймворків, а також класи, які забезпечують корисну функціональність і зручність в застосуванні.Він вміщує:

* Concurrent Collections
* Queues
* Synchronizers
* Executors
* Locks
* Atomics

Тут як і в низькорівневому пакеті можна використовувати **implements** *Runnable* для створення класу потоку, а в ньому повинен бути переписаний метод *run.* Тобто можна робити, наприклад, так:

*Transfer tr = new Transfer(b, i, balance);*

*Thread t = new Thread(tr);*

*t.start();*

*де*

class Transfer implements Runnable {

*…*

*//має метод run()*

*public void run() {*

*…*

Але при використанні *Concurrency* API для створення та керування потоком можна використати і додаткові засоби, які мають певні переваги (детально будуть розглянуті нижче): інтерфейс *Callable (*аналог *Runnable*, який повертає результат роботи, доступ до якого можна отримати через інтерфейс *Future* або клас-обгортку *FutureTask),* використання *пулу потоків* і його *виконавців* та інші.

В *java.util.concurrent* щоб заблокувати /розблокувати метод або його частину треба, скориставшись засобами пакету *Locks*, писати **lock/ unlock** (замість **synhronize**),щобпризупинити потік писати **await** (замість **wait),**  а щоб сповістити один або всі потоки треба писати **signal/signalAll (**замість ключових слів слова **notify /notifyAll** ).

Для створення блокування треба створити об'єкт:

private Lock bl = new ReentrantLock();

А потім можна встановити блокування, наприклад, на операції передачі коштів поки йде підрахунок загального балансу

bl.lock();

Після завершення підрахунку можна зняти блокування за допомогою

bl.unlock();

А між цими двома командами вставляється блок програми, який буде рахувати і поки він знаходить кінцевий результат розрахунку, то цей блок буде заблоковано для інших.

І так подібне блокування можна робити багато разів, тому клас і називається *ReentrantLock*.

Щоб ввести умову використання блокування, очикування або розблокування спочатку треба створити об'єкт класу ***Condition***:

private Condition con = bl.newCondition();

а потім писати

con.await();

//або

con.signal(); // або signalAll()

Пакет *Concurrency* API містить велику кількість корисних класів і методів для багатопотокового програмування.

Класи пакету реалізують такі інтерфейси:

• *Lock* - інтерфейс підтримує порядок блокування і дозволяє використовувати багаторазово пов'язаний умовний об'єкт *Condition*;

• *Condition* - інтерфейс описує пов'язані з блокуваннями змінні, які можуть виконувати функції монітора об'єкта;

• *ReadWriteLock* - інтерфейс підтримує пару пов'язаних блокувань: одну для читання і одну для запису. Одним з недоліків *synchronized* є те, що це ключове слово не забезпечує конкурентне читання, що зніжує продуктивність, особливо якщо воно використовується частіше ніж запис (оскільки читання безпечне з точки зору використання ресурсів). *java.util.concurrent.locks.ReentrantReadWriteLock* з API Lock забезпечує цю можливість.

Всі читачі можуть щось читати в пам’яті, а потім один записувач там може щось змінювати. Після цього всі читачі можуть знову повернутися до читання. Це застосовується коли багато читань і мало поновлень. Тоді буде виграш порівняно з звичайним *Reentrant***,** який буде блокувати на будь якому доступі – це буде ексклюзивний доступ. Треба буде писати код для читання і запису самостійно і окремо.

### Інтерфейси *Lock та Condition*

**Інтерфейс Lock**

*Lock* - це базовий інтерфейс в Lock API. Він забезпечує всі функції ключового слова *synchronized*, додаючи нові методи для зручної роботи.

Клас *ReentrantLock* - це конкретна і найбільш використовувана реалізація інтерфейсу *Lock*(аналог використанню ключевого слова *synchronized*).

При використанні ключового слова *synchronized* потік може бути заблокований в очікуванні "відкриття ресурсу" протягом невизначеного періоду часу і немає ніякого способу, щоб управляти цим очікуванням.

Інтерфейс *Lock* забезпечує різні можливості застосування блокування для управління очікуванням. Але програмісту треба не забувати виконувати розблокування. В наступному прикладі показано, як це можна зробити (в цьому прикладі *object* – об’єкт класу *ReentrantLock):*

object.lock(); // ввійти в монітор

try {

// фрагмент коду

} finally {

object.unlock(); //вийти из монітору

}

Ми використовуємо тут блок *try-finally*, щоб гарантувати, що блокування буде знято, навіть якщо фрагмент кодукине виняток.

*Lock* працює наступним чином: спочатку потік намагається отримати доступ до загального ресурсу. Якщо він вільний, то на нього накладає блокування. Після завершення роботи блокування з загального ресурсу знімається. Якщо ж ресурс не вільний і на нього вже накладено блокування, то потік очікує, поки це блокування не буде знято.

**Методи интерфейсу** *Lock*

* **lock ()** *–* чекати на отримання блокування*.* Якщо блокування не доступне, то поточний потік буде призупинено до тих пір, поки блокування не буде знято;
* **lockInterruptible ()** *-* чекати блокування, але надається можливість перервати очікуючий потік (для очікуючого блокування потоку викликається метод **interrupt** () і очікування буде перервано, а значить потік не робить спроб отримати доступ до захищеного ресурсу, а буде виконувати будь-які інші дії);
* **tryLock ()** *– намагається отримати блокування, але* якщо воно не доступне на момент виклику, то повертається ***false.***На відміну від методу **lock ()** - не очікує отримання блокування, а просто можна відмовитися від очікування і виконувати будь-які інші дії;
* **tryLock (long timeout, TimeUnit timeUnit)** *– призначений для очикування блокування на протязі певного часу* **timeout.** Якщо блокування не можна отримати після закінчення **timeout**, то повертається значення ***false***, що означає неможливість отримання блокування);
* **unlock ()** *-* зняти блокування*.*

**Клас ReentrantLock**

Клас *ReentrantLock*, який реалізує інтерфейс *Lock*, також, як і *synchronized*, забезпечує багатопоточність, але має додаткові можливості, пов'язані з опитуванням про блокування (lock polling), очікуванням блокування протягом певного часу і перериванням очікування блокування ( методи відповідного інтерфейсу описані вище). У перекладі *reentrant* може означати «той, що використовується повторно» ( повторний вхід). Тобто один і той же потік може декілька раз отримувати блокування. Але при цьому для того, щоб реально розблокувати необхідно вже буде стільки ж разів зняти блокування.

*ReentrantLock* слідкує**,** щоб в один блок коду не зайшло декілька потоків одночасно (на відміну, наприклад, від *Семафора*, який дозволяє не більше N потоків, а цей - тільки один потік).

Відмінність *ReentrantLock*від вбудованої конструкції синхронізації з ключовим cловом *synchronize* в тому, що методи *lock*і*unlock*дозволяють захватити потік в одному методі, а відпустити в іншому (*synhronize* такого не дозволяв; він працює тільки в середині одного методу.

В *ReentrantLock* з використанням *Condition* (див. нижче) єтакож розширені можливості методів *wait* i *notify*.

Тут також можна забезпечити справедливість черги, давши в конструктор параметр **true.**

*ReentrantLock* пропонує набагато більш високу ефективність функціонування в умовах жорсткого змаганя за ресурси. Іншими словами, коли кілька потоків намагаються отримати доступ до спільно використовуваного ресурсу, віртуальній машині JVM буде потрібно менше часу на встановлення черговості потоків і більше часу на її виконання.

**Інтерфейс *Condition***

Застосування умов блокування дозволяє домогтися контролю над управлінням доступом до потоків. Умова блокування являє собою об'єкт інтерфейсу *Condition* з пакета *java.util.concurrent.locks*.

Застосування об'єктів *Condition* багато в чому аналогічно використанню методів *wait / notify / notifyAll* класу *Object*:

* *await*: потік очікує, поки не буде виконана деяка умова і поки інший потік не викличе методи *signal / signalAll*.
* *signal*: сигналізує, що потік, у якого раніше був викликаний метод *await ()*, може продовжити роботу.
* *signalAll*: сигналізує всім потокам, у яких раніше був викликаний метод *await ()*, що вони можуть продовжити роботу.

Ці методи викликаються з блоку коду, який потрапляє під дію блокування *ReentrantLock.*

Таким чином, блокування *Lock* замінює використання *synchronized*. *Condition* в поєднанні з блокуванням *Lock* дозволяє замінити методи монітора / мьютекса (*wait, notify і notifyAll*) об'єктом, керуючим очікуванням подій.

Щоб отримати умови *Condition* для блокування *Lock* використовують метод *newCondition* ().

**Порівняння використання *Lock* та *synchronized***

На одному блоці може бути декілька *Conditions***.** Наприклад, в чергу щось можна додати, якщо вона пуста і щось з неї забрати, якщо вона не пуста. Можемо для цього створити два *Conditions***,** а не будити всі потоки, потім перевіряти і знову усипляти певні (як це будо у *wait* і *notify* )

Приклад.

// Conditions зв’язаний з Lock ,

Lock lock=new ReentrantLock ();

Condition сondition= lock.newСondition ();

lock.lock();

try {

While(..){

condition.await();

} finally

{ lock.unlock();}

//щось ще блокуємо в цій програмі

lock.lock();

try{condition.signal();

}finally{lock.unlock();}

Нижче показано два приклад програми: з використанням *synchronized* і більш ефективний аналог з використанням *Lock*

У нас є клас Resource для роботи з базою даних з потокобезпечним методом *doSomething*() і методом *doLogging*(), де потокобезпечність не вимагається.

public class Resource {

public void doSomething(){

        // якась робота з базою даних

    }

   public void doLogging(){

       // потокобезопечність для логування нам не потрібна

    }

}

З використанням *synchronized*

public class SynchronizedLockExample implements Runnable{

// екземпляр класу Resource для роботи з його методами

private Resource resource;

public SynchronizedLockExample(Resource r)

//конструктор присвоює ресурсу передане значення r

{

this.resource = r;

}

@Override

public void run() {

synchronized (resource) {

//блокується ресурс для виконання дій методом doSomething

resource.doSomething(); // потокобезопечність потрібна

}

resource.doLogging(); // потокобезопечність не потрібна,

//а чекати розблокування можно довго

}

З використанням *lock*

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class ConcurrencyLockExample implements Runnable{

private Resource resource;

private Lock lock;

public ConcurrencyLockExample(Resource r){

this.resource = r;

this.lock = new ReentrantLock();

}

@Override

public void run() {

try {

// намагаємось взяти блокування на протязі 10 секунд

if(lock.tryLock(10, TimeUnit.SECONDS)){

resource.doSomething();

}

} catch (InterruptedException e) {

e.printStackTrace();

}finally{

//знімаємо блокування

lock.unlock();

}

// Для логування не потрібна потокобезпечність

resource.doLogging(); }}

//Клас з методом main

public class ConcurrencyLockEx {

public static void main(String[] args) {

Resource res = new Resource ();

SynchronizedLockEx t=new SynchronizedLockEx(res);

Thread myThread = new Thread(t);

myThread.start();

ConcurrencyLockExample t1=new ConcurrencyLockExample(res);

Thread myThread1 = new Thread(t1);

myThread1.start();

}

}

Порівняння *synchronized і lock*для поточного прикладу.

Як видно з програми, ми використовуємо метод *tryLock*(), щоб переконатися в тому, що потік чекає лише певний час (10 секунд). Якщо він не отримує блокування на об'єкт, то просто виконує логування і виходить з методу.

Одним з цікавих методів інтерфейсу Lock і його реалізації *ReentrantLock* є запит на блокування з можливістю переривання процесу очікування. Тобто якщо потік запитує блокування методом *lockInterruptibly ()* і не отримує його відразу ж, то переходить в стадію очікування. Методом *interrupt* роботу потоку можна перервати. Тоді потік, який очікує блокування прокидається, і генерується виняткова ситуація *InterruptedException*. Після цього спроба отримати доступ до захищеного ресурсу (отримання блокування) не робиться і звільняти блокування не потрібно. Нижче представлений приклад використання блокування *lockInterruptibly*. Структура коду використання блокування *lockInterruptibly* має наступний вигляд:

//Приклад використання методу lockInterruptible () .

Lock l = new ReentrantLock();

try { l.lockInterruptibly();

try {

// робота с захищеним ресурсом

}

finally {

l.unlock();

}

}

catch (InterruptedException e)

{

System.err.println("Interrupted wait");

}

Внутрішній блок *try-finally* отримує блокування і доступ до захищених ресурсів; після завершення роботи блокування знімається.

Зовнішній блок *try-catch* обробляє виняткові ситуації запиту блокування. Якщо очікуючий потік перерваний в результаті виняткової ситуації (методом *interrupt* ), то виконується перехоплення *catch (InterruptedException)* і метод зняття блокування *unlock* не викликається.

**Загальні рекомендації по використанню *Lock***

* *Lock API* забезпечує більше можливостей для *очікування* блокування/розблокування, на відміну від *synchronized*, де потік може нескінченно очікувати монітору. У *Lock API* ми можемо використовувати метод *tryLock (),* щоб очікувати блокування тільки певний час.
* У разі використання *Lock API* ми *самі повинні знімати блокування* (на відміну від *synchronized,* де воно автоматично знімається після виходу з цього блоку або методу). Тому ми змушені використовувати блок **try-finally**, де в блоці **finally** ми пишемо *unlock*  і тим гарантуємо, що блокування буде обов’язково знято.
* Блоки синхронізації *synchronized* можуть застосовуватися *лише в одному методі*, в той час як *Lock API* дозволяє отримати *Lock* в одному методі, а зняти його в іншому.
* Об’єкт класу *String* є незмінним об'єктом, і будь-який рядок зберігається в пулі рядків. Якщо будь-яка інша частина коду буде використовувати той же рядок *String*, що і заблокована частина коду, вони обидві будуть заблоковані на одному і тому ж об'єкті, незважаючи на те, що будуть абсолютно не пов'язані, що може привести до несподіваного результату. Тому замість об'єкта *String* рекомендується для синхронізації використовувати *new Object ()*.
* В Java немає *synchronized* змінної, замість неї ви можете використовувати *volatile*  змінну, яка дасть вказівку JVM зчитувати її значення з основної пам'яті, а не кешувати її локально. Починаючі з Java 5 після зміни моделі пам'яті мови читання та запис є *атомарними* для всіх змінних, оголошених за допомогою ключового слова *volatile*  (включаючи змінні *long* та *double*). Простий доступ до атомарних змінних є більш ефективним замість доступу до цих змінних за допомогою синхронізованого коду. Але це вимагає більше уваги з боку програміста, щоб уникнути помилок узгодженості пам'яті.
* *Блокова* синхронізація в Java є кращою порівняно з синхронізацією *методів*, оскільки, використовуючи блокову синхронізацію, потрібно лише заблокувати критичну частину методу, а не весь метод. Оскільки це впливає на продуктивність, потрібно прагнути синхронізувати лише *минимально* необхідну частину коду.

### Інтерфейс *Callable*

В пакет *java.util.concurrent* входить інтерфейс *Callable (з методом call* замість *run)*, який схожий на *Runnable*. Основна відмінність полягає в тому, що *Callable* надає можливість повернути результат роботи методу *сall* в потік, який цей *Callable* запустив. Доступ до результату можна отримати через об'єкт типу **java.util.concurrent.Future**. Його метод **get (**) - встановлює блокування поки очікує завершення завдання *Callable*, щоб отримати результат або виключення в ситуації, якщо в процесі виконання завдання сталася помилка. Для зручності роботи з потоками різного типу (*Callable* і *Runnable*) створений клас **FutureTask**, який представляє собою клас-обгортку для класу **Future**.

Основні відмінності *Callable* та *Runnable*:

* Інтерфейс *Runnable* з'явився в Java 1.0, а інтерфейс *Callable* був введений в Java 5.0 в складі пакету *java.util.concurrent*.
* Класи, що реалізують інтерфейс *Runnable* повинні реалізовувати метод *run* () для виконання завдання. Класи, що реалізують інтерфейс *Callable* повинні реалізовувати метод *call* () для виконання завдання.
* Метод *Runnable*.*run* () не повертає ніякого значення, його тип *void*, а метод *Callable*.*call* () може повертати значення типу T. Інтерфейс *Callable* є параметризованим: <T> - тип значення, яке буде повертатися в методі *call* () задається цим параметром T.
* Метод *run* () не може кинути виключення, яке перевіряється, в той час як метод *call* () може кинути таке виключення. Інтерфейс *Runnable* взагалі не міг виконати викидання контрольованих виключень, а викидання неконтрольованого (*RuntimeException*) виключення приводило до зупинки потоку і всього застосунку.

Приклад (схематично):

public class CounterMath implements Callable <Integer> {

…

public Integer call() {

…

//Отримує список файлів директорії і підраховує кількість таких //файлів, у яких зустрічається певне слово, а потім повертає цю //кількість **count**

…

return count;

}

В класі *Main* цей потік-клас-рахівник упакується у клас **FutureTask** і уже цей потік можна стартувати. Послідовність команд така:

CounterMath counter = new CounterMath(new File(dir), word);  
FutureTask<Integer> task = new FutureTask <Integer> (counter);  
new Thread(task).start();  
try {  
 System.out.println(task.get() + " files.");// Тут чекати і отримати результат

} catch (ExecutionException | InterruptedException e) {  
 e.printStackTrace();  
}

}

Тут головна перевага - тіло потока повертає результат своєї роботи. Доступ до результату потока **Callable** - через Інтерфейс ***Future*** або ***FutureTask***

//Інтерфейс Callable (асинхронна робота з поверненням результату)

public interface Callable <T> {

// Тіло потока, який повертає результат своєї роботи

T call() throws Exception;

}

Інтерфейс Future (доступ до результату потока Callable)

public interface Future <T> {

T get();

T cancel(boolean mayInterrupt);

boolean isDone();

…………………….

}

Потоки (завдання) *Callable* повертають результат, доступ до якого можна отримати через об'єкт типу *java.util.concurrent.Future*:

* метод *get* () - встановлює блокування поки очікує завершення завдання *Callable*, щоб отримати результат або виключення ситуації, якщо в процесі виконання завдання сталася помилка;
* метод *cancel* () - скасовує обчислення, якщо завдання ще не стартувало і вже не буде запущене, і якщо обчислення вже йде, то воно переривається;
* метод *isDone* () – перевіряє, чи закінчено вже виконання завдання.

Для зручності роботи з потоками різного типу (*Callable* і *Runnable*) створений клас *FutureTask*, який представляє собою клас-обгортку.

Розглянемо більш детально схематичний приклад (див. вище) обробки файлової системи через потоки типу *Callable*:

* Необхідно підрахувати кількість файлів у вказаній директорії і її піддиректоріях, які містять деяке ключове слово;
* Є клас-потік *CounterMath*, який визначає всі елементи зазначеної директорії і обробляє файли, а для обробки піддиректорії запускає новий потік (тобто кожна директорія обробляється в окремому потоці);

Клас Main буде запускати завдання і збирати результат.

public class CounterMath implements Callable <Integer> {

private File dir; private String word;

public CounterMath(File dir, String word) {

this.dir = dir;

this.word = word;

}

public boolean search(File ff) {

try (Scanner sc = new Scanner(new FileInputStream(ff))) {

boolean flag = false;

while ( !flag && sc.hasNextLine()) {

String str = sc.nextLine();

if (str.contains(word))// Поки не знайдено слово прапор буде - false

flag = true;

}

return flag;

} catch (IOException e) { return false; }

}

// …

public Integer call() {  
 int count = 0;  
 try {  
 File[] files = dir.listFiles();//Отримати список елементів директорії

ArrayList< Future<Integer> > result = new ArrayList<>();  
 for (File ff : files)  
// Якщо елементом є директорія, то створити новий потік і зв'язати його з //FutureTask для отримання результату

if (ff.isDirectory()) {

CounterMath counter = new CounterMath(ff, word);  
 FutureTask<Integer> task = new FutureTask <Integer> (counter);  
 result.add(task);  
 new Thread(task).start();  
 }

// Якщо елементом є файл, то здійснити пошук: якщо слово присутнє, то //збільшити лічильник

else if ( search(ff) )  
 count++;  
 for (Future<Integer> rez : result)// Зібрать результати з усіх піддиректорій

count += rez.get();  
 } catch (ExecutionException | InterruptedException e) {  
 e.printStackTrace();  
 }  
 return count;  
}

}

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);  
System.out.print("Enter directiry -> ");  
String dir = sc.next();  
System.out.print("Enter keyword -> ");  
String word = sc.next();  
CounterMath counter = new CounterMath(new File(dir), word);//створити потік  
//Упакувати потік у тип FutureTask

FutureTask<Integer> task = new FutureTask <Integer> (counter);//  
new Thread(task).start();  
try {  
 System.out.println(task.get() + " files.");// Чекати і отримати результат

} catch (ExecutionException | InterruptedException e) {  
 e.printStackTrace();  
}

}

}

**Результат**

Enter directory -> d:\Temp\Example\

Enter keyword -> mov

14 files.

### Пул потоків та виконавці

*Пул потоків* - це група попередньо створених потоків, яка чекає завдань і виконує їх. Ідея полягає в тому, щоб завжди існували потоки, і нам не доводилося сплачувати накладні витрати для їх створення щораз. (Наприклад, це ефективно, при виконанні короткострокових серверних запитів, де число / виникнення клієнтських запитів не може бути визначено / передбачене). При цьому може бути якийсь час, коли взагалі немає завдань на виконання. Створений на початку пул надасть можливість всім його потокам постійно працювати поки ми всі не їх закриємо по *shutdown*.

Переваги використання пулу потоків:

* Підвищення продуктивністі, коли потоки повторно використовуються (серверний застосунок, що працює за принципом «потік-на-запит» буде не ефективним). Системні витрати створення нового потоку для кожного завдання значні (тобто буде витрачатися більше часу і споживатися більше системних ресурсів на створення та знищення потоків, ніж на їх виконання). (У низькорівневому пакеті потік не можна використовувати повторно);
* Більш ефективне використання пам'яті. Створення занадто великої кількості потоків в одній JVM може привести до нестачі системної пам'яті або пробуксовці через надмірне споживання пам'яті (необхідні заходи по обмеженню кількості завдань, що обробляються в заданий час). Значить необхідні заходи по обмеженню кількості завдань, що обробляються в заданий час – пул якраз це і робить, тому що він може працювати тільки з певним встановленим максимальним числом завдань;
* Краще побудовання програми: відмежування коду по створенню потоків і управління ними від іншої частини програми, а саме від виконуваних ним завдань типу *Runnable і Callable*.
* Тут не треба давати команду *start()* для запуску потоку типу *Runnable і Callable*(замість неї використовують декілька різних представників інтерфейсу *Executor*, які мають деякі додаткові можливості порівняно зі*start(*)**.**

У випадку створення нового потоку кожен раз при виконанні запиту клієнта, маємо два основних недоліки:

1) Затримка часу виконання для створення потоку. Створення потоку вимагає деякого часу, тому фактичне завдання не починається зразу, як тільки приходить запит і клієнт може помітити цю невелику затримку. Цей критерій має вирішальне значення в інтерактивних системах, де клієнт чекає негайних дій.

2) Неконтрольоване використання системних ресурсів: при створенні потоків споживають системні ресурси, тому система може вичерпати їх в разі великої кількості клієнтських запитів.

Пули потоків вирішують зазначені проблеми за рахунок:

1) Створення зазначеної кількості потоків завчасно (наприклад, при запуску сервера бази даних) замість їх створення під час виконання запитів. Оскільки коли приходить завдання потік вже існує, то затримка, яка відбувається через створення потоку, усувається (тобто застосунок швидше реагує).

2) Обмеження кількості потоків, що виконуються в будь-який момент часу.

Можна вважати, що потоки *thread* є одиничними працівниками, а *пули* потоків - групою працівників. Ви можете створювати кілька груп потоків для різних потреб. Наприклад, один пул може бути для виконання задач загального призначення, таких як фонові задачі, надсилання по електронній пошті і т.п., інший пул може використовуватися для одночасної обробки транзакцій, які не треба затримувати і так далі.

*Пул потоків* - це набір існуючих робочих потоків, який відділений від виконуваних ним завдань типу *Runnable* і *Callable*, який використовується для виконання декількох завдань одночасно.

Якщо правильно налаштувати кількість потоків в пулі, запобігаєся пробуксування ресурсів (якщо кількість завдань виходить за певні межі, то задача очікує до тих пір, поки потік не стане доступним, щоб її виконати).

*Concurrency* API вводить поняття *сервісу-виконавця (ExecutorService)* - високорівневу заміну засобам роботи з потоками безпосередньо. Є різні види виконавців. Всі вони можуть виконувати завдання асинхронно і зазвичай використовують пул потоків. Кожен потік з пулу може бути використаний повторно після виконання свого завдання. Можна виконати в застосунку кілька потоків-завдань, використовуючи тільки одного виконавеця.

**Інтерфейси і класи виконавців** (Executor)

Компоненти застосунків з пакету *Concurrency*, які інкапсулюють функції створення та управління потоками, називаються *виконавцями (Executors).*

Інтерфейси: *Executor, ExecutorService, ScheduledExecutorService.*

(Назва одного з них буде в лівій частині команди при створенні пулу потоків).

Класи: *ThreadPoolExecutor, ScheduledThreadPoolExecutor, ForkJoinPool*.

Кожен з наступних інтерфейсів в цьому списку має додаткові можливості порівняно з попереднім.

*Executor* - це простий інтерфейс, який підтримує запуск нових завдань;

*ExecutorService* - це підінтерфейс Executor, який додає функції, що допомагають керувати життєвим циклом, як завдань, так і самого виконавця.

*ScheduledExecutorService* - це підінтерфейс ExecutorService, який підтримує відкладене і / або періодично виконуване завдання.

*Executor* надає єдиний метод *execute*(), призначений, щоб бути заміною **Start (**для запуску потоку засобами низького рівня), тобто якщо ***rr*** - це об'єкт типу *Runnable*, а ***ex*** - це об'єкт типу *Executor*, тоді:

**ex.execute (rr);**

- це може бути те ж саме, що і **new Thread (rr)). Start ();** - створення і негайний запуск потоку на низькому рівні; або, швидше за все, розміщення ***rr*** в черзі для очікування доступності робочого потоку, тому що всі робочі потоки пулу можуть бути заняті.

Є різні варіанти створення виконавця для пулу. Найпростіший спосіб створити виконавця - використовувати фабричні методи класу **java.util.concurrent.Executors.**

Наприклад,

ExecutorService pool = Executors.newFixedThreadPool(4);

Це створення пулу потоку як об'єкта класу **ThreadPoolExecutor**, в якому буде максимум 4 потока і відповідно одночасно може виконуватися не більше 4 завдань.

Після цього (або до цього) треба створити клас (і його об'єкт) типу *Runnable* або *Callable*, типу

public class MyTestCallable implements Callable<String> {

...

public String call() {

…

return "work " + workNumber;

}

}

А потім

int numOfWorks = 20;

MyTestCallable works[] = new MyTestCallable[numOfWorks];

Future[] futures = new Future[numOfWorks];//щоб отримати результат

//від Callable (пояснення див.вище)

І нарешті в циклі створити 20 завдань типу MyTestCallable, які будуть подаватися в пул за допомогою його методу *submit*:

for (int i = 0; i < numOfWorks; ++i) {

works[i] = new MyTestCallable(i + 1);

futures[i] = pool.submit(works[i]);

}

Отримання результату по завершенню завданьтакож в циклі:

for (int i = 0; i < numOfWorks; ++i) {

try {

System.out.println(futures[i].get() + " ended");

} catch (Exception ex) {

ex.printStackTrace();

}

}

Універсальний метод *submit()* приймає і запускає на виконання завдання і типу *Runnable* і типу *Callable*:

* повертає об'єкт типу *Future*, який використовується для отримання значення, що повертається завданням типу *Callable* і управління станом обох завдань *Callable* і *Runnable*;
* Надає декілька методів для управління завершенням роботи виконавця:
* метод *shutdown ()* - очікує завершення запущених завдань;
* метод *shutdownNow ()* - зупиняє виконавця негайно, перериваючи запущені завдання.
* Забезпечує методи для подачі великих колекцій об'єктів *Callable*:
* метод *invokeAll (...)* - викликаючий потік блокується до завершення всіх переданих завдань;
* метод *invokeAny (...)* - викликаючий потік блокується до завершення будь-якого завдання.

Метод *execute* також як і *submit* подає завдання на виконання в пул потоків, але різниця між ними в наступному:

* *execute* – з інтерфейсу *Executor* (може приймати задачі типу *Runnable* , нічого не повертає);
* *submit* – з інтерфейсу *ExecutorService* (може приймати задачі типу *Runnable* і *Callable* і повертає об'єкт типу *Future*).

*ScheduledExecutorService* надає можливість відкладати початок виконання завдань на певний проміжок часу, а також планувати виконання завдань через заданий часовий інтервал:

* метод *schedule* (...) - виконує завдання типу *Callable* і *Runnable*, які стають доступними після зазначеної тимчасової затримки;
* метод *scheduleAtFixedRate (...)* - виконує завдання типу *Runnable*, з вказаною початковою затримкою і повторює його через вказаний інтервал часу (завершення відбудеться по завершенню завдання, інтервал повторення не враховує час виконання завдання);
* метод *schedule WithFixedRate (...)* - виконує завдання типу *Runnable*, з вказаною початковою затримкою і повторює його через вказаний інтервал між закінченням одного виконання і початком наступного.

Таким чином, для використання пулу потоків треба виконати наступні кроки:

* Створіть потік-завдання (об'єкти типу *Runnable* і *Callable*);
* Створіть пул потоків необхідного типу;
* Передайте завдання пулу потоків викликом відповідного методу;
* Збережіть посилання на об'єкт типу *Future* для можливості відслідковувати прогрес виконання завдання (при необхідності);
* Завершіть роботу з пулом потоків (наприклад, викликавши метод *shutdown* ()).

Пул потоків представляється через об’єкти класів: *ThreadPoolExecutor, ScheduledThreadPoolExecutor, ForkJoinPool*:

* ThreadPoolExecutor – реалізує інтерфейс ExecutorService
* ScheduledExecutorService - реалізує интерфейс ScheduledExecutorService
* ForkJoinPool (з Java 8), доступний через ForkJoinPool.commonPool(). Цей пул створює кількість потоков у відповідності до кількості ядер процесора. Якщо, наприклад, їх 4, то значить буде використано три потоки:

System.out.println(ForkJoinPool.getCommonPoolParallelism());// буде 3 потоки

Тим самим ми можемо краще балансувати навантаження на процесори. Це значення може бути спеціально змінено за допомогою параметра JVM.

*ForkJoinTask* - це варіант *ExecutorService*, який, наприклад, можна застосувати для виконання задач, які можуть динамично породжувати виконувати підзадачі. Типовою задачею може бути сортування злиттям, коли спочатку задача розбивається на дві підзадачі, потім ще на дві, після виконання яких їх результати зливаються. Вся декомпозиція може відбуватися в середині ForkJoinPool

**Виконавець користувача**

Якщо жоден з виконавців, що надаються фабричними методами, не задовольняє ваші потреби, то будуйте *ThroeadPolExecutor* або *ScheduledThreadPoolExecutor* самостійно. Один з параметрів конструктора такого виконавця є *workQueue* - черга для утримання завдань, перш ніж вони виконаються (ця черга буде містити тільки завдання *Runnable* на подачу методу *execute* ()). Черга повинна бути одним з типів блокуючих черг(про такі черги читай нижче). Наприклад, можна створити виконавця з такими параметрами:

ThreadPoolExecutor tpe = new ThreadPoolExecutor( 5, 10, 30L, TimeUnit.SECONDS,  
 new LinkedBlockingQueue<Runnable>());

Тут такі параметри:

* corePoolSize - число робочих потоків пулу, які зберігаються навіть якщо вони простоюють;
* maximumPoolSize - максимальна кількість потоків в пулі;
* keepAliveTime - коли число потоків більше ніж використовується, то це максимальний час, який недіючі потоки будуть чекати нових завдань перед завершенням;
* unit - одиниця виміру часу для параметра keepAliveTime;
* workQueue - черга для утримання завдань, перш ніж вони виконаються (ця черга буде містити тільки завдання Runnable на подачу методу execute ()).

### Блокуючі черги

*Блокуючі черги* - інструмент для координації дій кількох потоків. (Наприклад, для завдань коли одні потоки створюють дані, а інші їх використовують**). Черга підтримує обмеження no розміру (кількості елементів в черзі) та операцію очікування.**  Типова задача для таких черг «постачальник-споживач».

Інтерфейс *BlockingQueue* визначає блокуючу чергу, яка успадковує властивості інтерфейсу *Queue*, в якому елементи зберігаються в порядку «перший прийшов, перший вийшов» (*FIFO* - *first in, first out*). Реалізація даного інтерфейсу забезпечує блокування потоку в двох випадках:

* при спробі отримання елемента з порожньою черги;
* при спробі розміщення елемента в заповненій черзі.

Коли потік намагається отримати елемент з порожньої черги, то він переводиться в стан очікування до тих пір, поки який-небудь інший потік не розмістить елемент в черзі. Аналогічно при спробі покласти елемент в повну чергу; потік ставиться в очікування до тих пір, поки інший потік не забере елемент з черги і, таким чином, не звільнить місце в ній. Природно, поняття "повна черга" має на увазі обмеження розміру черги.

**Класи для створення блокуючої черги**

* Клас *ArrayBlokingQueue* (обов'язково вказати розмір)
* Клас *LinkedBlokingQueue* (необмежений розмір) – односпрямована черга на звязаних вузлах
* Клас *LinkedBlokingDeque* (необмежений розмір) – двоспрямована черга на звязаних вузлах
* Клас *PriorityBlokingQueue* (необмежений розмір) реалізація черги на основі [*PriorityQueue*](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html)
* Клас *DelayQueue* (необмежений розмір) - реалізує інтерфейс *Delayed*;
* Клас [*SynchronousQueue*](http://java-online.ru/concurrent-queue-block.xhtml) ( черга без ємкості - операція додавання одного потока знаходиться в очикуванні відповідної операції видалення в іншому потоці)
* Клас [*LinkedTransferQueue*](http://java-online.ru/concurrent-queue-block.xhtml) ( реализація черги на основі інтерфейса TransferQueue)

Таблиця 2.8.1. Основні методи інтерфейса BlockingQueue

|  |  |
| --- | --- |
| **Метод** | **Опис** |
| boolean add(E e) | Негайне додавання елемента в чергу. |
| boolean contains(Object o) | Перевірка наявності об'єкта в черзі; якщо об'єкт знайдений в черзі метод поверне true. |
| boolean offer(E e) | Негайне розміщення елемента в черзі при наявності вільного місця. |
| boolean remove(Object o) | Видалення об'єкта з черги, якщо він в ній присутній. |
| E take() | Отримання з видаленням елемента з черги, при необхідності очікування поки елемент не стане досяжним. |
| void put(E e) | Розміщення елемента в черзі, очікування при необхідності звільнення вільного місця. |
| int remainingCapacity() | Отримання кількості елементів, яке можна розмістити в черзі без блокування. |

**Основні конструктори LinkedBlockingQueue**

LinkedBlockingQueue()

LinkedBlockingQueue(int capacity) LinkedBlockingQueue(Collection<? extends E> c)

Перший конструктор створює порожню чергу фіксованою ємності.

Другий конструктор створює чергу з фіксованою ємністю capacity.

Третій конструктор створює чергу з набором елементів. Якщо використовується конструктор без вказівки ємності черзі, то використовується значення за замовчуванням Integer.MAX\_VALUE.

* Класс *LinkedBlockingQueue* і його iterator реалізують всі опціональні методи Collection і Iterator.
* Метод *toArray()* повертає масив елементів черги типу *Object[]*.  *Object[]*.

Створення блокуючої черги за допомогою класу*ArrayBlokingQueue:*

* реалізує класичний буфер обмеженого розміру FIFO - «першим прибув - першим вибув».
* Нові елементи вставляються в хвіст черги; операції вилучення - з голови черги.
* Створювана ємність черги не може бути змінена. Спроби вставити (*put*) елемент в повну чергу призведе до блокування роботи потоку; спроба взяти (*take*) елемент з порожньою черзі також блокує потік.

Особливості створення блокуючої черги за допомогою класу*DelayQueue:*

* вибираються тільки ті елементи, для яких час затримки закінчився;
* розташовуються елементи за спаданням терміну закінчення затримки;
* якщо ні у одного елемента в черзі час затримки не закінчився, черга блокується для вибірки.

Для черги треба створити клас типу *Runnable* (один з параметрів цього класу має бути блокуючи черга) з методом *run*, і в цьому методі *run* вставити в чергу певний об’єкт для подальшої обробки в потоці. Це вставлення в чергу виконується методом *put*(або схожими *add* чи *offer*), а взяття з черги методом *take*. Тобто після створення черги

BlockingQueue <File> que = new ArrayBlockingQueue <File> (10);

і створення потоку типу *Runnable*

FileRunTask running = new FileRunTask(que, new File(dir));

який вставляється в чергу *que*, далі запускаємо потік на виконання

new Thread(running).start();

Якщо останній оператор виконати 5 раз в циклі, то ми запустимо 5 потоків на виконання.

Вигідно використовувати чергу, коли одночасно декілька потоків вводять, наприклад, файли в чергу, і декілька потоків їх з черги вибирають, обробляють і виводять результуючий файл, наприклад, на консоль, або записують в іншу папку. Тут черга буде буфером, яка дозволяє обробляти файли в певному порядку (перший зайшов-перший вийшов), крім того може утримувати від створення великої кількості завдань, які споживачі не встигатимуть обробляти. В нашому прикладі, якщо обробка файлів складна, то можна запустити один потік на введення в чергу і 2– на виведення (якщо наприклад є 4 ядра в процесорі), або 2 - на введення в чергу і 4 – на виведення. А в черзі може зберігатися наприклад 10 задач. Якщо обробники-спочивачі будуть не встигати вибирати нові задачі по команді *take* і черга буде повна, то нові задачі будуть чекати коли звільнится місто в черзі. Якщо навпаки черга буде пуста, то *take* буде чекати на нові завдання. Таким чином можна підібрати кількість вхідних і вихідних потоків, щоб вони працювали з мінімальними очикуваннями.

### Синхронізуючі класи-колекції

Класи *java.util.ArrayList* і *java.util.HashMap* не забезпечують безпечну роботу з потоками. Для реалізаціїї потокобезпечної роботи з даними класами, починаючи з версії JDK 1.2 була додана можливість створювати потокобезпечні обгортки (*synchronization wrapper*) для класів-колекцій.

Створені обгортки виконують роль декораторів, в яких методи роботи з елементами колекцій, такі як *get* і *put,* синхронізовані за допомогою блокування.

Методи типу *synchronizedСollection*() можна застосувати до певних звичайних однопотокових об`єктів, після цього вони поміщаються в обгортку, яка дозволяє зробити всі їх методи синхронізованими, тобто з ними можна працювати декільком потокам, але вони будуть блокуватися на кожну операцію. Вони дадуть змогу обробляти колекцію іншим ітераторам після того, як попередні в колекціях щось змінили.

Наприклад, для створення потокобезпечної обгортки для списку можна застосувати статичний метод *synchronizedList* класу java.util.Collections.

Класи з пакету *Concurrent* більш ефективні, ніж повністю синхронізовані колекції *java.util.Collections.synchronizedСollection().* Ось їх список:

* ConcurrentLinkedQueue
* ConcurrentHashMap
* ConcurrentSkipListMap
* ConcurrentSkipListSet
* CopyOnWriteArrayList
* CopyOnWriteArraySet

**Клас ConcurrentLinkedQueue**

Якщо є клас з одним методом для отримання елементів з черги і іншим методом для додавання елементів в чергу, методи не потрібно синхронізувати, і вам не потрібно визначати будь-які методи; вони вже знаходяться в *ConcurrentLinkedQueue*.

Типова задача для черги – виробник і споживач. Споживач буде брати те, що поставляє виробник. Якщо товару немає, то потік споживача очикує. Якщо споживач переповнений, то виробник очикує. У *ConcurrentLinkedQueue* не можно організувати очікування*.* Томудля такої задачі можно скористатися блокуючою чергою *java. util. concurrent. BlockingQueue <E>*(див. відповідний розділ книги).

**Клас ConcurrentHashMap**

До моменту появи *ConcurrentHashMap* Java-розробники потребували подальшої реалізації хеш-карти:

* Потокобезпечність;
* відсутність блокувань всієї таблиці на час доступу до неї;
* бажано, щоб були відсутні блокування таблиці при виконанні операції читання.

Після появи *ConcurrentHashMap* з'явились додаткові можливості:

* карта має схожий з hashmap інтерфейс взаємодії;
* операції читання не вимагають блокувань і виконуються паралельно;
* операції запису часто також можуть виконуватися паралельно без блокувань;
* при створенні вказується необхідний *concurrencyLevel*, який визначається за статистикою читання і запису;
* елементи карти мають значення *value*, оголошене як *volatile.*

Кожен сегмент являє собою потокобезпечну таблицю елементів карти. Але *ConcurrentHashMap* повинен застосовуватися помірковано, з попередньою оцінкою співвідношення читання і записи в карту. Також як і раніше має сенс використовувати *HashMap* в програмах, де немає множинного доступу від декількох потоків до інформації, що зберігається карті.

Між хеш-кодами ключів і відповідними їм сегментами встановлюється залежність на основі застосування до старших розрядів хеш-коду бітової маски.

Класи *ConcurrentLinkedQueue* і *ConcurrentHashMap* дозволяють уникнути блокування всієї структури даних і мінімізувати конфлікти, допускаючи одночасний доступ до різних частин структури даних.

Одночасно зі структурою даних може працювати до 16 записуючих потоків. Зазвичай застосовуються для організації роботи з даними як з «кешем».

У Табл. 2.8.2 наведено порівняння продуктивності деякої програми, яка використовувала класи *ConcurrentHashMap* і *Hashtable*.

Таблиця 2.8.2. Порівняння продуктивності використання класів *ConcurrentHashMap* і *Hashtable*:

|  |  |  |
| --- | --- | --- |
| **Кількість потоків** | ***ConcurrentHashMap*** | ***Hashtable*** |
| 1 | 1.0 | 1.03 |
| 2 | 2.59 | 32.4 |
| 4 | 5.58 | 78.23 |
| 8 | 13.21 | 163.48 |
| 16 | 27.58 | 341.21 |
| 32 | 57.27 | 778.41 |

**Відмінності між *ConcurrentHashMap* та *Collections.synchronizedMap***

Основна відмінність між цими двома класами полягає в тому, що *ConcurrentHashMap* блокуватиме тільки частину даних, які оновлюються, а інша частина даних може бути доступна іншим потокам. Проте, *Collections.synchronizedMap ()* блокує всі дані при оновленні, інші потоки можуть отримувати доступ до даних тільки при звільненні блокування. Якщо є багато операцій оновлення і відносного невелика кількість операцій читання, краще вибрати *ConcurrentHashMap*.

Ще одна відмінність полягає в тому, що *ConcurrentHashMap* не збереже порядок елементів у переданій карті (схожий на *HashMap* при зберіганні даних). Напроти, *Collections.synchronizedMap ()* збереже порядок елементів переданої карти. Наприклад, якщо ви передаєте TreeMap в ConcurrentHashMap, то порядок елементів в ConcurrentHashMap може відрізнятися від порядку в TreeMap, а Collections.synchronizedMap () збереже порядок.

*ConcurrentHashMap* може гарантувати, що не буде *ConcurrentModificationException*, якщо один потік оновлює карту, а інший потік перетинає итератор, отриманий з карти. *Collections.synchronizedMap ()* цього не гарантує.

Таким чином, якщо, наприклад, потрібна карта, яка може одночасно змінюватися декільками потоками без необхідності її повного блокувння, то краще використовувати *ConcurrentHashMap. Collections.synchronizedMap(map)*  створює повністю блоковану карту, яка погіршує продуктивність, хоча і забезпечує узгодженість (якщо використовується правильно). Використовуйте цей варіант, якщо вам потрібно забезпечити узгодженість даних, і кожен наступний потік повинен мати оновлений вигляд карти. Використовуйте *ConcurrentHashMap*, якщо продуктивність критична, і кожен потік тільки вставляє дані на карту, причому читання відбувається не дуже часто.

**Класи CopyOnWriteArrayList та CopyOnWriteArraySet**

Ці два дозволяють багатопотокове читання і запис: вони створюють копію колекції і дозволять іншому потоку виконати аналогічну операцію.

### Синхронізатори загального призначення

Класи, які призначені для управління набором взааємодіючих потоків наведені в Табл.2.8.3.

Таблиця 2.8.3. Класи для взааємодії потоків

|  |  |  |
| --- | --- | --- |
| **Клас** | **Призначення** | **Використання** |
| Semaphore | Кілька потоків запитує, очікує і отримує дозвіл на подальшу роботу | Коли необхідно обмежити загальну кількість потоків, що мають доступ до ресурсів |
| CyclicBarrier | Набір потоків очікує, поки певна їх кількість не досягне бар'єру, після чого можливе виконання дії, пов'язаного з цією точкою | Коли результати можуть бути використані тільки після того, як декілька потоків завершить своє завдання |
| CountDownLatch | Набір потоків очікує, поки деякий лічильник не стане рівним нулю | Коли потоки повинні очікувати вихідних даних для своєї роботи |
| Exchanger | Два потоки обробляють два примірника однієї структури даних | Коли потоки повинні обмінюватися своїми даними |
| SynchronousQueue | Один потік повинен передати об'єкт іншому потоку | Коли потік повинен передати об'єкт іншому потоку при готовності обох виконати таку передачу |

**Клас Semaphore**

*Семафор (клас Semaphore)* - це управління доступом до об'єкту на основі заборон / дозволів. Найчастіше використовуються для обмеження кількості потоків при роботі з апаратними ресурсами або файлової системою.

*Семафор* управляє доступом до загального ресурсу за допомогою лічильника. Якщо значення лічильника більше нуля, то потік отримує доступ до ресурсу, при цьому лічильник зменшується на одиницю. Після закінчення роботи з ресурсом потік звільняє семафор, і лічильник збільшується на одиницю. Якщо він дорівнює нулю, то в доступі буде відмовлено.

Насправді цей лічильник підраховує дозволи, що відкривають доступ до загального ресурсу. Отже, щоб отримати доступ до ресурсу, потік виконання повинен отримати у семафора дозвіл на доступ.

Наприклад, семафор може дозволити виконуватися не більше трьох потоків-задач і для цього встановлює рахівник=3. Як тільки зявляється чергова задача-потік на виконання, то рахівник стає=2 і так до нуля. Як тільки один з потоків виконання звільнить ресурси, тобто завершить виконання певної ділянки коду, то значення лічильника семафора збільшується на одиницю. Якщо в цей час очікує дозволу інший потік виконання, то він відразу ж його отримує. Тобто головне, щоб в будь-який момент часу працювало не більше трьох завдань. Як тільки одна задача закінчується, на її місце можения бути завантажена інша. Зручно, наприклад, для обмеження кількості запитів до бази даних.

Таким чином, у Семафорі:

- лічильник завжди спочатку встановлюється на максимальне число потоків, одночасне функціонування яких може бути дозволено;

- при перевищенні цього значення всі бажаючі виконуватися потоки будуть призупинені до звільнення семафора одним з потоків, що працюють по його дозволу;

- при кожному новому дозволі лічильник допущених завдань на виконання зменшується на одиницю;

- при звільненні семафора лічильник збільшується на одиницю.

Для отримання дозволу використовується метод *acquire* ().

Для звільнення викликається метод *release* ().

Стандартна взаємодія методів *acquire() і release():*

public void run() {

try {

semaphore.acquire();

// код використання захищеного ресурсу

} catch (InterruptedException e) { }

finally {

semaphore.release(); // звільнення семафора

}

}

Семафор з максимальним значенням 1, використовується в якості затвора (наприклад, деякий застосунок виконує роботу частинами, щоб дати можливість користувачеві ознайомитися з проміжними результатами і ініціювати продовження роботи).

Конструктори:

* *Semaphore* *(int permits)* - створює семафор з зазначеним числом дозволів – це кількість потоків виконання.
* *Semaphore* *(int permits, boolean fair)* - створює семафор з зазначеним числом дозволів і з установкою зазначеної послідовності виконання (*true* – означає надання очікуючим потокам дозволу на виконання в тому порядку, в якому вони просили дозвіл).

Приклад.

public class Main {

public static final int ITEMS\_COUNT = 15;

public static double items[];

// семафор, контролюючий дозвіл на доступ до елементів масиву

public static Semaphore sortSemaphore = new Semaphore(0, true);

public static void main(String[] args) {

items = new double[ITEMS\_COUNT];

for(int i = 0 ; i < items.length ; ++i)

items[i] = Math.random();

new Thread(new ArraySort(items)).start();

for(int i = 0 ; i < items.length ; ++i) {

sortSemaphore.acquireUninterruptibly();

System.out.println(items[i]);

}

}

}

class ArraySort implements Runnable {

private double items[];

public ArraySort(double items[]) { this.items = items; }

public void run(){

for(int i = 0 ; i < items.length - 1 ; ++i) {

for(int j = i + 1 ; j < items.length ; ++j) {

if( items[i] < items[j] ) { double tmp = items[i];

items[i] = items[j];

items[j] = tmp;

} }

// звільнення семафора

Main.sortSemaphore.release();

try { Thread.sleep(71);

} catch (InterruptedException e) { System.err.print(e); }

}

Main.sortSemaphore.release();

}

}

**Клас CyclicBarrier**

Потоки, один за одним досягаючи бар'єру, переходять в стан «очікування» (викликається метод *wait* ()), поки вся їх зазначена певна кількість не досягне бар'єру для того, щоб всім разом почати виконуватися. Це нагадує паром, який стане рухатися тількі після того, як на нього заїде, наприклад, встановлена кількість авто. Потім паром може повернутися і забрати чергову партію авто. Так само і CyclicBarrier знову почне працювати після завершення роботи попередньої партії потоків, тому його і називають *циклічним*.

З точки зору API *циклічний* бар'єр *CyclicBarrier* має тільки метод самоблокування *await()* і не має методу *декрементаціі* лічильника, а також він дозволяє підключити і автоматично запускати додатковий потоковий клас при досягненні бар'єру всіх потоків.

Конструктори:

* CyclicBarrier(int parties);
* CyclicBarrier(int parties, Runnable barrierAction);

де параметр *parties* визначає число потоків, які повинні досягти деякого бар'єру до того, як їх виконання буде продовжено.

У другій формі конструктора параметр *barrierAction* визначає потік, який буде виконуватися після досягнення бар'єру.

Бар'єр можна використовувати повторно після звільнення всіх потоків.

Коли потік виконання досягне бар'єру, слід викликати для нього метод *await()*. В результаті виконання цього потоку буде призупинено до тих пір, поки метод *await()* не буде викликаний у всіх інших потоках, які досягли бар'єру.

Як тільки вказана кількість потоків досягне бар'єру, відбудеться повернення з методу *await* () для всіх потоків, і їх виконання буде відновлено. А якщо додатково вказати яку-небудь дію, то буде виконаний відповідний потік.

У методу *await* () є такі загальні форми: без параметрів і з параметрами.

У першій формі очікування триває до тих пір, поки кожен потік виконання не досягне бар'єрної точки.

А в другій формі очікування триває лише протягом певного періоду часу, обумовленого параметром очікування. Час очікування вказується в одиницях, які встановлюються параметром «одиниця\_часу».

В обох формах повертається значення, яке вказує порядок, в якому потоки виконання досягатимуть бар'єрної точки. Перший потік виконання повертає значення, яке дорівнює кількості очікуваних потоків мінус 1, а останній потік повертає нульове значення.

Приклад.

Обчислення суми елементів матриці при паралельному обчисленні суми елементів кожного рядка:

public class Main {

private static int matrix[][] = new int [5][6];

private static int rez[];

private static Boolean flag = true;

static {

for(int i=0; i<matrix.length; i++)

for(int j=0; j<matrix[i].length; j++)

matrix[i][j] = (int)(Math.random()\*20);

}

private static class Summator extends Thread {

int row; CyclicBarrier barrier;

public Summator(int row, CyclicBarrier barrier) {

this.row = row; this.barrier = barrier;

}

public void run() {

int colum = matrix[row].length;

for(int i=0; i<colum; i++)

rez[row] += matrix[row][i];

System.out.println("Summa (" + (row+1) + ") = " + rez[row]);

try {

barrier.await();

} catch(BrokenBarrierException e) { e.printStackTrace(); }

catch(InterruptedException e) { e.printStackTrace(); }

}

}

private static class Summator extends Thread {

int row; CyclicBarrier barrier;

public Summator(int row, CyclicBarrier barrier) {

this.row = row; this.barrier = barrier;

}

public void run() {

int colum = matrix[row].length;

for(int i=0; i<colum; i++)

rez[row] += matrix[row][i];

System.out.println("Summa (" + (row+1) + ") = " + rez[row]);

try {

barrier.await();

} catch(BrokenBarrierException e) { e.printStackTrace(); }

catch(InterruptedException e) { e.printStackTrace(); }

}

}

public static void main(String[] args) {

final int roww = matrix.length;

rez = new int [roww];

printMatrix(); // статичний метод виведення матриці

Runnable merger = new Runnable() {

public void run() {

int summa = 0;

for(int i=0; i<roww; i++)

summa +=rez[i];

synchronized (flag) {

flag.notifyAll(); flag = true;

}

System.out.println("Summa elements matrix = " + summa);

}

};

CyclicBarrier barrier = new CyclicBarrier(roww, merger);

for(int i=0; i<roww; i++)

new Summator(i,barrier).start();

synchronized (flag) {

while (!flag)

try { flag.wait(); }

catch (InterruptedException e) {

e.printStackTrace(); }

}

}

// опис виведення матриці

}

*Результат:*

Matrix:

8 19 4 12 8 12

11 5 17 3 15 15

0 17 14 11 17 16

12 15 6 1 8 0

9 9 19 4 0 8

Summa (1) = 63

Summa (2) = 66

Summa (3) = 75

Summa (4) = 42

Summa (5) = 49

Summa elements matrix = 295

**Клас CountDownLatch**

Клас *CountDownLatch* (можна перекласти, ук «засов з рахівником») є ефективним і простим у використанні засобом синхронізації, яке виявиться корисним в тих випадках, коли потік виконання повинен перебувати в стані очікування до тих пір, поки не відбудеться одна чи кілька подій. Він підтримує очікування для набору потоків до тих пір, поки значення деякого лічильника не буде дорівнювати нулеві.

*CountDownLatch* також, як і *Semaphore*, працює з лічильником, обнулення якого знімає самоблокування виконуваних потоків і після цього дає можливість працювати іншим потокам, тобто якщо задано CountDownLatch (int number) і number = 7, то поки не виконається 7 завдань (з наявних 20 в пулі), інші виконуватися не будуть. В міру ініціалізації кожного з 7 завданнь лічильник зменшується на 1, але на 1 буде збільшуватися і число задач, які будуть чекати початку виконання. Помірі ініціалізації нових завдань лічильник буде послідовно зменшуватися до 0. Після цього він вже не відновить своє значення до 7, а це значить, що після виконання 7 задач далі все решта завдань, наприклад, в пулі завдання, будуть виконуватися без всяких обмежень на кількість.

*CountDownLatch* має два перевантажених методи *await()* для самоблокування. У першому методі очікування триває до тих пір, поки лічильник *CountDownLatch* не досягне нуля. А в другому методі очікування триває лише протягом певного періоду часу, обумовленого параметром очікування *wait*. Час очікування вказується в одиницях *unit* об'єкта-перерахування *TimeUnit*.

Щоб зменшити лічильник об'єкта *CountDownLatch* слід викликати метод *countDown.*

Не завжди зрозуміло, в якому порядку і в яких об’ємах ОС буде запускати потоки на виконання в цьому класі. Але в будь-якому випадку треба, щоб спочатку всі потоки були ініціалізовані (дочекались один одного), а потім уже всі N –потоків сінхронно почали свою роботу. Це N повинно бути попередньо відомо.

Нехай, наприклад, ми створюємооб’єкт *CountDownLatch* **і** в конструктор передаємо кількість потоків, наприклад, N = 4. (Для створення затвору викликається конструктор *CountDownLatch(int count)).* В кожному потоці ми викличемо *countDown* і*await***.** Перший потік після завершення ініціалізації зменшує рахівник на 1, тобто після завершення ініціалізації першого потоку N=3. А *await* буде блокувати поточний потік (1-й потік), поки рахівник не стане =0. Тобто 1-й потік зависне на цьому методі *await***,** поки рахівник не стане 0. А це відбудеться тоді, коли кожен з залишившихся потоків також зменшить рахівник на 1 і зависне на своєму *await*. В кінці 4-й потік доведе рахівник до 0 і метод *await*відновить роботу у 4-му потоці, а також і у всіх інших потоках.

Таким чином, циклічний бар'єр *CyclicBarrier* схожий на *CountDownLatch*, але є і наступні відмінності:

* для зняття «засову» *CountDownLatch* необов'язково, щоб всі потоки набору перебували в стані «очікування»;
* значення лічильника може бути зменшено обробниками зовнішніх подій;
* «засов» *CountDownLatch* встановлюється один раз (повторне використання після обнуління лічильника неможливо).

Приклад.

public class Main {

public static void main(String args[]) {

//у нас буде запущено 5 потоків (це початкове значення рахівника)

CountDownLatch cdl = new CountDownLatch(5);

long timeS = 0, timeF = 0;

for (int i=0;i<5;++i){//запускаємо всі 5 потоків

new MyThread(cdl);//запускаємо один потік з рахівником

}

try {

System.out.print(«Черговий потік чекає-> ");

timeS = System.nanoTime();

cdl.await();//чекаємо закінчення решти потоків

timeF = System.nanoTime();

} catch (InterruptedException exc) {

System.out.println(exc);

}

double time = (timeF-timeS)/1000000000.0;

System.out.printf("%7.5f секунд\n", time);

}}

class MyThread implements Runnable {

CountDownLatch latch;

MyThread(CountDownLatch c) {

latch = c;

new Thread(this).start();

}

public void run() {

for(int i = 0; i<5; i++) {

try {

Thread.sleep((int)(Math.random()\*20));

} catch(InterruptedException e) { e.printStackTrace(); }

latch.countDown(); // decrement count- зменшуємо рахівник }

}}

Результат:

run:

Головний потік чекає -> 0,03949 секунд

**Клас Exchanger**

Клас *Exchanger*, призначений для спрощення процесу обміну даними між двома потоками виконання.

Принцип дії класу *Exchanger*: він очікує до тих пір, поки два окремих потоки виконання не викличуть його метод *exchange* (). Як тільки це відбудеться, він зробить обмін даними, наданими обома потоками.

Використання об'єкта синхронізації Exchanger можна уявити на прикладі роботи двох листонош, які з пунктів А і Б відправляються в сусідні селища В і Г доставити листи. Кожен з листонош повинен доставити по листу в кожне з селищ. Щоб не робити зайву ходку, вони зустрічаються в проміжному селищі Д і обмінюються одним листом. В результаті цього кожному з листонош доведеться доставити листи тільки в одне селище.

Цей клас підтримує передачу даних між потоками в дві сторони, не піклуючись про синхронізацію. Для виконання обміну викликається метод

<Type> exchange(<Type> sendlingValue)

Після виклику цього методу потік переходить в стан «очікування», поки інший потік не викличе цей же метод на цьому ж об'єкті.

Головна особливість методу *exchange*() полягає в тому, що він не завершиться успішно до тих пір, поки він не буде викликаний для одного і того ж об'єкта типу *Exchanger* з двох окремих потоків виконання. Таким чином, метод *exchange* () синхронізує обмін даними.

Наприклад:

public static void main(String[] args) {

…………………………

int result = 0;

final Exchanger <Integer> myExchanger = new Exchanger <Integer>();

…………………………

public void run() {

…………………………

try { myExchanger.exchange(summa); }

catch (InterruptedException e) { e.printStackTrace(); }

…………………………

}

…………………………

try { result = myExchanger.exchange(0); }

catch (InterruptedException e) { e.printStackTrace(); }

System.out.println("Summa elements matrix = " + result);

}

**Клас SynchronousQueue**

Цей клас виконує передачу даних між потоками в одну сторону, тобто один потік віддає дані, інший - отримує.

Для надання значення викликається метод:

void put(<Type> item)

Для отримання значення викликається метод:

<Type> take()

Передача даних може здійснюватися тільки в разі, коли є потік, який очікує надходження.

Приклад.

class Producer implements Runnable {

private SynchronousQueue <String> drop;

List <String> messages = Arrays.asList( "Mares eat oats",

"Does eat oats","Little lambs eat ivy",

"Wouldn't you eat ivy too?");

public Producer(SynchronousQueue <String> drop) { this.drop = drop; }

public void run() {

try {

for (String s : messages)

drop.put(s);

drop.put("DONE");

}

catch (InterruptedException intEx) {

System.out.println("Interrupted! " + "Last one out, turn out the lights!");

}

}

}

class Producer implements Runnable {

private SynchronousQueue <String> drop;

List <String> messages = Arrays.asList( "Mares eat oats",

"Does eat oats","Little lambs eat ivy",

"Wouldn't you eat ivy too?");

public Producer(SynchronousQueue <String> drop) { this.drop = drop; }

public void run() {

try {

for (String s : messages)

drop.put(s);

drop.put("DONE");

}

catch (InterruptedException intEx) {

System.out.println("Interrupted! " + "Last one out, turn out the lights!");

}

}

}

class Consumer implements Runnable {

private SynchronousQueue <String> drop;

public Consumer(SynchronousQueue <String> d) { this.drop = d; }

public void run() {

try {

String msg = null;

while (!((msg = drop.take()).equals("DONE")))

System.out.println(msg);

}

catch (InterruptedException intEx) {

System.out.println("Interrupted! " + "Last one out, turn

out the lights!");

}

}

}

public class Main {

public static void main(String[] args) {

SynchronousQueue <String> drop =

new SynchronousQueue<String>();

(new Thread(new Producer(drop))).start();

(new Thread(new Consumer(drop))).start();

}

}

Результат:

run:

Mares eat oats

Does eat oats

Little lambs eat ivy

Wouldn't you eat ivy too?

**Клас Phaser**

*Phaser*, як і *CyclicBarrier*, є реалізацією об'єкта синхронізації типу «Бар'єр». На відміну від *CyclicBarrier*, *Phaser* надає більше гнучкості. Щоб краще зрозуміти *Phaser*, можна навести два приклади, які наочно демонструють його використання.

В якості першого прикладу можна розглянути кілька потоків виконання, що реалізують процес обробки замовлень, який складається з трьох стадій. На першій стадії окремі потоки перевіряють відомості про клієнта, наявність товару на складі і їх вартість. На другій стадії обчислюється вартість замовлення і вартість доставки. На заключній стадії підтверджується оплата і визначається орієнтовний час доставки.

У другому прикладі кілька потоків реалізують перевезення пасажирів міським транспортом. Пасажири чекають транспорт на різних зупинках. Транспорт, зупиняючись на зупинках, одних пасажирів «садить», інших «висаджує».

### Атомарні класи та змінні

*Атомарні змінні -* це змінні атомарних класів. Атомарні класи гарантують, що певні операції будуть виконуватися потокобезпечно (наприклад, операції інкремента і декремента, оновлення та додавання значення). Атомарні класи описані в пакеті *java.util.concurrent.atomic* (включає такі класи як *AtomicInteger, AtomicBoolean, AtomicLong, AtomicIntegerArray* і інші). Атомарні змінні необхідні при таких операціях читання / запису, як інкремент / декремент без зовнішньої синхронізації; на змінних ***volatile*** вони не виконуються.

Наприклад, наступна операція на змінній *volatile є* небезпечною при використанні декількома потоками*:*

myVolatileVar++;

Це все рівно, що:

int temp = 0;

synchronized (myVolatileVar) {

temp = myVolatileVar; // Читання синхронізовано (атомарно)

}

temp++; // ця операція інкремента не атомарна

synchronized (myVolatileVar) {

myVolatileVar = temp; // Запис синхронізований (атомарно)

}

Розглянемо приклад використання змінних: *звичайна, атомарна і volatile* в класі-рахівнику.

class MyCounter {

public int count1;

public volatile int count2;

public AtomicInteger count3 =

new AtomicInteger(0);

}

// Потік який збільшує рахівник meter number разів

class MyCountThread extends Thread {

MyCounter meter;

int number;

public MyCountThread(MyCounter m, int n) {

this.meter = m;

this.number = n;

}

public void run() {

for(int i=0; i<number; i++) {

this.meter.count1++;

this.meter.count2++;

this.meter.count3.getAndIncrement();

}

}

}

// …

public static void main(String[] args) {

MyCounter meter = new MyCounter();

MyCountThread[] tgs = new MyCountThread[100];

for (int i = 0 ; i < 100; i++)

tgs[i] = new MyCountThread(meter,1\_000\_000);

for (MyCountThread thread : tgs)

thread.start(); //створюється и запускається 100 потоків

try {

for (MyCountThread thread : tgs)

thread.join();//наступний чекає завершення попереднього

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("int: " + meter.count1 + "\nvolatile: "

+ meter.count2 + "\nAtomic: " + meter.count3);

}

**Результат**

int: 49793826

volatile: 99998132

Atomic: 100000000

В цій програмі в класі-рахівнику MyCounter використовуються зміні трьох типів: *звичайна, атомарна і volatile***,** в кожній з яких нарощується значення. Створюються 100 потоків – об’єктів класу MyCountThread, в якому використовується клас MyCounter. Після виконання циклів по нарощенню рахівників у трьох потоках, їх значення виводиться на консоль. Як видно всі вони різні, тому що операція інкремента – не атомарна. Вірний результат тільки у змінній типу *AtomicInteger***.**

Змінна *volatile*оновлюється неявно, тобто значення читається, змінюється, а потім присвоюється як нове.

Розглянемо приклад міжпоточної взаємодії з використанням високорівневого API керування потоками (аналог прикладу в розділі 1.15).

Є клас Bank, який надає можливість:

* передачі коштів з одного рахунку на інший усередині цього банку (метод trans (...));
* перевірки, що загальний баланс коштів не змінюється при передачі коштів (метод getBalance ());

Є клас-потік Transfer, який відповідає за роботу з одним рахунком банку.

import java.util.concurrent.locks.\*;

public class Main {

public static final int count = 100; //кількість рахунків

public static final double balance = 1000.0; //сума на кожному рахунку

public static void main(String[] args) {

Bank b = new Bank(count, balance);

for (int i=0; i<count; i++) {

Transfer tr = new Transfer(b, i, balance);

Thread t = new Thread(tr);//потік для роботи з одним рахунком

t.start();

}

}

}

// …

В *main* був створений потік t, який намагається отримати доступ до загального ресурсу Bank, екземпляр цього класу також був створений в main. При створенні банку створюється масив рахунків в кількості 100, на кожний з яких кладеться сума грошей balance = 1000.0. Для кожного рахунку в циклі створюється свій потік. Потік створюється на основі *Transfer implements Runnable* , в якому реалізовано метод run. В цьому методі в циклі на певний рахунок перераховується певна сума грошей bb.trans(fromCh, toCh, ss) . Обидва ці числа вираховуються рандомно) .

При створенні об’єкту Bank в конструкторі на основі інтерфейсу Lock був використаний клас *ReentrantLock*(), який реалізує цей інтерфейс і цей клас має методи, призначені для реалізації блокування. Також створено екземпляр класу *Condition*, який керує блокувавнням. При виконанні чергової транзакції *trans* викликається метод *bl.lock()* для встановлення блокування (монітор), тобто на початку роботи з загальним ресурсом – певним рахунком, на нього накладається блокування, щоб з ним не проводилось інших транзакцій. (Якщо на рахунку не вистачає коштів, то зняти монітор і почекати *con.await().*

Інші потоки чекають розблокування.У блоці *finally* після закінчення всієї основної роботи потоку це блокування знімається. Причому робиться це обов'язково в блоці finally, так як в разі виникнення помилки всі інші потоки виявляться заблокованими.

class Bank {

private double kol[];//масив рахунків

private Lock bl;

private Condition con;

Bank(int n, double bal) {

kol = new double[n];

for (int i=0; i< kol.length; i++)

kol[i] = bal;

bl = new ReentrantLock();//для створення блокування

con = bl.newCondition(); //умова використання блокування

}

// …

public void trans(int from, int to, double sum) throws InterruptedException{

bl.lock();//Встановлення блокування (монітор)

try {

// Якщо на рахунку не вистачає коштів, то зняти монітор і почекати

while (kol[from] < sum)

con.await();

System.out.print(Thread.currentThread());

kol[from] -= sum;

System.out.print(sum + " from " + from + " to" + to);

kol[to] += sum;

System.out.println(" Загальний рахунок = " + getBalance());

con.signalAll();//Сповістити всі потоки

} finally { bl.unlock(); }

}

// …

public double getBalance() {

// Встановити блокування на операції передачі коштів поки йде підрахунок загального балансу

bl.lock();

try {

double summa = 0.0;

for (double aa : kol)

summa += aa;

return summa;

} finally {

bl.unlock();

}

}

}

class Transfer implements Runnable {

private Bank bb; private int fromCh; private double maxSum;

Transfer(Bank b, int ch, double sum) {

bb = b; fromCh = ch; maxSum = sum;

}

public void run() {

try {

while (true) {

int toCh = (int)(Math.random() \* bb.length);//номер рахунку

double ss = Math.random() \* maxSum;//сума перерахування

bb.trans(fromCh, toCh, ss);

Thread.sleep((int)(Math.random() \* 10));

}

} catch (InterruptedException e) { }

}

}

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Які стандартні фреймворки вміщує пакет *java.util.concurrent*?
2. Які основні переваги використання пакету *java.util.concurrent*?
3. Яке призначення інтерфейсу Lock та його основні методи?
4. Чим інтерфейс *Callable* відрізняється від інтерфейсу *Runnable?*
5. Яке призначення *пулу потоків* та *виконавців*?
6. Яке призначення б*локуючих черг?*
7. Яке призначення *синхронізуючих колекцій*?
8. Які існують синхронізатори загального призначення?
9. Яке признаення пакету *java.util.concurrent.atomic?*

## Stream API

### Концепція потоку stream

Мова Java була спроектована як *імперативна* мова, тобто вона складається з набору команд, які послідовно виконуються. В програмуванні також є *декларативний* стиль, коли замість набору команд описується той результат, який ми хочемо отримати.

У *Java 8,* задяки введенню *Stream API і лямбда-виразів,* був впроваджений цей декларативний стиль, який ще називають *функціональний* стиль програмування, а дехто називає його *Stream-стиль*, щоб підкреслити те, що *Java* була і залишається все ж таки *імперативною* мовою програмування. Строго визначення потоку *Stream* (*стрім*у)немає. Можна зазначити тільки основні ознаки і можливості *Stream.*

В основs *Stream API* лежить интерфейс *BaseStream*. Його повне визначення:

interface BaseStream<T , S extends BaseStream<T , S>>

Тут параметр T означає тип даних в *Stream*, а S - тип потоку, який успадковується від інтерфейсу *BaseStream*.

*BaseStream* визначає базовий функціонал для роботи з потоками, який реалізується через його методи. Від інтерфейса *BaseStream* похідними є кілька інтерфейсів для створення конкретних потоків:

* *Stream<T>:* для потоків даних будь якого типу-посилання
* *IntStream*: для потоків з типом даних int
* *DoubleStream*: для потоков з типом даних double
* *LongStream*: для потоків з типом даних long

Крім об'єктних стрімів  *Stream<T>*, є спеціальні (примітивні спеціалізації): *IntStream, LongStream, DoubleStream*. Для *boolean, byte, short і char* типів даних спеціальних стрімів немає, але замість них можна використовувати *IntStream*, а потім приводити його до потрібного типу. Для *float* теж доведеться скористатися *DoubleStream*. Примітивні стріми корисні, тому що не потрібно витрачати час на боксинг / анбоксінг, до того ж у них є ряд спеціальних операторів, що спрощують роботу.

В більшості випадків, як правило, робота відбувається з більш складними даними, ніж дані примітивних типів, для яких призначений інтерфейс *Stream* <T>.

*Stream* працює, в загальному випадку, з невпорядкованим набором будь яких даних. (Хоча стріми можуть бути як невпорядкований так і впорядкованими. Останні краще використовувати, якщо цього вимагає задача). *Stream* хоча і тісно взаємодіє зі структурами (колекціями) даних, але не є структурою, яка призначена для зберігання даних. *Stream* може бути безкінечним. Він, як правило, не мутує (не змінює) джерело даних (наприклад, структура даних після обробки стрімом не змінюється). *Stream* – одноразовий (його не можна використати двічі), тобто при роботі з ним відходу назад немає.

Стріми можуть бути *послідовними* (sequential) і *паралельними* (parallel) і можуть бути перетворені один в інший. Послідовні обробляються тільки в поточному потоці виконання, а паралельні використовують загальний пул *ForkJoinPool.commonPool().* При цьому дані розбиваються (якщо це можливо) на кілька груп і обробляються в кожному потоці окремо. Потім на потрібному етапі групи об'єднуються в одну для надання кінцевого результату.

*В Stream-*інтерфейсі об’явлені всі операції по роботі з ним (ці операції поділяються на «*проміжні*» і «*термінальні*»)*.* Наприклад, можна виконати такий ланцюг операцій: вибрати лише деякі елементи вихідного набору даних (filter), перетворити кожен елемент (map), порахувати суму елементів або об'єднати все в один об'єкт (reduce). Інші приклади:

* зі списку співробітників, вибрати тих, хто молодше 40 років, впорядкувати за прізвищами і помістити в новий список.
* прочитати всі json-файли, що знаходяться в папці books, десеріалізувати в список об'єктів книг, обробити елементи всіх цих списків, а потім згрупувати книги по автору.

При цьому ми вказуємо, які операції хочемо провести, не піклуючись про деталі реалізації.

Стрім виконує послідовність проміжних операцій-дій, але остаточний результат виконання відкладений до останньої дії (в цьому сенсі всі його проміжні операції називають «лінивим» (lazy)). Будь-який метод-операція, яка в результаті породжує новий стрім – це *проміжна* операція. Всі інші (які повертають не стрім) – *термінальні*. Ми не можемо розгалужувати цей ланцюжок операцій. Кожна чергова операція може породжувати новий стрім, який передається наступній операції.

Осклільки *проміжні* *або конвеєрні* (*intermediate*) операції обробляють дані і знову повертають стрім (в ланцюжку обробки їх може бути багато), то, тим самим, вони працюють як *builder*.

*Термінальні* - обробляють дані і завершують роботу стріму ( так що термінальний оператор в ланцюжку може бути тільки один). В результаті своєї роботи вони повертають інший об'єкт, такий як колекція, примітив, об'єкт, Optional і т.д.

Стрім, як і колекція, може бути порожнім, а значить в такому випадку операторам нічого буде обробляти.

Є різні способи створення стрімів – з колекцій, масивів, зі значень, з файлів і т.д. Тобто, стрім може використовувати існуючі колекції для отримання нових даних, але це не буде якась нова структура даних.

Для створення «джерела»-стріма можна також використовувати генератори-функції, які, наприклад, за допомогою інкремента можуть створити безкінечний стрім.

Поява *Stream API* дозволила програмістам писати істотно коротше те, що раніше займало багато рядків коду, а саме - спростити роботу з наборами даних, зокрема, спростити операції фільтрації, сортування та інші маніпуляції з даними. Якщо у вас проміжних операцій немає, часто можна і потрібно обійтися без стріму.

### Способи створення стрімів

Крім останніх двох способів створення стріму, перелічених нижче, всі вони схожі на створення колекцій. Останні два способи використовуються для генерації нескінченних стрімів.

1. з колекції

**collection**.**stream**()

Collection<String> collection=Arrays.asList("a1","a2","a3");

Stream<String> streamFromCollection = collection.stream();

2. зі значень

**Stream.of**(*значение1*,… *значениеN*)

Stream<String> streamFromValues =Stream.of("a1","a2", "a3");

3. з масива

**Arrays.stream**(*массив*)

String[] array = {"a1","a2","a3"};

Stream<String> streamFromArrays = Arrays.stream(array);

4. з файла (кожен рядок у файлі буде окремим елементом у стримі)

**Files.lines**(*шлях\_до\_файлу*)

Stream<String> streamFromFiles = Files.lines(Paths.get("file.txt"))

5. з рядка

«рядок».**chars**()

IntStream streamFromString = "123".chars()

6. за допомогою Stream.builder

**Stream.builder**().**add**(...)....**build**()

Stream.builder().add("a1").add("a2").add("a3").build()

7. Створення паралельного стрима

**collection**.**parallelStream**()

Stream<String> stream = collection.parallelStream();

8. Створення безкінечного стрима за допомогою *Stream.iterate*  
**Stream.iterate**(*початкова\_умова*, *вираз\_генерації*)

Stream<Integer> streamFromIterate=Stream.iterate(1, n-> n+1)

В *iterate* задається початкова умова і вираз отримання наступного значення з попереднього, тобто *Stream.iterate* (1, n -> n + 1) буде видавати значення 1, 2, 3, 4, ... N .

9. Створення безкінечного стрима за допомогою *Stream.generate*

**Stream.generate**(*вираз\_генерації*)

Stream<String> streamFromGenerate =Stream.generate(()->"a1")

*Stream.generate* служить для генерації константних і випадкових значень, він просто видає значення у відповідності до виразу.

### Методи роботи зі стримами

*Java Stream API* пропонує два основних вида методів роботи зі стримами:

*Проміжні* (*конвеєрні)* - повертають інший *stream*, тобто працюють як *builder*.

*Термінальні* - повертають інший об'єкт, такий як колекція, примітиви, об'єкти, Optional і т.д.

Таблиця 2.9.1. Проміжні (конвеєрні) методи роботи зі стримами

|  |  |  |
| --- | --- | --- |
| **Метод stream** | **Опис** | **Приклад** |
| filter | Stream<T> **filter**(Predicate<? super T> predicate): фільтрує елементи у відповідності з умовою в предикаті. | collection.stream().filter(«a1»::equals).count() |
| skip | Stream<T> **skip**(long n):повертає потік, в якому відсутні перші n елементів. | collection.stream().skip(collection.size() — 1).findFirst().orElse(«1») |
| distinct | Stream<T> **distinct**(): повертає потік, в якому є тільки унікальні дані з типом T. (для метода equals) | collection.stream().distinct().collect(Collectors.toList()) |
| map | Претворює кожний елемент стрима  <R> Stream<R> **map**(Function<? super T,? extends R> mapper): перетворює елементи типу T в елементи типу R та повертає потік з елементами R. | collection.stream().map((s) -> s + "\_1").collect(Collectors.toList()) |
| peek | повертає той же стрим, але застосовує функцію до кожного елементу стрима | collection.stream().map(String::toUpperCase).peek((e) -> System.out.print("," + e)). collect(Collectors.toList()) |
| limit | Stream<T> **limit**(long maxSize): залишає в потоці тільки maxSize елементів | collection.stream().limit(2).collect(Collectors.toList()) |
| sorted | повертає відсортований потік (можна задавати Comparator) | collection.stream().sorted().collect(Collectors.toList()) |
| mapToInt, mapTo Double, mapTo Long | Аналог map, але повертає числовий стрим (тобто стрим з числових примітивів) | collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray() |
| flatMap, flatMap ToInt, flatMap ToDouble flatMap ToLong | Схоже на map,але може створювати з одного елемента декілька  <R> Stream<R> **flatMap**(Function<? super T, ? extends Stream<? extends R>> mapper): дозволяє перетворити елемент типу T в декілька елементів типу R та повертає потік з елементами R. | collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::new) |
| takeWhile | Stream <T> **takeWhile** (Predicate <? Super T> predicate): вибирає з потоку елементи, поки вони відповідають умові в predicate. Вибрані елементи повертаються у вигляді потоку. |  |
| dropWhile | Stream<T> **dropWhile**(Predicate<? super T> predicate): пропускає елементи, які відповідають умові в *predicate*, поки не попадеться елемент, який не відповідає умові. Вибрані елементи повертаються у вигляді потоку. |  |
| concat | Stream<T> **concat**(Stream<? extends T> a, Stream<? extends T> b) об'єднує два потоки. |  |

Таблиця 2.9.2. Термінальні методи роботи зі стримами

|  |  |  |
| --- | --- | --- |
| **Метод stream** | **Опис** | **Приклад** |
| findFirst | Optional<T> **findFirst**(): повертає перший елемент з потоку (повертає Optional) | collection.stream().findFirst().orElse(«1») |
| findAny | Optional<T> **findAny**(): повертає перший підходяший елемент з потоку | collection.stream().findAny().orElse(«1») |
| collect | Представлення результатів в вигляді колекций і інших структур даних  <R,A> R **collect**(Collector<? super T,A,R> collector): додає елементи незмінний контейнер з типом R. T представляє тип даних з вхідного потоку, а A - тип даних в контейнері. | collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList()) |
| count | long **count**(): повертає кількість елементів у потоці | collection.stream().filter(«a1»::  equals).count() |
| anyMatch | boolean **anyMatch**(Predicate<? super T> predicate): повертає true, якщо хоч один елемент потоку задовольняють умові в предикате. | collection.stream().anyMatch(«a1»::equals) |
| noneMatch | boolean **noneMatch**(Predicate<? super T> predicate): повертає *true*, якщо ні один з елементів у потоці не задовольняє умові в предикате | collection.stream().noneMatch(«a8»::equals) |
| allMatch | boolean **allMatch**(Predicate<? super T> predicate): повертає true, якщо всі елементи потоку задовольняють умові в предикаті | collection.stream().allMatch((s) -> s.contains(«1»)) |
| min | Optional<T> min(Comparator<? super T> comparator): повертає мінімальний елемент з потоку. Для порівняння елементів застосовується компаратор comparator. | collection.stream().min(String::compareTo).get() |
| max | Optional<T> **max**(Comparator<? Super T> comparator): повертає максимальний елемент з потоку. Для порівняння елементів застосовується компаратор *comparator* | collection.stream().max(String::compareTo).get() |
| forEach | void **forEach**(Consumer<? super T> action): для кожного елемента виконується дія *action*, порядок при паралельному виконанні не гарантується | set.stream().forEach((p) -> p.append("\_1")); |
| forEachOrdered | Застосовує функцію до кожного об’єкту стрима, порядок при паралельному виконанні гарантується | list.stream().forEachOrdered((p) -> p.append("\_new")); |
| toArray | Повертає масив значень стрима | collection.stream().map(String::toUpperCase).toArray(String[]::new); |
| reduce | Дозволяє виконати агрегатні функції на всій колекції і повертає один результат | collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0 |

|  |  |  |
| --- | --- | --- |
| **Метод** | **Опис** | **Приклад** |
| Sum | Повертає суму всіх чисел | collection.stream().mapToInt((s) -> Integer.parseInt(s)).sum() |
| average | Повертає середнє арифметичне всіх чисел | collection.stream().mapToInt((s) -> Integer.parseInt(s)).average() |
| mapToObj | Перетворює числовий стрим в об’єктний | intStream.mapToObj((id) -> new Key(id)).toArray() |

Таблиця 2.9.3. Додаткові методи для числових стримів

Таблиця 2.9.4. Інші корисні методи

|  |  |
| --- | --- |
| **Метод** | **Опис** |
| isParallel | З’ясовує, чи є стрим паралельним |
| parallel | Повертає паралельний стрим, а якщо стрим уже паралельний, то может повернути самого себе |
| sequential | Повертає послідовний стрим, я якщо стрим уже послідовний, то может повернути самого себе |

**Рекомендації по використанню деяких методів**

Методи *findFirst* і *findAny* повертають новий тип *Optional*, що з'явився в Java 8, для того щоб уникнути *NullPointerException*.

Метод *filter* зручно використовувати для вибірки лише певної множини значень, а метод *skip* дозволяє пропускати певну кількість елементів.

Метод *distinct* повертає *stream* без дублікатів, при цьому для упорядкованого стріму (наприклад, колекція на основі *list*) порядок стабільний, для невпорядкованого - порядок не гарантується.

Якщо ви використовуєте *distinct* з класом, у якого перевизначений *equals*, обов'язково так само коректно треба перевизначити *hashCode* у відповідністі до пари *equals* / *hashCode* (найголовніше, щоб *hashCode* для всіх *equals* об'єктів повертав однакове значення), інакше *distinct* може не видалити дублікати (аналогічно, при використанні *HashSet* / *HashMap*),

Якщо ви використовуєте паралельні стріми і вам не важливий порядок елементів після видалення дублікатів - набагато краще для продуктивності зробити спочатку стрім невпорядкованим з допомогу *unordered* (), а вже потім застосовувати *distinct* (), тому що підтримання стабільності сортування при паралельному стрімі досить коштовно по ресурсам і *distinct* () на впорядкованому стрімі буде виконуватися значно довше ніж на невпорядкованиму.

Всі *map* функції можуть повернути об'єкт іншого типу (класу), тобто *map* може працювати зі стрімом рядків, а на виході дати *Stream* зі значеннями *Integer* або отримувати клас людей *People*, а повертати клас *Office*, де ці люди працюють і т.п.. *flatMap* (*flatMapToInt* і т.п.) на виході повинні повертати стрім з одним, кількома або жодним елементів для кожного елемента вхідного стріму.

Обидві функції *ForEach* і *Peek* по суті роблять одне і теж, змінюють властивості об'єктів у стрімі, єдина різниця між ними в тому що *ForEach* термінальна і вона закінчує роботу зі стрімом, в той час як *Peek* конвеєрна і робота зі стрмом триває. Наприклад, є колекція:

Collection <StringBuilder> list = Arrays.asList (new StringBuilder ( "a1"), new StringBuilder ( "a2"), new StringBuilder ( "a3"));

І потрібно додати до кожного елементу "\_new", тоді для ForEach код буде

list.stream (). forEachOrdered ((p) -> p.append ( "\_ new")); // list - містить

// [a1\_new, a2\_new, a3\_new]

а для *peek* код буде

List <StringBuilder> newList = list.stream (). Peek ((p) -> p.append ( "\_ new")). Collect (Collectors.toList ()); // і list і newList містять [a1\_new, a2\_new, a3\_new]

Метод *reduce* дозволяє виконувати агрегатні функції на всій колекції (такі як сума, знаходження мінімального або максимального значення і т.п.), він повертає одне значення для стріму. Функція отримує два аргументи - значення отримане на минулих етапах і поточне значення.

Можна викликати *toArray* () і отримати Object [], або *toArray* (T [] :: new) – і отримати масив типу T.

**Інтерфейс Collector та його методи**

Інтерфейс *java.util.stream.Collector* служить для збору елементів стріму в певний мутабельний контейнер.

Характеристики:

- CONCURRENT - колектор підтримує багатопоточність, а значить окремі частини стріму можуть бути успішно покладені в контейнер з іншого потоку.

- UNORDERED - колектор не залежить від порядку надходження елементів.

- IDENTITY\_FINISH - функція finish () має стандартну реалізацію (Function.identity ()), а значить її можна не викликати.

**Методи Collectors**

*R collect (Collector collector)* - один з найпотужніших операторів Stream API. З його допомогою можна виконати обернену операцію - перетворити стрім в колекцію (тобто зібрати, наприклад, всі елементи стрім в список, множину або іншу колекцію), згрупувати елементи по якомусь критерію, об'єднати все в рядок і т.д . У класі *java.util.stream.Collectors* є багато статичних методів:

* **toList, toCollection, toSet** представляють стрім у вигляді списку, колекції або множини (наприклад перетворення в *List* буде *stream.collect (Collectors.toList ()).*
* **toConcurrentMap, toMap** дозволяють перетворити стрім в *map*
* **averagingInt, averagingDouble, averagingLong** повертають середнє значення
* **summingInt, summingDouble, summingLong** повертає суму
* **summarizingInt, summarizingDouble, summarizingLong** повертають SummaryStatistics з різними агрегатними значеннями
* **partitioningBy** розділяє колекцію на дві частини по відповідній умові і повертає їх як Map <Boolean, List>
* **groupingBy** розділяє колекцію на кілька частин і повертає Map <N, List <T >>
* **mapping** додаткові перетворення значень для складних Collector

Крім *Collector* вже визначених у *Collectors* можна так само створити власний *Collector*. Розглянемо приклад як його можна створити.

Метод визначення користувальницького Collector:

Collector <Тип\_джерела, Тип\_акумулятору, Тип\_результату> сollector = Collector.of (

метод\_ініціалізаціі\_акумулятора,

метод\_обробки\_кожного\_елемента,

метод\_об’єднання\_двох\_акумуляторів,

[Метод\_останньої\_обробки\_акумулятора]

);

Як видно з коду вище, для реалізації свого Collector потрібно визначити три або чотири методи (Метод\_останньої\_обробки\_акумулятора не обов'язковий). Розглянемо наступний приклад коду, який ми писали до Java 8, щоб об'єднати всі рядки колекції:

StringBuilder b = new StringBuilder(); //метод\_ініціалізаціі\_акумулятора

for(String s: strings) {

b.append(s).append(" , "); //метод\_обробки\_кожного\_елемента }

String joinBuilderOld= b.toString();//Метод\_останньої\_обробки\_акумулятора

І аналогичний код, який буде написаний в Java 8  
  
String joinBuilder = strings.stream().collect(

Collector.of(

StringBuilder::new, // метод\_ініціалізаціі\_акумулятора

(b ,s) -> b.append(s).append(" , "), // метод\_обробки\_кожного\_елемента

(b1, b2) -> b1.append(b2).append(" , "), //метод\_об’єднання\_двох\_акумуляторів

StringBuilder::toString // Метод\_останньої\_обробки\_акумулятора

)

);

Метод\_об’єднання\_двох\_акумуляторів - це метод який потрібен для паралельної обробки Collector, в даному випадку при паралельному стрімі колекція може бути розділеною на дві частини (або більше частин), в кожній з яких буде свій акумулятор StringBuilder і потім необхідно буде їх об'єднати, то код до Java 8 при 2 потоках буде таким:

StringBuilder b1 = new StringBuilder();

for(String s: stringsPart1) {

b1.append(s).append(" , ");

}

StringBuilder b2 = new StringBuilder();

for(String s: stringsPart2) {

strings b2.append(s).append(" , ");

}

StringBuilder b = b1.append(b2).append(","), //метод\_з’єднання\_двох\_акумуляторів

String joinBuilderOld = b.toString();

Напишемо свій аналог Collectors.toList() для роботи з рядковим стрімом:

// Напишемо свій аналог toList

Collector<String, List<String>, List<String>> toList = Collector.of( ArrayList::new,

List::add,

(l1, l2) -> { l1.addAll(l2); return l1; }

);

List<String> distinct1 = strings.stream().distinct().collect(toList);

За допомогою, методів *parallel* і *sequential* можна визначати які операції можуть бути паралельними, а які тільки послідовними. Так само з будь-якого послідовного стріму можна зробити паралельний і навпаки, тобто:

collection.stream(). peek(...). // операція послідовна

parallel(). map(...). // операція може виконуватися паралельно,

sequential(). reduce(...) // операція знову послідовна

**Рекомендації по** **використанню паралельних стрімів**

Щоб отримати паралельний стрім, потрібно або викликати метод *parallelStream* () замість *stream* (), або перетворити звичайний стрім в паралельний, викликавши проміжний оператор *parallel*.

Робота з потоконебезопаснимі колекціями, розбиття елементів на частини, створення потоків, об'єднання частин воєдино, все це криється в реалізації Stream API. Від нас лише потрібно звернутися до потрібного методу простежити, щоб функції в операторах не залежали від будь-яких зовнішніх факторів, інакше є ризик отримати невірний результат або помилку.

Вкрай не рекомендується використовувати паралельні стріми для скільки-небудь тривалих операцій (отримання даних з бази, мережевих з'єднань), так як всі паралельні стріми працюють c одним пулом *fork / join* і такі довгі операції можуть зупинити роботу всіх паралельних стрімів в JVM з- за відсутності доступних потоків в пулі, тобто паралельні стріми варто використовувати лише для коротких операцій, де рахунок йде на мілісекунди, але не для тих, де рахунок може йти на секунди і хвилини.

Не використовуйте паралельні стрім всюди, де тільки можна. Витрати на розбиття елементів, обробку в іншому потоці і подальше їх злиття часом більше, ніж виконання в одному потоці.

При використанні паралельних стрімів, переконайтеся, що ніде немає блокуючих операцій або чогось, що може перешкодити обробці елементів.

**Інтерфейс Spliterator**

Елементи стримів потрібно не тільки итерувати, але ще й розділяти на частини і віддавати іншим потокам. За ітерацію і розбиття відповідає *Spliterator*. Він навіть звучить як Iterator, тільки з приставкою Split — розділяти.

Методи інтерфейсу:

- trySplit — як випливає з назви, намагається розділити елементи на дві частини. Якщо це зробити не виходить, або елементів недостатньо для поділу, то поверне null. В інших випадках повертає ще один Spliterator з частиною даних.

- tryAdvance(Consumer action) — якщо є елементи, для яких можна застосувати дію, то воно застосовується і повертає true, в іншому випадку повертається значення false, але дія не виконується.

- estimateSize() — повертає приблизну кількість елементів, що залишилися для обробки, або Long.MAX\_VALUE, якщо стрім нескінченний або порахувати кількість неможливо.

- characteristics() — повертає характеристики сплитератора.

**Характеристики Spliterator**

У методі sorted і distinct було згадано, що якщо стрім позначений як відсортований або містить унікальні елементи, то відповідні операції проводитися не будуть. Ось характеристики сплитератора і впливають на це.

- DISTINCT — всі елементи унікальні. Сплитератори всіх реалізацій Set містять цю характеристику.

- SORTED — всі елементи відсортовані.

- ORDERED — порядок має значення. Сплитератори більшості колекцій містять цю характеристику, а у HashSet, приміром, її немає.

- SIZED — кількість елементів точно відомо.

- SUBSIZED — кількість елементів кожної розбитої частини точно відомо.

- NONNULL — в елементах не зустрічається null. Деякі колекції з java.util.concurrent, в які можна покласти null, містять цю характеристику.

- IMMUTABLE — джерело є іммутабельним і в нього можна додати більше елементів, або видалити їх.

- CONCURRENT — джерело лояльний до будь-яких змін.

Зрозуміло, характеристики можуть бути змінені при виконанні ланцюжка операторів. Наприклад, після sorted додається характеристика SORTED, після filter втрачається SIZED і т. д.

## Lambda-вирази

### 2.10.1 Концепція Lambda-виразу

Мова Java з самого спочатку свого існування була і залишається повністю об'єктно-орієнтованою мовою. Тому до Java 8 не існувало можливості визначити окремо поза класом якийсь метод (функцію). І формально не було можливості передати метод як аргумент або повернути тіло методу як результат іншого методу.

Тому приходилося, наприклад, у *Swing* створювати анонімні класи для обробників подій, щоб скоротити код при передачі функціональності метода. Так, у прикладі нижче, передається деякий код в метод-слухач події миші. Для цього визначається анонімний клас MouseAdapter і відразу створюється об'єкт з нього. Описавши потім метод mouseClicked цього анонімного класу, ми тим самим передали його функціональність в метод *addMouseListener*.

someObject.addMouseListener(new MouseAdapter() {

public void mouseClicked(MouseEvent e) {

//тут реалізація обробки події слухача...

}

});

Аналогічна ситуація - при створенні потоку виконання:

new Thread(new Runnable() {

public void run() {

...

}

}).start();

Тут ми об’єднуємо створення класу – нового потоку потоку Thread і створення нового класу на основі інтерфейса *Runnable* для того щоб перевизначити його метод *run*. А оскільки цей клас ніде більше не використовується, то ми можемо створити анонімний клас (без назви) .

Як видно з прикладів, не дуже просто в Java передати простий метод (функціональність) в якості аргумента. Це обмеження змусило розробників *Java 8* додати в специфікацію мови таку конструкцію як *Lambda-вираз*.

Взагалі в функціональній мові *lambda-вирази* - це функції; але в Java, *lambda*-вирази - представляються об'єктами, і повинні бути пов'язані з конкретним об'єктним типом, який називається *функціональний інтерфейс*.

Виходячі з попередніх прикладів, можна сказати, що в Java Lambda-вираз - це анонімна функція, тобто, це метод без оголошення, без модифікаторів доступу, який повертає значення і ім'я. Lambda-вираз дозволяє написати метод і відразу ж використати його. Особливо це корисно в разі одноразового виклику методу, тому що скорочується час на оголошення і написання методу без необхідності створювати клас.

Lambda-вирази в Java зазвичай мають наступний синтаксис:

***(аргументи) -> (тіло)****.*

Наприклад:

(int a, int b) -> { return a + b; }

() -> System.out.println("Hello World");

(String s) -> { System.out.println(s); }

() -> 42

() -> { return 3.1415 };

В *Java 8* введено спеціальний вид інтерфейсів - *функціональні* інтерфейси (*Functional* Interface) , які мають тільки один абстрактний метод. Саме завдяки цій властивості, як ми побачимо далі, їх дуже зручно використовувати в *Lambda-виразах*. (Крім них є ще так звані *маркерні* інтерфейси (*Marker* interface) - це інтерфейси без оголошення методів і полів, тобто порожні інтерфейси).

*java.lang.Runnable* - це приклад функціонального інтерфейсу. У ньому оголошений тільки один метод *void run ()*. Інтерфейс *ActionListener* - теж функціональний. Раніше нам доводилося використовувати анонімні класи для створення об'єктів, що реалізують функціональний інтерфейс. З *Lambda*-виразами, все стало простіше. Кожний lambda-вираз може бути неявно прив'язаний до якогось функціональному інтерфейсу. Наприклад, можна створити посилання на *Runnable* інтерфейс, як показано в наступному прикладі:

Runnable r = () -> System.out.println("hello world");

Створити потік і запустити його з використанням *lambda*-виразу можна не вказуючі назву інтерфейса Runnable і його метода тому що вони тільки в одному екземплярі і компілятор сам розпізнає їх:

new Thread(

() -> System.out.println("hello world")

).start();

Допоможе компілятору ще й анотація @*FunctionalInterface*, додана в Java 8 згідно Java Language Specification. Вона примушує компілятор перевірити, чи є оголошений інтерфейс функціональним. @*FunctionalInterface* видасть помилку компіляції, якщо оголошений інтерфейс не буде функціональним.

Немає необхідності в *lambda*-виразі, також, вказувати тип значення, яке повертається методом і його модифікатор доступу – компілятор і це «побачить» сам.

Крім того, в Java 8 включено декілька готових функціональних інтерфейсів для використання з *Lambda*-виразами.

Приклади використання функціонального інтерфейсу класичним способом та за допомогою *Lambda*-виразу показані нижче:

@FunctionalInterface

public interface WorkerInterface {

public void doSomeWork(int a, int b);

}

public class WorkerInterfaceTest {

public static void **main**(String [] args) {

// перевизначення методу doSomeWork інтерфейсу WorkerInterface з //використанням створеного об’єкту на основі інтерфейсу

// (класичний спосіб)

WorkerInterface operation =new WorkerInterface(){

@Override

public long doSomeWork(int a, int b){

return a+b;

}

};

// перевизначення методу doSomeWork через Lambda-вираз

// (нововведення Java 8)

WorkerInterface **lambda=(a,b)-> a \* b;**

// виклик метода doSomeWork через об’єкт operation

// (класичний спосіб)

System.out.println(operation.doSomeWork(2, 3));

// виклик метода doSomeWork через об’єкт lambda**,** в якому також

// як і в operation перевизначений метод doSomeWork, тільки

//за допомогою Lambda-виразу

System.out.println(lambda.doSomeWork(2, 3));

}}

Результат: 5

6

Тут ми визначили свій власний функціональний інтерфейс і скористалися *lambda*-виразом для перевизначення його методу.

Як видно з прикладу, використовуючи новий синтаксис Java 8 у круглих дужках ми вказали **набір параметрів, а після стрілки тіло методу.**

Спеціальний символ **стрілка** (дефіс і кутова дужка) вставляється, щоб розділити набір параметрів і тіло метода. Такий вираз з трьох елементів і називається *Lambda*-виразом. Зліва від стрілки – *параметри* Lambda-виразу, а праворуч його *тіло*.

Розглянемо ще кілька варіантів запису *Lambda*-виразу, які дозволяють скоротити його запис.

Якщо є інтерфейс з методом, який приймає два параметри з певними типами, як ми бачили в попередньому прикладі, то типи можна не вказувати (компілятор сам це з’ясує) і тоді лямбда-вираз буде мати такий вигляд:

(a, b) -> {

// …

}

Якщо у методі інтерфейсу немає параметрів, то треба писати пусті круглі дужки, тобто:

() -> {

// …

}

Якщо у методі інтерфейсу є *тільки один параметр*, то можна взагалі *не вставляти круглі дужки*.

Якщо метод інтерфейсу має тип, що повертається, наприклад,

interface MySecondInterface {

abstract int onSecondMethod(int a, int b);

}

то лямбда-вираз буде мати такий вигляд:

(a, b) -> {

return a + b;

}

Якщо тіло методу містить тільки один рядок коду, можна видалити фігурні дужки {}. Якщо ви видаляєте фігурні дужки, вам також необхідно видалити крапку з комою, залишивши наступне:

(a, b) -> return a + b

Якщо в методі тільки один рядок коду і він повертає значення за допомогою слова *return*, то і його також можна не писати (компілятор це зрозуміє):

(a, b) -> a + b

Якщо у методі багатостроковий код, то тоді ми повинні вставляти фігурні дужки:

(a, b) -> {

a+=1;

return a + b;

}

Розглянемо ще посилання на методи з використанням Lambda-виразів. Припустимо, у нас є інтерфейс і метод, який цей інтерфейс приймає як параметр:

public interface Callback {

public void onEvent(int event);

}

public void myMethod(Callback callback){

}

Для того, щоб в методі myMethod викликати метод onEvent інтерфейсу Callback (з перевизначенням методу onEvent), то без лямбда-виразу це виглядало б так:

myMethod(new Callback() {

@Override

public void onEvent(final int state) {

System.out.println(state);

}

});

Додавши лямбда-вираз, як ми це робили раніше, отримаємо наступне:

myMethod(state -> System.out.println(state)); (1)

В нашому прикладі в методі використовується однорядковий код і зовнішній метод, що приймає один параметр і він співпадає з параметром Lambda-виразу, то ми можемо передавати посилання на метод в такому вигляді:

myMethod(System.out::println); (2)

Рядок (1) і рядок (2) – ідентичні. Замість *System.out* могла би стояти назва будь якого статичного класу (або назва об’єкту нестатичного класу), а після крапки назва його методу.

Параметр буде передаватися автоматично, без необхідності використання проміжного іншого кода, який непотрібний.

Чому тут використовується спеціальний символ ::, а не крапка? У *Java* дозволяється мати метод і поле з однаковим іменем, тому, якщо ми поставимо крапку, то компілятор може не зрозуміти, що буде за нею. Символ :: вказує на те, що тут викликається метод.

Ще одна особливість лямбда-виразів: ефективно фіналізовані зовнішні змінні або final-змінні можна вставляти в лямбда-вирази. Слово «ефективно» означає, що перед змінною не обов’язково повинно стояти слово «final», але ця змінна не повинна змінюватися. Якщо ми її спробуємо змінити, то компілятор виведе повідомлення про помилку.

### 2.10.2 Використання *stream* з *Lambda-виразами*

В java 8 з’явився клас *Stream*. В цьому класі є набір методів, які в якості параметрів приймають функціональні інтерфейси, а значить на їх місце ми можемо підставити Lambda-вирази і їх функції будуть виконуватися для кожного об’єкта колекції, якщо застосувати stream до колекції.

У stream є метод *forEach(Person p)*, параметром якого може бути інтерфейс Consumer з пакету *java.util.function* - це новий пакет java 8. У цього абстрактного метода є один параметр – черговий елемент колекції, а тіло цього методу ми повинні написати самі.

Для пркладу створимо клас:

public class Person {

private String name;

private int age;

//Constructor and standard getters

@Override()

public String toString(){

return String.format("%s-%d", this.getName(), this.getAge());

}

}

І використаємо його для створення колекції людей, до якої застосуємо різні методи класу stream, а також Lambda-вирази.

public class Program {

public static void main(String[] args) {

List<Person> persons=new ArrayList<>();

Person se = new Person("Сергій", 38);

Person d = new Person("Даша", 7);

Person g = new Person("Глаша", 3);

Person sa = new Person("Саша", 6);

Person a = new Person("Анна", 18);

persons.add(se);

persons.add(d);

persons.add(g);

persons.add(sa);

persons.add(a);

//повний варіант для виведення

persons.stream().forEach(

(Person p)->{System.out.println(p);});

//або скорочений варіант

// p->System.out.println(p));

//або ще коротше

// persons.stream().forEach(System.out::println);

}

}

Тобто, тут для кожного елемента колекції повинен бути викликаний метод *System.out::printl*n. Таким чином, використання класу *stream* з колекцією дуже ефективне.

Поки що ми просто записали іншим способом те, що могли б реалізувати. класичним оператором *forEach.*

Суттєві переваги використання цих нових можливостей java 8 будуть виявлятися в операціях *фільтрації* та *сортування*. Наприклад, в наступному записі з усіх людей нашого класу ми виберемо тільки тих, які є старшими за 18 років:

persons.stream().filter(p -> p.getAge() >= 18).

forEach(System.out::println);

Тут крім фільтрації можна виконати ще й сортування, тобто через крапку за фільтром написати ще й: *sorted().*

Ми дописуємо ці методи через крапку, тому що вони і багато інших методів належать *stream.* Метод *sorted* в якості параметра приймає так званий компаратор. Це також функціональній інтерфейс, реалізація якого дозволяє порівнювати два об’єкти. Є інший метод Sorted() без параметра-компаратора, але він працює тільки в ситуації, коли сам клас об’єктів (в нашому прикладі це клас *Person)* реалізує інтерфейс *comparable*, і тоді об’єкти цього класу можуть порівнюватися один з одним.

У компаратора два параметра (оскільки він порівнює два об’єкти) і їх типи можна не писати. Повертає він ціле значення: додатнє, якщо перший елемент більше другого, від’ємне, якщо напраки і нуль, якщо вони рівні. А оскільки цей параметр повертається за допомогою слова *return,* то його ми можемо не писати, як і фігурні дужки.

Реалізацію метода порівняння для нашого прикладу ми зробимо такою, щоб елементи колекції сортувалися по імені людини. Тоді повний вираз для фільтрації і сортування колекції об’єктів *Person по імені* буде таким:

persons.stream().filter(p -> p.getAge() >= 18).

sorted((p1,p2)-> p1.getName().compareTo(p2.getName())).

forEach(System.out::println);

Такий короткий декларативний опис нашої програми став можливий завдяки використанню лямбда-виразів. Якби ми в дужках фільтра використовували анонімні класи, то вирази були б набагато довшими.

Методи класа stream, які ми тут використовуємо, називаються *агрегатними операціями*. Іншими методами є:

*mapTo* – його різні варіанти дозволяють виводити не об’єкти, а, наприклад, поля класа

*map(p->p.getName())*

Тут результатом буде не колекція об’єктів, а колекція рядків-імен, які теж можна вивести за допомогою *println* (навіть без перевизначення методу toString , як у нашому класі *Person*). Тобто тут ми маємо перетворення колекції.

Тут уже після методу *map* параметром для forEach буде рядок-ім’я і його ми виводимо за допомогою System.out::println(name)

forEach(

(String name)->System.out.println(name);

Тут тип парамета лямбда-виразу співпадає з типом параметра метода *println,* виклик якого в одному екземплярі записано в *тілі* лямбда виразу.Тобто метод *println* фактично є реалізацією лямбда-виразу. І той і другий мають тип *String.*Тоді ми можемо скоротити запис*:*

forEach(System.out::println);

Можна ще застосувати і функцію average для того, щоб обчислити середній вік людей в колекції. Тоді повний вираз з фільтрацією, перетворенням типів даних і обчисленням середнього віку буде мати вигляд:

double averageAge = persons.stream().

filter(p -> p.getAge() >= 18).

mapToInt(p -> p.getAge()).

average().getAsDouble();

System.out.println(averageAge);

## Література

1. Кетті Сьєрра, Берт Бейтс. Head First Java. Легкий для сприйняття довідник. 2 видання оновлене, - Фабула, 2022. – 720 с.
2. Васильєв О. Програмування мовою Java. – Навчальна книга – Богдан, 2020. – 696 с.
3. Тарнавський Ю. А. Java-програмування: комп’ютерний практикум: навч. посіб. Київ: КПІ ім. Ігоря Сікорського, 2021. – 95 с.
4. Галкін О.В., Катеринич Л.О., Шкільняк О.С. Програмування на Java 8: Навчальний посібник для студентів факультету комп’ютерних наук та кібернетики. – К.: ЛОГОС, 2017. – 186 с.
5. Ткаченко О.А., Ткаченко О.І., Овчарук І.В. Сучасні парадигми програмування. Навч. посіб. Ч.2. – К.: КНУКіМ, 2017. – 305 с.
6. [C. Ullenboom](https://www.amazon.com/Christian-Ullenboom/e/B004585BNO/ref=dp_byline_cont_book_1). Java: The Comprehensive Guide to Java Programming for Professionals. – Rheinwerk Computing, 2022. – 1128 p.
7. B. Evans, J. Clark, M. Verburg. The Well-Grounded Java Developer. – Paperback, 2022. – 704 p.
8. Y. Daniel Liang. Introduction to Java Programming and Data Structures. – Pearson, 2020. – 1240 p.
9. Bruce Eckel. On Java 8. – MindView LLC, 2017. – 1316 p.
10. Downey A.B., Mayfield C. Think Java: How to Think Like a Computer Scientist Version 6.1.3 /A.B.Downey, C. Mayfield. – Massachusetts: Green Tea Press, 2016. – 291 p.
11. Evans В. Java: The Legend /В. Evans. – Publisher: O'Reilly Media, 2015. – 769 p.
12. Aniche M. Effective Software Testing: A developer's guide. Shelter Island, USA: Manning, 2022. – 328 p.
13. [www.java.com](http://www.java.com)
14. <https://docs.oracle.com/javase/tutorial/>
15. http://www.iwanoff.inf.ua/java\_ua/
16. [https://www.cs.cmu.edu/afs/cs.cmu.edu/user/gchen/www/download/java/ LearnJava.pdf](https://www.cs.cmu.edu/afs/cs.cmu.edu/user/gchen/www/download/java/%20LearnJava.pdf)
17. <https://book.huihoo.com/goalkicker.com/JavaBook/JavaNotesForProfessionals.pdf>
18. <https://document.kdu.edu.ua/metod/2019_1443.pdf>
19. http://ni.biz.ua/14/
20. <https://www.digitalocean.com/community/tutorials/java-lock-example-reentrantlock>
21. <https://introcs.cs.princeton.edu/java/10elements/>
22. https://www.oracle.com/java/technologies/
23. <http://uk.wikipedia.org/wiki/Java/>
24. <https://javaprogramming.language-tutorial.com/>
25. <https://kievoit.ippo.kubg.edu.ua/kievoit/2016/46_java/index.html>
26. <https://vdocuments.mx/java-questions-answers.html>

## Зміст

[Вступ 3](#_Toc162676204)

[Частина 1. Основи мови програмування JAVA 4](#_Toc162676205)

[1.1. Загальна характеристика мови Java 4](#_Toc162676206)

[1.1.1 Історія створення мови JAVA 4](#_Toc162676207)

[1.1.2 Відмінності мов JAVA та С++ 5](#_Toc162676208)

[1.2. Лексичні основи мові JAVA 6](#_Toc162676209)

[1.2.1 Структура програми на мові JAVA 6](#_Toc162676210)

[1.2.2 Лексичні основи мови 7](#_Toc162676211)

[1.3. Консольне введення/виведення 13](#_Toc162676212)

[1.3.1 Консольне виведення 13](#_Toc162676213)

[1.3.2 Консольне введення 15](#_Toc162676214)

[1.4. Типи даних 16](#_Toc162676215)

[1.4.1 Прості типи даних 16](#_Toc162676216)

[1.4.2 Визначення локальних змінних 18](#_Toc162676217)

[1.4.3 Класи – оболонки типів 20](#_Toc162676218)

[1.5. Арифметичні та логічні оператори 29](#_Toc162676219)

[1.5.1 Арифметичні оператори 29](#_Toc162676220)

[1.5.2 Бітові оператори 31](#_Toc162676221)

[1.5.3 Оператори порівняння 32](#_Toc162676222)

[1.5.4 Булеві логічні оператори 33](#_Toc162676223)

[1.5.5 Пріоритети операторів 36](#_Toc162676224)

[1.6. Класи і об’єкти 36](#_Toc162676225)

[1.6.1 Класи 36](#_Toc162676226)

[1.6.2 Об’єкти 39](#_Toc162676227)

[1.6.3 Посилання. Збирання сміття 40](#_Toc162676228)

[1.7. Методи і конструктори 41](#_Toc162676229)

[1.7.1 Методи 41](#_Toc162676230)

[1.7.2 Конструктори 45](#_Toc162676231)

[1.7.3 Суміщення (перевантаження) методів і конструкторів 46](#_Toc162676232)

[1.7.4 Передача параметрів методам 49](#_Toc162676233)

[1.7.5 Вбудовані методи (функції) 51](#_Toc162676234)

[1.8. Перерахування 53](#_Toc162676235)

[1.9. Оператори управління виконанням програми 54](#_Toc162676236)

[1.9.1 Умовні оператори 54](#_Toc162676237)

[1.9.2 Оператори циклу 57](#_Toc162676238)

[1.10. Масиви 62](#_Toc162676239)

[1.10.1 Одновимірні масиви 62](#_Toc162676240)

[1.10.2 Багатовимірні масиви 68](#_Toc162676241)

[1.10.3 Класи Arrays та ArrayList 69](#_Toc162676242)

[1.11. Обробка виняткових ситуацій 70](#_Toc162676243)

[1.11.1 Поняття виключення 70](#_Toc162676244)

[1.11.2 Типи виключень 70](#_Toc162676245)

[1.11.3 Розділи try і catch 72](#_Toc162676246)

[1.11.4 Оператор throw 74](#_Toc162676247)

[1.11.5 Ключове слово throws 75](#_Toc162676248)

[1.11.6 Розділ finally 76](#_Toc162676249)

[1.12. Пакети 78](#_Toc162676250)

[1.13. Абстрактні класи і інтерфейси 81](#_Toc162676251)

[1.13.1 Абстрактні класи 81](#_Toc162676252)

[1.13.2 Інтерфейси 83](#_Toc162676253)

[1.14. Робота з рядками 88](#_Toc162676254)

[1.14.1 Клас String 88](#_Toc162676255)

[1.14.2 Класи StringBuilder та StringBuffer 95](#_Toc162676256)

[1.14.3 Клас StringTokenizer 98](#_Toc162676257)

[1.15. Потоки виконання 99](#_Toc162676258)

[1.15.1 Поняття потоку виконання 99](#_Toc162676259)

[1.15.2 Створення потоку - клас Thread 100](#_Toc162676260)

[1.15.3 Створення потоку - інтерфейс Runnable 101](#_Toc162676261)

[1.15.4 Пріоритети потоків 103](#_Toc162676262)

[1.15.5 Синхронізація потоків 104](#_Toc162676263)

[1.15.6 Взаємодія потоків 110](#_Toc162676264)

[1.15.7 Стан потоку 116](#_Toc162676265)

[1.15.8 Використання загальних ресурсів – змінних 117](#_Toc162676266)

[1.15.9 Фонові потоки 120](#_Toc162676267)

[1.16. Робота с базами даних 122](#_Toc162676268)

[1.16.1 ODBC і JDBC 122](#_Toc162676269)

[1.16.2 Встановлення з'єднання з БД 123](#_Toc162676270)

[1.16.3 Створення та виконання запитів 124](#_Toc162676271)

[1.16.4 Обробка результатів запиту 125](#_Toc162676272)

[1.16.5 Створення бази даних 127](#_Toc162676273)

[1.16.6 Створення таблиці 128](#_Toc162676274)

[1.17 Реалізація принципів об’єктно-орієнтованого програмування 130](#_Toc162676275)

[1.17.1 Спадкування 130](#_Toc162676276)

[1.17.2 Інкапсуляція 135](#_Toc162676277)

[1.17.3 Поліморфізм 139](#_Toc162676278)

[1.17.4 Приведення типів об’єктів 140](#_Toc162676279)

[Частина 2. Розширені можливості мови Java 141](#_Toc162676280)

[2.1. Потокове введення / виведення даних 141](#_Toc162676281)

[2.1.1 Потоки для різних типів даних 141](#_Toc162676282)

[2.1.2 Потоки байтового введення/виведення 142](#_Toc162676283)

[2.1.3 Потоки символьного введення/виведення 145](#_Toc162676284)

[2.1.4 Файлові потоки введення/виведення 145](#_Toc162676285)

[2.1.5 Буферезоване введення/виведення символів та рядків 148](#_Toc162676286)

[2.1.6 Потоки для роботи з примітивними типами 149](#_Toc162676287)

[2.1.7 Об’єктні потоки 152](#_Toc162676288)

[2.1.8 Звільнення ресурсів 153](#_Toc162676289)

[2.2. Вкладені класи 155](#_Toc162676290)

[2.2.1 Типи вкладених класів 155](#_Toc162676291)

[2.2.2 Статичний клас 155](#_Toc162676292)

[2.2.2 Внутрішній клас 156](#_Toc162676293)

[2.2.3 Локальний клас 158](#_Toc162676294)

[2.2.4 Анонімний локальний клас 158](#_Toc162676295)

[2.2.5 Використання вкладених класів 160](#_Toc162676296)

[2.2.6 Вкладені класи і віртуальна машина 161](#_Toc162676297)

[2.3. Узагальнення 162](#_Toc162676298)

[2.3.1 Поняття узагальнення 162](#_Toc162676299)

[2.3.2 Узагальнений метод 167](#_Toc162676300)

[2.3.3 Обмеження на параметр типу 168](#_Toc162676301)

[2.3.4 Обмеження на узагальнення 169](#_Toc162676302)

[2.3.5 Узагальнення і спадкування 170](#_Toc162676303)

[2.3.6 Підстановлювальний тип 171](#_Toc162676304)

[2.4. Серіалізація 173](#_Toc162676305)

[2.4.1 Серіалізація і десеріалізація 173](#_Toc162676306)

[2.4.2 Серіалізація складних об’єктів 174](#_Toc162676307)

[2.4.3 Користувальницька серіалізація 176](#_Toc162676308)

[2.5. Колекції 178](#_Toc162676309)

[2.5.1 Інтерфейс Collection та клас Collections 178](#_Toc162676310)

[2.5.2 Інтерфейс Iterator 181](#_Toc162676311)

[2.5.3 Спискові інтерфейси: List, Queue, Deque 181](#_Toc162676312)

[2.5.4 Множина Set 184](#_Toc162676313)

[2.5.5 Карти відображень Map 185](#_Toc162676314)

[2.6. Рефлексія 188](#_Toc162676315)

[2.6.1 Тип Class 188](#_Toc162676316)

[2.6.2 Отримання інформації про модифікатори 192](#_Toc162676317)

[2.6.3 Отримання інформації про елемент класу 193](#_Toc162676318)

[2.7 Графічний інтерфейс користувача 196](#_Toc162676319)

[2.7.1 AWT 196](#_Toc162676320)

[2.7.2 Swing 197](#_Toc162676321)

[2.7.3 JavaFX 207](#_Toc162676322)

[2.8 Багатопотокове програмування - пакет java.util.concurrent 212](#_Toc162676323)

[2.8.1 Переваги пакету java.util.concurrent 212](#_Toc162676324)

[2.8.2 Інтерфейси *Lock та Condition* 214](#_Toc162676325)

[2.8.3 Інтерфейс *Callable* 221](#_Toc162676326)

[2.8.4 Пул потоків та виконавці 225](#_Toc162676327)

[2.8.5 Блокуючі черги 231](#_Toc162676328)

[2.8.6 Синхронізуючі класи-колекції 234](#_Toc162676329)

[2.8.7 Синхронізатори загального призначення 237](#_Toc162676330)

[2.8.8 Атомарні класи та змінні 249](#_Toc162676331)

[2.9 Stream API 253](#_Toc162676332)

[2.9.1 Концепція потоку stream 253](#_Toc162676333)

[2.9.2 Способи створення стрімів 256](#_Toc162676334)

[2.9.3 Методи роботи зі стримами 257](#_Toc162676335)

[2.10 Lambda-вирази 266](#_Toc162676336)

[2.10.1 Концепція Lambda-виразу 266](#_Toc162676337)

[2.10.2 Використання *stream* з *Lambda-виразами* 271](#_Toc162676338)

[Література 274](#_Toc162676339)

[Зміст 275](#_Toc162676340)