МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

М. В. Олексів

КРОСПЛАТФОРМНІ ЗАСОБИ ПРОГРАМУВАННЯ

КОНСПЕКТ ЛЕКЦІЙ

для студентів Інституту комп’ютерних технологій, автоматики та метрології

Затверджено
на засіданні кафедри
”Електронні обчислювальні машини”
Протокол № 1 від 28.08.2023 р.

Львів 2023

Олексів М. В. Кросплатформні засоби програмування: конспект лекцій для студентів Інституту комп'ютерних технологій, автоматики та метрології – Львів, Видавництво Національного університету «Львівська політехніка», 2023. – 166 с.

Конспект лекцій складається з двох частин.

У першій частині висвітлено питання термінології Java, наведено класифікацію платформ Java, проаналізовано характеристики різних версій мови Java, а також наведено засоби для розробки програм мовою Java. Розглянуто основні типи даних та мовні конструкції їх опрацювання. Наведено особливості реалізації парадигм об'єктно-орієнтованого програмування мовою Java, а також розглянуто механізм рефлексії, анотацій, виключень, лямбда виразів, потокової обробки даних, роботу з файлами та параметризоване програмування.

У другій частині висвітлюються основи розробки програмного забезпечення мовою Python. Наведено різновиди платформ Python. Розглянуто основні типи даних та мовні конструкції їх опрацювання. Наведено особливості реалізації парадигм об'єктно-орієнтованого програмування мовою Python, а також розглянуто механізм виключень та роботу з файлами.

Відповідальний за випуск: Мельник А. О., професор, завідувач кафедри

Рецензент

Зміст

Частина 1.

Основи Java

1. Вступ до Java

План

1. Термінологія Java.
2. Загальна характеристика мови Java.
3. Класифікація платформ Java.
4. Аналіз версій платформи Java SE.
5. Середовище виконання Java програм.
6. Засоби для розробки програм мовою Java.
7. Загальні вимоги до програм мовою Java.
8. Структура консольної програми на мові Java.
9. Коментарі та автоматична генерація документації.

Термінологія Java

Java – кросплатформенна об'єктно-орієнтована мова програмування, що була початково розроблена Джеймсом Гослінгом (James Gosling) з Sun Microsystems (у 2010 р. куплена Oracle). Дата офіційного випуску 23.05.1995 р.

Таблиця 1.1.

Зведена таблиця усталеної термінології Java

Термін	Абревіатура	Зміст темріну
Java Development Kit	JDK	Програмне забезпечення для розробки програм мовою Java.
Software Development Kit	SDK	Застарілий термін для JDK
Java Runtime Environment	JRE	Програмне забезпечення, що необхідне користувачам для виконання Java програм.
Java Virtual Machine	JVM	Середовище в якому виконується байт-код Java. Разом з множиною Java API (стандартні бібліотеки Java) реалізує JRE.
Just-in-time compiler	JIT compiler	Компілятор, який здійснює компіляцію байт-коду у машинний код під час виконання програми, збільшуючи цим самим швидкодію програм.
Java 2	J2	Застарілий термін для версій Java, які були випущені починаючи з 1998 року.
Standard Edition	SE	Платформа Java для настільних систем і простих серверних задач.
Enterprise Edition	EE	Платформа Java для складних серверних задач.
Micro Edition	ME	Платформа Java для мобільних пристроїв
Update	u	Термін Sun для випусків з виправленням помилок (наприклад, Java SE 7u4)

Загальна характеристика мови Java

Автори Java визначили 11 характерних особливостей мови:

1. **Проста** – синтаксис Java, по суті, являє собою спрощений синтаксис C++ в якому відсутні файли заголовків, вказівники, структури, об'єднання, перевантаження операцій, віртуальні базові класи тощо. Іншим аспектом простоти є компактність: програми на мові Java можна запускати на комп'ютерах з обмеженими ресурсами.
2. **Об'єктно-орієнтованість** – мова Java є об'єктно-орієнтованою мовою.
3. **Підтримка розподілених обчислень** – мова Java надає розробнику обширну бібліотеку класів для передачі даних на базі протоколів TCP/IP, HTTP, FTP. Програми на мові Java можуть відкривати об'єкти і одержувати до них доступ через мережу по URL.
4. **Надійність** – програми на мові Java працюють надійно за будь-яких обставин, що забезпечується відсутністю вказівників та механізмами: раннього виявлення помилок; контролю в процесі виконання програм; усунення ситуацій, які можуть викликати помилки.
5. **Безпека** – мова Java дозволяє розробляти системи, які є захищеними від вірусів і несанкціонованого доступу. Система безпеки Java захищає від переповнень стеку, пошкоджень ділянок пам'яті за межами адресного простору процесу, несанкціонованого читання та зміни файлів. Мова Java дозволяє розробляти класи з цифровим підписом.
6. **Кросплатформенність** – байт-код, який утворюється в результаті компіляції програми на мові Java, може виконуватися на будь-яких комп'ютерах на яких встановлена віртуальна машина Java.
7. **Переносимість** – на відміну від C/C++ розмір основних типів даних і операції над ними строго визначені та не залежать від реалізації компілятора.
8. **Інтерпретованість** – інтерпретатор Java може виконувати байт-код Java на будь-якій машині.
9. **Швидкодія** - сьогоднішні реалізації інтерпретаторів Java – JIT-компілятора (just-in-time), транслюють байт-код Java у машинний код в процесі виконання програми, відслідковують і оптимізують ділянки коду, які виконуються найчастіше, відслідковують завантаження класів, широко використовують платформенно орієнтований код в стандартних бібліотеках. Завдяки цьому піднято швидкодію у порівнянні з першими реалізаціями інтерпретаторів Java. На сьогодні програми написані на Java в середньому в 1,5-2 рази повільніші за програми написані на мові C/C++, а подекуди швидші за них. Процесори з архітектурою ARM забезпечують апаратне виконання байт-коду через опцію Jazelle.
10. **Багатопотоковість** – мова Java має вбудовані засоби розробки багатопотокових програм, проте реалізація багатопотоковості лягає на плечі ОС і бібліотеки потоків.
11. **Динамічність** – у бібліотеки Java можна вільно включати нові методи і об'єкти не зачіпаючи запущені на виконання програми, які ними користуються, а також мова дає засоби для аналізу об'єктів в процесі виконання програм.

Класифікація платформ Java

Java має 5 основних платформ:

- **Java SE** – базова платформа Java, що містить компілятори, API, JRE; вона підходить для створення користувацьких додатків, в першу чергу - для настільних систем.
- **Java EE** – платформа Java для створення складного програмного забезпечення рівня підприємства.
- **Java ME** – платформа Java для пристроїв з низькою обчислювальною потужністю, наприклад, мобільних телефонів, КПК, вбудованих систем.
- **JavaFX** - платформа, що є наступним кроком в еволюції Java як Rich Client Platform; вона призначена для створення графічних інтерфейсів корпоративних додатків і бізнесу.
- **Java Card** - платформа Java для пристроїв з сильно обмеженими обсягами пам'яті і потужності.

Аналіз версій платформи Java

Перше покоління Java: JDK 1.0 (23 січня 1996 р.), JDK 1.1 (19 лютого 1997 р.).

Друге покоління Java: J2SE 1.2 (8 грудня 1998 р.), J2SE 1.3 (8 травня 2000 р.), J2SE 1.4 (6 лютого 2002 р.), J2SE 5.0 (30 вересня 2004 р.), Java SE 6 (11 грудня 2006 р.). Подальший розвиток платформи Java SE відображено у таблиці нижче [11].

Таблиця 1.2.

Дорожня карта підтримки Oracle Java SE

Версія	Дата випуску	Преміум підтримка до	Розширена підтримка до
7 (LTS)	Липень 2011	Липень 2019	Липень 2022
8 (LTS)	Березень 2014	Березень 2022	Грудень 2030
9 (non-LTS)	Вересень 2017	Березень 2018	Відсутня
10 (non-LTS)	Березень 2018	Вересень 2018	Відсутня
11 (LTS)	Вересень 2018	Вересень 2023	Вересень 2026
12 (non-LTS)	Березень 2019	Вересень 2019	Відсутня
13 (non-LTS)	Вересень 2019	Березень 2020	Відсутня
14 (non-LTS)	Березень 2020	Вересень 2020	Відсутня
15 (non-LTS)	Вересень 2020	Березень 2021	Відсутня
16 (non-LTS)	Березень 2021	Вересень 2021	Відсутня
17 (LTS)	Вересень 2021	Вересень 2026	Вересень 2029
18 (non-LTS)	Березень 2022	Вересень 2022	Відсутня
19 (non-LTS)	Вересень 2022	Березень 2023	Відсутня
20 (non-LTS)	Березень 2023	Вересень 2023	Відсутня
21 (LTS)	Вересень 2023	Вересень 2028	Вересень 2031

Починаючи з **J2SE 1.2** платформа Java підтримує:

- бібліотеку графічного інтерфейсу Swing;
- колекції;
- Policy файли;
- цифрові сертифікати користувача;
- бібліотеку Accessibility;
- Java 2D;
- технологію drag-and-drop;
- повністю підтримує Unicode;
- відтворення аудіо файлів декількох популярних форматів;
- технологію CORBA;
- JIT-компілятор;
- удосконалені інструментальні засоби JDK, включаючи підтримку профілювання Java-програм.

Починаючи з **J2SE 5.0** платформа Java підтримує:

- типи перелічення (enum), які на відміну від C++ є класами.
- анотації - можливість додавання в текст програми метаданих, які не впливають на виконання коду, але допускають використання для отримання різних відомостей про код і його виконання. З'явився інструментарій для використання анотованого коду. Одне з застосувань анотацій - спрощення створення тестових модулів для Java-коду.
- засоби параметризованого програмування (generics) - механізм, аналогічний Eiffel (пізніше також з'явилися і в C #, принципово відрізняються від шаблонів C + +), що дає можливість створювати класи та методи з полями і параметрами довільного об'єктного типу. Даний механізм використано у новій версії колекцій стандартної бібліотеки Java.
- методи з невизначеною кількістю параметрів.
- Autoboxing / Unboxing - автоматичне перетворення між скалярними типами Java та відповідними типами-оболонками (наприклад, між int і Integer). Наявність такої можливості спрощує код, оскільки виключає необхідність у виконанні явних перетворень типів в очевидних випадках.
- імпорт статичних змінних.
- цикл по колекціям об'єктів (ітератор, foreach).

Починаючи з **Java SE 6** платформа Java:

- перестала підтримувати версії Windows 9x;
- одержала API для підтримки скриптових мов;
- підтримує JDBC 4.0;
- одержала Java Compiler API – API для програмного вибору і виклику компілятора Java;
- одержала удосконалені GUI і JVM.

У **Java SE 7** платформа Java одержала:

- підтримку віртуальною машиною динамічних мов в рамках багатомовної віртуальної машини;
- стиснуті 64-бітові вказівники;
- зміни в рамках Project Coin. Можливості, включені в Project Coin:
    - рядки в switch;
    - автоматичне керування ресурсами;
    - виведення типів при створенні екземпляра параметризованого класу;
    - спрощено виклик методу зі змінним числом аргументів;
    - підтримка підкреслень в якості роздільників цифр для чисел (можливість розбивати візуально числа на розряди, покращує читабельність);
    - підтримка колекцій на рівні мови;
- засоби для паралельного виконання;
- нову бібліотеку вводу / виводу для поліпшення кросплатформенності і підтримки метаданих і символьних посилань (пакети: java.nio.file і java.nio.file.attribute);
- підтримку алгоритмів еліптичної криптографії на рівні бібліотеки;
- XRender для Java 2D, що поліпшує керування можливостями сучасних GPU;
- нове графічне API, яке планували випустити в Java версії 6u10;
- розширення підтримки мережевих протоколів (включаючи SCTP і Sockets Direct Protocol) на рівні бібліотеки;
- оновлення в XML і Unicode.

Від лямбда-функцій, проекту Jigsaw та деяких інших можливостей проекту Coin в Java 7 було вирішено відмовитися. Вони будуть реалізовані в Java 8, випуск якого очікується в середині 2014 року.

У **Java SE 8** платформа Java підтримує:

- замикання (лямбда-функції, аналог анонімних методів у C#);
- анотацію типів;
- тісну інтеграцію з JavaFX;
- методи віртуального розширення;
- потокове API;
- зміни, що не ввійшли в Java SE 7.

У **Java SE 9** платформа Java підтримує:

- Java 9 REPL (JShell);
- методи-фабрики для незмінних List, Set, Map і Map.Entry;
- приватні методи у інтерфейсах;
- Java 9 Module System (проект Jigsaw);
- удосконалення API для роботи з процесами ОС;
- удосконалення оператора Try-With-Resources;
- удосконалення CompletableFuture API;
- reactive Streams;

- diamond operator для анонімних внутрішніх класів;
- удосконалення класу Optional;
- удосконалення Stream API;
- розширення аннотації @Deprecated;
- новий HTTP 2 клієнт;
- додано multi-resolution image API;
- удосконалено збирач сміття;
- інші зміни.

У **Java SE 10**:
- додана підтримка ключового слова var;
- розширено клас Optional;
- уддосконалено API для створення незмінних колекцій;
- удосконалено збирач сміття.

У **Java SE 11**:
- додана підтримка синтаксису локальних змінних для параметрів лямбда виразів;
- додана можливість запуску однофайлових програм без необхідності їх компіляції.
- Версія випущена як найнадійніша заміна Java SE 8.

Короткий огляд ключоових нововведень у інших версіях Java SE наведено у [12-13].

Середовище виконання Java програм (JRE)

Програма, що написана мовою Java, міститься у файлах з розширенням **.java*. Для виконання на віртуальній машині Java (JVM) її слід скомпілювати за допомогою компілятора *javac* у стандартизований кросплатформенний бінарний формат, який типово є **.class* файлом. Програма може складатися з багатьох класів у різних файлах. Для полегшення розгортання програм **.class* файли архівують в **.jar* файл (скорочено від Java архів). Для запуску на виконання програми використовуються інструменти `java` або `javaw`. Вони запускають на виконання JRE та завантажують в нього вказаний клас або архів. Різниця між інструментами полягає в тому, що перший виводить консольне вікно на екран, а другий – ні. Синтаксис команд запуску цих інструментів, які виконуються у командному рядку операційної системи, наступний:

```
java [опції] <class> [ агрументи ... ]
java [опції] -jar <file.jar> [агрументи... ]

javaw [опції] <class> [агрументи... ]
javaw [опції] -jar <file.jar> [агрументи... ]
```

Параметри:

`[опції]` – опції командного рядка ОС, які передаються в JRE.
`<class>` – клас, який містить `main` метод.

`<file.jar>` – ім'я **.jar* файлу, який необхідно запустити на виконання. Використовується лише з опцією `-jar`. Вказаний **.jar* файл має містити файли класів і ресурсів програми. Клас, який містить `main` метод, має бути вказаний в декларативному заголовку Main-Class (Main-Class manifest header).

`[аргументи...]` – аргументи командного рядка, які мають бути передані `main` методу.

Після запуску програми в JRE середовищі JVM здійснює перевірку байт коду на безпечність, яка складається з трьох головних перевірок:

- перевірка чи переходи відбуваються в допустимі місця програми;
- перевірка чи дані є ініціалізовані і посилання є типізовані;
- жорстка перевірка можливості доступу до приватних секцій класів і пакетів.

Перші дві перевірки переважно відбуваються при завантаженні програми в JRE, а остання – в процесі виконання програми, коли одні класи звертаються до полів інших класів.

Після успішного завершення перевірки байт-коду JVM емулює (виконує) набір інструкцій, написаних для JVM, інтерпретуючи їх або використовуючи JIT компілятор. Сучасні JVM використовують JIT компілятор, який одночасно компілює частини байт-коду, що мають аналогічну функціональність. Завдяки цьому зменшується кількість часу, що необхідна для компіляції порівняно з використанням інтерпретатора. Керування пам'яттю при виконанні програми здійснює менеджер пам'яті, що є частиною JVM.

До складу JRE крім JVM входить множина Java API – стандартних бібліотек класів Java (рис. 1.1).

Засоби для розробки програмного забезпечення мовою Java

JDK - крім набору бібліотек для платформ Java SE і Java EE містить компілятор командного рядка *javac* і набір утиліт, які працюють в режимі командного рядка.

NetBeans IDE - безкоштовне інтегроване середовище розробки для всіх платформ Java - Java ME, Java SE і Java EE. Пропагувалося Sun Microsystems, як базовий засіб для розробки ПЗ на мові Java і інших мовах (C, C++, PHP, Fortran тощо).

Eclipse IDE - безкоштовне інтегроване середовище розробки для Java SE, Java EE і Java ME. Пропагується IBM, як базовий засіб для розробки ПЗ на мові Java і інших мовах (C, C++, Ruby, Fortran тощо).

IntelliJ IDEA - інтегроване середовище розробки для платформ Java SE, Java EE і Java ME. Розробник - компанія JetBrains. Розповсюджується в двох версіях: безкоштовна (Community Edition) та комерційна пропрієтарна (Ultimate Edition).

JDeveloper - інтегроване середовище розробки для платформ Java SE, Java EE і Java ME. Розробник - компанія Oracle.

BlueJ - інтегроване середовище розробки програм на мові Java, створене в основному для використання в навчанні, але також підходить для розробки невеликих програм.

Visual Studio Code – універсальне середовище розробки програм з використанням різноманітних технологій. Підтримка мови Java забезпечуєтсья за допомогою різноманітних розширень.

XCode – середовище для розробки програмного забезпечення з використанням різних технологій для Mac OS.

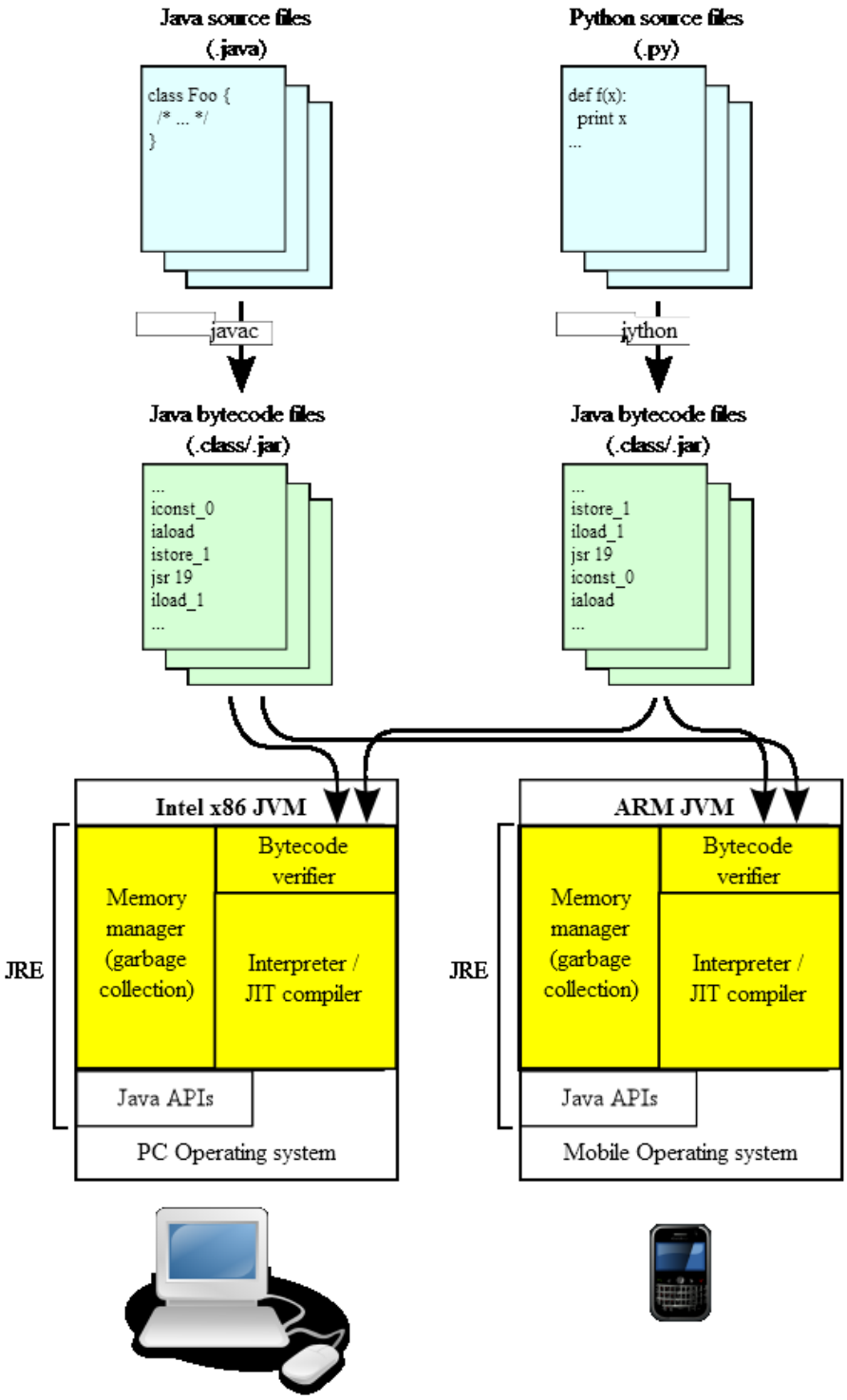

Рис.1.1. Архітектура JRE.

Загальні вимоги до програмування мовою Java

Програми мовою Java розміщуються у файлах з розширенням **.java*, назва яких співпадає з назвою загальнодоступного класу, який міститься у файлі. Програма може складатися з кількох класів. Якщо клас є загальнодоступним, то він розміщується в окремому файлі.

Синтаксис мови Java є регістрозалежним. Назви ідентифікаторів мають починатися з букви, за якою може йти довільна комбінація букв, цифр або знаків підкреслень. Буквою може бути будь-який символ, що відповідає букві з Unicode (в тому числі і українська). Використовувати зарезервовані слова для іменування ідентифікаторів заборонено.

Іменування класів в Java прийнято здійснювати за правилом, яке називається *UpperCamelCase*. Згідно цього правила назва класу починається з великої букви за якою слідують малі букви або цифри. Якщо назва класу складається з кількох слів, то кожне зі слів записується з великої букви підряд без символів-розділювачів.

Назви методів, полів (властивостей), змінних і екземплярів класів (об'єктів) записуються за правилом *lowerCamelCase*, згідно якого назва починається з маленької букви за якою слідують малі букви або цифри. Якщо назва складається з кількох слів, то кожне з них, крім першого, починається з великої літери.

Константи слід іменувати великими літерами.

Кожен оператор у мові Java закінчується крапкою з комою.

Методи можуть приймати 0 або більше *параметрів*. Всі елементи програми мовою Java обов'язково мають знаходитися в середині класів. Оператори в тілі методів розділяються крапкою з комою. Оператори можна записувати у кілька рядків.

Структура консольної Java програми

Структура простої консольної Java програми, яка складається з одного класу, має наступний вигляд:

```
public class НазваКласу
{
      public static void main(String[] args)
      {
            ...
            оператори
            ...
      }
}
```

Модифікатор доступу `public` перед назвою класу робить клас загальнодоступним. Таким чином цей клас має знаходитися у файлі, що називається НазваКласу.java.

Проста консольна програма має містити як мінімум один загальнодоступний клас в якому є загальнодоступний статичний метод `main`, який приймає рядок параметрів, що передаються в програму при її запуску на виконання. З цього методу почнеться виконання програми. Оскільки метод є статичним, то він є доступний для виконання, навіть якщо не створено жодного екземпляру класу, в якому він міститься. Це дозволяє запустити на виконання програму не створюючи попередньо об'єкт класу. Метод `main`, на відміну від C/C++, не повертає код завершення програми в ОС. Якщо програма виконалася коректно, то в ОС повертається 0. Для повернення відмінного від 0 коду завершення програми слід використати метод `exit` класу `System`.

Коментарі та автоматична генерація документації

Java дозволяє використовувати 3 види коментарів:

- рядкові (`//`);
- блочні (`/*…*/`);
- коментарі для автоматичної генерації документації (`/**…*/`); на початку кожного з рядків коментарю може розташовуватися зірочка.

При автоматичній генерації документації використовується утиліта `javadoc`, яка аналізує вміст між `/**` і `*/` та на його базі генерує документацію у форматі *.html. Коментарі між `/**` і `*/` прийнято починати з описового тексту, за яким слідують дескриптори. Використання дескрипторів полегшує як автоматичну генерацію документації, так і розуміння коду, до якого відноситься коментар. Дескриптор, на відміну від решти коментарів, починається з символу `@` за яким слідує ім'я дескриптора. Оскільки документація генерується у форматі *.html, то між `/**` і `*/` допускається розташування html-тегів, включаючи рисунки. При використанні рисунків, їх необхідно розташовувати у каталозі doc-files, на який слід посилатися у адресі рисунку, наприклад, `<img src="doc-files/uml.jpg" alt="UML diagram">`. Щоб не спотворити автоматично згенеровану документацію слід уникати використання тегів `<h1>-<h6>` та `<hr>`.

Для автоматичної генерації документації між `/**` і `*/` можна розмістити:

- коментарі до класу;
- коментарі до методів;
- коментарі до полів;
- загальні коментарі.

Коментарі до класу мають бути розміщені після директив `import` безпосередньо перед визначенням класу. Найчастіше цей коментар має вигляд одного або кількох коротких речень:

```
/**
     Об'єкт класу <code>Person</code> описує особу.
     Особа має властивості: ім'я, прізвище та стать.
*/
```

Коментарі до методів розташовуються безпосередньо перед методами, які вони описують. Крім дескрипторів загального призначення для коментування методів використовуються дескриптори:

- `@param змінна опис`
  Цей дескриптор додає в опис методу розділ "parameters". Опис цього елементу може складатися з кількох рядків та містити html-теги. Всі дескриптори `@param`, що відносяться до одного методу слід групувати разом.
- `@return опис`
  Цей дескриптор додає в опис методу розділ "returns". Опис цього елементу може складатися з кількох рядків та містити html-теги.
- `@throws опис_класу`
  Цей дескриптор додає в опис методу інформацію про класи об'єкти яких можуть генеруватися при виключних ситуаціях. Відомості про кожен клас слід описувати в окремому дескрипторі `@throws`.

Коментарі до полів (властивостей) застосовуються, якщо поля є загальнодоступними:

```
/**
     Чоловіча стать
*/
public static final int SEX=1;
```

Загальні коментарі відображають інформацію загального характеру за допомогою дескрипторів:

- `@author ім'я`
  Цей дескриптор створює розділ "author" у якому відображаються відомості про авторів. Відомості про кожного з авторів записуються в окремих дескрипторах.
- `@version текст`
  Цей дескриптор створює розділ "version" у якому відображається інформація про версію (переважно її номер).
- `@since текст`
  Цей дескриптор створює розділ "since" у якому відображається інформація про версію у якій вперше появилася дана властивість. Наприклад,

`@since version 1.4.4`

- `@deprecated текст`

  Цей дескриптор інформує про те, що дана властивість чи метод застарілі, та часто містить інформацію про те, що слід застосовувати замість них. Наприклад, `@deprecated Use <code>SetVisible(true)</code> instead.`

- `@see посилання`

  Цей дескриптор створює розділ “See also”. Використовується як для класів так і для методів. Посилання записується у одному з трьох можливих виглядів:

  ```
  пакет.клас#елемент мітка
  <a href="…"> мітка </a>
  "текст"
  ```

  В першому варіанті елемент класу, на який відбувається посилання, відділяється від класу, в якому він міститься знаком #. Мітка використовується як якір для посилання і є необов’язковою. При її відсутності якорем для посилання буде ім’я коду або URL. У третьому варіанті текст, який є між подвійними лапками відобразиться в розділі “See also”.

- Спеціальний дескриптор `{link пакет.клас#елемент мітка}` дозволяє включати гіпертекстові посилання в будь-якому місці коментарю. Його застосування підпорядковується таким же самим правилам, що й `@see`.

Пакетні коментарі – застосовуються для автоматичного документування пакетів. Для їх використання слід в кожен каталог пакету добавити файл `package.html`. Весь текст, що міститься між `<body>` і `</body>` буде екстраговано утилітою `javadoc`.

Оглядові коментарі відображаються на екран, коли користувач вибере пункт Overview в панелі навігації. Для їх генерації у кореневому каталозі з лістингами програми слід створити файл `overview.html` в якому між тегами `<body>` і `</body>` слід розмістити інформацію, яку екстрагуватиме утиліта `javadoc`.

Для генерування документації по пакету слід ввести в консолі ОС:

```
javadoc –d каталог_doc ім’я_пакету
```

Опція `–d каталог_doc` задає каталог, де слід розмістити згенеровану документація до пакету.

Висновок

У лекції висвітлено усталену термінологію Java; наведено 11 загальних особливостей мови Java, визначених її авторами; показано класифікацію платформ Java. Відображено історію розвитку Java SE починаючи з версії 1.2 до версії Java SE 8. Розглянуто засоби для розробки програм мовою Java та середовище виконання Java програм. Наведено загальні вимоги до програм мовою Java. Розглянуто приклад

мінімальної консольної програми на мові Java, а також засоби автоматичної генерації супровідної документації до програм.

2. Основні типи даних

План

1. Типи даних.
2. Змінні.
3. Виведення типу локальної змінної.
4. Константи.
5. Масиви.

Типи даних

Мова Java є строго типізованою. Це означає, що тип кожної змінної має бути оголошеним. Мова має 8 основних (простих) типів, які не є класами та однаково представляються на будь-якій машині, де виконується програма (див. табл. 2.1).

Таблиця 2.1.

Основні типи даних мови Java

Тип	Розмір, байти	Діапазон значень	Приклад запису
boolean	1	true, false	true
char	2	\u0000…\uFFFF	\u0041 або 'A'
byte	1	-128...127	15
short	2	-32768…32767	15
int	4	$-2^{31}...2^{31}-1$	15
long	8	$-2^{63}...2^{63}-1$	15L
float	4	±3.4E+38	15.0F
double	8	±1.79E+308	15.0 або 15.0D

Тип `boolean` використовується лише в логічних виразах та не приводиться до інших типів.

Тип `char` використовує кодування UTF-16. Константи цього типу можна задавати символами, або їх кодами з використанням префіксу `\u` та двобайтного коду символьної константи. Спеціальними символами є: `\b`, `\t`, `\n`, `\r`, `\'`, `\"` і `\\`.

Для задання довгої цілочисельної константи (типу `long`) використовується суфікс `L`. У Java передбачено задання цілих чисел у вісімковому (010), десятковому (8) та шістнадцятковому (0x8) форматах. На відміну від C/C++ в Java відсутній беззнаковий тип.

Для задання дробової константи в форматі з одинарною точністю (типу `float`) використовується суфікс `F`. Дробові константи в форматі з подвійною точністю (типу `double`) використовуються по замовчуванню, або явно через використання суфіксу `D`. У

Java передбачено задання дробових чисел у шістнадцятковому вигляді у форматі знак-мантиса-експонента. В цьому записі мантиса представляється шістнадцятковим числом, а експонента – десятковим, наприклад, $0.125 == 2^{-3}$ можна записати як 0x1.0p-3. Всі обчислення, що відбуваються з числами з рухомою комою, відповідають стандарту IEEE-754. Крім значущих чисел в форматі з рухомою комою Java за допомогою класу `Double` підтримує 3 спеціальні значення з плаваючою комою:

- позитивна нескінченність;
- негативна нескінченність;
- нечисло (NaN).

Ці значення можуть використовуватися при перевірці результату обчислення виразів, наприклад:

```
if (Double.isNaN(result)) // перевірка чи результат є числом
{...}
```

У JDK 7 появилася можливість групувати цифрові константи знаками підкреслення «_», яких може бути кілька, але лише між сусідніми цифрами. Наприклад: 123_456__789 еквіваленте запису 123456789, запис 123_456.123__456 еквівалентний запису 123456.123456, а запис 0b1101_0111 еквівалентний запису 11010111b.

Змінні

У мові Java кожна змінна обов'язково має мати тип. Імена змінних є регістрозалежними. Назви змінних мають починатися з букви, за якою може йти довільна комбінація букв, цифр та символів підкреслення. Буквою може бути будь-який символ, що відповідає букві з Unicode (в тому числі і українська). Використовувати зарезервовані слова для іменування змінних заборонено. При іменуванні змінних прийнято використовувати правило *lowerCamelCase.* Обмеження на довжину назви змінної - відсутні. Змінні простих типів зберігаються у стеку. Синтаксис оголошення змінних:

```
тип назваЗмінної[=значення] {, назваЗмінної [= значення]};
```

Наприклад,

```
int i;
double x, y;
boolean isZero = false;
```

Перед використанням змінну слід обов'язково ініціалізувати.

Виведення типу локальної змінної

Починаючи з версії Java SE 10 підтримується виведення типу локальної змінної (local variable type inference) при оголошенні *локальних* (лише локальних) змінних. Завдяки цьому будь-яку локальну змінну, для якої можна динамічно визначити тип під час виконання програми можна оголосити за допомогою слова `var`. Слово `var` не є

ключовим словом, тому якщо у вас є старий код зі змінною `var` ви зможете використовувати слово `var` для виведення типу локальної змінної, і навіть для оголошення змінної var, наприклад:

```
var var = 5;
```

Згідно новітніх напрямків розвитку стилів програмування рекомендовано дотримуватися стилю AAA (almost always auto), який полягає у використанні автоматичного виведення типів майже всюди де тільки можливо при написанні коду. Бувають ситуації, коли без його використання майже неможливо обійтися, наприклад, коли результатом операції може бути або якийсь конкретний тип, або проксі об'єкт. На відміну від звичного оголошення змінних з вказанням типів при використанні виведення типу локальної змінної змінні після слова `var` мають бути ініціалізовані у місці оголошення значенням, тип якого можна однозначно визначити. Це є можливим у наступних ситуаціях:

1. При створенні нового екземпляру класу:

```
var myVarTypeVar = new MyClass();
```

2. У заголовку зміної:

```
for (var i = 1; i < 10; i++){
   //операції
}
```

у даному випадку компілятор виведе для змінної `i` тип `int`. Щоб компілятор вивів інший тип слід використати префікс, який відповідає конкретному типу (див. Типи даних), або явне приведення до типу, наприклад,

```
var a = (short) 5.
```

3. В блоці try-with-resources:

```
void copyFile(File src, File dest) throws IOException {
   try (var reader = new BufferedReader(new
FileReader(src));
      var writer = new BufferedWriter(new
FileWriter(dest))) {
      String s;
      while ((s = reader.readLine()) != null) {
         writer.write(s);
         writer.newLine();
      }
   }
}
```

Наступне оголошення не дає змоги динамічно визначити тип змінної і призведе до помилки компіляції:

```
var a;
a = 5;
```

але наступний код скомпілюється:

```
var a
      = 5;
```

Слово `var` можна застосовувати в середині:

- конструкторів;
- блоків ініціалізації;
- методів;

Слово `var` **не** можна застосовувати:

- при оголошенні полів класу

```
class MyClass
{
      public var a = "Text";
}
```

- при оголошенні статичних полів класу

```
class MyClass
{
      public static var a = "Text";
}
```

- при оголошенні методів класу:

```
public int add(var x, var y) {
  return x + y;
}
```

- при ініціалізації null

```
var myNullVar = null;
```

але наступний код допустимий:

```
var myNullVar = (MyClass) null;
```

- при оголошенні кількох змінних:

```
int a, var b =0;
var a = 0, b = 1;
var a = 0, var b = 1;
```

При використанні ключового слова `final` при оголошенні змінної, вона залишиться константою не залежно від того чи використано слово `var` при її оголошенні чи ні.

Константи

В мові Java для оголошення констант використовується ключове слово `final`, яке вказує що даний ідентифікатор може бути ініціалізований лише в місці оголошення, а його значення в процесі виконання програми не може бути зміненим. Іменувати константи прийнято великими літерами. У назві константи можна використовувати такі ж символи, як і у назвах змінних.

Константи у мові Java можна поділити на дві групи: *константи* і *константи класів*. Константа оголошується у методі і видима лише в ньому. Константа класу оголошується як член класу і видима у всіх методах класу. Константа класу використовується тоді, коли необхідно використовувати іменоване константне значення в кількох методах класу або в інших класах.

Синтаксис оголошення константи:

```
final тип НАЗВА_КОНСТАНТИ = значення;
```

Приклад оголошення константи:

```
public class Constants1
{
      public static void main(Strings[] args)
      {
            final double CM_PER_INCH_1 = 2.54;
            …
      }
}
```

Синтаксис оголошення константи класу:

```
модифікатор_доступу static final тип НАЗВА_КОНСТАНТИ = значення;
```

Приклад оголошення константи класу:

```
public class Constants2
{
```

```
        public static void main(Strings[] args)
        {
              …
        }
        public static final double CM_PER_INCH_2 = 2.54;
}
```

На відміну від константи `CM_PER_INCH_1` константа класу `CM_PER_INCH_2` є статичною (`static`) загальнодоступною (`public`) і оголошеною як член класу. Тому вона видима в усіх методах класу `Constants2` та за його межами навіть за відсутності об'єкту класу `Constants2`, а не лише у методі `main`. З-за меж класу `Constants2` до `CM_PER_INCH_2` можна доступитися так: `Constants2.CM_PER_INCH_2`.

Масиви

Масив – структура даних, що зберігає набір значень однакового типу. Пам'ять під масив виділяється у *керованій кучі*. При завершенні життєвого циклу масиву пам'ять, яку він займав, вивільняється збирачем сміття. Доступ до елементів масиву здійснюється за допомогою індексів. Індексація масивів у Java починається з 0. Для створення масиву у Java необхідно оголосити змінну-масив та ініціалізувати її. При створенні за допомогою оператора `new` масиву чисел всі його елементи ініціалізуються нулями (масиви типу `boolean` ініціалізуються значеннями `false`, масиви об'єктів ініціалізуються значеннями `null`). Після створення масиву змінити його розмір неможливо.

Одновимірні масиви

Синтаксиси оголошення неініціалізованого одновимірного масиву:

```
тип[] змінна;
тип змінна[];
```

Приклади оголошення неініціалізованого одновимірного масиву типу `int`:

```
int[] arr;
int arr[];
```

Синтаксиси оголошення та ініціалізації одновимірного масиву:

```
тип[] змінна = new тип[кількість_елементів_масиву];
тип[] змінна = {значення1, значення2,..., значенняN};
тип змінна[] = new тип[кількість_елементів_масиву];
тип змінна[] = {значення1, значення2,..., значенняN};
```

Приклади оголошення та ініціалізації одновимірного масиву типу `int`:

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

```
int arr[] = {1,2,3,4,5};
```

Java дозволяє створювати і *анонімні масиви* (без іменні). При створенні анонімного масиву відбувається виділення необхідної кількості пам'яті для збереження елементів масиву та ініціалізація масиву значеннями зі списку ініціалізації. Синтаксис створення анонімного масиву:

```
new тип [] { значення1, значення2,..., значенняN };
```

Анонімні масиви корисні тоді, коли необхідно ініціалізувати існуючий масив новими значеннями без створення нової змінної, коли необхідно повернути з методу масив нульової довжини (пустий масив), або коли в метод необхідно передати масив наперед відомих значень. Анонімні масиви дозволяють записати код:

```
int[] arr = {1, 2, 3};
...
int[] anonym = {4, 5, 6}
arr = anonym;
```

у скороченому вигляді:

```
int[] arr = {1, 2, 3};
...
arr = new int[] {4, 5, 6};
```

```
int[] arr = {1, 2, 3};
obj.met(arr);
```

у скороченому вигляді:

```
obj.met(new int[] {1, 2, 3});
```

Багатовимірні масиви

Багатовимірний масив – це масив, який складається з множини масивів. У Java нема багатовимірних масивів в принципі, а багатовимірні масиви реалізуються як множина одновимірних. Кількість вимірів масиву задається парами закриваючих і відкриваючих прямокутних дужок. Як і одновимірнім масиви багатовимірні масиви перед використанням необхідно оголосити і ініціалізувати.

Синтаксиси оголошення неініціалізованого двовимірного масиву:

```
тип[][] змінна;
тип змінна[][];
```

Приклади оголошення неініціалізованого двовимірного масиву типу `int`:

```
int[][] arr;
int arr[][];
```

Синтаксиси оголошення та ініціалізації двовимірного масиву:

```
тип[][] змінна = new тип[розмір_виміру_1][розмір_виміру_2];
тип[][] змінна = {{значення11, значення12,..., значення1N},
                  {значення21, значення22,..., значення2N}
                   ...}
```

```
                    {значенняM1, значенняM2,..., значенняMN}};
тип змінна[][] = new тип[розмір_виміру_1][розмір_виміру_2];
тип змінна[][] = {{значення11, значення12,..., значення1N},
                   {значення21, значення22,..., значення2N}
                    ...
                   {значенняM1, значенняM2,..., значенняMN}};
```

Приклади оголошення та ініціалізації двовимірного масиву типу `int`:

```
int[][] arr = new int[2][5];
int[][] arr = {{1,2,3,4,5},{11,12,13,14,15}};
int arr[][] = new int[2][5];
int arr[][] = {{1,2,3,4,5},{11,12,13,14,15}};
```

Зубчаті масиви

Завдяки тому, що багатовимірні масиви у Java реалізуються як множина одновимірних масивів, стає можливим реалізувати багатовимірні масиви з різною кількістю елементів у межах виміру. Синтаксис оголошення зубчатого масиву нічим не відрізняється від синтаксису оголошення звичайного багатовимірного масиву. Різниця є лише у способі ініціалізації, де використовується виділення пам'яті під різну кількість елементів у межах виміру.

Синтаксис оголошення та ініціалізації зубчатого масиву:

```
тип[][] змінна = new тип[N][];
змінна[0] = new тип[розмір_виміру_20];
змінна[1] = new тип[розмір_виміру_21];
...
змінна[N-1] = new тип[розмір_виміру_2N-1];
```

Приклад оголошення та ініціалізації зубчатого масиву:

```
int[][] arr = new int[3][];
arr[0]= new int[3];
arr[1]= new int[1];
arr[2]= new int[2];
```

Особливості використання масивів

Розмір масиву зберігається у властивості `length`.

Копіювання масивів не можна здійснити звичайним присвоюванням однієї змінної-масиву іншій. У цьому випадку обидві змінні-масиви посилатимуться на одну і ту саму область пам'яті, тобто фізично на один і той самий масив. Для коректного копіювання масивів слід скористатися методом `copyOf` класу `Arrays`. Цей метод створює копію масиву, що переданий через перший параметр методу, у пам'яті та повертає посилання на

нього. Кількість елементів масиву, що підлягають копіюванню, передається через другий параметр методу. Приклад копіювання масиву:

```
int [] copiedArray = Arrays.copyOf(originalArray, originalArray.length);
```

Для перевірки масивів на ідентичність використовується метод `equals` класу `Arrays`. Цей метод приймає через параметри 2 масиви простих типів, та повертає `true`, якщо вони ідентичні, та `false`, якщо ні. Приклад порівняння масивів на ідентичність:

```
boolean compared = Arrays.equals(array1, array2);
```

Сортування масивів можна здійснювати за допомогою статичного методу `sort` класу `Arrays`. Даний метод приймає посилання на масив, який необхідно відсортувати. Відсортований масив повертається за цим же посиланням. Для застосування даного методу масив має містити елементи, які реалізують інтерфейс `Comparable`, що визначає єдиний метод `compareTo`, або прості типи. Приклад сортування масиву:

```
String students[] = {"Микола","Петро","Іван"};
Arrays.sort(students);
```

У Java 8 з'явилися множина перевантажених статичних методів класу `Arrays`, що дозволяють здійснювати сортування масивів паралельно – методи `parallelSort`.

Висновок

У лекції розглянуто основні прості типи даних мови Java, а також методи оголошення та використання змінних, констант та масивів різних видів.

3. Основні операції над даними

План

1. Операції.
2. Приведення типів.
3. Керуючі конструкції.
4. Деякі способи реалізації вводу/виводу інформації.

Операції

Операції мови Java за своєю суттю схожі на операції мови C/C++. Проте є деякі особливості:

- ділення цілого на 0 генерує виключну ситуацію, а ділення числа з рухомою комою на 0 генерує NaN;
- для забезпечення гарантії однакових результатів обчислення виразів з рухомою комою на різних комп'ютерах слід використовувати в оголошеннях методів, які здійснюють ці обчислення, ключове слово `strictfp`; використання `strictfp` призводить до сповільнення обчислень;
- тип `boolean` не можна привести до іншого типу; для приведення `boolean` до іншого типу можна використати конструкцію: `x = b ? 1 : 0`;

- >> - арифметичний зсув праворуч;
- >>> - логічний зсув праворуч;
- операція ” , ” відсутня (крім циклу for);
- операція sizeof() відсутня;
- математичні операції реалізуються з використанням статичних членів класу Math.

Таблиця. 3.1.

Операції мови Java

Пріоритет	Операції	Асоціативність
1	[] . ()(виклик функції)	Зліва направо
2	! ~ ++ -- +(унарний) -(унарний) ()(приведення типу) new	Справа наліво
3	* / %	Зліва направо
4	+ -	Зліва направо
5	<< >> >>>	Зліва направо
6	< <= > >= instanceof	Зліва направо
7	== !=	Зліва направо
8	&	Зліва направо
9	^	Зліва направо
10	|	Зліва направо
11	&&	Зліва направо
12	||	Зліва направо
13	?:	Справа наліво
14	= += -= *= /= %= |= ^= <<= >>= >>>=	Справа наліво

Приведення типів

При виконанні бінарних операцій над даними різних типів відбувається неявне приведення типів за правилом:

- якщо хоч один з операндів має тип double, то інший також приводиться до типу double;
- інакше, якщо хоч один з операндів має тип float, то інший також приводиться до типу float;
- інакше, якщо хоч один з операндів має тип long, то інший також приводиться до типу long;
- інакше, операнди приводяться до типу int.

Приведення типів з втратою точності відбувається лише явно. Для виконання явного приведення типів перед змінною, яку слід привести до іншого типу, необхідно в круглих дужках вказати до якого типу її необхідно привести. Приведення типів з формату з рухомою комою до цілого типу відбувається шляхом відкидання дробової частини числа з рухомою комою.

Керуючі конструкції

Блочний оператор. Блочний оператор – це довільна кількість простих операторів, які розташовані між фігурними дужками. Блоки можуть бути вкладеними один в одного. Блоки визначають області видимості змінних. У Java не можна оголошувати змінні з однаковими іменами у вкладених блоках.

Приклад блочного оператора:

```
public static void main(String[] args)
{
      // зовнішній блок
      int n;
      {
            //вкладений блок
            int k;
            int n;     // при компіляції згенерується помилка
      }
}
```

Умовний оператор. Умовний оператор `if` призначений для керування виконанням операторів мови Java при здійснені або не здійснені певної логічної умови. Можливі синтаксичні конструкції умовного оператора `if` у Java і C/C++ співпадають та мають наступний вигляд:

```
if (логічна умова)
   оператор_1
```

```
if (логічна умова)
   оператор_1
else
   оператор_2
```

Оператор циклу `while`. Оператор циклу `while` забезпечує виконання виразу, або блоку операторів поки умова є істинно. Синтаксис оператора циклу `while` у Java і C/C++ співпадають та мають наступний вигляд:

```
while (логічна умова)
      оператор
```

Оператор циклу `do-while`. Оператор циклу `do-while` забезпечує виконання виразу, або блоку операторів як мінімум один раз і продовжує виконувати їх поки умова є істинно. Це відбувається завдяки перевірці умови після виконання тіла циклу. Синтаксис оператора циклу `do-while` у Java і C/C++ співпадають та мають наступний вигляд:

```
do
      оператор
while (логічна умова)
```

Оператор циклу for. Оператор циклу `for` призначений для виконання наперед заданої кількості ітерацій. У Java цей оператор має 2 різновиди:

- конструкція в стилі C/C++ з полем ініціалізації, логічною умовою та кроком;
- конструкція з *синтаксисом foreach*.

Синтаксис оператора `for` в стилі C/C++ має такий вигляд:

```
for (ініціалізація лічильника; логічна умова; модифікація лічильника)
            оператори
```

Робота оператора циклу `for` в стилі C/C++ починається з виконання операторів поля ініціалізації лічильника, після чого відбувається перевірка логічної умови, виконання операторів тіла циклу та модифікація лічильника. Після першої ітерації, поки логічний вираз є істинним, циклічно послідовно виконуються лише операції перевірки умови, тіла циклу та модифікації лічильника. Область видимості змінних, що оголошені в полі ініціалізації лічильника та час їх життя обмежені тілом циклу for.

Приклад оператора оператора `for` в стилі C/C++:

```
for (int i = 0; i < 100; i++)
     System.out.println(i);
```

Починаючи з Java 5.0 мовою підтримується конструкція оператора циклу `for` з синтаксисом foreach, який дозволяє послідовно перебирати всі елементи набору даних без застосування лічильника. Таким набором даних може бути будь-який клас, що реалізує інтерфейс `Iterable`, або масив. Оператор циклу `for` з синтаксисом foreach має наступний вигляд:

```
for (змінна : набір даних)
            оператори
```

При опрацюванні циклу змінній послідовно присвоюється кожен елемент набору даних (наприклад, елемент масиву) після чого виконується оператор.

Приклад використання оператора `for` з синтаксисом foreach:

```
for (int elem: arr)
     System.out.println(elem);
```

Цей фрагмент коду є аналогічним наступному:

```
for (int i = 0; i < arr.length; i++)
     System.out.println(arr[i]);
```

Оператор кома. Оператор кома може застосовуватися лише у полях ініціалізації та модифікації лічильника у циклах `for` в стилі C/C++. У цих полях за допомогою оператора кома можна розділити кілька команд, які будуть виконуватися послідовно. Єдиною вимогою оператора кома є те, що всі змінні, які визначаються за його участю, мають бути однакового типу.

Приклад використання оператора кома:

```
for (int i = 0, j = 0; i < arr.length; i++, j += 2)
      System.out.println(arr[i] + j);
```

Оператори переривання потоку виконання. До операторів переривання потоку виконання відносяться оператори `break` і `continue`. Вони призначені для переривання послідовності виконання операцій в циклах. У циклах дані оператори можуть використовуватися з мітками і без них.

Оператор `break` без мітки перериває виконання циклу та передає керування на першу інструкцію, що стоїть після циклу. Крім цього `break` застосовується для переривання послідовності виконання операцій у операторі `switch` та передачі керування на наступну інструкцію після `switch`. Оператор `continue` без мітки перериває виконання поточної ітерації циклу та передає керування наступній ітерації циклу.

Оператори `break` і `continue` з міткою застосовуються у вкладених циклах для переривання роботи вкладеного і зовнішнього циклів. Мітка у цих операторах призначена для того, щоб визначити зовнішній цикл, на роботу якого впливатиме оператор `break` або `continue`. Для цього вона ставиться перед оператором зовнішнього циклу, на який має впливати `break` або `continue`. Міткою може бути будь-який ідентифікатор, що відповідає правилам оголошення ідентифікаторів (змінних) та закінчується двокрапкою.

Оператор `break` з міткою перериває виконання вкладеного циклу та передає керування на першу інструкцію, що стоїть після циклу, який маркований міткою. Оператор `continue` з міткою перериває виконання поточної ітерації циклу та передає керування наступній ітерації зовнішнього циклу, що маркований міткою.

Приклад використання операторів `break` і `continue`:

```
label1:
зовнішній цикл
{
      внутрішній цикл
      {
            …
            break;                  // (1)
            …
```

```
            continue;             // (2)
            …
            break label1;         // (3)
            …
            continue label1;      // (4)
      }
}
оператор1
```

При виконанні (1) відбудеться вихід з внутрішнього циклу. При виконанні (2) відбудеться перехід на наступну ітерацію внутрішнього циклу. При виконанні (3) відбудеться вихід з внутрішнього і зовнішнього циклів та перехід на мітку `label1` з передачею керування на `оператор1`. Тобто, наступним після виконання `break label1` виконається `оператор1`. При виконанні (4) відбудеться вихід з внутрішнього і зовнішнього циклів та перехід на мітку `label1` з передачею керування наступній ітерації зовнішнього циклу. Тобто, після виконання `continue label1` виконається наступна ітерація зовнішнього циклу.

Оператор багатоваріантного гілкування. Оператор багатоваріантного гілкування `switch` використовується в тих випадках, коли необхідно реалізувати вибір з багатьох можливих наперед відомих константних цілочисельних або рядкових варіантів вибору. Синтаксис оператора `switch` у Java відповідає синтаксису оператора `switch` у мовах C/C++:

```
switch (змінна)
{
      case константа_1:
            оператор_1
      break;
      case константа_2:
            оператор_2
      break;
      ...
      case константа_n:
            оператор_n
      break;
      default:
            оператор_def
}
```

Оператор `switch` аналізує `змінну`, яка може бути змінною типу `byte`, `short`, `int`, `char`, `String` або тип-перелічення. В залежності від її значення передає керування на відповідну мітку `case`. Міткою `case` може бути цілочисельна константа, константа перелічення або рядок. У випадку використання константи перелічення не треба застосовувати ім’я перелічення у мітках, оскільки воно виводиться із `змінної` у `switch`

автоматично. Якщо значення змінної не співпадає із жодним зі значень константи у case, то керування передається на мітку default. Якщо після case не поставити оператор break, то після виконання операторів, що належать даному case, виходу з оператора switch не відбудеться, а наступним виконається наступний оператор у switch, навіть якщо він відноситься до іншої мітки case або default.

Приклад використання оператора switch:

```
Size s = …;
switch (s)
{
      case SMALL:
      …
      break;
      …
      default:
}
```

Деякі способи реалізації вводу і виводу інформації

Ввід з консолі

Для введення інформації з консолі необхідно створити об'єкт класу Scanner і зв'язати його з стандартним потоком вводу System.in, наприклад:

```
Scanner in = new Scanner(System.in);
```

Зробивши це ми отримаємо доступ до методів класу Scanner, які призначені для введення даних простих типів і рядків:

- hasNext – перевіряє чи є доступні дані для введення;
- nextBoolean – вводить дані типу boolean;
- nextByte – вводить дані типу byte;
- nextShort – вводить дані типу short;
- nextInt – вводить дані типу int;
- nextLong – вводить дані типу long;
- nextFloat – вводить дані типу float;
- nextDouble – вводить дані типу double;
- nextLine – вводить весь рядок і представляє його як об'єкт класу String;
- next – вводить одне слово (токен) і представляє його як об'єкт класу String.

Вивід на консоль

Популярним механізмом виводу на консоль є використання методу print об'єкту out з пакету System, який виводить переданий через параметр текстовий рядок на екран:

```
System.out.print("Hello!!!");
```

Недоліком цього методу є неможливість здійснити форматований вивід на консоль. Для здійснення форматованого виводу на консоль використовується метод `printf` синтаксис якого повністю співпадає з синтаксисом функції `printf` у мові С за винятком кількох доданих функціональних можливостей по виводу дати, часу, логічних значень і хеш-кодів, наприклад:

```
System.out.printf("Hello %s!!! ", str);
```

Іншим способом виводу на консоль є використання методів класу `java.io.Console`, який серед інших також містить метод `printf`, що також наслідує синтаксис однойменної функції мови С. На відміну від методу `printf` класу `System.out`, методу `printf` класу `Console` коректніше виводить симлови Unicode на консоль. Для одержання об'єкту даного класу слід викликати статичний метод `console` класу `System`:

```
Console cons = System.console();
cons.printf("Hello %s!!! ", str);
```

Крім цього даний клас містить методи для введення паролів, які не відображаються на екрані під час введення (`readPassword`), методи одержання об'єктів класів `Reader` і `PrintWriter`, асоційованих з консоллю та інші.

Ввід з текстового файлу

Для введення інформації з файлу необхідно підключити пакет `java.io` та створити об'єкт класу `Scanner` з об'єкту `File`:

```
Scanner fin = new Scanner(File("MyFile.txt"));
```

При компіляції цього рядка коду може виникнути виключна ситуація неіснування файлу з якого має проходити ввід даних (`FileNotFoundExeption`), яка розглядається компілятором серйозніше, ніж, скажімо, ділення на нуль. Тому, компілятор не дасть нам просто так скомпілювати цей рядок коду. Для коректної компіляції цього рядку коду необхідно або вказати, що наш метод може генерувати це виключення, або реалізувати код для перехоплення і обробки виключення. Для того, щоб вказати, що наш метод може генерувати це виключення, у оголошенні методу, де відбувається створення об'єкту класу `Scanner` з файлу, необхідно додати наступний код:

```
throws FileNotFoundExeption
```

наприклад,

```
public static void main(String[] args) throws FileNotFoundException
```

Пошук файлу відбувається у директорії з якої була запущена на виконання програма. Після відкривання файлу інформацію з нього можна читати використовуючи методи класу `Scanner`.

Вивід у текстовий файл

Для виведення інформації у текстовому вигляді у файл треба підключити пакет `java.io` та створити об'єкт класу `PrintWriter` в конструкторі якого необхідно вказати назву файлу, що відкривається на запис, наприклад:

```
PrintWriter fout = new PrintWriter ("MyFile.txt");
```

Зробивши це ми отримаємо доступ до методів класу `PrintWriter`, які призначені для виведення даних простих типів і рядків:

`print` – виводить значення простих типів і рядків у текстовому вигляді;
`write` – призначений для виводу даних типу `char` і `String` у текстовий файл.

При створенні об'єкту класу `PrintWriter` з іменем файлу, який не може бути створений буде згенероване виключення `FileNotFoundExeption`. Для його обробки необхідно виконати такі самі дії, що і в розглянутому вище способі введення даних з файлу.

Висновок

У лекції розглянуто операції мови Java, схему неявного приведення простих типів, керуючі конструкції та їх особливості, а також оглянуто прості способи реалізації вводу/виводу інформації.

4. Класи у мові Java

План

1. Класи та об'єкти.
2. Методи.
3. Конструктори.
4. Поля.
5. Порядок роботи конструктора.
6. Знищення об'єктів за допомогою методу `finalize`.

Класи та об'єкти

Оскільки мова Java є повністю об'єктно-орієнтованою мовою програмування, то вона дозволяє писати програми лише з використанням об'єктно-орієнтованих парадигм програмування, що базуються на понятті класів. Між класами мов Java і C++ є багато спільного, проте і багато відмінного. Так класи у мовах Java і C++ складаються з конструкторів, методів та полів (властивостей). Проте, на відміну від C++, у мові Java:

- реалізація і оголошення класів є нероздільними, тобто реалізація класу може знаходитися лише між `{}`, які розташовані після назви класу;
- використовується однокоренева архітектура класів згідно якої всі класи походять від класу `Object`;
- відсутні шаблони, деструктор та оператор `delete`;
- створення нових об'єктів відбувається у *кучі* за допомогою оператора `new`;
- об'єкт класу знищується за допомогою *збирача сміття*;
- кожен файл програми з розширенням *.java може мати один єдиний *загальнодоступний клас* назва якого співпадає з назвою файлу в якому він міститься.

Синтаксис оголошення класу в мові Java має наступний вигляд:

```
[public] [abstract] [final] class НазваКласу [extends Клас] [implements Інтерфейси]
{
	[конструктори]
	[методи]
	[поля]
}
```

Приклад оголошення загальнодоступного класу:

```
public class StartClass
{
	public StartClass()
	{
		str = "Hello";
	}

	public StartClass(String initString)
	{
		str = initString;
	}

	public void showMessage()
	{
		System.out.print(str);
	}

	private String str;
}
```

Необов'язковий специфікатор доступу `public` робить клас загальнодоступним. У кожному файлі з кодом програми може бути лише один загальнодоступний клас, ім'я якого співпадає з назвою файлу, та безліч класів без специфікатора `public`.

Ключове слово `abstract` вказує, що клас є абстрактним базовим класом.

Використання ключового слова `final` робить метод і клас - *герметизованим* (метод – недоступним для перевизначення у похідному класі при спадкуванні, а клас – недоступним для використання у ролі базового), а поле – незмінним після ініціалізації при створенні об'єкту. Це ключове слово слід застосовувати при оголошені полів простих типів або *незмінних класів* (класів, що не мають методів, які можуть змінити стан їх об'єктів (значення, яке має об'єкт), наприклад, клас `String`). В протилежному випадку використання `final` може стати джерелом помилок, оскільки лише посилання на об'єкт буде константним, а значення самого об'єкту можна буде змінити.

Після ключового слова `extends` вказується єдиний базовий клас, що успадковується.

Після ключового слова `implements` вказуються інтерфейси, які реалізує даний клас.

Створення об'єкту класу, як і масиву, складається з двох етапів: оголошення та ініціалізації посилання на об'єкт. Оголошення посилання на об'єкт класу має синтаксис:

```
НазваКласу назваПосилання;
```

Приклад оголошення посилання на об'єкт класу `StartClass`:

```
StartClass obj;
```

Ініціалізація посилання на об'єкт класу здійснюється за допомогою оператора `new` і вказування конструктора, який має збудувати об'єкт. Одержаний в результаті цих операцій об'єкт розташується у області оперативної пам'яті що зветься *"куча"*. Ініціалізація посилання на об'єкт класу за допомогою конструктора за замовчуванням має такий синтаксис:

```
назваПосилання = new НазваКонструтора();
```

Приклад ініціалізації посилання на об'єкт класу `StartClass`:

```
obj = new StartClass();
```

При створенні об'єктів дозволяється суміщати оголошення та ініціалізацію об'єктів, а також створювати анонімні об'єкти. Якщо посилання на об'єкт не посилається на жоден об'єкт, то йому слід присвоїти значення `null`. На відміну від полів-посилань на об'єкти, локальні змінні-посилання на об'єкти не ініціалізуються значенням `null` при оголошенні. Для них ініціалізацію посилання слід проводити явно.

Методи

Метод – функція-член класу, яка призначена маніпулювати станом об'єкту класу. Оголошення методу може складається з: специфікатора доступу; ключового слова

static; ключового слова final; типу значення, що повертається; назви методу; параметрів; переліку виключень, що можуть бути згенеровані методом, але не опрацьовані; тіла методу. Допускається перевантаження методів, в тому числі і конструкторів. *Перевантаження методу* відбувається тоді, коли клас містить кілька методів, що мають однакові імена та відрізняються типами і кількістю параметрів. Тип значення, що повертається методом, не впливає на перевантаження. Тому, методи не можуть бути перевантаженими, якщо вони відрізняються лише значенням, що повертається. Синтаксис оголошення методу наступний:

```
[СпецифікаторДоступу] [static] [final] Тип назваМетоду([параметри]) [throws класи]
{
	[Тіло методу]
	[return [значення]];
}
```

Конструктори, методи, та поля класу можуть бути відкритими (public), закритими (private) та захищеними (protected), що визначається специфікатором доступу.

Специфікатор доступу public робить елемент класу загальнодоступним в межах пакету (набору класів, з яких складається програма).

Специфікатор доступу private робить елемент класу закритим (недоступним) для всіх зовнішніх відносно даного класу елементів програми (включаючи похідні класи).

Специфікатор доступу protected робить елемент класу закритим (недоступним) для всіх зовнішніх відносно даного класу елементів програми, проте цей елемент буде загальнодоступним для похідних класів (утворених з даного через спадкування).

Якщо будь-який елемент класу не має специфікатора доступу, то цей елемент автоматично стає відкритим та видимим у межах пакету (не плутати з public).

Всі елементи класу, що оголошені без використання ключового слова static, належать об’єкту класу. Тобто, кожен об’єкт класу містить власну копію цих елементів класу. Ключове слово static робить поле або метод членом класу, а не об’єкту, тобто вони є спільними для всіх об’єктів класу. Оскільки клас існує завжди, на відміну від об’єктів, які створюються в процесі роботи програми, то статичні елементи класу доступні навіть тоді, коли ще не створено жодного об’єкту класу. Цей підхід використовується при написанні методу main з якого починається виконання консольної програми, бо на момент її запуску ще не існує жодного об’єкту.

Різниця між статичними і не статичними методами полягає в тому, що крім явних параметрів, нестатичний метод приймає додатково неявний параметр – посилання this на об’єкт, що викликав метод, а статичні методи цього не роблять. Таким чином статичні методи не можуть доступатися до нестатичних елементів класу, проте вони мають доступ

до статичних полів класу. Для більшої наочності до нестатичних членів класу в середині методів можна звертатися за допомогою посилання `this`, наприклад:

```
this.step = 5;
```

Статичні методи слід застосовувати у двох випадках:

- коли методу не потрібен доступ до інформації про стан об'єкту, оскільки всі дані задаються через параметри (наприклад метод `Math.pow`);
- коли метод доступається лише до статичних полів класу.

Метод може генерувати виключення. Якщо виключення не перехоплюється у тілі методу, то воно повинно бути описаним в оголошені методу після ключового слова `throws`. Якщо виключення перехоплюється у тілі методу, то цього робити не потрібно.

Передача параметрів у метод відбувається по значенню шляхом копіювання значень реальних параметрів у формальні параметри методу. Якщо ці значення є простими типами, то відбудеться копіювання значень. Якщо ці значення є посиланнями, то копіюватимуться не об'єкти, а посилання на об'єкти. Таким чином зміна значення посилання формального параметру в середині методу не вплине на значення посилання за його межами.

Вихід та повернення значення з методу відбувається за допомогою оператора `return`. Якщо метод не повертає значення, то оператор `return` можна опустити. Перед поверненням з методу значення об'єкту, що може змінювати свій стан, слід обов'язково скористатися методом `clone` об'єкту, який створює його копію.

Синтаксис виклику нестатичного методу:

```
НазваОб'єкту.назваМетоду([параметри]);
```

Синтаксис виклику статичного методу має 2 види:

```
НазваОб'єкту.назваМетоду([параметри]);     // через об'єкт класу
НазваКласу.назваМетоду([параметри]);       // через назву класу
```

Конструктори

Конструктор – спеціальний метод класу, який не повертає значення, носить ім'я класу та призначений для початкової ініціалізації об'єктів класу. Синтаксис оголошення конструктора:

```
[СпецифікаторДоступу] НазваКласу([параметри])
{
     Тіло конструктора
}
```

Конструкторів може бути кілька. Конструктор без параметрів називається *конструктором за замовчуванням*. Якщо у класі не визначено жодного конструктора, то конструктор за замовчуванням генерується автоматично при компіляції (неявно). Він здійснює ініціалізацію полів об'єкту класу *значеннями за замовчуванням* (оскільки поля об'єкту на момент створення обов'язково мають бути ініціалізованими). Ці значення рівні:

- для полів чисел простих типів – `0`;
- для логічних полів – `false`;
- для посилань на об'єкти – `null`.

Якщо ж клас має хоч один конструктор з параметрами, то конструктор за замовчуванням має бути визначений явно.

У мові Java допускається виклик одного конструктора з іншого конструктора. Для цього у тілі першого конструктора першим оператором має бути виклик іншого конструктора:

```
this(сигнатура необхідного конструктора);
```

Наприклад:

```
public class StartClass
{
      public StartClass(String initString)
      {
            str = initString;
      }
      public StartClass(int val)
      {
            this("Hello User " + val);
      }
      public void showMessage()
      {
            System.out.print(str);
      }

      private String str;
}
```

Поля

Поле (властивість) – це дані-члени класу, що призначені для зберігання стану об'єкту. Поле може бути статичним (в цьому випадку воно називається *полем класу*), незмінним (*константне поле*), простим типом чи об'єктом та мати різні рівні доступу, що визначаються специфікатором доступу. Допускається ініціалізація поля в місці оголошення. Синтаксис оголошення поля наступний:

```
[СпецифікаторДоступу] [static] [final] Тип НазваПоля [= ПочатковеЗначення];
```

Приклад оголошення поля:

```
private int i;
```

Приклад оголошення константного поля:

```
private final int i;
```

Ініціалізацію полів при створенні об’єкту можна здійснювати трьома способами:

- у конструкторі;
- явно при оголошені поля;
- у блоці ініціалізації (виконується перед виконанням конструктора).

Якщо поле не ініціалізується жодним з цих способів, то йому присвоюється значення за замовчуванням.

Приклад ініціалізації поля `str` у конструкторі:

```
public class StartClass
{
      public StartClass()
      {
            str = "Hello";
      }
      ...
      private String str;
}
```

Приклад явної ініціалізації поля `str` константою при оголошенні:

```
public class StartClass
{
      ...
      private String str = "Hello";
}
```

Приклад явної ініціалізації поля `str` статичним методом при оголошенні:

```
public class StartClass
{
      ...
      static String assignUser()
      {
            String tmpStr = "Hello user " + nextId++;
            return tmpStr;
      }
      ...
      private String str = assignUser ();
      private static int nextId;
```

```
}
```

Приклад ініціалізації поля `str` у блоці ініціалізації:

```
public class StartClass
{
	...
	private String str;



	// блок ініціалізації
	{
		str = "Hello";
	}
}
```

Ініціалізацію полів класу (статичних полів) можна здійснювати двома способами:

- явно при оголошені поля класу;
- у статичному блоці ініціалізації.

Статичний блок ініціалізації виконується коли клас завантажується вперше. Якщо поле класу не ініціалізується жодним з цих способів, то йому присвоюється значення за замовчуванням.

Приклад явної ініціалізації поля класу `str` константою при оголошенні:

```
public class StartClass
{
	...
	private static String str = "Hello";
	...
}
```

Приклад ініціалізації поля класу `str` у статичному блоці ініціалізації:

```
public class StartClass
{
	...
	private String str;

	// статичний блок ініціалізації
	static
	{
		str = "Hello";
	}
}
```

Порядок роботи конструктора

Дії, що виконуються при створенні об'єкту та виклику конструктора:

1. Всі поля ініціалізуються значеннями за замовчуванням (`0`, `false`, `null`).
2. Ініціалізатори усіх полів та блоки ініціалізації виконуються в порядку слідування в оголошенні класу.
3. Якщо в конструкторі викликається інший конструктор, то виконується він.
4. Виконується тіло конструктора.

Знищення об'єктів за допомогою методу `finalize`

У мові Java відсутній деструктор, оскільки, на відміну від C++, видалення об'єктів покладене на збирач сміття і не може бути виконане явно. Більше того, нема ніякої гарантії, що збирач сміття відпрацює негайно після того, як об'єкт необхідно знищити. Такий підхід усуває помилки пов'язані з необхідністю програмісту контролювати час життя об'єкту, проте може призвести до проблем пов'язаних з використанням спільних ресурсів (наприклад, файлів, ресурсів ОС). Для усунення цього недоліку в Java було введено можливість включення методу `finalize()` в будь-який клас. Цей метод викликається перед запуском збирача сміття. Проте, якщо необхідно звільнити ресурси і негайно використати їх знову, використання цього методу є невиправданим через непередбачуваність часу початку роботи збирача сміття. Для того, щоб звільнити ресурс зразу ж після його використання необхідно створити метод, який це робитиме і викликати його самостійно в потрібних ситуаціях не покладаючись на метод `finalize()`.

Для примусового запуску збирача сміття використовується статичний метод `gc()` класу `System`, або цей же метод з об'єкту класу `Runtime`. Методи працюють однаково, оскільки метод `gc()` класу `System` у свому тілі викликає метод `gc()` з об'єкту класу `Runtime`. Приклад виклику збирача сміття:

```
System.gc();
Runtime.getRuntime().gc();
```

Клас `Runtime` є надзвичайно потужним системним класом мови Java, який дозволяє керувати процесом виконання програми, завантажувати динамічні бібліотеки, зупиняти/запускати віртуальну машину та програми, перевіряти наявні апаратні ресурси, примусово запускати на виконання методи `finalize()`, які очікують на виконання, тощо.

Висновок

У лекції розглянуто реалізацію класів, методів і полів, створення об'єктів, запуск на виконання методів, особливі методи класу – конструктори та порядок їх роботи, принципи роботи з полями, знищення об'єктів за допомогою методу `finalize` та основні можливості класу `Runtime`.

5. Внутрішні класи, пакети, лямбда вирази

План

1. Статичні внутрішні класи.
2. Звичайні внутрішні класи.
3. Локальні класи.
4. Анонімні класи.
5. Лямбда вирази.
6. Потокова обробка даних.
7. Створення пакетів.
8. Використання пакетів.
9. Статичний імпорт пакетів.
10. Області видимості.

Статичні внутрішні класи

Статичні внутрішні класи визначаються в середині основного класу з використанням ключового слова `static`. Вони мають доступ лише до статичних членів класу. На відміну від інших видів внутрішніх класів вони не мають посилання на об'єкт зовнішнього класу та можуть містити статичні поля, методи і класи.

Приклад внутрішнього статичного класу:

```
class OuterClass
{
   public OuterClass(){}
   static int staticOuterField;
   static class InnerClass
   {
      int getStaticOuterField()
      {
         return OuterClass.staticOuterField;
      }
   }
}
```

Звичайні внутрішні класи

Звичайні внутрішні класи визначаються в середині основного класу. На відміну від статичних внутрішніх класів вони мають доступ до членів зовнішнього класу, проте не можуть містити визначення (але можуть успадковувати) статичних полів, методів і класів (за винятком констант). Звертання до членів зовнішнього класу відбувається за допомогою посилання на зовнішній клас, або простим звертанням до члену зовнішнього класу за допомогою його імені (як до звичайного члену класу). Посилання на зовнішній клас з внутрішнього класу має наступний вигляд:

```
НазваЗовнішньогоКласу.this
```

Якщо звичайний внутрішній клас є загальнодоступним, то на нього можна посилатися в будь-якому місці програми за допомогою посилання, що записане у вигляді:

```
НазваЗовнішньогоКласу.НазваВнутрішньогоКласу
```

Синтаксично створення об’єктів звичайних внутрішніх класів у членах зовнішнього класу нічим не відрізняється від створення об’єктів будь-яких інших класів. Проте при створенні об’єкту звичайного внутрішнього класу за межами об’єкту зовнішнього класу слід явно здійснювати виклик конструктора внутрішнього класу з-під об’єкту зовнішнього класу:

```
НазваОб’єктуЗовнішньогоКласу.new НазваВнутрішньогоКласу(параметри);
```

Приклад внутрішнього класу:

```
class OuterClass
{
   public OuterClass()
   {
      innerObj = new InnerClass();
   }

   private int outerField;
   private InnerClass innerObj;

   class InnerClass
   {
       int getOuterField()
       {
         return OuterClass.this.outerField;
       }
   }
}
```

Зауважимо, що внутрішніх класів у скомпільованій Java програмі не існує. Всі внутрішні класи, що оголошені в коді програми, під час компіляції перетворюються компілятором у звичайні класи. Проте на відміну від звичайних класів, оброблені компілятором внутрішні класи мають доступ до всіх полів зовнішніх класів (до приватних у тому числі). Але ця перевага несе в собі небезпеку, оскільки знаючи структуру скомпільованої програми на мові Java можна в class-файлі за допомогою внесення команд віртуальної машини для виклику додатково згенерованих компілятором методів одержати доступ до приватних полів зовнішніх класів. Або можна створити атакуючий клас, додати його в той самий пакет, в якому знаходиться атакований клас, та одержати доступ до приватних полів класу, що атакується.

Локальні класи

Локальні класи визначаються в середині методів основного класу та можуть бути застосовані лише в середині цих методів. Вони не можуть містити визначення (але можуть успадковувати) статичних полів, методів і класів (за винятком констант).

Локальні класи мають доступ до:

- членів зовнішнього класу;
- локальних змінних та параметрів методу за умови, що ці змінні і параметри визначені з використанням ключового слова `final`.

Якщо у локальному класі, необхідно змінювати значення локальних змінних або переданих у метод параметрів, то їх слід оголошувати як `final` масиви. В такому випадку лише посилання на масив є константним, а значення, що містяться у масиві, є змінними.

Приклад локального класу:

```
class OuterClass
{
   public OuterClass(){}

   private int outerField;
   InnerClass inner;                    // Помилка компіляції

   void methodWithLocalClass (final int parameter)
   {
      InnerClass innerInsideMehod;    // OK
      int notFinal = 0;
      class InnerClass
      {
         int getOuterField()
         {
            return OuterClass.this.outerField; // OK
         }

         notFinal++;                    // Помилка компіляції

         int getParameter()
         {
             return parameter;          // OK
         }
      }
   }
}
```

Анонімні класи

Анонімні класи визначаються в середині методів основного класу або безпосередньо в основному класі та можуть бути використані лише в середині цих методів або класів. На відміну від локальних класів, анонімні класи не мають назви. Головна вимога до анонімного класу – він повинен успадковувати існуючий клас або реалізовувати існуючий інтерфейс (інакше об'єкт такого класу не можна було б створити). Як звичайні внутрішні та локальні класи вони не можуть містити визначення (але можуть успадковувати) статичних полів, методів і класів (за винятком констант).

Оголошення анонімного класу синтаксично схоже на виклик конструктора. Різниця між ними полягає у тому, що після вказування назви конструктора ставиться не крапка з комою, а відкриваюча фігурна дужка, після якої вказується тіло анонімного класу та закриваюча фігурна дужка. В результаті такого запису створюється об'єкт анонімного класу, що реалізує вказаний у назві «конструктора» інтерфейс або розширює вказаний у назві «конструктора» суперклас.

Анонімні класи корисні тоді, коли в програмі необхідно створити лише один об'єкт певного класу і при реалізації механізму обробки подій та зворотних викликів (callback).

Приклад анонімного класу:

```
class OuterClass
{
   public OuterClass() {}

   void methodWithAnonymousClass (final int interval)
   {
       // ActionListener - інтерфейс який ми використаємо для
       // створення анонімного класу
      ActionListener listener = new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
        System.out.println("Цей рядок виводиться на екран що " +
        interval + " секунд");
         }
      }
     // listener  - єдина на всю програму реалізація об'єкту
     // анонімного класу, який використано при створені
     // об'єкту класу Timer для опрацювання події проходження
     // вказаного часового інтервалу
     Timer tmr = new Timer(interval, listener);
      tmr.start();
   }
}
```

Об’єкт класу `Timer` при створенні приймає 2 параметри – часовий інтервал і об’єкт, що розширює інтерфейс `ActionListener`. У цьому інтерфейсі визначається єдиний метод `actionPerformed`, який буде автоматично викликатися з-під переданого другим параметром об’єкту, коли пройде заданий змінною `interval` часовий інтервал. Оскільки при створенні об’єкту `tmr` нам в усій програмі потрібен лише один об’єкт, що реалізовує вказаним чином інтерфейс `ActionListener`, то для створення даного об’єкту нам якнайкраще підходить анонімний клас. Завдяки використанню анонімного класу відбувається зменшення необхідного для написання коду, оскільки не треба окремо визначати клас, та покращується читабельність коду.

Наприклад, без застосування анонімного класу даний приклад мав би таку реалізацію:

```
class TmrHandler implements ActionListener
{
      public TmrHandler (int tInt)
      {
            interval = tInt;
      }
      public void actionPerformed(ActionEvent event)
      {
         System.out.println("Цей рядок виводиться на екран що " +
         interval + " мілісекунд");
      }

      private int interval;
}

class OuterClass
{
   public OuterClass() {}
   void methodWithAnonymousClass (final int interval)
   {   TmrHandler listener = new TmrHandler (interval);
       Timer tmr = new Timer(interval, listener);
      tmr.start();
   }
}
```

Лямбда вирази

У Java 8 на зміну анонімним класам приходять анонімні методи – *лямбда вирази*. Лямбда вирази покликані спростити реалізацію так званих *функціональних інтерфейсів* – інтерфейсів, що містять оголошення лише одного абстрактного методу (у Java 8 не всі методи в інтерфейсі можуть бути абстрактними) та однометодних абстрактних класів. У Java вони є надзвичайно поширеними. До них відносяться: `Comparable`, `ActionListener`, `Runnable`, `Closeable` класи типу `EventListener` тощо. Тепер

замість того, щоб реалізовувати повноцінні класи, або дані інтерфейси достатньо використати лямбда вираз. Синтаксис лямбда виразу має кілька видів:

```
([[типАргументу] аргумент{, [типАргументу] аргумент}]) ->
(“{“тілоМетоду”}” | оператор)

[типАргументу] аргумент -> (“{“тілоМетоду”}” | оператор)
```

Приклади лямбда виразів:

```
(int x, int y) -> x + y      // повертається сума x+y
(x, y) -> x + y              // повертається сума x+y
() -> 35                     // повертається 35
(String str) -> { System.out.println(str); } // нічого не повертається
str -> System.out.println(str)                // нічого не повертається
```

Зверніть увагу! Якщо лямбда вираз використовується без фігурних дужок, то крапка з комою в кінці не ставиться.

Проілюструємо переваги лямбда виразів на прикладі розглянутого у темі присвяченій анонімним класам методу `methodWithAnonymousClass`:

```
void methodWithAnonymousClass (final int interval)
{
       ActionListener listener = () -> {System.out.println("Цей
рядок виводиться на екран що " + interval + " секунд");}
       Timer tmr = new Timer(interval, listener);
      tmr.start();
}
```

Як видно з синтаксису лямбда виразів якщо параметрів немає, або якщо їх більше ніж один, то необхідно параметри вказувати у дужках. Якщо параметр один, то дужки використовувати не обов’язково. Це саме стосується фігурних дужок для тіла методу.

У лямбда виразах дозволяється застосовувати мітки, але лише в межах лямбда виразу. Перехід за межі лямбда виразу заборонений. Генерування виключних ситуацій в середині лямбда виразу – заборонено. Лямбда вирази не можуть змінювати локальні змінні. Лямбда вираз не можна перервати за допомогою `return`. Якщо у лямбда виразі використовується гілкування, то результат має повертатися за допомогою оператора `return` у кожній з гілок. Якщо код у лямбда виразі може генерувати виключення, то його слід перехоплювати у лямбда виразі, або використовувати інтерфейс з відповідним методом.

Java підтримує так званий “Target typing”, який дозволяє динамічно визначати тип даних, що приймає участь в операціях, для:

- оголошень змінних;
- присвоєнь;

- оператора `return`;
- ініціалізаторів масивів;
- аргументів методів чи конструкторів;
- тіл лямбда виразів;
- тернарного оператора `?:` ;
- приведень значень.

Завдяки “Target typing” з’являється змога не вказувати типи даних для аргументів, які передаються у лямбда вираз:

```
str -> System.out.println(str)
```

В даному лямбда виразі тип для `str` буде виведено автоматично.

Потокова обробка даних

У Java 8 додано підтримку потокової обробки даних (streaming API). Вона дозволяє зменшити кількість коду необхідного для обробки великих масивів даних, а також дозволяє полегшити розпаралелення цієї обробки та виконувати дану обробку використовуючи паттерн “Lazy”, тобто не у місці оголошення потоку, а у місці виклику схеми обробки даних. Недоліком є те що оголошений потік можна використати леше 1 раз для обробки певного набору даних. При потребі повторити обчислення потік треба створювати наново.

Потік Java не зберігає дані. Він працює на структурі вихідних даних (колекції і масиви) і виробляє конвеєрні дані, які ми можемо використовувати і над якими ми можемо виконувати конкретні операції. Наприклад, ми можемо створити потік зі списку (List) і фільтрувати його за умовою.

Операції Java Stream використовують функціональні інтерфейси, що робить їх придатним для функціонального програмування за допомогою лямбда-виразів.

Принцип внутрішньої ітерації Java 8 Stream допомагає реалізувати пошук даних згідно паттерну “Lazy” у деяких потокових операціях. Наприклад, фільтрація, відображення (mapping) або видалення дублікатів можуть бути реалізовані як Lazy, дозволяючи більш високу продуктивність і можливості для оптимізації.

Потоки Java є споживачами, тому немає можливості створити посилання на потік для майбутнього використання. Оскільки дані потоку надаються на вимогу, то повторне використання одного і того ж потоку неможливе.

Java 8 Stream підтримує послідовну та паралельну обробку, паралельна обробка може бути дуже корисною для досягнення високої продуктивності для великих колекцій даних.

Всі інтерфейси та класи Java Stream API знаходяться в пакеті java.util.stream. Оскільки ми можемо використовувати примітивні типи даних, такі як int, long в колекціях за допомогою auto-boxing, і ці операції можуть зайняти багато часу, існують специфічні класи для примітивних типів - `IntStream`, `LongStream` і `DoubleStream`.

Для полегшення написання програм з використанням Java Stream у Java 8 з'явився пакет `java.util.function`, який містить множину корисних інтерфейсів для вирішення задач, що часто виникають при потоковій обробці даних. Це інтерфейси:

- `Predicate` – містить метод, що повертає значення типу Boolean для переданого аргументу;
- `Consumer` – містить метод, що виконує дію над об'єктом, що переданий через аргумент;
- `Function` – містить метод, що перетворює об'єкт типу `T` в об'єкти типу `U`;
- `Supplier` – містить метод, що повертає об'єкт класу `T` (по суті є фабрикою);
- `UnaryOperator` – містить метод, що виконує унарну операцію над типом `T`, повертає `T`;
- `BinaryOperator` – містить метод, що виконує бінарну операцію над об'єктами типу (`T`, `T`) повертає об'єкт типу `T`.

`java.util.Optional` - контейнерний об'єкт, який може містити або не містити ненульове значення. Якщо значення присутнє, `isPresent()` поверне true і `get()` поверне значення. Багато термінальних методів повертають об'єкт класу `Optional`. Деякі з цих методів:

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
- `Optional<T> min(Comparator<? super T> comparator)`
- `Optional<T> max(Comparator<? super T> comparator)`
- `Optional<T> findFirst()`
- `Optional<T> findAny()`

Для підтримки паралельного виконання в Java 8 Stream API використовується інтерфейс `java.util.Spliterator`. Метод `trySplit` повертає новий Spliterator, який керує підмножиною елементів оригінального сплітератора.

Операції Java Stream API, що повертають новий потік, називаються проміжними операціями. У більшості випадків ці операції носять Lazy характер, тому вони починають виробляти нові елементи потоку та відправляють його до наступної операції. Проміжні операції ніколи не є кінцевим результатом виконання потоку операцій. Зазвичай використовуються проміжні операції - filter і map.

Операції Java Stream API, які повертають результат називаються кінцевими. Після виклику кінцевого методу в потоці він споживає потік і після цього ми не можемо використовувати потік. Кінцеві операції обробляють всі елементи потоку перед поверненням результату. Найчастіше використовуються кінцеві методи `forEach`, `toArray`, `min`, `max`, `findFirst`, `anyMatch`, `allMatch` і т.д. Термінальні методи можна ідентифікувати з типу результату, що повертається. Вони ніколи не повертають Stream.

Частина проміжних операцій є операціями короткого замикання. Проміжну операцію називають коротким замиканням, якщо вона може виробляти скінченний потік для нескінченного потоку. Наприклад, `limit()` і `skip()` є двома проміжними операціями короткого замикання.

Кінцеві операції є операціями короткого замикання, якщо вони можуть завершуватися в скінченний час для нескінченного потоку. Наприклад, `anyMatch, allMatch, noneMatch, findFirst` і `findAny` є кінцевими операціями короткого замикання.

Створення потоків.

Існує кілька способів, за допомогою яких можна створити потік java з масиву та колекцій.

1. Для створення потоку з набору однотипних даних ми можемо використати метод `Stream.of()`. Наприклад, ми можемо створити Java Stream цілих чисел з групи об'єктів `int` або `Integer`:

```
Stream<Integer>  stream = Stream.of(1, 2, 3, 4);
```

2. Ми можемо використовувати `Stream.of()` з масивом об'єктів (успадкованих від класу `Object`) для повернення потоку. Зауважте, що він не підтримує автоматичне пакування, тому ми не можемо передавати матрицю типу примітив.

```
Stream<Integer> stream = Stream.of (new Integer[] {1,2,3,4}); // працює нормально

Stream<Integer> stream = Stream.of (new int[] {1,2,3,4}); // Помилка часу компіляції, невідповідність типу: не можна перетворити з потоку <int []> на потік <Integer []>
```

3. Ми можемо використовувати метод `stream()`, який реалізований у колекціях для створення послідовного потоку і `parallelStream()` для створення паралельного потоку.

```
List<Integer> myList = new ArrayList<>();
for(int i = 0; i < 100; i ++) myList.add(i);

// послідовний потік
Stream<Integer> sequentialStream = myList.stream();

// паралельний потік
Stream <Integer> parallelStream = myList.parallelStream();
```

4. Ми можемо використовувати `Stream.generate()` і `Stream.iterate()` методи для створення потоку.

```
Stream<String> stream1 = Stream.generate(() -> {return "abc";});
Stream<String> stream2 = Stream.iterate("abc", (i) -> i);
```

5. Ми можемо використовувати методи `Arrays.stream()` та `String.chars()`.

```
LongStream = Arrays.stream(new long [] {1,2,3,4});
IntStream is2 = "abc".chars();
```

Перетворення потоків Java в колекції або масиви.

Є кілька способів, за допомогою яких ми можемо отримати колекцію або масив з потоку Java.

Ми можемо використовувати метод потоку Java під назвою `collect()` для отримання `List`, `Map` або `Set` з потоку.

```
Stream<Integer> intStream = Stream.of(1,2,3,4);
List<Integer> intList = intStream.collect(Collectors.toList());
System.out.println(intList); // виводить [1, 2, 3, 4]

intStream = Stream.of(1,2,3,4); // Потік закритий, тому нам потрібно його знову створити
Map<Integer, Integer> intMap = intStream.collect(Collectors.toMap (i -> i, i -> i + 10));
System.out.println(intMap); // виводить {1 = 11, 2 = 12, 3 = 13, 4 = 14}
```

Клас `Collectors` містить багато методів які дозволяють реалізовувати різну додаткову обробку даних, наприклад, `averagingInt` – вираховує середнє значення операцій переданих через аргумент, `summingInt` – вираховує суму операцій переданих через аргумент, `groupingBy` – здійснює групування елементів відповідно до переданої функції, `collectingAndThen` – дозволяє здійснювати додаткову обробку значень.

Для створення масиву з потоку ми можемо використовувати метод потоку Java під назвою `toArray()`.

```
Stream<Integer> intStream = Stream.of(1,2,3);
Integer[] intArray = intStream.toArray(Integer[]::new);
```

```
System.out.println(Arrays.toString(intArray)); // виводить [1, 2, 3]
```

Проміжні операції Java Stream

1. Фільтрація потоків здійснюється методом `filter()`, який перевіряє кожен елемент елемент потоку на відповідність умові та залишає лише ті елементи у потоці, які задовільняють умові.

```
List<Integer> myList = new ArrayList <> ();
for(int i = 0; i <100; i ++) myList.add(i);
Stream <Integer> sequentialStream = myList.stream();

Stream <Integer> highNums = sequentialStream.filter (p -> p> 90); // залишає в потоці елементи, що більші за 90
System.out.print("Числа більше 90 =");
highNums.forEach(p -> System.out.print (p + "")); // виводить "числа більше 90 = 91 92 93 94 95 96 97 98 99"
```

2. Відображення (map) застосовується щоб виконувати над кожним елементом потоку функції, наприклад приведення списку рядків до верхнього регістру:

```
Stream<String>  names = Stream.of("aBc", "d", "ef");
System.out.println (names.map(s -> {
                    return s.toUpperCase ();
                    }).collect(Collectors.toList ()));
// виводить [ABC, D, EF]
```

3. Сортування потоків здійснюється за допомогою методу `sorted()`, який приймає параметром об’єкт класу, що реалізує інтерфейс `Comparator`.

```
Stream<String>  names2  =  Stream.of("aBc",  "d",  "ef", "123456");
List<String>  reverseSorted  =  names2.sorted(Comparator. reverseOrder()).Collect (Collectors.toList());
System.out.println(reverseSorted); // виводить [ef, d, aBc, 123456]

Stream<String>  names3  =  Stream.of("aBc",  "d",  "ef", "123456");
List<String>  naturalSorted  =  names3.sorted().Collect (Collectors.toList());
System.out.println (naturalSorted); // виводить [123456, aBc, d, ef]
```

4. Метод flatMap() використовується для створення однорангових потоків з потоків списків.

```
Stream<List<String>> namesOriginalList = Stream.of(
     Arrays.asList ("Pankaj"),
     Arrays.asList ("David", "Lisa"),
     Arrays.asList ("Amit"));

// одноранговий потік зі списку <String> до потокового рядка
Stream<String>  flatStream  =  namesOriginalList.flatMap
(strList -> strList.stream ());

flatStream.forEach(System.out::println);
```

Кінцеві операції Java Stream

1. Скорочення елементів потоку здійснюється за допомогою методу `reduction()` використовуючи функцію асоціативного накопичення. Метод повертає об'єкт класу `Optional`. Приклад множення чисел потоку між собою.

```
Stream<Integer>  numbers = Stream.of(1,2,3,4,5);

Optional<Integer> intOptional = numbers.reduce((i, j) ->
{return i * j;});
if       (intOptional.isPresent())       System.out.println
("Multiplication =" + intOptional.get ()); // 120
```

2. Підрахунок кількості елементів потоку здійснюється кінцевою функцією `count()`:

```
Stream<Integer> numbers = Stream.of (1,2,3,4,5);

System.out.println("Кількість елементів у потоці =" +
numbers.count()); // 5
```

3. Перебір елементів потоку можна робити за допомогою функції `forEach()`. Метод пиймає параметром `Consumer<? super T> action`, який здійснює обробку кожного елементу даних. Цей метод може використовуватися замість ітератора. Приклад використання функції, щоб вивести усі елементіи потоку на екран.

```
Stream<Integer> numbers = Stream.of(1,2,3,4,5);
numbers2.forEach (i -> System.out.print (i + ",")); // 1,2,3,4,5,
```

4. Пошук елементів у потоці можна робити за допомогою методу `match()`:

```
Stream<Integer> numbers = Stream.of(1,2,3,4,5);
System.out.println("Потік містить 4?" + Numbers3.anyMatch (i -> i == 4));
// Потік містить 4? true

Stream<Integer> numbers = Stream.of(1,2,3,4,5);
System.out.println("Потік містить всі елементи менше 10?" + numbers.allMatch (i -> i <10));
// Потік містить всі елементи менше 10? true

Stream<Integer> numbers = Stream.of(1,2,3,4,5);
System.out.println("Потік не містить 10?" + numbers.noneMatch (i -> i == 10));
// Потік не містить 10? true
```

5. Пошук першого входження елементів здійснюється за допомогою методу `findFirst()`, який відноситься до кінцевих операцій короткого замикання. Приклад використання методу `findFirst()` для пошуку першого рядка з потоку, що починається з D.

```
Stream <String> names = Stream.of ("Pankaj", "Amit", "David", "Lisa");
Optional <String> firstNameWithD = names4.filter (i -> i.startsWith ("D")). FindFirst ();
if (firstNameWithD.isPresent ()) {
System.out.println ("Ім'я, що починається з D =" + firstNameWithD.get ()); // David
}
```

Розглянуті вище операції мають багато варіацій, які дозволяють здійснювати більш спеціалізовану обробку даних.

Крім цього у Java 8 з’явилася можливість використання посилання на методи замість звичайного лямбда синтаксису шляхом вказання `тип :: НазваМетоду;` є три основні варіанти використання даного синтаксису:

- захоплюючий метод об’єкту - object::instanceMethod
- статичний метод класу - Class::staticMethod
- не статичний метод класу - Class::instanceMethod

у перших двох випадках посилання на метод еквівалентне лямбда виразу, що представляє параметри методу, тобто `System.out::println` еквівалентне `x -> System.out.println(x)`, а `Math::pow` еквівалентне `(x, y) -> Math.pow(x,`

y); а в третьому випадку - перший параметр стає цільовим об'єктом методу, наприклад, String::compareToIgnoreCase – це те ж саме, що й (x, y) -> x.compareToIgnoreCase(y); слід зазначити, що усі посилання на методи перетворюються компілятором у відповідні лямбда вирази, а при наявності кількох перевантажених методів (конструкторів) компілятор самостійно вибере найвідповідніший контексту з них; крім цього дозволяється як object використовувати this та super.

Таблиця 5.1.

Приклади реалізації посилань на методи

Тип методу	Посилання на метод	Лямбда вираз
Статичний метод	String::valueOf	x -> String.valueOf(x)
Не статичний метод	Object::toString	x -> x.toString()
Захоплюючий метод	x::toString	() -> x.toString()
Конструктор	ArrayList::new	() -> new ArrayList<>()

Інші приклади для Java streams:

```
public class TestForEach {
   public static void main(String[] args) {
      List<SomeClass> pl = ...;

      // метод forEach і стандартний лямбда синтаксис
      pl.forEach(p -> p.someMethod());
      // метод forEach і посилання на метод someMethod
      pl.forEach(SomeClass::someMethod);

      // при використанні вкладених лямбда виразів назва параметру
      // міняється з p на r
      pl.forEach(p -> { System.out.println(p.printCustom(r -> "Name: "
+ r.getFName() + " Second Name: " + r.getSName())); });

      // створення колекції List, на основі відфільтрованих даних
      List< SomeClass > pList = pl
                  .stream()
                  .filter(FileringMethod())
                  .collect(Collectors.toList());

      // Сумування даних
      long summ = pl
                  .stream()
                  .filter(FileringMethod())
                  .mapToInt(p -> p.getData())
                  .sum();

      // Середнє арифметичне
```

```
        Optional<Double> averageData = pl
                    .parallelStream()
                    .filter(FileringMethod())
                    .mapToDouble(p -> p.getData())
                    .average();

        }
}
```

У Java SE 9 додала чотири корисні нові методи до інтерфейсу java.util.Stream.

- dropWhile
- takeWhile
- ітерація
- ofNullable

Оскільки Stream є інтерфейсом то, перші два методи є методами за замовчуванням, а останні два - статичними методами. Два з них дуже важливі: `dropWhile` і `takeWhile`.

У Stream API метод `default Stream<T> takeWhile(Predicate<? super T> predicate))` повертає найдовшу послідовність елементів префікса, які відповідають умові предиката.

В якості аргументу береться предикат. Предикат є булевим виразом, який повертає або `true`, або `false`. Він поводиться по-різному для упорядкованих і невпорядкованих потоків.

Приклад для впорядкованого потоку:

```
        Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9,10)
        stream.takeWhile(x -> x <4).forEach(a -> System.out.println
(a))
        // вивід
        // 1
        // 2
        // 3
```

Оскільки цей потік є упорядкованим, метод `takeWhile()` повертає перші три елементи, які збігаються з нашим предикатом. Тут наш предикат полягає в тому, що «Елемент повинен бути менше 4».

Приклад для невпорядкованого потоку:

```
        Stream<Integer> stream = Stream.of(1,2,4,3,5,6,7,8,9,10)
```

```
stream.takeWhile(x -> x <4).forEach(a -> System.out.println
(a))
// вивід
// 1
// 2
```

Оскільки цей потік є невпорядкованим, метод `takeWhile()` повертає перші два елементи, які збігаються з нашим предикатом.

Це означає, що `takeWhile()` повертає всі префіксні елементи, поки вони не відповідають умові предиката. Коли цей предикат повертає `false` для першого елемента, він припиняє оцінювання і повертає елементи підмножини. Цей предикат обчислюється до тих пір, поки він не поверне false.

Метод `DropWhile()` працює з точністю до навпаки. Він видаляє найдовші елементи префікса, які збігаються з предикатом, і повертає інші елементи. Наприклад, код

```
Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9,10)
stream. DropWhile (x -> x <4).forEach(a ->
System.out.println (a))
```

виведе на екран числа від 4 до 10, а код

```
Stream<Integer> stream = Stream.of(1,2,4,3,5,6,7,8,9,10)
stream. DropWhile (x -> x <4).forEach(a ->
System.out.println (a))
```

виведе на екран всі елементи послідовності крім перших двох.

Метод `static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)` повертає потік елементів, які починаються з початкового значення, яке визначається першим параметром, співпадають з предикатом (2-й параметр) і генерує наступний елемент за допомогою 3-го параметра.

Він подібний до циклу for: перший параметр - ініціалізація, наступний параметр - умова, а остаточний параметр - створює наступний елемент (наприклад, операція інкременту або декремента).

Приклад

```
IntStream.iterate(2, x -> x < 20, x -> x *
x).forEach(System.out::println)
```

На екран виведеться 2, 4 , 16.

Метод `static <T> Stream<T> ofNullable(T t)` повертає послідовний потік, що містить один елемент `t`, якщо він не є null, інакше повертає порожній потік.

Пакети

Пакет – це механізм мови Java, що дозволяє об'єднувати класи в простори імен. Об'єднання класів в пакети дозволяє відділяти класи, що розроблені одними розробниками, від класів, що розроблені іншими розробниками, забезпечуючи тим самим унікальність імен класів в межах програми та усуваючи можливі конфлікти імен класів. Пакети можуть бути вкладеними одні в одних, утворюючи цим самим ієрархії пакетів. Будь-який зв'язок між вкладеними пакетами відсутній. Всі стандартні пакети належать ієрархіям `java` і `javax`, наприклад, `java.lang`, `java.util`, `java.net` тощо.

Створення пакетів

Створення пакетів відбувається за допомогою оператора `package` з вказуванням назв пакету і підпакетів (за необхідності), що розділені крапкою. Оператор `package` вказується на початку тексту програми перед операторами `import` та визначенням класу. Синтаксис оператора `package`:

```
package НазваПакету{.НазваПідпакету};
```

Приклад:

```
package mypack.subpack;

public class StartClass
{
      ...
}
```

Якщо оператор `package` в файлі з кодом програми не вказаний, то класи, що описані в цьому файлі розміщуються у *пакеті за замовчуванням*. Пакет за замовчуванням не має імені.

Використання пакетів вимагає, щоб файли і каталоги проекту та їх ієрархія були строго структурованими. Так назви пакету і його підпакетів мають співпадати з назвами каталогів, де вони розміщуються. Назви загальнодоступних класів мають співпадати з назвами файлів, де вони розміщуються. Ієрархія каталогів і файлів проекту має співпадати з ієрархією пакетів. Після компіляції ієрархія каталогів, де містяться файли класів, співпадає з ієрархією каталогів проекту. Наприклад, для пакету:

```
package mypack.subpack;

public class StartClass
{
      ...
```

```
}
```

ієрархія каталогів і файлів буде наступною:

. (кореневий каталог проекту)
- (файли пакету за замовчуванням)
- mypack/
    -- subpack/
        --- StartClass.java
        --- StartClass.class

У будь-якому випадку, використовуємо ми пакети чи ні, для компіляції проекту необхідно перейти в кореневий каталог проекту (в той, що містить каталог кореневого пакету) та викликати компілятор `javac`, передавши йому в параметрах шлях до файлу з методом `main` програми. Якщо програма використовує інші пакети, то, завдяки строгій ієрархії каталогів пакетів, шлях до їх файлів, що містять код відповідних класів, буде визначений компілятором автоматично. Наприклад, для компіляції згаданого вище пакету слід виконати команду:

```
javac mypack/subpack/StartClass.java
```

Для запуску програми на виконання слід в параметрах команди `java` передати повну назву пакету і клас, який слід запустити на виконання. Наприклад, для запуску згаданого вище пакету слід виконати команду:

```
java mypack.subpack.StartClass
```

Зверніть увагу! Компілятор працює з *файлами*, а інтерпретатор з *класами*.
Todo: Правила іменування пакетів.

Використання пакетів

Клас може використовувати всі класи з власного пакету і всі *загальнодоступні класи* з інших пакетів. Доступ до класів з інших пакетів можна отримати двома шляхами:

1. вказуючи повне ім’я пакету перед іменем кожного класу, наприклад,

```
java.util.Date today = new java.util.Date();
```

2. використовуючи оператор `import`, що дозволяє підключати як один клас так і всі загальнодоступні класи пакету, позбавляючи необхідності записувати імена класів з вказуванням повної назви пакету перед ними.

Оператор `import` слід розміщувати в коді програми після оператора `package` та перед оголошенням класів.

Для *підключення одного загальнодоступного класу пакету* необхідно за допомогою оператора `import` через крапку вказати повну ієрархію пакету та назву класу, який має бути імпортовано, наприклад,

```
import java.util.Date
Date today = new Date();
```

Для *підключення всіх загальнодоступних класів пакету* необхідно за допомогою оператора `import` через крапку вказати повну ієрархію пакету та символ зірочка (*), наприклад,

```
import java.util.*
Date today = new Date();
```

Який би з шляхів ви не обрали на розмір скомпільованої програми це не вплине. Однак слід зауважити, що оператор `import` з зірочкою можна застосовувати для імпортування лише одного пакету. Не можна використовувати, наприклад: "`import java.*`, `import java.*.*`", - щоб імпортувати всі пакети, що містять префікс `java`. В більшості випадків імпортується весь пакет не залежно від його розміру. Єдина ситуація, коли не можна імпортувати весь пакет – це конфлікт імен. Конфлікт імен виникає тоді, коли в пакетах, що підключаються, є класи з однаковими іменами. В цій ситуації, при одночасному використанні конфліктуючих класів, слід явно вказувати якому з пакетів він належить:

```
import java.util.*;
import java.sql.*;

java.util.Date today = new java.util.Date();
java.sql.Date dbToday = new java.sql.Date();
```

Якщо ж в програмі використовується клас, що належить тільки одному з кількох пакетів, то можна обмежитися явним вказуванням в операторі `import` приналежності класу потрібному пакету:

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

Date today = new Date();
```

Статичний імпорт пакетів

Починаючи з Java SE 5.0 у мову додано можливість імпортувати статичні методи і поля класів. Для цього при підключені пакету слід вжити ключове слово `static` та вказати назву пакету, або назву пакету класу та статичного методу чи поля, які ви хочете підключити:

```
import static НазваПакету{.НазваПідпакету}.НазваКласу.
    НазваСтатичногоМетодуАбоПоля;
import static НазваПакету{.НазваПідпакету}.*;
```

Наприклад,

```
// підключити тільки java.lang.Math.sqrt
import static java.lang.Math.sqrt;
// підключити всі статичні члени пакету java.lang
import static java.lang.*;
```

Статичний імпорт дозволяє не вживати явно назву класу при звертанні до статичного поля або методу класу, наприклад:

```
sqrt(pow(x, 3));           //  замість Math.sqrt(Math.pow(x, 3));

dow = d.get(DAY_OF_WEEK); //  замість dow = d.get(Calendar.DAY_OF_WEEK);
```

Області видимості

Класи, що оголошені без використання модифікаторів доступу (`public`, `private`, `protected`) видимі лише в меж пакету. Областю видимості елементів класу керують модифікатори доступу (`public`, `private`, `protected`). Області видимості, що визначаються кожним з модифікаторів доступу відображені у наступній таблиці.

Таблиця 5.1.

Вплив модифікаторів доступу на області видимості

Область видимості / Модифікатор	Клас	Пакет	Похідний клас	Зовнішнє середовище
`private`	Так	-	-	-
без модифікатора	Так	Так	-	-
`protected`	Так	Так	Так	-
`public`	Так	Так	Так	Так

Висновок

У лекції розглянуто види та особливості внутрішніх класів мови Java. Розглянуто питання створення та використання пакетів, а також області видимості модифікаторів.

6. Спадкування та поліморфізм

План

1. Спадкування.
2. Поліморфізм.
3. Приведення об’єктних типів.

Спадкування

Спадкування в ООП призначене для розширення функціональності існуючих класів шляхом утворення нових класів на базі вже існуючих. У Java реалізована однокоренева архітектура класів згідно якої всі класи мають єдиного спільного предка (кореневий клас в ієрархії класів) – клас `Object`. Решта класів мови Java утворюються шляхом успадковування даного класу. Будь-яке спадкування у мові Java є відкритим, при цьому аналогів захищеному і приватному спадкуванню мови C++ не існує. На відміну від C++ у Java можливе спадкування лише одного базового класу (множинне спадкування відсутнє). Спадкування реалізується шляхом вказування ключового слова `class` після якого вказується назва *підкласу*, ключове слово `extends` та назва *суперкласу*, що розширюється у новому підкласі. Синтаксис реалізації спадкування:

```
class Підклас extends Суперклас
{
     Додаткові поля і методи
}
```

В термінах мови Java базовий клас найчастіше називається *суперкласом*, а похідний клас – *підкласом*. Дана термінологія запозичена з теорії множин, де підмножина міститься у супермножині.

При наслідувані у Java дозволяється перевизначення і перевантаження методів та полів. При цьому область видимості методу, що перевизначається, має бути не меншою, ніж область видимості цього методу у суперкласі, інакше компілятор видасть повідомлення, про обмеження привілеїв доступу до даних. Перевизначення методу полягає у визначені у підкласі методу з сигнатурою методу суперкласу. При виклику такого методу з-під об’єкта підкласу викличеться метод цього підкласу. Якщо ж у підкласі немає визначеного методу, що викликається, то викличеться метод суперкласу. Якщо ж у суперкласі даний метод також відсутній, то згенерується повідомлення про помилку.

Перевизначення у підкласах елементів суперкласів (полів або методів) призводить до їх приховування новими елементами. Бувають ситуації, коли у методах підкласу необхідно звернутися до цих прихованих елементів суперкласів. У цій ситуації слід використати ключове слово **`super`**, яке вказує, що елемент до якого йде звернення, розташовується у суперкласі, а не у підкласі. Синтаксис звертання до елементів суперкласу:

```
super.назваМетоду([параметри]); // виклик методу суперкласу
super.назваПоля                 // звертання до поля суперкласу
```

Використання ключового слова `super` у конструкторах підкласів має дещо інший сенс, ніж у методах. Тут воно застосовується для виклику конструктора суперкласу. Виклик конструктора суперкласу має бути першим оператором конструктора підкласу.

Конкретний конструктор, який необхідно викликати, вибирається по переданим параметрам. Явний виклик конструктора суперкласу часто є необхідним, оскільки підкласи не мають доступу до приватних полів суперкласів. Тож ініціалізація їх полів значеннями відмінними від значень за замовчуванням без явного виклику відповідного конструктора суперкласу є неможливою. Якщо виклик конструктора суперкласу не вказаний явно у підкласі або суперклас не має конструкторів, тоді автоматично викликається конструктор за замовчуванням суперкласу. Синтаксис виклику конструктора суперкласу з конструктора підкласу:

```
public НазваПідкласу([параметри])
{
      super([параметри]);
      оператори конструктора підкласу
}
```

Поліморфізм

Механізм поліморфізму забезпечує можливість присвоєння об'єктним змінним суперкласу об'єктів похідних класів та звертання з-під цих змінних до перевизначених у підкласі членів суперкласу. У Java всі об'єктні змінні є поліморфними. Поліморфізм реалізується за допомогою механізму динамічного (пізнього) зв'язування, який полягає у тому, що вибір методу, який необхідно викликати, відбувається не на етапі компіляції, а під час виконання програми. Для глибшого розуміння поліморфізму розглянемо покроково виклик методу класу:

1. Компілятор визначає об'явлений тип об'єкту і ім'я методу та нумерує всі методи з однаковою назвою у класі та всі загальнодоступні методи з такою ж назвою у його суперкласах.

2. Компілятор визначає типи параметрів, що вказані при виклику методу. Якщо серед усіх методів з вказаним іменем є лише один метод, типи параметрів якого співпадають з вказаним, то відбувається його виклик. Цей процес називається *дозволом перевантаження*. Якщо компілятор не знаходить жодного методу з підходящим набором параметрів або в результаті перетворення типів виявлено кілька методів, що відповідають даному виклику, то видається повідомлення про помилку.

3. Якщо метод є приватним, статичним, фінальним або конструктором, то для нього застосовується механізм *статичного зв'язування*. Механізм статичного зв'язування передбачає визначення методу, який необхідно викликати, на етапі компіляції. В протилежному випадку метод, що необхідно викликати, визначається по фактичному типу неявного параметру (*динамічне зв'язування*).

4. Якщо для виклику методу використовується динамічне зв'язування, то віртуальна машина повинна викликати версію методу, що відповідає фактичному типу об'єкту на який посилається об'єктна змінна.

Оскільки на пошук необхідного методу потрібно багато часу, то віртуальна машина заздалегідь створює для кожного класу таблицю методів, в якій перелічуються сигнатури всіх методів і фактичні методи, що підлягають виклику. При виклику методу віртуальна

машина просто переглядає таблицю методів. Якщо відбувається виклик методу з суперкласу за допомогою ключового слова `super`, то при виклику методу переглядається таблиця методів суперкласу неявного параметру.

Приведення об'єктних типів

Приведення типів у Java відбувається вказуванням у круглих дужках перед змінною, яку необхідно привести до іншого типу, типу до якого її необхідно привести. Синтаксис приведення змінної до іншого типу:

```
(новийТип) змінна
```

У Java усі об'єктні змінні є типізовані. Механізми наслідування і поліморфізму дозволяють створювати нові типи (класи та інтерфейси) на базі вже існуючих та присвоювати об'єкти цих типів посиланням на об'єкти супертипу. В цьому випадку об'єкти підтипів мають ті самі елементи, що й об'єкти супертипу, тож таке висхідне приведення типів є безпечним і здійснюється компілятором автоматично. Проте присвоєння посиланню на об'єкт підтипу об'єкту супертипу не завжди є коректним, тому таке приведення вимагає явного приведення типів. При такому приведенні типів можливі дві ситуації:

- якщо посилання на об'єкт супертипу реально посилається на об'єкт підтипу, то приведення посилання на об'єкт супертипу до типу підтипу є коректним;
- якщо посилання на об'єкт супертипу посилається на об'єкт супертипу, то приведення посилання на об'єкт супертипу до типу підтипу викличе виключну ситуацію `ClassCastExeption`.

Наявність бодай одної такої виключної ситуації призводить до аварійного завершення програми. Щоб уникнути цього слід перед приведенням типів використати оператор `instanceof`, який повертає `true`, якщо посилання посилається на об'єкт фактичний тип якого є не вищим в ієрархії типів, ніж вказаний у операторі `instanceof`, і `false` у протилежному випадку. Синтаксис оператора `instanceof`:

```
посилання instanceof Ім'яТипу
```

Таким чином, основні правила приведення типів є наступними:

1. Приведення типів можна виконувати лише в ієрархії спадкування.
2. Щоби перевірити коректність приведення супертипу до підтипу слід використовувати оператор `instanceof`.

Висновок

У лекції розглянуто особливості реалізації спадкування, поліморфізму та приведення об'єктних типів у мові Java.

7. Ієрархія класів мови Java. Інтерфейси

План

1. Абстрактні класи.
2. Глобальний суперклас `Object`.
3. Об'єктні оболонки та автопакування.
4. Методи зі змінним числом параметрів.
5. Класи-перелічення.
6. Інтерфейси

Абстрактні класи

Абстрактні класи призначені бути основою для розробки ієрархій класів та не дозволяють створювати об'єкти свого класу. Вони реалізуються за допомогою ключового слова `abstract`. На відміну від звичайних класів абстрактні класи можуть містити абстрактні методи (а можуть і не містити). *Абстрактні методи* – це методи, що оголошені з використанням ключового слова `abstract` і не містять тіла. Розширюючи абстрактний клас можна залишити деякі або всі методи невизначеними. При цьому підклас автоматично стане абстрактним. Перевизначення у підкласі усіх абстрактних методів призведе до того, що підклас не буде абстрактним, що дозволить створювати на його основі об'єкти класу. Синтаксис оголошення абстрактного класу наведено в пункті «Класи та об'єкти». Синтаксис оголошення абстрактного методу:

```
[СпецифікаторДоступу] abstract Тип назваМетоду([параметри]);
```

Абстрактні класи корисні тоді, коли в ієрархії класів необхідно реалізувати методи з однаковими назвами, проте різною функціональністю і можливістю поліморфного виклику методів підкласів з-під посилання на абстрактний суперклас.

Глобальний суперклас `Object`

Глобальний суперклас `Object` є предком всіх класів – кожен клас у мові Java розширює клас `Object`. Однак явно відображати цей факт нема потреби, оскільки якщо суперклас явно не вказаний, то ним є клас `Object`. Наявність такого глобального суперкласу дозволяє застосовувати його для посилання на будь-який клас мови Java. Цей клас не є порожнім, а володіє кількома методами, які є корисними при роботі з підкласами даного класу. Декілька з цих методів наведено нижче:

- `equals (Object obj)` – перевіряє два об'єкти на еквівалентність; корисний при порівнянні об'єктів на рівність; обов'язково перевизначається, якщо похідний клас використовується у структурах даних на базі хеш-таблиць;
- `hashCode()` – обчислює хеш-код об'єкту класу; корисний при порівнянні об'єктів на рівність; обов'язково перевизначається, якщо похідний клас

використовується у структурах даних на базі хеш-таблиць; детальніше можна прочитати, наприклад, тут;

- `toString()` – перетворює об'єкт класу у рядок; автоматично викликається при використанні операції конкатенації рядка з об'єктом;
- `clone()` – створює та повертає копію об'єкту;
- `finalize()` – викликається збирачем сміття при знищенні об'єкту.

Усі ці методи при потребі можна перевизначити і використовувати для полегшення процесу написання та налагодження програми.

Об'єктні оболонки та автопакування

Бувають ситуації, коли необхідно перетворити змінну простого типу, наприклад, `int`, у об'єкт. Для цього усі прості типи мають класи-аналоги та існує механізм автопакування.

Таблиця 7.1.

Прості типи та їх класи-аналоги

Простий тип	Клас-оболонка
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`void`	`Void`

Перші 6 класів мають спільний суперклас `Number`. Класи-оболонки є фінальними, що не дозволяє перевизначати методи цих класів. Крім цього змінити значення, що зберігається в об'єкті, неможливо.

Автопакування полягає у автоматичному перетворенні змінної простого типу у об'єкт відповідного їй класу. Воно відбувається тоді, наприклад, коли відбувається розміщення числових констант у об'єктах-контейнерах, що приймають об'єкти класів-оболонок, а не числа простих типів, наприклад, клас `ArrayList`. При присвоєнні об'єктів класів-оболонок відповідним змінним простих типів відбувається процес, що називається *авторозпаковування*, в результаті якого числове значення, що зберігається у об'єкті, виймається з об'єкту і присвоюється змінній простого типу. Автопакування і авторозпаковування можуть відбуватися під час обчислення арифметичних виразів. При роботі з об'єктами класів-оболонок слід бути обережним, оскільки, наприклад, операція порівняння двох об'єктів може дати некоректний результат, бо при її виконанні порівнюються не значення у об'єктах, а посилання на об'єкти. Слід також відмітити, що за

процес автопакування і авторозпаковування відповідає не віртуальна машина, а компілятор, який генерує під час компіляції виклики необхідних методів. Класи-оболонки є корисними ще по тій причині, що вони містять статичні методи по приведенню чисел у текстові рядки і навпаки.

Методи зі змінним числом параметрів

Бувають ситуації, коли в метод необхідно передати невизначене число параметрів. У таких ситуаціях на порятунок приходять методи зі змінним числом параметрів. Синтаксис оголошення такого методу наступний:

```
[СпецифікаторДоступу] [static] [final] Тип назва методу ( {ТипПар параметр}, ТипПар...змінна ) [throws класи]
{
      [Тіло методу]
      [return [значення]];
}
```

Тут три крапки (`...`) – це частина коду. Вони вказують, що в цьому місці може бути довільна кількість параметрів. Перед довільною кількістю параметрів певного типу можуть зустрічатися звичайні параметри перелічені через кому. Змінна з довільною кількістю параметрів – це насправді масив, у який розміщуються передані у довільній кількості параметри. При передачі змінних простих типів у змінну-масив об'єктного типу, наприклад, `Object`, відбувається автопакування. Далі параметри можна опрацьовувати як звичайний масив. Приклад методу з довільною кількістю параметрів, що визначає максимальне передане значення:

```
public static double maxDoub (double ... values)
{
      double largest = Double.MIN_VALUE;
      for (double v: values)
            if (v > largest)
                  largest = v;
      return largest;
}
```

Виклик цього методу може виглядати так:

```
double m = maxDoub (3.1, 40.0, -3, 20);
```

При цьому компілятор здійснить перетворення цього виклику у наступний:

```
double m = maxDoub (new double[] {3.1, 40.0, -3, 20});
```

Класи-перелічення

Починаючи з Java 5.0 мовою Java підтримуються типи перелічення. Тип перелічення насправді є підкласом класу `Enum`. При цьому, можливе існування стількох екземплярів класу, скільки констант-перелічень задано при визначені типу перелічення. Таким чином для перевірки констант-перелічень на рівність можна застосувати операцію ==, а не метод `equals()`. Оскільки тип перелічення є класом, то він успадковує відкриті члени класу `Enum` та може містити свої поля, методи і конструктори. Очевидно, що конструктори можуть викликатися лише при створенні констант-перелічень, тому їх специфікатор доступу слід визначати як `private`.

Скорочений синтаксис оголошення типу перелічення:

```
[СпецифікаторДоступу]    enum    НазваТипу    {ОБ'ЄКТ_КОНСТАНТА1,
      ОБ'ЄКТ_КОНСТАНТА2,...};
```

Повний синтаксис оголошення типу перелічення:

```
[СпецифікаторДоступу] enum НазваТипу
{
      ОБ'ЄКТ_КОНСТАНТА1([параметри]),
ОБ'ЄКТ_КОНСТАНТА2([параметри]),...;
      [конструктори]
      [методи]
      [поля]
}
```

Приклад оголошення типу перелічення:

```
enum Size
{
      SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

      private Size (String abrr) {abrreviation = abrr;}
      public String getAbrreviation() {return abbreviation;}

      private String abbreviation;
}
```

Інтерфейси

Інтерфейси вказують що повинен робити клас не вказуючи як саме він це повинен робити. Інтерфейси покликані компенсувати відсутність множинного спадкування у мові Java та гарантують визначення у класах оголошених у собі прототипів методів. Синтаксис оголошення інтерфейсів:

```
[public] interface НазваІнтерфейсу
{
      Прототипи методів та оголошення констант інтерфейсу
}
```

Інтерфейси можуть містити прототипи методів, які мають визначатися у класі, що відповідає цьому інтерфейсу, і константи, які автоматично успадковуються класом, що реалізує цей інтерфейс. Всі методи інтерфейсу вважаються загальнодоступними, тому оголошувати методи як `public` у інтерфейсі нема необхідності. Всі поля, що оголошені у інтерфейсі автоматично вважаються такими, що оголошені як `public static final`, тому додавати ці модифікатори самостійно необхідності також нема. Інтерфейси можуть успадковувати інші інтерфейси, утворюючи таким чином ієрархії інтерфейсів. Синтаксис реалізації спадкування у інтерфейсів співпадає з синтаксисом реалізації спадкування у класах.

Оскільки інтерфейс не є класом, то створити його об'єкт за допомогою оператора `new` неможливо. Проте можна оголосити посилання на інтерфейсний тип та присвоїти цьому посиланню об'єкт, що реалізує цей інтерфейс.

На відміну від спадкування клас може реалізувати кілька інтерфейсів. Ця особливість і компенсує відсутність множинного спадкування у мові Java.

При іменуванні інтерфейсів загальноприйнятим є використання великої букви «`I`» на початку назви інтерфейсу за якою іде назва інтерфейсу з великої букви, наприклад, `IReadable`.

Щоб клас реалізував інтерфейс необхідно:

1. Оголосити за допомогою ключового слова `implements`, що клас реалізує інтерфейс. Якщо клас реалізує кілька інтерфейсів, то вони перелічуються через кому після ключового слова `implements`.
2. Визначити у класі усі методи, що вказані у інтерфейсі.

Приклад використання інтерфейсів:

```
// оголошуємо інтерфейс IMoveable
interface IMoveable
{
      void move (double x);   // прототип методу
}

// оголошуємо інтерфейс IPowered, що успадковує інтерфейс IMoveable
interface IPowered extends IMoveable
{
      double milesToFueling (); // прототип методу
      int SPEEDLIMIT = 100;   // константа
}

// оголошуємо клас, що реалізує інтерфейс IPowered
public class Car extends Object implements IPowered
{
      public Car (double lFuel, double lMilesPerGalon)
      {
```

```
            fuel = lFuel;
            milesPerGalon = lMilesPerGalon;
            distance = 0.0;
      }

      private Car ()
      {
            fuel = 0.0;
            milesPerGalon = 0.0;
            distance = 0.0;
      }

      public void move (double x)
      {
            distance = distance + x;
            fuel = fuel - distance / milesPerGalon;
      }

      public double getDistance()
      {
            return distance;
      }

      public double milesToFueling ()
      {
            return fuel*milesPerGalon;
      }

      private double distance;
      private final double milesPerGalon;
      private double fuel;
}
```

У Java 8 з’явилася можливість оголошення інтерфейсів з статичними методами з тілом та методами за замовчуванням (методи віртуального розширення - *virtual extension methods*). Дана функціональність покликана вирішити проблему, коли метод є оголошений у інтерфейсі, але в програмі не використовується, проте його необхідно постійно перевизначати в силу специфіки механізму інтерфейсів. Застосування методів за замовчуванням дає змогу задати методу тіло за замовчуванням, яке буде використовуватися коли метод не буде перевизначатися в класі, або буде замінятися на перевизначене у класі, коли метод у класі перевизначає метод у інтерфейсі.

Синтаксис методів за замовчуванням у інтерфейсах:

```
[public] interface НазваІнтерфейсу
{
default модифікатори типРезультату назваМетоду([параметри]) "{"тіло"}"
модифікатори типРезультату назваМетоду([параметри]) default "{"тіло"}"
}
```

Приклад реалізації:

```
public interface ISomeInterface
{
      default void prnt1(String str) {System.out.println(str);}
      void prnt2(String str) default {System.out.println(str);}
}
```

При використанні методів за замовчуванням може виникнути ситуація, коли клас реалізує два інтерфейси, які мають методи з однаковою назвою і тілом за замовчуванням, що призводить до помилки компіляції. Для вирішення цієї помилкової ситуації, слід перевизначити ці методи у класі, що реалізує дані інтерфейси. Якщо при цьому виникне ситуація, коли треба викликати метод за замовчуванням з інтерфейсу, а не перевизначений у класі, то при виклику у тілі перевизначеного методу через крапку слід вказати назву інтерфейсу з якого слід викликати метод, ключове слово `super`, назву методу, який треба викликати і його параметри:

```
public class SomeClass implements A, B {
    public void someMethod(){
       A.super.someMethod(); }}
```

У Java 9 з'явилася можливість оголошення інтерфейсів з приватними методами та приватними статичними методами. Синтаксис оголошення даних методів не відрізняється від синтаксису їх оголошення в класах.

Приклад реалізації:

```
public interface Card {
  private Long createCardID(){
    // тіло
  }
  private static void displayCardDetails(){
    // тіло
  }
}
```

Висновок

У лекції розглянуто особливості створення та застосування абстрактних класів, методів зі змінним числом параметрів, класів-перелічення та інтерфейсів. Проаналізовано глобальний суперклас `Object`, особливості об'єктних оболонок та автопакування.

8. Рефлексія

План

1. Визначення рефлексії.
2. Клас `Class`.
3. Клас `Field`.
4. Клас `Modifier`.

5. Клас `Method`.
6. Клас `Constructor`.
7. Вказівники на методи.
8. Проксі-класи

Визначення рефлексії

Рефлексія – це механізм, за допомогою якого програма може відслідковувати і модифікувати власну структуру і поведінку під час виконання. Рефлексивно-орієнтоване програмування, або рефлексивне програмування – функціональне розширення парадигми об'єктно-орієнтованого програмування. Рефлексивно-орієнтоване програмування включає в себе самоперевірку, самомодифікацію і самоклонування. Головне достоїнство рефлексивно-орієнтованої парадигми полягає в динамічній модифікації програми, яка може бути визначена і виконана під час роботи програми. Тобто програмна архітектура сама визначає, що саме можна робити під час роботи виходячи з даних, сервісів і специфічних операцій. Рефлексія – не лише потужний, але й складний механізм, який перш за все цікавий програмістам, що розробляють інструментальні засоби. При розробці прикладних програм його переважно не застосовують.

Рефлексія дозволяє отримувати і досліджувати інформацію про поля, методи і конструктори класів. Також можна виконувати операції над полями і методами які досліджуються. Інтерфейс Java Reflection API дозволяє зробити наступне:

- визначити клас об'єкта;
- отримати інформацію про модифікатори класу, поля, методи, конструктори і суперкласи;
- з'ясувати, які константи і методи належать інтерфейсу;
- створити екземпляр класу, ім'я якого невідоме до моменту виконання програми;
- отримати і встановити значення поля об'єкта.
- викликати метод об'єкта.
- створити новий масив, розмір і тип компонентів якого невідомі до моменту виконання програм.

Java Reflection API містить такі основні класи:

- `Class` – об’єкт цього класу описує властивості конкретного класу.
- `Field` – об’єкт цього класу описує поля конкретного класу;
- `Method` – об’єкт цього класу описує методи конкретного класу, а також дозволяє реалізувати вказівники на методи;
- `Constructor` – об’єкт цього класу описує конструктори конкретного класу;
- `Array` – дозволяє динамічно створювати масиви.

Клас `Class`

Клас `Class` належить пакету `java.lang`. Об'єкт класу `Class` описує властивості конкретного типу (класу або примітивного типу). Даний об'єкт можна отримати трьома способами:

- за допомогою методу `getClass()` класу `Object`;
- за допомогою статичного методу `forName()` класу `Class`, який можна застосовувати, якщо ім'я класу або інтерфейсу знаходиться в текстовому рядку;
- за допомогою оператора `.class`, який можна застосовувати навіть з примітивними типами даних, наприклад, `int.class`.

Таким чином наступні три записи повертають один і той самий об'єкт класу `Class`, що описує тип (клас) `Date`:

`Date d = new Date();` `Class cl = d.getClass();`	`String str = "java.util.Date";` `Class cl = Class.forName(str);`	`Class cl = Date.class;`

Починаючи з Java 5.0 клас `Class` є параметризованим, тобто `Date.class` насправді повертає `Class<Date>`. Проте на практиці цей параметр цілком можна ігнорувати і працювати з звичайним класом `Class`.

Віртуальна машина Java підтримує унікальний об'єкт `Class` для кожного типу. Внаслідок цього для порівняння об'єктів можна використовувати операції порівняння, наприклад, ==.

Для одержання назви класу у вигляді текстового рядка використовують метод `getName()` класу `Class`. Якщо клас знаходиться у пакеті, то даний метод поверне ім'я класу з врахуванням назви пакету, який містить цей клас. Наприклад, якщо `d` – об'єкт класу `Date`, то `d.getClass().getName()` поверне рядок `"java.util.Date"`. Проте слід зауважити, що для масивів даний метод повертає доволі дивні імена, наприклад:

- `Double[].class.getName()` повертає `"[Ljava.lang.Double;"`;
- `int[].class.getName()` повертає `"[I"`.

Іншим корисним методом даного класу є `newInstance()`. Він дозволяє створювати новий об'єкт класу, який описує екземпляр класу `Class` під час роботи програми. Наприклад,

```
Object obj = Class.forName("java.util.Date").newInstance();
```

Для ініціалізації новоствореного об'єкту даний метод використовує конструктор за замовчуванням. Якщо конструктор за замовчуванням відсутній, то генерується виключення. Якщо при створенні об'єкту на основі імені класу треба передати

конструктору будь-які параметри, то слід застосувати метод `newInstance()` класу `Constructor`.

Для визначення типів елементів масиву використовується метод `getComponentType()`, що повертає об'єкт класу `Class`, який описує тип масиву.

Клас `Field`

Клас `Field` належить пакету `java.lang.reflect`. Об'єкт класу `Field` описує властивості полів класу. Для одержання масиву загальнодоступних полів класу, що аналізується, і його суперкласів використовується метод `getFields()` класу `Class`, а для одержання всіх полів – метод `getDeclaredFields()`. Іншими методами класу `Field`, що часто застосовуються є:

- `getName()` – повертає ім'я класу, що аналізується;
- `getType()` – повертає об'єкт класу `Class`, що описує тип поля;
- `getModifiers()` – повертає ціле число, що відповідає використовуваним модифікаторам полів (`public`, `static` і т.п.); використовується сумісно з методами `isPublic()`, `isStatic()` і т.п., які здійснюють декодування ознак, що повертає `getModifiers()`;
- `get(Object obj)` – повертає значення поля об'єкту переданого через параметр, яке (поле) описує даний об'єкт класу `Field`; даний метод можна застосовувати тільки для загальнодоступних полів; щоб зробити поле примусово загальнодоступним (якщо програма не контролюється диспетчером захисту) можна використати метод `setAccessible(true)`, який відноситься до класу `AccessibleObject` – спільному суперкласу класів `Field`, `Method` і `Constructor`;
- `set(Object obj, Object value)` – встановлює значення поля об'єкту `obj`, яке описується даним об'єктом класу `Field` рівним значенню, що міститься в `value`; даний метод можна застосовувати тільки для загальнодоступних полів; щоб зробити поле примусово загальнодоступним (якщо програма не контролюється диспетчером захисту) можна використати метод `setAccessible(true)`.

Приклад методу, що аналізує поля класу, який йому передається через параметр.

```
public static void printFields(Class cl)
{
      Field[] fields = cl.getDeclaredFields();
      for(Field f: fields)
      {
            Class type = f.getType();
            String name = f.getName();
            System.out.print(" ");
            String modifiers = Modifier.toString(f.getModifiers());
            if (modifiers.length()>0) System.out.print(modifiers + " ");
```

```
                System.out.println(type.getName() + " " + name + ";");
        }
}
```

Клас `Modifier`

Клас `Modifier` належить пакету `java.lang.reflect`. Він містить статичні методи, які перевіряють розряди числа `modifiers`, які відповідають модифікаторам, що визначаються іменами методів:

- `static boolean isAbstract(int modifiers)`
- `static boolean isFinal(int modifiers)`
- `static boolean isInterface(int modifiers)`
- `static boolean isNative(int modifiers)`
- `static boolean isPrivate(int modifiers)`
- `static boolean isProtected(int modifiers)`
- `static boolean isPublic(int modifiers)`
- `static boolean isStatic(int modifiers)`
- `static boolean isStrict(int modifiers)`
- `static boolean isSynchronized(int modifiers)`
- `static boolean isVolatile(int modifiers)`

`static String toString(int modifiers)` – повертає рядок з модифікаторами, що відповідають бітам у цілому числі `modifiers`.

Клас `Method`

Клас `Method` належить пакету `java.lang.reflect`. Об'єкт класу `Method` описує властивості методів класу. Для одержання масиву загальнодоступних методів класу, що аналізується використовується метод `getMethods()` класу `Class`, а для одержання всіх методів – метод `getDeclaredMethods()`. Іншими методами класу `Method`, що часто застосовуються є:

- `getName()` – повертає ім'я класу, що аналізується;
- `getType()` – повертає об'єкт класу `Class`, що описує тип поля;
- `getModifiers()` – повертає ціле число, що відповідає використовуваним модифікаторам методів (`public`, `static` і т.п.); використовується сумісно з методами `isPublic()`, `isStatic()` і т.п., які здійснюють декодування ознак, що повертає `getModifiers()`
- `getReturnType()`– повертає об'єкт класу `Class`, що описує тип значення, яке повертає аналізований метод;
- `getParameterTypes()` – повертає масив об'єктів класу `Class`, що описують типи параметрів, які приймає аналізований метод;
- `getExceptionTypes()` – повертає масив об'єктів типу `Class`, які являють собою типи виключень, що генерує даний метод.

Приклад методу, що аналізує методи класу, який йому передається через параметр.

```
public static void printMethods(Class cl)
{
      Method[] methods = cl.getDeclaredMethods();

      for (Method m: methods)
      {
            Class retType = m.getReturnType();
            String name = m.getName();

            System.out.print(" ");

            /* Вивід модифікаторів, типу значення, що повертається, і
            назви  методу */

            String modifiers = Modifier.toString(m.getModifiers());
            if(modifiers.length()>0) System.out.print(modifiers + " ");
            System.out.print(retType.getName() + " " + name + "(");

            // Вивід типів параметрів аналізованого методу
            Class[] paramTypes = m.getParameterTypes();
            for (int j = 0; j < paramTypes.length; j++)
            {
                  if (j>0) System.out.print(", ");
                  System.out.print(paramTypes[j].getName());
            }
            System.out.println(");");
      }
}
```

Клас `Constructor`

Клас `Constructor` належить пакету `java.lang.reflect`. Об'єкт класу `Constructor` описує властивості конструкторів класу. Для одержання масиву загальнодоступних конструкторів класу, що аналізується використовується метод `getConstructors()` класу `Class`, а для одержання усіх конструкторів – метод `getDeclaredConstructors()`. Іншими методами класу `Constructor`, що часто застосовуються є:

- `getName()` – повертає ім'я класу, що аналізується;
- `getType()` – повертає об'єкт класу `Class`, що описує тип поля;
- `getModifiers()` – повертає ціле число, що відповідає використовуваним модифікаторам конструкторів (`public`, `private` і т.п.); використовується сумісно з методами `isPublic()`, `isPrivate()` і т.п., які здійснюють декодування ознак, що повертає `getModifiers()`
- `getParameterTypes()` – повертає масив об'єктів класу `Class`, що описують типи параметрів, які приймає аналізований конструктор.
- `getExceptionTypes()` – повертає масив об'єктів типу `Class`, які являють собою типи виключень, що генерує даний конструктор.

Приклад методу, що аналізує конструктори класу, який йому передається через параметр.

```
public static void printConstructors(Class cl)
{
      Constructor[] constructs = cl.getDeclaredConstructors();

      for (Constructor c: constructs)
      {
            String name = c.getName();
            System.out.print(" ");
            String modifiers = Modifier.toString(c.getModifiers());
            if(modifiers.length()>0) System.out.print(modifiers + " ");
            System.out.print(name + "(");

            // Вивід типів параметрів аналізованого конструктора
            Class[] paramTypes = c.getParameterTypes();
            for (int j = 0; j < paramTypes.length; j++)
            {
                  if (j>0) System.out.print(", ");
                  System.out.print(paramTypes[j].getName());
            }
            System.out.println(");");
      }
}
```

Вказівники на методи

Одним з побічних ефектів рефлексії у мові Java є можливість реалізації вказівників на методи, які відсутні у самій мові. Для одержання методу, який слід викликати використовується метод

```
Method getMethod(String name, Class…parameterTypes),
```

що належить класу `Class`. Перший параметр містить назву методу, другий – змінну кількість типів параметрів. Однак методів з однаковими назвами може бути багато, тому слід ретельно вибирати потрібний.

Одержавши об'єкт класу `Method`, що описує метод, який слід викликати, можна здійснити виклик даного методу використавши метод

```
Object invoke(Object obj, Object…args),
```

що належить класу `Method`. Перший параметр – неявний параметр будь-якого методу, який є посиланням на об'єкт з-під якого здійснюється виклик методу. Якщо метод є статичним, то цей параметр рівний `null`. Якщо тип що повертається є простим, то він перетворюється в інтерфейсний клас, наприклад, тип `double` перетвориться в об'єкт класу `Double`. Починаючи з Java 5.0 таке перетворення здійснюється автоматично.

Приклад застосування вказівників на методи для табулювання будь-яких функцій:

```
public static void PrintTable(double from, double to, int n, Method f)
{
	double dx = (to - from) / (n-1);
	for(double x = from; x <=to; x+=dx)
	{
		try
		{
			double y = (Double) f.invoke(null, x);
			System.out.printf("%10.4f | %10.4f /n", x, y);
		}
		catch (Exception e)
		{
			e.printStackTrace();
		}
	}
}
```

Використовуючи дану техніку можна робити те саме, що й за допомогою вказівників у мові C\C++ і делегатів у мові C#.

Проксі-класи

Механізм проксі-класів (див. паттерн Замісник), або класів посередників, використовується для того, щоб під час виконання програми створювати нові класи, що реалізують задані інтерфейси, причому на етапі компіляції ці інтерфейси можуть бути невідомими. В прикладному програмуванні така ситуація виникає дуже рідко, тому цей механізм більше пригодиться системним програмістам.

Проксі-клас містить всі методи, які задекларовані у вказаних інтерфейсах, а також всі методи класу `Object`. Однак задати новий код для цих методів під час виконання програми не можна. Замість цього програміст має створити обробник викликів, який являє собою об'єкт класу, що реалізує інтерфейс `InvocationHandler`. В цьому інтерфейсі оголошений єдиний метод:

```
Object invoke (Object proxy, Method method, Object[] args).
```

При виклику будь-якого методу з проксі-об'єкту автоматично викликається метод `invoke` обробника викликів, який отримує на вхід об'єкт класу `Method`, що описує метод, виклик якого призвів до виклику методу `invoke`, і параметри що були передані даному методу. Обробник виклику має вміти опрацьовувати виклик.

Для створення проксі-об'єкту використовується метод `newProxyInstance` класу `Proxy`. Цей метод отримує 3 параметри:

- Посилання на завантажувач класу. Для використання завантажувача класів по замовчуванню слід вказати `null`.

- Масив об’єктів класу `Class` – по одному для кожного інтерфейсу, який має бути реалізований.
- Обробник викликів.

Методи пакету `java.lang.reflect.Proxy` для роботи з проксі-класами:

- `static Class getProxy(ClassLoader loader, Class[] args)` – повертає проксі-клас, що реалізує задані інтерфейси.
- `static Object newProxyInstance (ClassLoader loader, Class[] interfaces, InvocationHandler handler)` – створює новий екземпляр проксі-класу, що реалізує задані інтерфейси. Всі методи викликають метод `invoke` обробника викликів.
- `static Boolean isProxyClass(Class c)` – повертає значення `true`, якщо `c` є проксі-класом.

Прилад використання проксі-класу для відстежування звертань до методів.

```
import java.lang.reflect.*;
import java.util.*;

/**
 * Обробник виклику, що виводить назву методу і його параметри, а потім
 * викликає фактичний метод
 */
class TraceHandler implements InvocationHandler
{
   /**
    * Конструктор TraceHandler
    * @param t Неявний параметр виклику методу
    */
   public TraceHandler(Object t)
   {
      target = t;
   }

   public Object invoke(Object proxy, Method m, Object[] args) throws
Throwable
   {
      // вивести неявний параметр
      System.out.print(target);
      // вивести назву методу
      System.out.print("." + m.getName() + "(");
      // вивести явний параметр
      if (args != null)
      {
         for (int i = 0; i < args.length; i++)
         {
            System.out.print(args[i]);
            if (i < args.length - 1) System.out.print(", ");
         }
      }
```

```
      System.out.println(")");

      // виклик фактичного методу
      return m.invoke(target, args);
   }

   private Object target;
}

/**
 * Програма демонструє використання проксі-класу для відстежування
 * звертань до методів.
 * @version 1.00 2000-04-13
 * @author Cay Horstmann
 */

public class ProxyTest
{
   public static void main(String[] args)
   {
      Object[] elements = new Object[1000];

      // Заповнення проксі-об'єктами для цілих чисел від 1 до 1000
      for (int i = 0; i < elements.length; i++)
      {
         Integer value = i + 1;
         // Створюємо обробник для заданого об'єкту
         InvocationHandler handler = new TraceHandler(value);
         //Створюємо об'єкт проксі класу і назначаємо для нього обробник
         Object proxy = Proxy.newProxyInstance(null, new Class[] {
Comparable.class } , handler);
         elements[i] = proxy;
      }

      // Генерація випадкового числа
      Integer key = new Random().nextInt(elements.length) + 1;

      // пошук індексу числа key у масиві elements
      int result = Arrays.binarySearch(elements, key);

      // вивести число, якщо таке існує
      if (result >= 0) System.out.println(elements[result]);
   }
}
```

Результат роботи програми:

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
...
288.compareTo(288)
288.toString()
```

Особливості проксі-класів

Проксі-класи володіють такими особливостями:

1. Проксі-класи створюються під час виконання програми. Після створення, вони стають звичайними класами, які опрацьовує віртуальна машина.
2. Всі проксі-класи розширюють клас `Proxy`. Такий клас містить лише одну змінну екземпляру, яка посилається на обробник викликів, що визначений у суперкласі `Proxy`. Будь-які додаткові дані мають зберігатися в обробнику викликів.
3. В усіх проксі-класах перевизначаються методи `toString()`, `equals()` і `hashCode()` класу `Object`. Ці методи лише викликають метод `invoke()`, що належить обробнику подій. Інші методи класу `Object` НЕ перевизначаються.
4. Імена проксі-класів невизначені.
5. Для конкретного завантажувача класів і заданого набору інтерфейсів може існувати тільки один проксі-клас.
6. Проксі-клас завжди є загальнодоступним і фінальним. Якщо всі інтерфейси, які реалізуються проксі-класом оголошені як `public`, то цей клас не належить жодному конкретному пакету. Інакше всі інтерфейси, що не оголошені як `public`, як і проксі-клас мають належати одному пакету.

Висновок

У лекції дано визначення рефлексії, розглянуто класи `Class`, `Field`, `Modifier`, `Method` та `Constructor`. Наведено особливості реалізації вказівників на методи та проксі-класів за допомогою рефлексії.

9. Анотації

План

1. Вступ
2. Оголошення та використання анотацій.
3. Обмеження при оголошені анотацій.
4. Політика утримування анотації.
5. Одержання анотацій під час виконання програми засобами рефлексії.
6. Інтерфейс `AnnotatedElement`.
7. Анотація-маркер.
8. Одноелементні анотації.
9. Вбудовані анотації

Вступ

Починаючи з JDK 5 мова Java підтримує засоби включення службової інформації в вихідні файли. Ця інформація не змінює роботу програми і називається *анотацією* (*метадані*). Анотації перш за все призначені для використання різноманітними

інструментальними засобами як в період розробки, наприклад, генераторами вихідного коду, так і в період розгортання програм.

Оголошення та використання анотацій

Анотації створюються з використанням механізму, що оснований на інтерфейсі.

Синтаксис оголошення анотації:

```
@interface НазваАнотації {
     {типДанихАнотації назваМетоду();}
}
```

Приклад оголошення анотації:

```
@interface FirstAnno{
     String description();
     int val();
}
```

Розглянемо особливості оголошення анотацій детальніше. Оголошення типу анотації починається з символу "@" за яким слідує ключове слово `interface`, після якого іде назва типу анотації. Тіло анотації може бути порожнім або містити лише методи, при цьому тіла цих методів визначаються не програмістом, а самою платформою Java.

Анотація не може містити ключового слова `extends`, але всі анотації автоматично розширюють інтерфейс `Annotation`. Тобто в анотаціях не використовується механізм спадкування, а лише суперінтерфейс `Annotation`, який оголошений в пакеті `java.lang.annotation`. У інтерфейсі `Annotation` перевизначаються методи `hashCode()`, `euals()`, `toString()`, які визначені в класі `Object`; а також визначається метод `annotationType()`, який повертає об'єкт класу `Object`, що в свою чергу представляє собою анотацію, з-під якої даний метод був викликаний.

Об'явивши анотацію, ви можете використовувати її для анотування будь-яких оголошень, наприклад, класів, полів, методів, параметрів і констант перечислень, навіть самих анотацій. Для анотування будь-якого оголошення анотацію слід розмістити безпосередньо перед цим оголошенням. Наприклад:

```
@FirstAnno (description = "Приклад анотації", val = 5)
public void someMethod() {}
```

В даному прикладі відбувається створення анотації для методу `someMethod()`. Створення анотації починається з символу "@" за яким слідує назва попередньо оголошеного типу анотації `FirstAnno`. В круглих дужках, які слідують за назвою типу анотації, через кому вказуються список ініціалізаторів елементів анотації (`description` і `val`). Як видно з прикладу, при роботі з анотаціями методи анотацій більше схожі на поля класу, ніж на звичайні методи. Так, щоб записати окремий елемент значення анотації

використовується синтаксис присвоєння значень полям або змінним, при цьому тип значення, яке записується в елемент анотації, має співпадати з типом параметра, що повертається методом з відповідною назвою.

Java дає можливість задавати членам анотацій значення за замовчуванням. Значення за замовчуванням вказується додаванням ключового слова `default` після оголошення складового елементу анотації з вказуванням після цього слова самого значення за замовчуванням. Синтаксис оголошення значення за замовчуванням:

```
тип назваЕлементу() default значення;
```

Приклад:

```
@interface FirstAnno{
     String description() default "Hello World :)";
     int val() default 1;
}

@FirstAnno() //description і val зі значеннями за замовчуванням
public void someMethod() {}
```

Обмеження при оголошені анотацій

Існує кілька обмежень, що стосуються оголошення анотацій. По-перше, одна анотація не може спадкувати іншу. По-друге, усі методи, що оголошені в анотаціях не повинні приймати параметрів, більше того, вони повинні повертати один наступних типів:

- елементарний тип (`int`, `boolean`,…);
- об’єкт класу `String` або `Class`;
- тип-перелічення;
- тип іншої анотації;
- масив одного з вищезгаданих типів.

По-третє, анотації не можуть бути параметризованими, тобто вони не можуть приймати параметри типу. І на сам кінець, в методах анотації не може бути вказана конструкція `throw`.

Політика утримування анотації

Політика утримування анотації (annotation retention policies) визначає на якому етапі обробки програми платформою Java дана анотація відкидається. Java визначає три такі політики (`SOURCE`, `CLASS`, `RUNTIME`), що інкапсульовані в перечисленні `java.lang.annotation.RetentionPolicy`.

Анотації з політикою типу SOURCE містяться лише в вихідному файлі з текстом програми і відкидаються при компіляції.

Анотації з політикою типу CLASS зберігаються у файли .class під час компіляції, але є недоступними віртуальній машині під час виконання.

Анотації з політикою типу RUNTIME зберігаються у файли .class під час компіляції, та доступні віртуальній машині під час виконання.

Увага! *Анотації оголошень локальних змінних у файли .class не зберігаються.*

Визначення політики утримування анотації здійснюється за допомогою вбудованої анотації @Retention, яка вказується перед оголошенням анотації. В загальному випадку визначення політики утримування анотації має наступний синтаксис:

```
@Retention (політикаУтримування)
```

Наприклад, щоб задати для анотації @FirstAnno політику утримування CLASS слід перед оголошенням даної анотації додати анотацію @Retention (RetentionPolicy.CLASS):

```
@Retention (RetentionPolicy.CLASS)
@interface FirstAnno{
     String description();
     int val();
}

@FirstAnno (description = "Приклад анотації", val = 5)
public static void someMethod() {}
```

УВАГА!!! Кожна анотація має мати визначену політику утримування анотації. Компілювання анотації, яка не має визначеної політики утримування, призведе до полилок компіляції.

Одержання анотацій під час виконання програми засобами рефлексії

Хоч анотації призначені для використання інструментальними засобами, проте якщо вони використовують політику утримування RUNTIME, то до них можна звернутися під час виконання програми використовуючи рефлексію. Одержання анотації під час виконання програми засобами рефлексії виконується в кілька кроків.

На першому кроці слід одержати об’єкт класу `Class`, що представляє клас, анотацію якого хочемо одержати. Для цього можна викликати метод `getClass()`, який визначений в класі `Object`.

На другому кроці за допомогою рефлексії слід одержати об’єкт того елемента (метод, поле,…), анотацію якого хочемо одержати та викликати з-під цього об’єкта (об’єкту класу `Method`, `Field`, `Constructor`…) метод `getAnnotation(типАнотації)`, який поверне об’єкт-анотацію, тип якої був переданий методу через параметр і визначений для цього об’єкту. Якщо ж необхідно одержати анотацію класу, то можна одразу викликати метод `getAnnotation()` з-під об’єкту, що повернув метод `getClass()` на попередньому кроці. Якщо анотація для даного елементу не знайдена, то метод повертає `null`.

Повний синтаксис методу `getAnnotation()`:

```
<T extends Annotation> getAnnotation(Class<T> типАнотації)
```

Щоб одержати конкретне значення елементу анотації, яке було йому задане, слід викликати з-під одержаного на попередньому кроці об’єкту-анотації відповідний метод, який і поверне шукане значення.

Для одержання всіх анотацій використовується метод `getAnnotations()`. Синтаксис методу:

```
Annotation[] getAnnotations().
```

Приклад програми, що містить метод, який виводить на екран свою анотацію:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

//оголошення типу анотації
@Retention(RetentionPolicy.RUNTIME)
@interface FrstAnno{
   String descr();
   int val();
}

class Test{
   // Задаємо методу анотацію
   @FrstAnno(descr = "Це є метод someMethod", val = 5)
   public static void someMethod()
   {
        Test obj = new Test();
```

```
        // Одержуємо анотацію методу і відображаємо
        // значення її елементів на екрані
        try{
              // 1. Спочатку одержимо об’єкт Class,
              // що описує клас Test
              Class<?> c = obj.getClass();

              // 2. Одержимо об’єкт Method, що
              // представляє метод someMethod
              Method m = c.getMethod(“someMethod”);

              // 3. Одержуємо анотацію методу
              FrstAnno anno = m.getAnnotation(FrstAnno.class);

              // 4. Відображуємо анотацію
              System.out.println(anno.descr() + “ ” + anno.val());
        }
        catch (NoSuchMethodException ex)
        {
              System.out.println(“Метод не знайдено”);
        }
    }

    public static void main (String args[])
    {
        someMethod();
    }
}
```

Інтерфейс AnnotatedElement

Інтерфейс `AnnotatedElement` визначений у пакеті `java.lang.reflect`. Цей інтерфейс реалізований у класах `Method`, `Field`, `Constructor`, `Class` і `Package`, забезпечуючи цим самим підтримку рефлексії для анотацій. Інтерфейс визначає 4 методи:

- `<T extends Annotation> getAnnotation(Class<T> annotationClass)` – повертає анотацію одного конкретного заданого параметром типу для елементу з-під якого даний метод був викликаний;
- `Annotation[] getAnnotations()` – повертає всі анотації для елементу з-під якого даний метод був викликаний;
- `Annotation[] getDeclaredAnnotations()` – повертає всі неуспадковані анотації, що стосуються елементу з-під якого даний метод був викликаний;
- `boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)` – повертає істину, якщо задана анотація присутня у елементі, з-під якого даний метод був викликаний; інакше повертає хибу.

Анотація-маркер

Анотація-маркер – це спеціальний вид анотацій, що не містять членів. Єдине призначення анотації-маркера – це помітити (промаркувати) оголошення. На відміну від інших анотація при створені анотації-маркера круглі дужки після імені маркера можна не ставити, хоча їх наявність не буде помилкою . Наприклад:

```
@Retention(RetentionPolicy.RUNTIME)
@interface MarkAnno{}

@MarkAnno  // круглі дужки можна не ставити
public static void someMethod() {}
```

Для того, щоб перевірити чи елемент є промаркований можна використати метод `isAnnotationPresent`.

Одноелементні анотації

Одноелементна анотація – це анотація, що містить один елемент. На відміну від інших анотацій вона допускає скорочену форму вказування значення даному елементу. Дана форма передбачає, що можна не вказувати назву елементу, якому присвоюється значення, а просто вказати саме значення, наприклад:

```
@Retention(RetentionPolicy.RUNTIME)
@interface MarkAnno{int value();}

@MarkAnno(15) // назва елемента не вказується
public static void someMethod() {}
```

Однак, для того щоб використати дану форму, елемент, якому буде присвоюватися значення має обов’язково мати назву `value`.

Синтаксис одноелементних анотацій можна застосовувати і для інших анотацій, якщо вони задовольняють наступним вимогам: 1) елемент до якого буде застосовуватися синтаксис одноелементних анотацій має мати назву `value`; 2) всі інші елементи мають мати значення за замовчуванням. Наприклад:

```
@Retention(RetentionPolicy.RUNTIME)
@interface MarkAnno{int value(); boolean active() default false;}

@MarkAnno(15) // назва елемента не вказується
public static void someMethod() {}
```

Вбудовані анотації

У Java визначено багато вбудованих анотацій. Більшість з них є спеціалізовані, але 8 з них мають загальне призначення. Чотири з них імпортуються з пакету `java.lang`: `@FunctionalInterface`, `@Override`, `@Deprecated`, `@SafeVarargs` і `@SuppressWarnings`. Чотири інших: `@Retention`, `@Documented`, `@Target` і `@Inherited` – включені в пакет `java.lang.annotation`.

Анотація `@Override` – це анотація-маркер, яка може застосовуватися лише в методах. Метод анотований як `@Override` має перевизначати метод суперкласу. Якщо він цього не робить, то генерується помилка компіляції. Вона використовується для гарантування того, що метод суперкласу буде дійсно перевизначений, а не просто перевантажений.

Анотація `@Deprecated` – це анотація-маркер, яка вказує на те, що оголошення є застарілим і має бути замінене новою формою.

Анотація `@SafeVarargs` – це анотація-маркер, яка застосовується до методів і конструкторів. Вона вказує на те, що ніякі небезпечні дії, що пов'язані з параметром зі змінною кількістю аргументів, недопустимі. Вона використовується для подавлення невідмічених попереджень в решті безпечного коду відносно нессилочних типів (по великому рахунку параметричних типів) з змінною кількістю аргументів і параметричним створенням екземпляру класу. Застосовується лише до методів і конструкторів зі змінною кількістю аргументів, які оголошені як `static` або `final`.

Анотація `@SuppressWarnings` – ця анотація вказує на те, що одне чи більше попереджень, які можуть бути видані компілятором, слід подавити. Попередження, які слід подавити, задаються іменами в рядковій формі. Ця анотація може бути застосована до оголошення будь-якого типу.

Анотація `@Retention` – застосовується лише як анотація до інших анотацій. Визначає політику утримання анотацій.

Анотація `@Documented` – це маркер-інтерфейс, який повідомляє інструменту, що анотація має бути документована. Застосовується лише як анотація до оголошення анотації.

Анотація `@Target` – задає типи оголошень до яких може бути застосована анотація. Застосовується лише як анотація до інших анотацій. Дана анотація приймає один елемент, який має бути константою з перелічення `ElementType`. Цей аргумент задає типи оголошень, до яких може бути застосована анотація. Ці константи описані нижче.

Константа	Анотація може бути застосована до
ANNOTATION_TYPE	Іншої анотації
CONSTRUCTOR	Конструктора

FIELD	Поля
LOCAL_VARIABLE	Локальної змінної
METHOD	Методу
PACKAGE	Пакету
PARAMETER	Параметру
TYPE	Класу, інтерфейсу або типу-переліченню

Можна задати одне або кілька значень в анотації `@Target`. Щоб задати множину значень, слід помістити їх в середину обмеженого фігурними дужками списку, наприклад:

```
@Target ( {ElementType.FIELD, ElementType.METHOD} )
```

Анотація `@Inherited` – це анотація-маркер, яка дозволяє анотації суперкласу бути успадкованою в підкласі. Вона стосується лише тих анотацій, що будуть використовуватися в оголошеннях класів. Коли відбувається запит до анотації у підкласі і у ньому її нема, то перевіряється суперклас. Якщо шукана анотація в суперкласі присутня і вона промаркована як `@Inherited`, то вона буде повернута як результат запиту.

У Java 8 з'явилася ще одна корисна анотація – `@FunctionalInterface`, яка вказує компілятору, що анотований інтерфейс є функціональним, тобто він містить лише єдиний абстрактний метод. Такий інтерфейс також буде позначений утилітою Javadoc у документації як функціональний.

Висновок

У лекції розглянуто особливості оголошення та використання анотацій, політику утримування анотацій, одержання анотацій під час виконання програми засобами рефлексії, а також вбудовані анотації.

10. Виключення

План

1. Означення виключення.
2. Ієрархія класів виключень.
3. Створення власних класів виключень.
4. Оголошення контрольованих виключень.
5. Генерація контрольованих виключень.
6. Перехоплення виключень.
7. Мультиобробник виключень і фінальна повторна передача.

Означення виключення

Виключення – це механізм мови Java, що забезпечує негайну передачу керування блоку коду опрацювання критичних помилок при їх виникненні уникаючи процесу розкручування стеку. Основне призначення цього блоку коду є:

- повернутися у безпечний стан і дозволити користувачу продовжити виконання програми, або
- дати користувачу можливість коректно зберегти результати своєї роботи після чого акуратно завершити виконання програми.

Хоч виключення є потужним механізмом по обробці помилок, проте його робота вимагає набагато більше ресурсів, ніж проста перевірка на помилки, що реалізується, наприклад, за допомогою оператора `if` і поверненням коду помилки з методу. Тому заміна звичайної перевірки на помилки генерацією виключної ситуації є невиправданою. Проте бувають ситуації, коли розрізнити повернуті з методу ознаку помилки виконання методу і цілком коректний результат його виконання є неможливим. У такому випадку без застосування виключень не обійтися. Розглянемо основні помилки для усунення яких генерація виключних ситуацій є оправданою:

- *помилки введення*, наприклад, введення назви неіснуючого файлу або Інтернет адреси з подальшим зверненням до цих ресурсів, що призводить до генерації помилки системним програмним забезпеченням;
- *збої обладнання;*
- *помилки пов'язані з фізичними обмеженнями комп'ютерної системи*, наприклад, заповнення оперативної пам'яті або жорсткого диску;
- *помилки програмування*, наприклад, некоректна робота методу, читання елементів порожнього стеку, вихід за межі масиву тощо.

Для реалізації виключень застосовуються 5 ключових слів: `throw`, `throws`, `try`, `catch` і `finally` та особлива ієрархія класів, що описують виключні ситуації.

Ієрархія класів виключень

Всі виключення в мові Java поділяються на *контрольовані* і *неконтрольовані* та спадкуються від суперкласу `Throwable`. Безпосередньо від цього суперкласу спадкуються 2 класи `Error` і `Exception` (див. рис. 10.1).

Ієрархія класів, що спадкує клас `Error`, описує внутрішні помилки і ситуації, що пов'язані з браком ресурсів у системі підтримки виконання програм. Жоден об'єкт цього типу самостійно згенерувати неможна. При виникненні внутрішньої помилки можна лише відобразити повідомлення користувачу та спробувати коректно завершити виконання програми. Такі помилки є нечастими.

Ієрархія класів, що спадкує клас `Exception` поділяється на клас `RuntimeException` та інші. Виключення типу `RuntimeException` виникають внаслідок помилок програмування. Всі інші помилки є наслідком непередбачених подій, що виникають під час виконання коректної програми, наприклад, помилок вводу/виводу.

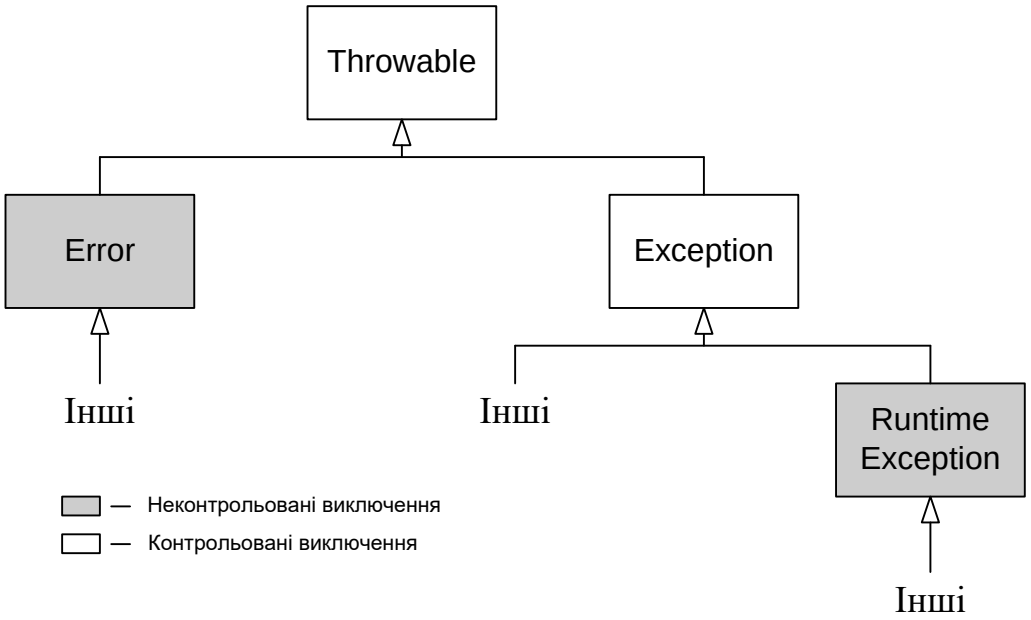

Рис. 10.1. Ієрархія класів виключень.

Класи, що спадкуються від `Error` та `RuntimeException`, відносяться до неконтрольованих виключень. Всі інші класи відносяться до контрольованих виключень. Лише контрольовані виключення можуть бути згенеровані програмістом у коді програми явно за допомогою ключового слова `throw`. Для всіх контрольованих виключень компілятор перевіряє наявність відповідних обробників.

Створення власних класів виключень

Як правило, власні класи контрольованих виключень використовуються для конкретизації виключних ситуацій, що генеруються стандартними класами контрольованих виключень, з метою їх точнішого опрацювання. Для створення власного класу контрольованих виключень необхідно обов'язково успадкувати один з існуючих класів контрольованих виключень та розширити його новою функціональністю. Найчастіше власні класи оснащують конструктором по замовчуванню та конструктором, що приймає детальний опис ситуації, яка призвела до генерації виключення. Для відображення опису помилкової ситуації можна використати метод `toString()` класу `Throwable`. Для цього необхідно викликати відповідний конструктор класу, що розширяється. Після цього створений клас можна застосовувати для генерації виключень.

Приклад власного класу виключень:

```
class FileFormatException extends IOException
{
      public FileFormatException()
      {}

      public FileFormatException(String message)
      {

            /*
```

```
                Даний виклик конструктора суперкласу дозволяє
                використовувати метод toString() класу Throwable
                */
                super(message);
        }
    }
```

Оголошення контрольованих виключень

Виключення можуть генеруватися лише методами. Якщо метод може генерувати виключення певного класу, то назву цього класу слід вказати в заголовку методу після ключового слова `throws`. Якщо метод може генерувати кілька видів виключень, то всі вони перелічуються через кому.

Приклад оголошення методу, що може генерувати виключення:

```
public int loadData(String fName) throws EOFException, MalformedURLException
{
    …
}
```

Слід зауважити, що оголошення всіх можливих виключень, які може генерувати метод, є поганим стилем програмування. Оголошувати слід лише всі контрольовані виключення. Якщо цього не зробити, то компілятор видасть повідомлення про помилку. Якщо метод оголошує, що він може генерувати виключення певного класу, то він може також генерувати виключення і його підкласів. Наприклад, оголошення методу `int loadData(String fName)` можна переписати наступним чином, не конкретизуючи, яке саме виключення похідне від класу `IOException` ми генеруватимемо:

```
public int loadData(String fName) throws IOException, MalformedURLException
{
    …
}
```

Проте зауважимо, що метод підкласу не може генерувати більш загальне контрольоване виключення, ніж метод суперкласу, що перевизначається. Виключення можуть лише конкретизуватися або не виникати взагалі. Зокрема, якщо метод суперкласу не генерує контрольованих виключень взагалі, то і метод підкласу не може їх генерувати. Проте в цьому випадку виключення можуть бути згенеровані і перехоплені в середині такого методу.

Генерація контрольованих виключень

Генерація контрольованих виключень відбувається за допомогою ключового слова `throw` після якого необхідно вказати об’єкт класу виключення який і є власне виключенням, що генерує метод. Це можна зробити двома шляхами, використовуючи іменовані або анонімні об’єкти:

```
1. throw new IOException();
```

```
2. IOException ex = new IOException();
   throw ex;
```

Деякі класи контрольованих виключень мають конструктори, що приймають рядок з описом причини виникнення виключення. Такі конструктори корисно застосовувати для полегшення пошуку місця виникнення помилки під час виконання програми.

Перехоплення виключень

Перехоплення виключень здійснюється з метою коректного опрацювання критичних помилок повідомлення про які були згенеровані у процесі виконання програми. Якщо виключення виникає і ніде не перехоплюється, то програма припиняє свою роботу і виводить на консоль повідомлення про тип виключення та вміст стеку (stack trace). Графічні програми (аплети і прикладні програми) виводять те ж саме повідомлення, повертаючись після цього у цикл обробки користувацького інтерфейсу.

Перехоплення виключень відбувається за допомогою блоків `try/catch/finally`, що мають такий синтаксис:

```
try
{
      // етап 1
      Код, що може згенерувати виключення
      // етап 2
}
catch (тип 1 виключення)
{
      // етап 3
      Опрацювання виключення типу 1
      // етап 4
}
catch (тип 2 виключення)
{
      Опрацювання виключення типу 2
}
....
catch (тип N виключення)
{
      Опрацювання виключення типу N
}
finally
{
      // етап 5
      Код, що виконується за будь-яких обставин
}
// етап 6
```

Блоки `catch` і `finally` у цій конструкції є необов'язковими.

Розглянемо можливі варіанти роботи цієї конструкції у різних ситуаціях.

1. Якщо код у блоці `try` не генерує ніяких виключень, то програма спочатку повністю виконає блок `try`, а потім блок `finally`. Тобто виконаються етапи 1, 2, 5 і 6.

2. Якщо код у блоці `try` згенерував виключення, то подальше виконання коду в цьому блоці припиняється і відбувається пошук блоку `catch` тип у заголовку якого співпадає з типом виключення після чого виконується блок `finally`. Таким чином виконуються етапи 1, 3, 4, 5 і 6.

Якщо виключення генерується у блоці `catch`, то після виконання блоку `finally` керування передається викликаючому методу і виконуються лише етапи 1, 3 та 5.

3. Якщо виключення не опрацьовується у жодному з блоків `catch`, то виконується блок `finally` і керування передається викликаючому методу. Таким чином виконуються етапи 1 і 6.

4. Якщо ж блоки `finally` і `catch` відсутні, то керування передається викликаючому методу.

Мова Java дає розробнику вибір або самому перехопити і опрацювати виключення у методі, або делегувати це право іншому методу. Як правило, слід перехоплювати лише ті виключення, які ви самі можете опрацювати, або, які були створені вами. Решту виключень слід передавати далі по ланцюгу викликаних методів.

Для того щоб передати виключення далі по ланцюгу викликаних методів у заголовку методу, що розробляється, за допомогою ключового слова `throws` слід вказати типи виключень, що генерує цей метод але не опрацьовує, передаючи це право викликаючому методу. Якщо ж метод містить код опрацювання виключень, то вказувати типи виключень, що перехоплюються у методі, в заголовку методу не треба.

Якщо ви перевизначаєте метод суперкласу, що не генерує виключень, то ви зобов'язані перехоплювати всі контрольовані виключення, що генеруються в цьому методі.

Виключення можна генерувати у блоці `catch` створюючи цим самим ланцюг виключень. Зазвичай це робиться тоді, коли розробнику необхідно змінити тип виключення на тип виключення вищого рівня. Це може бути корисним, наприклад, тоді коли розробляється підсистема і розробнику, що використовуватиме її в подальшому слід знати, що помилка відбулася саме у цій підсистемі, а тип помилки значення не має. При цьому, у виключені вищого рівня за допомогою методу `setCause(Throwable)` можна розмістити первинне виключення як причину виникнення помилки. Одержати первинне виключення з виключення вищого рівня можна за допомогою методу `Throwable getCause()`. Для одержання повідомлення, що розміщено у виключені, слід використати метод `getMessage()`. Фактичний тип об'єкту виключення можна отримати за допомогою рефлексії: `назваОб'єкту.getClass().getName()`.

Блоки `try/catch` і `try/finally` рекомендовано розділяти розміщуючи у внутрішньому блоці `try/finally` код для звільнення ресурсів, а у зовнішньому `try/catch` код для опрацювання помилок. Це дозволить коректно опрацьовувати

ситуацію, коли виключна ситуація генерується у блоці `finally` і не опрацьовується в подальшому ніяким з блоків `catch`.

Розміщення оператора `return` у блоці `finally` може призвести до неочікуваних результатів, тому цього слід уникати.

Мультиобробник виключень і фінальна повторна передача

У JDK 7 з'явилися 3 нові типи виключень – це: оператор `try-з-ресурсами`, який дозволяє автоматизувати процес вивільнення ресурсу, наприклад, закривання файлу; *мультиобробник* і *фінальна (точніша) повторна передача*. Розглянемо останні 2 механізми (оператор `try-з-ресурсами` розглядається у темі присвяченій файлам).

Часто бувають ситуації, коли опрацювання кількох виключень проходить за однаковим алгоритмом. В такому випадку мультиобробник виключення дозволяє опрацьовувати кілька виключень в одному операторі `catch`. Для цього параметри оператора `catch` слід розділити оператором побітового «або» (`|`). При використанні мультиобробника кожен його параметр стає неявно фінальним, тобто йому не можна присвоювати нове значення. Приклад:

```
…
try {
      int result = a/b;
      arr[i] = i;
}
catch (ArithmeticException | ArrayIndexOutOfBoundsException ex)
{ System.out.println(" Виключення " + ex); }
```

У випадку, коли `b = 0` або `i` виходить за межі масиву, будуть згенеровані виключення `ArithmeticException` або `ArrayIndexOutOfBoundsException` відповідно, які будуть оброблені в одному блоці `catch`. Параметр `ex` є неявно фінальним.

Механізм точнішої повторної передачі пов'язаний з точнішою обробкою виключень у JDK 7, ніж у попередніх версіях, що дає змогу точніше вказувати після ключового слова `throws` в оголошені методу які виключення даний метод може генерувати. Механізм точнішої повторної передачі активується компілятором якщо компілятор шляхом аналізу коду впевниться, що:

- виключення може бути згенероване у блоці `try`;
- немає вищестоящих блоків `catch`, що можуть обробити це виключення;
- повторно згенероване виключення є підтипом або супертипом параметру;
- параметр не змінюється у блоці `catch`.

Якщо ці умови не задовольняються, то механізм точнішої повторної передачі не застосовується компілятором. Приклад:

```
// повторна передача виключень у версіях до JDK 7
static class FirstException extends Exception { }
static class SecondException extends Exception { }

public void rethrowException(String exceptionName) throws Exception {
     try {
          if (exceptionName.equals("First")) {
               throw new FirstException();
          } else {
               throw new SecondException();
          }
     } catch (Exception e) {
          throw e; // повторна генерація виключення
     }
}

// повторна передача виключень у JDK 7
static class FirstException extends Exception { }
static class SecondException extends Exception { }

public    void    rethrowException(String    exceptionName)    throws
FirstException, SecondException{
     try {
          if (exceptionName.equals("First")) {
               throw new FirstException();
          } else {
               throw new SecondException();
          }
     } catch (Exception e) {
          throw e; // повторна генерація виключення
     }
}
```

Висновок

У лекції розглянуто ієрархію класів виключень, показано принципи створення власних і використання існуючих класів виключень.

11. Файли

План

1. Ієрархія класів для роботи з файлами.
2. Принципи роботи з файловими потоками.
3. Читання з текстових потоків. Запис у текстові потоки.
4. Читання і запис двійкових даних.
5. Файли з довільним доступом.
6. Автоматичне керування ресурсами.
7. Засоби мови Java для роботи з файловою системою.

Ієрархія класів для роботи з файлами

Бібліотека класів мови Java має більше 60 класів для роботи з потоками. Потоками у мові Java називаються об'єкти з якими можна здійснювати обмін даними. Цими об'єктами найчастіше є файли, проте ними можуть бути стандартні пристрої вводу/виводу, блоки пам'яті і мережеві підключення тощо. Класи по роботі з потоками об'єднані у кілька ієрархій, що призначені для роботи з різними видами даних, або забезпечувати додаткову корисну функціональність, наприклад, підтримку ZIP архівів.

Класи, що спадкуються від абстрактних класів `InputStream` і `OutputStream` призначені для здійснення байтового обміну інформацією. Підтримка мовою Java одиниць Unicode, де кожна одиниця має кілька байт, зумовлює необхідність у іншій ієрархії класів, що спадкується від абстрактних класів `Reader` і `Writer`. Ці класи дозволяють виконувати операції читання/запису не байтних даних, а двобайтних одиниць Unicode.

Принцип здійснення читання/запису даних нічим не відрізняється від аналогічного принципу у інших мовах програмування. Все починається з створення потоку на запис або читання після чого викликаються методи, що здійснюють обмін інформацією. Після завершення обміну даними потоки необхідно закрити щоб звільнити ресурси.

Абстрактний клас `InputStream` визначає абстрактний метод `abstract int read()` та його перевизначення для читання байтової інформації. Цей метод має повертати значення прочитаного байту в діапазоні [0; 255] або -1, якщо потік закінчився. Інші методи класу:

- `skip` – пропускає вказану кількість байт у потоці;
- `available` – визначає кількість байт, що доступні без блокування;
- `close` – закриває потік;
- `mark` – встановлює на поточні позиції у потоці вхідних даних маркер;
- `reset` – повертається до позиції у потоці, що позначена маркером;
- `markSupported` – перевіряє чи потік підтримує маркер.

Абстрактний клас `OutputStream` визначає абстрактний метод `abstract void write(int n)` та його перевизначення для запису байтової інформації. Цей метод має приймати значення, яке необхідно записати у потік. Інші методи класу:

- `close` – закриває потік;
- `flush` – записує у потік дані, що знаходяться у буфері.

Абстрактний клас `Reader` визначає абстрактний метод `abstract int read()` та його перевизначення для читання одиниці коду Unicode. Цей метод має повертати значення прочитаного байту або -1, якщо потік закінчився. Інші методи класу:

- `skip` – пропускає вказану кількість байт у потоці;
- `available` – визначає кількість байт, що доступні без блокування;
- `close` – закриває потік;

- `mark` – встановлює на поточній позиції у потоці вхідних даних маркер;
- `reset` – повертається до позиції у потоці, що позначена маркером;
- `markSupported` – перевіряє чи потік підтримує маркер.

Абстрактний клас `Writer` визначає абстрактний метод `abstract void write(int n)` та його перевизначення для запису байтової інформації. Цей метод має приймати значення, яке необхідно записати у потік. Інші методи класу:

- `close` – закриває потік;
- `flush` – записує у потік дані, що знаходяться у буфері.

Методи `read` та `write` є синхронними (блокуючими). Тобто, їх виконання призупиняє виконання програми на час роботи методів.

Починаючи з версії 5.0 у мові Java для розглянутих класів доступними стали 4 інтерфейси: `Closeable`, `Flushable`, `Appendable` і `Readable`. Дані інтерфейси визначають методи, що закривають потік (`Closeable`), примусово записують дані з буфера в потік (`Flushable`), читають дані у буфер типу `CharBuffer` (`Readable`), приєднують дані до потоку (`Appendable`).

Клас `InputStream` реалізує інтерфейс: `Closeable`;
Клас `OutputStream` реалізує інтерфейси: `Closeable` і `Flushable`.
Клас `Reader` реалізує інтерфейси: `Closeable` і `Readable`.
Клас `Writer` реалізує інтерфейси `Closeable`, `Flushable` і `Appendable`.

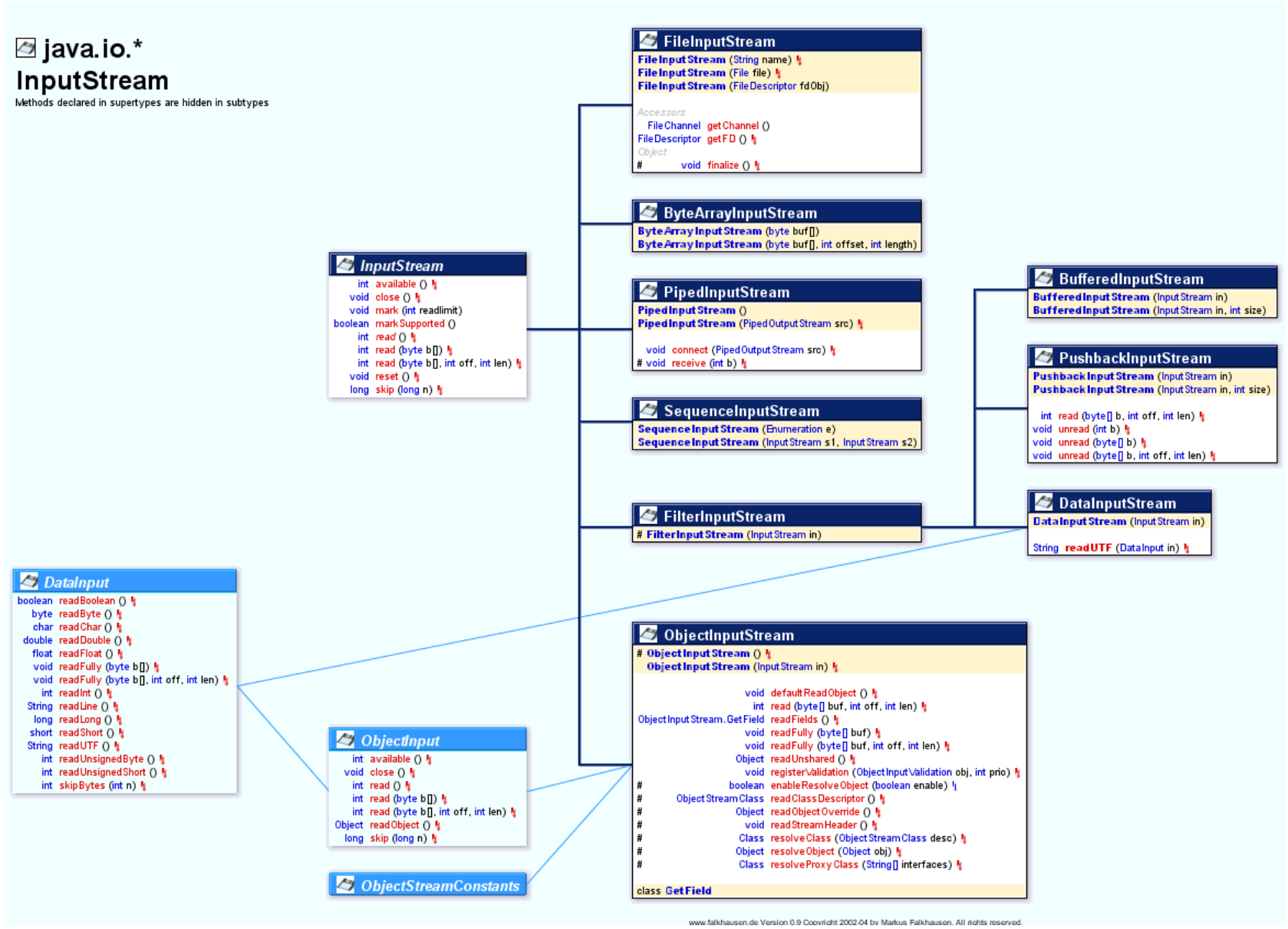

Рис. 11.1. Ієрархія класів Java.io.InputStream.

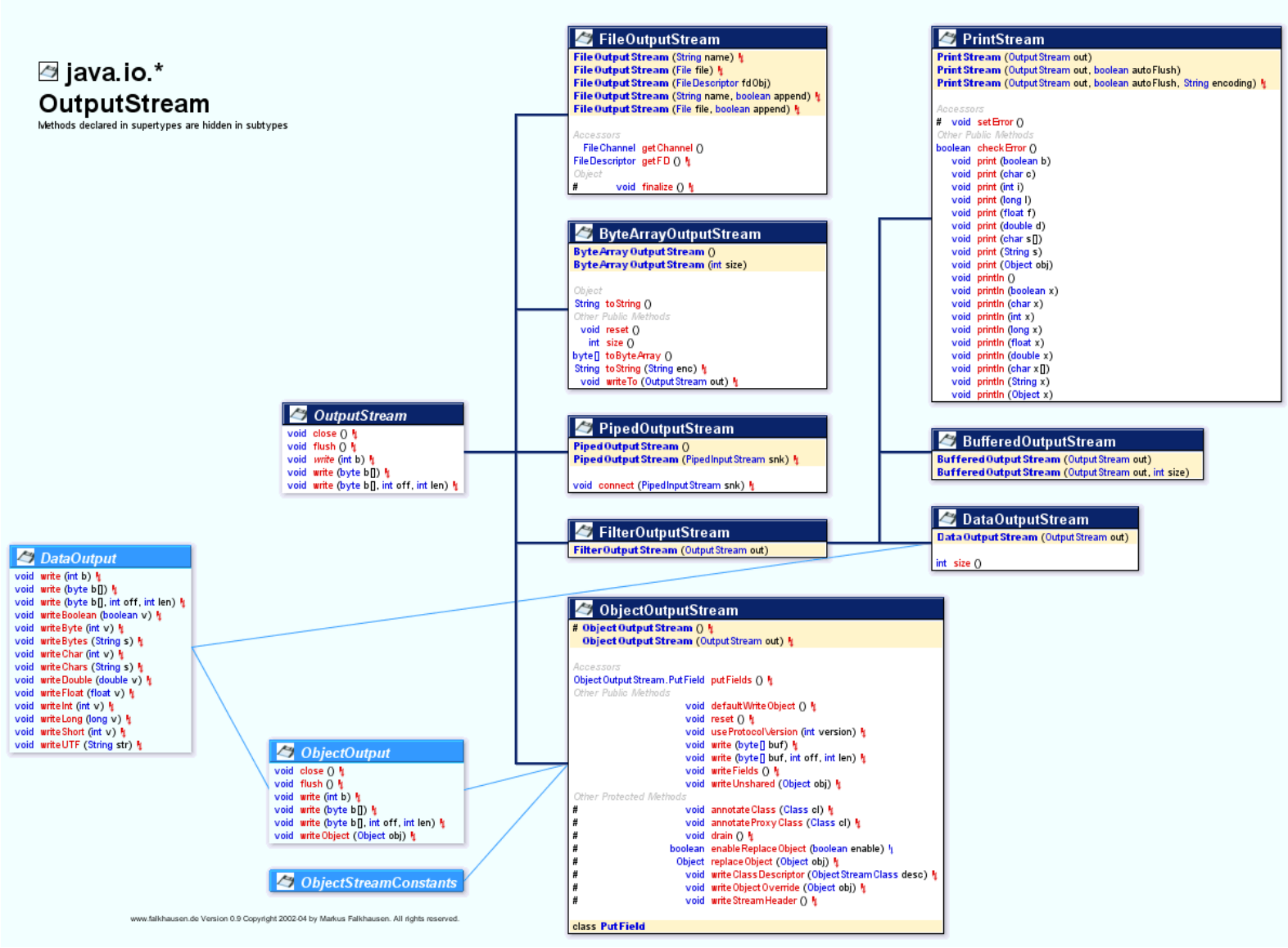

Рис. 11.2. Ієрархія класів Java.io.OutputStream.

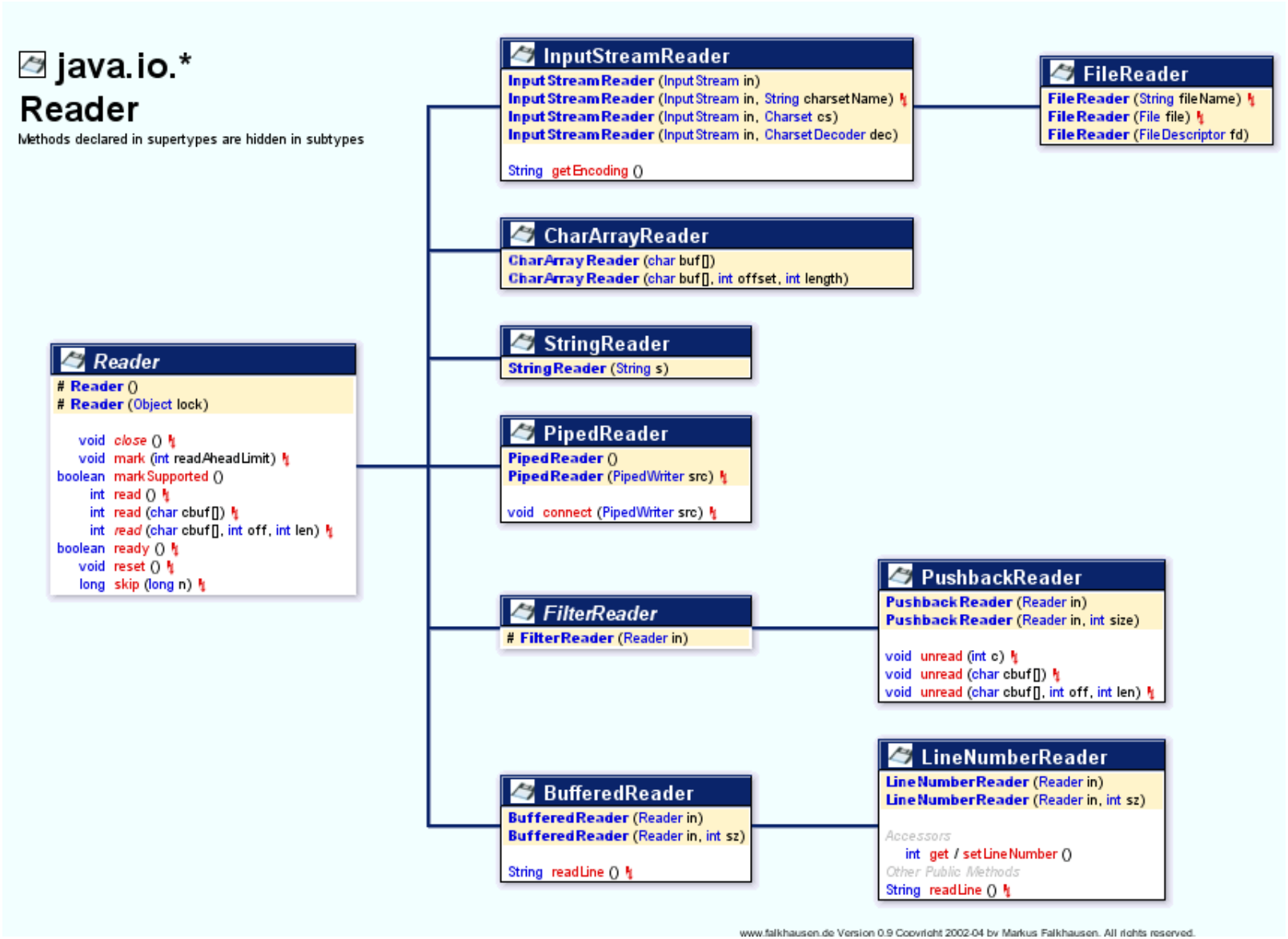

Рис. 11.3. Ієрархія класів Java.io.Reader.

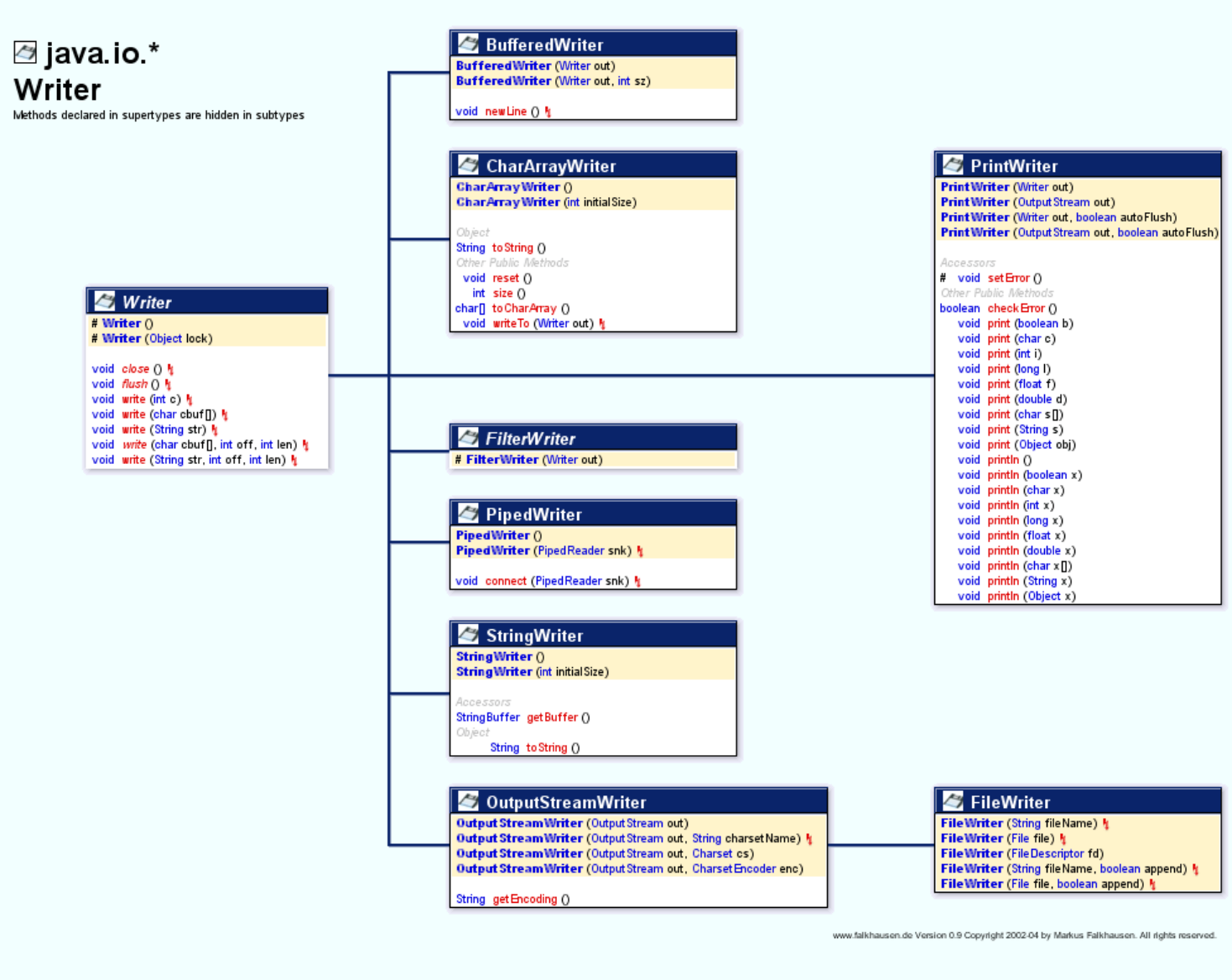

Рис. 11.4. Ієрархія класів Java.io.Writer.

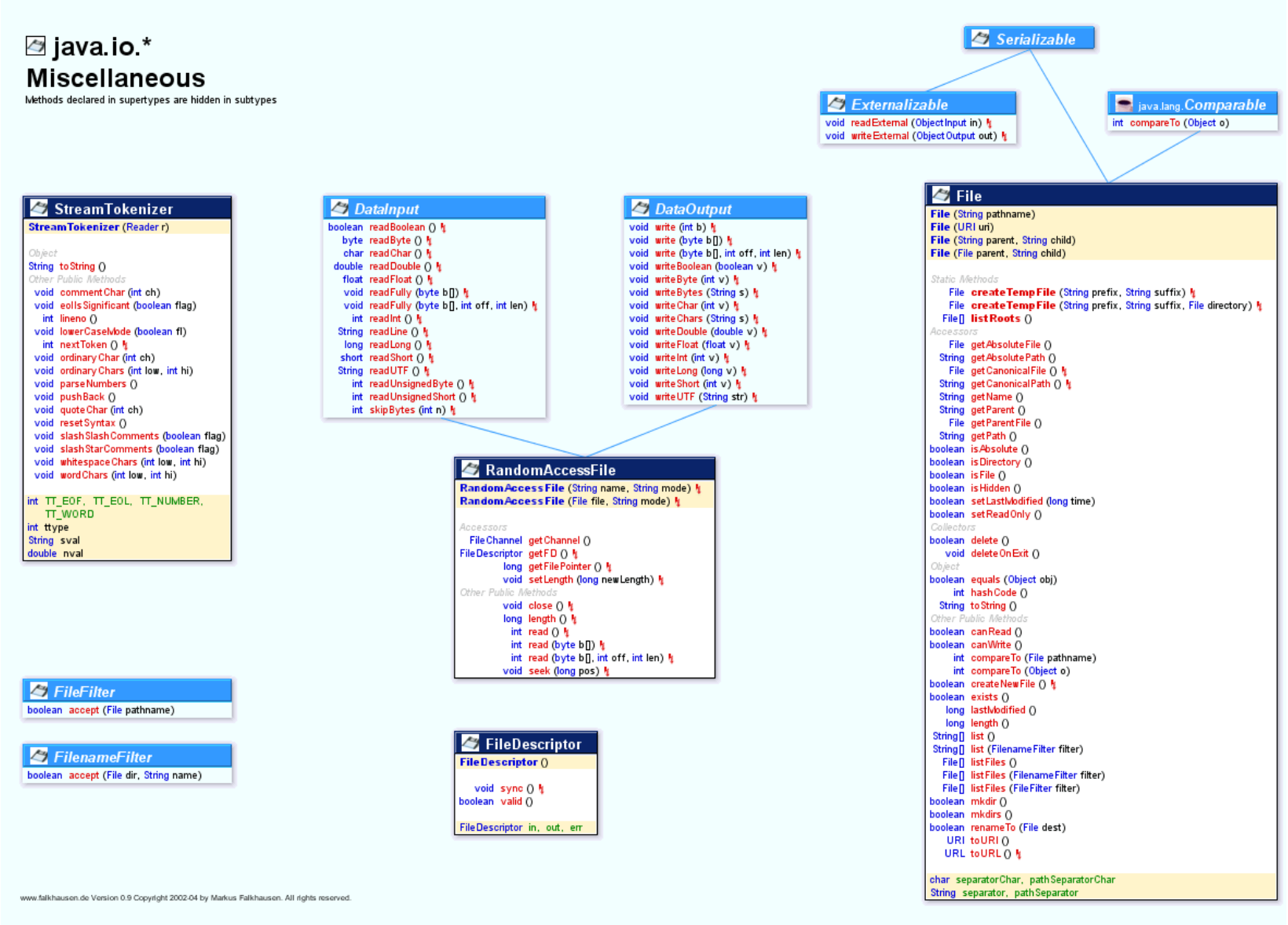

Рис. 11.5. Ієрархія допоміжних класів Java.io.*.

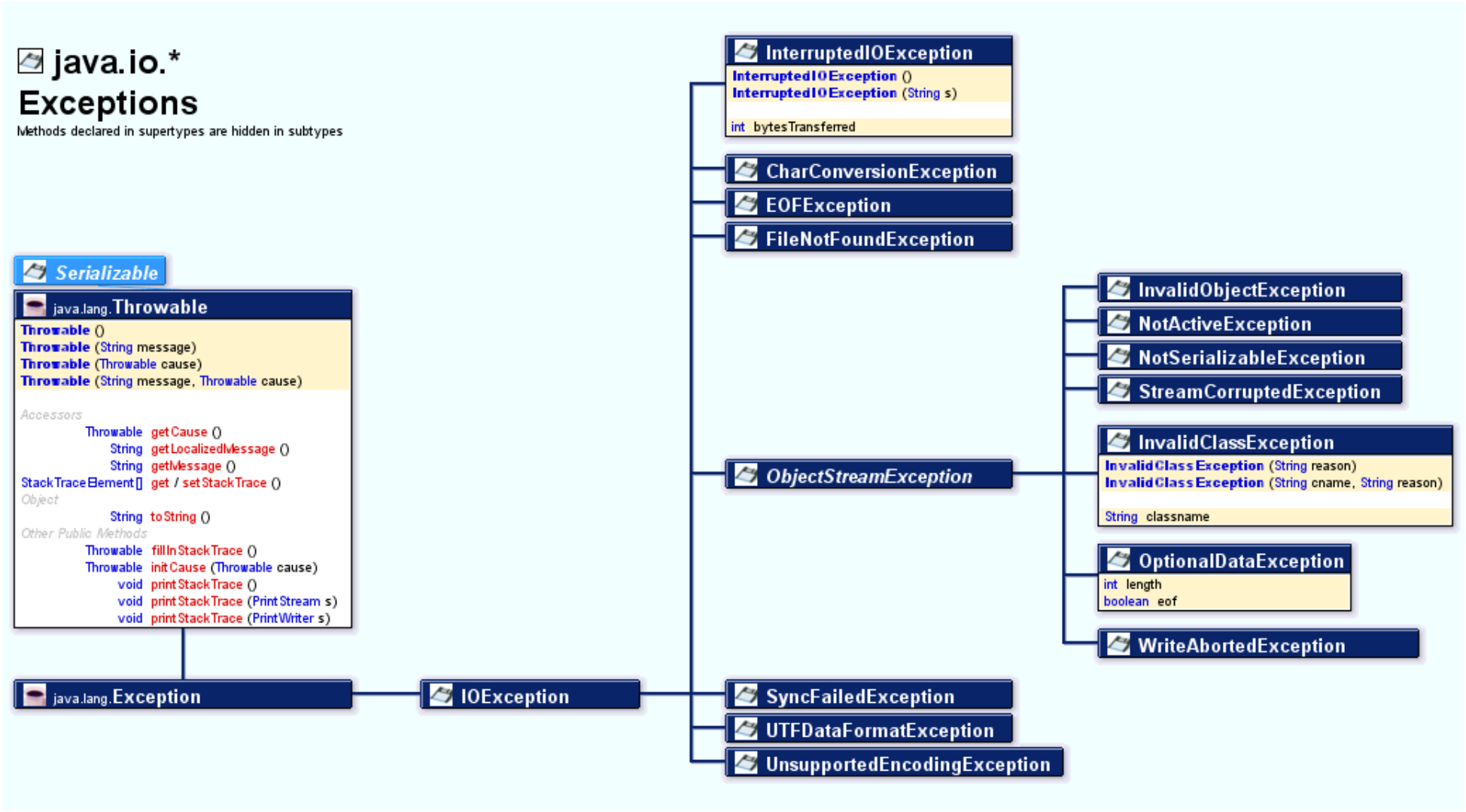

Рис. 11.6. Ієрархія класів виключень вводу/виводу.

Принципи роботи з файловими потоками

Для створення файлових потоків і роботи з ними у Java є 2 класи, що успадковані від `InputStream` і `OutputStream` це - `FileInputStream` і `FileOutputStream`. Як і їх суперкласи вони мають методи лише для байтового небуферизованого блокуючого відповідно читання і запису даних та керуванням потоками. На відміну від, наприклад, мови програмування С, де для виконання усіх можливих операцій з файлами необхідно мати один вказівник на FILE, у мові Java реалізовано інший набагато складніший і гнучкіший підхід, який дозволяє формувати такі властивості потоку, які найкраще відповідають потребам рішення конкретної задачі. Так у Java розділено окремі функціональні можливості потоків на різні класи. Компонуючи ці класи між собою і досягається необхідна кінцева функціональність потоку. Так одні класи, як `FileInputStream`, забезпечують елементарний доступ до файлів, інші, як `PrintWriter`, надають додаткової функціональності по високорівневій обробці даних, що пишуться у файл. Ще інші, наприклад, `BufferedInputStream` забезпечують буферизацію. Таким чином, наприклад, щоб отримати буферизований файловий потік для читання інформації у форматі примітивних типів (`char`, `int`, `double`,…) слід створити потік з одночасним сумісним використанням функціональності класів `FileInputStream`, `BufferedInputStream` і `DataInputStream` (див. паттерн Декоратор). Для цього слід здійснити наступний виклик:

```
DataInputStream din = new DataInputStream(
      new BufferedInputStream(
            new FileInputStream)));
```

Класи типу `BufferedInputStream`, `DataInputStream`, `PushbackInputStream` (дозволяє читати з потоку дані і повертати їх назад у потік) успадковані від класу `FilterInputStream`. Вони виступають так званими фільтрами, що своїм комбінуванням забезпечують додаткову лише необхідну функціональність при читанні даних з файлу. Аналогічний підхід застосовано і при реалізації класів для обробки текстових даних, що успадковані від `Reader` і `Writer`.

Особливості читання і запису даних у текстових потоках

Для читання з текстового потоку у мові Java є клас `InputStreamReader`, що призначений для читання байтів з потоку і перекодовування їх у символи відповідно до заданого кодування. Для зворотного процесу переведення символів з певним кодуванням у масив байтів і запису цього масиву у потік використовується клас `OutputStreamWriter`. Конструктори цих класів можуть приймати параметрами посилання на потік і кодування, що в ньому використовується. Приклад відкривання текстового файлу у форматі ISO8859-5 на читання:

```
InputStreamReader in = new InputStreamReader(
      new FileInputStream ("file.txt"), "ISO8859-5");
```

Приклад одержання доступу до стандартного потоку вводу:

```
InputStreamReader in = new InputStreamReader(System.in);
```

Якщо потоками є текстові файли і вони використовують стандартне у системі кодування, то можна використати класи: `FileWriter` і `FileReader`. Дані класи на відміну від параметризованого класу `InputStreamReader` здатні самостійно відкривати файл на читання без посередництва `FileInputStream`, хоча також можуть використовувати посилання на потоки. Приклад відкривання текстового файлу на читання:

```
FileReader in = new FileReader(“file.txt”);
```

У Java 5.0 з’явився ще один клас, що називається `Scanner`, який найкраще підходить для читання тестових потоків. На відміну від `InputStreamReader` і `FileReader`, що дозволяють лише читати текст, він має велику кількість методів, які здатні читати як рядки, так і окремі примітивні типи з подальшим їх перекодуванням до цих типів, робити шаблонний аналіз текстового потоку, здатний працювати без потоку даних та ще багато іншого. Приклад читання даних за допомогою класу `Scanner` з стандартного потоку вводу:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

Приклад читання даних за допомогою класу `Scanner` з текстового файлу:

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
   long aLong = sc.nextLong();
}
```

Приклад аналізу рядка тексту одержаного з рядка за допомогою `Scanner`:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

Результат виконання коду:

```
1
2
red
blue
```

Для буферизованого запису у текстовий потік найкраще використовувати клас PrintWriter. Цей клас має методи для виводу рядків і чисел у текстовому форматі: print, println, printf, - принцип роботи яких співпадає з аналогічними методами Systen.out.

Приклад використання класу PrintWriter:

```
PrintWriter out = new PrintWriter (“file.txt”);
out.print(“Hello ”);
out.print(1070);
out.println(“! I’m World.”);
out.close();
```

Читання і запис двійкових даних

Читання двійкових даних примітивних типів з потоків здійснюється за допомогою класів, що реалізують інтерфейс DataInput, наприклад класом DataInputStream. Інтерфейс DataInput визначає такі методи для читання двійкових даних:

- readByte;
- readInt;
- readShort;
- readLong;
- readFloat;
- readDouble;
- readChar;
- readBoolean;
- readUTF.

Приклад читання двійкових даних з файлу:

```
DataInputStream in = new DataInputStream(new FileInputStream
 (“binarydata.dat”));
int n = in.readInt();
```

Запис двійкових даних примітивних типів у потоки здійснюється за допомогою класів, що реалізують інтерфейс DataOutput, наприклад класом DataOutputStream. Інтерфейс DataOutput визначає такі методи для запису двійкових даних:

- writeByte;
- writeInt;
- writeShort;
- writeLong;
- writeFloat;

- writeDouble;
- writeChar;
- writeChars;
- writeBoolean;
- writeUTF.

Метод writeUTF здійснює запис рядка у модифікованому машиннонезалежному форматі UTF-8, який крім Java мало хто використовує. Тому для потоків, які не призначені суто для Java, слід використовувати метод writeChars.

Приклад запису двійкових даних у файл:

```
DataOutputStream out = new DataOutputStream(new FileOutputStream
 (“binarydata.dat”));
out.writeInt(5);
```

Файли з довільним доступом

Керування файлами з можливістю довільного доступу до них здійснюється за допомогою класу RandomAccessFile. Відкривання файлу в режимі читання або читання/запису здійснюється за допомогою конструктора, що приймає 2 параметри – посилання на файл (File file) або його адресу (String name) та режим відкривання файлу (String mode):

```
RandomAccessFile(File file, String mode);
RandomAccessFile(String name, String mode).
```

Параметр mode може приймати такі значення:

- "r" – читання;
- "rw" – читання/запис;
- "rws" – читання/запис даних з негайним синхронним записом змін у файл або метадані файлу;
- "rwd" – читання/запис даних з негайним синхронним записом змін у файл, метадані файлу не міняються одразу.

Файли, що керуються класом RandomAccessFile, оснащені вказівником на позицію наступного байту, що має читатися або записуватися. Для того, щоб перемістити даний вказівник на довільну позицію в межах файлу використовується метод void seek(long pos). Параметр long pos визначає номер байту, що має читатися або записуватися.

Щоб дізнатися поточну позицію вказівника на позицію наступного байту слід використати метод long getFilePointer().

Для визначення довжини файлу використовується метод `long length()`, а для закриття файлу – `void close()`.

Для читання і запису даних клас `RandomAccessFile` реалізує методи інтерфейсів `DataInput` і `DataOutput`, які описані у попередньому пункті.

Довільний доступ до файлу є найповільнішим способом доступу до файлів. Трішки швидшим є звичайний потік даних і набагато швидшим є буферизований потік даних.

Автоматичне керування ресурсами

Автоматичне керування ресурсами базується на удосконаленій версії оператора `try`. Воно призначене для попередження ситуацій, коли деякий ресурс, наприклад, файл, виділений, але після його використання ресурс не звільняється (закривається). Узагальнена форма використання оператора `try-з-ресурсами`:

```
try(специфікація_ресурсу) “{”
//використання ресурсу
“}”
{catch({параметр{| параметр}}) “{” {код} } “}”
[finally “{” {код} “}”]
```

Тут `специфікація_ресурсу` – це оператор, який оголошує і ініціалізує ресурс, наприклад, файловий потік. Він складається з оголошення змінної, в якому змінна ініціалізується посиланням на об’єкт (ресурс), що використовуватиметься. Ресурс, що оголошений у блоці `try` є неявно фінальним, область його видимості обмежується блоком `try` і після завершення блоку `try` ресурс автоматично звільняється. У випадку файлу це означає, що файл автоматично закривається, тобто необхідність явно викликати метод `close()` зникає.

Оператор `try-з-ресурсами` можна застосовувати лише з тими ресурсами, що реалізують інтерфейс `AutoCloseable` пакету `java.lang`, в якому визначається метод `close()`. Інтерфейс `AutoCloseable` в свою чергу спадкується інтерфейсом `Closeable` пакету `java.io`. Обидва інтерфейси реалізуються потоковими класами. Таким чином, оператор `try-з-ресурсами` може бути використаним при роботі з потоками, включаючи файлові потоки.

Оператор `try-з-ресурсами` має ще одну особливість. При виконанні оператора `try` є ймовірність того, що після виникнення виключення і подальшому виконанні блоку `finally` під час вивільнення ресурсу може бути згенероване інше виключення. При цьому, якщо використовується звичайний блок `try`, то перше виключення буде втрачене, оскільки воно буде *витіснене* другим виключенням. Натомість при використанні блоку `try-з-ресурсами` друге виключення буде *подавлене* першим. Однак воно не

втратиться, а буде добавлене в список подавлених виключень, що пов'язані з першим виключенням. Доступ до цього списку виключень можна отримати за допомогою методу `getSuppressed()`, що визначається у класі `Throwable`.

Приклад програми по копіюванню одного файлу в інший:

```
import java.io.*;

class CopyFile
{
      public static void main (String args[]) throws IOException
      {
            int i;
            try(FileInputStream fin = new FileInputStream(args[0]);
                FileOutputStream fout = new FileOutputStream(args[1]))
            {
                  do{
                        i = fin.read();
                        if (i != -1) fout.write(i);
                  } while (i != -1);
            }
            catch (IOException ex)
            {
                  System.out.println("Помилка " + ex);
            }
      }
}
```

Зверніть увагу, що в даному прикладі відкриття обох файлів відбувається у блоці `try`, а їх закриття відбудеться автоматично після завершення виконання блоку `try`.

У Java SE 9, якщо у нас є ресурс, який вже оголошений поза оператором `try-з-ресурсами` як final або фактично final, то нам не потрібно оголошувати локальну змінну. Ми можемо використовувати попередньо створену змінну в операторі `try-з-ресурсами`. Порівняємо ці 2 реалізації:

```
void exampleBeforeJava9() throws IOException{
      BufferedReader freader1 = new BufferedReader(new
      FileReader("somefile.txt"));
      try (BufferedReader freader2 = freader1) {
            System.out.println(freader2.readLine());
      }
}
```

та

```
void exampleJava9() throws IOException{
```

```
        BufferedReader freader1 = new BufferedReader(new
        FileReader("somefile.txt"));
        try(freader1) {
              System.out.println(freader1.readLine());
        }
}
```

Засоби мови Java для роботи з файловою системою

Робота з файловою системою ОС з використанням мови Java здійснюється за допомогою класу `File`. Даний клас використовується як для роботи з файлами так і для роботи з каталогами ОС. Для створення об'єкту типу `File` використовується один з кількох конструкторів, які можуть приймати:

- шлях до файлу;
- окремо шлях і назву файлу;
- об'єкт типу `File`, що є каталогом, та назву файлу;
- лише універсальний ідентифікатор ресурсів, що є об'єктом класу `URI`.

При вказуванні шляху для кросплатформенних програм слід використовувати значення статичного поля `File.separator`, що містить значення символу розділювача каталогів. Наприклад, з умови використання ОС Windows наступний запис створює об'єкт `File` для файлу з шляхом `Docs\data.txt`:

```
File fDoc = new File ("Docs"+File.separator+"data.txt").
```

Наступний запис створює об'єкт `File` для каталогу `Docs`:

```
File dDoc = new File (File.separator+"Docs").
```

При створені об'єкту класу `File` фізично на носії інформації файл чи каталог не створюються, не відкриваються і навіть можуть не існувати. Для того, щоб перевірити наявність каталогу чи файлу з заданим шляхом слід використати метод `boolean exists()`, який поверне `true`, якщо даний файл чи каталог існує, або `false` у протилежному випадку. Якщо такого файлу чи каталогу немає, то їх можна створити за допомогою методів:

- `boolean createNewFile()` – створити файл;
- `boolean mkdir()` – створити каталог.

Якщо створення пройшло успішно, то методи повернуть `true`.

Для перевірки чи об'єкт класу `File` є файлом чи каталогом є 2 методи, що повертають `true` якщо об'єкт є відповідно файлом і каталогом:

- `boolean isFile()` – перевірка чи об'єкт є файлом;
- `boolean isDirectory()` – перевірка чи об'єкт є каталогом.

Для перегляду вмісту каталогу використовуються методи:

- `String [] list()` – повертає вміст каталогу;
- `String [] list(FilenameFilter filter)` – повертає вміст каталогу, що задовольняє фільтр.

Фільтр задається класом, що має реалізовувати інтерфейс `FilenameFilter`. Цей інтерфейс містить єдиний метод `boolean accept(File dir, String name)`, де параметр `name` задає назву файлу, що перевіряється на відповідність шаблону. Якщо файл відповідає шаблону, то метод повертає `true`.

Крім цих методів у класі `File` реалізовано методи, що перевіряють та встановлюють прапорці файлів і каталогів, роблять файл тимчасовим, видаляють файли і каталоги, повертають шлях та назву файлу або каталогу, визначають розмір файлу, час останньої модифікації, перелік файлів вказаного каталогу, здійснюють перейменування, перетворюють об'єкт `File` в `URL`, показують інформацію про розділи (об'єкт `File` має бути розділом).

Висновок

У лекції розглянуто принципи роботи з файловими потоками та ієрархії відповідних класів. Показано особливості роботи з текстовими і байтовими потоками. Розглянуто засоби мови Java для роботи з файловою системою.

12. Параметризоване програмування

План

1. Вступ.
2. Визначення простого параметризованого класу.
3. Параметризовані методи.
4. Встановлення обмежень для змінних типів.
5. Взаємодія параметризованого коду та віртуальної машини.
6. Трансляція коду при звертанні до параметризованих методів і полів.
7. Тонкощі параметризованого програмування.
8. Правила спадкування параметризованих типів.
9. Підстановочні типи.

Вступ

Параметризоване програмування полягає у написанні коду, що можна багаторазово застосовувати з об'єктами різних класів. Користувачів параметризованого програмування можна поділити на 3 рівні кваліфікації:

1. ті, що користуються готовими класами;
2. ті, що користуються готовими класами і вміють виправляти помилки, що виникають при їх використанні;

3. ті, що пишуть власні параметризовані класи.

Для успішного застосування параметризованого програмування слід навчитися розуміти помилки, що генерує середовище при компіляції програми, що можуть стосуватися, наприклад, неоднозначності визначення спільного суперкласу для всіх переданих об'єктів.

Визначення простого параметризованого класу

Параметризований клас – це клас з однією або більше змінними типу.

Синтаксис оголошення параметризованого класу:

```
[public] class НазваКласу <параметризованийТип{,параметризованийТип}>
{…}
```

Іменувати змінні параметризованих типів прийнято великими літерами. У бібліотеці Java використовується літера `E` для позначення типу колекції, `K` і `V` – для типів ключа і значення таблиці, `T`, `U`, `S` та сусідні літери – для позначення довільних типів.

Приклад оголошення параметризованого класу:

```
class GenericClass<T, U>
{
      public GenericClass(T first, U second)
      {
            this.first = first;
            this.second = second;
      }
      public void setFirst(T first)
      {
            this.first = first;
      }
      public T getFirst()
      {
            return first;
      }
      ...
      private T first;
      private U second;
}
```

Тут `T` і `U` – це змінні параметризованих типів, що використовуються по всьому тілу класу для специфікації типу повернення методів, типів полів і локальних змінних. При створенні об'єкту параметризованого класу замість них підставляються реальні типи, що визначаються в трикутних дужках у місці створення об'єкту параметризованого класу.

Синтаксис створення об’єкту параметризованого класу:

```
НазваКласу < перелікТипів > змінна = new НазваКласу < перелікТипів >
(параметри);
```

Приклад створення об’єкту параметризованого класу:

```
GenericClass<String, Integer> obj = new GenericClass<String, Integer> ();
```

Параметризовані методи

Параметризовані методи визначаються в середині як звичайних класів так і параметризованих. На відміну від звичайних методів параметризовані методи мають параметризований тип, що дозволяє за їх допомогою опрацьовувати різнотипні набори даних. Реальні типи для методів, як і для класів, визначаються у місці виклику методу шляхом передачі реального типу у трикутних дужках.

Синтаксис оголошення параметризованого методу:

```
Модифікатори <параметризованийТип{,параметризованийТип}> типПовернення
назваМетоду(параметри);
```

Приклад оголошення параметризованого методу:

```
class ArrayAlg
{
      public static<T> T getMiddle(T[] a)
      {
            return a[a.length / 2];
      }
}
```

Синтаксис виклику параметризованого методу:

```
(НазваКласу|НазваОб’єкту).[<перелікТипів>] НазваМетоду(параметри);
```

У мові Java компілятор здатний самостійно визначати типи, що підставляються замість параметризованих типів, тому у трикутних дужках вказувати реальні типи не обов’язково. Проте це може призвести до помилок, якщо компілятор не зможе однозначно визначити єдиний супертип для всіх параметрів.

Приклад виклику параметризованого методу:

```
String[] names = {“Ivan”, “Ivanovych”, “Ivanov”};

String middle = ArrayAlg.<String>getMiddle(names);
```

або

```
String middle = ArrayAlg.getMiddle(names);
```

Приклад виклику параметризованого методу, що може призвести до помилок, оскільки параметри методу можна привести як до класу `Double` так і до інтерфейсу `Comparable`:

```
ArrayAlg.<String>getMiddle(3.75, 123, 0);
```

Встановлення обмежень для змінних типів

Бувають ситуації, коли клас або метод потребують накладення обмежень на змінні типів. Наприклад, може бути ситуація, коли метод у процесі роботи викликає з-під об'єкта параметризованого типу метод, що визначається у деякому інтерфейсі. У такому випадку немає ніякої гарантії, що цей метод буде реалізований у кожному класі, що передається через змінну типу. Щоб вирішити цю проблему у мові Java можна задати обмеження на множину можливих типів, що можуть бути підставлені замість параметризованого типу. Для цього після змінної типу слід використати ключове слово `extends` і вказати один суперклас, або довільну кількість інтерфейсів (через знак `&`), від яких має походити реальний тип, що підставляється замість параметризованого типу. Якщо одночасно вказуються інтерфейси і суперклас, то суперклас має стояти першим у списку типів після ключового слова `extends`.

Синтаксис оголошення параметризованого методу з обмеженнями типів:

```
Модифікатори <параметризований тип extends обмежуючийТип {& обмежуючий тип}
{, параметризований тип extends обмежуючийТип {& обмежуючий тип} }>
типПовернення назваМетоду(параметри);
```

Приклад оголошення параметризованого методу з обмеженнями типів:

```
class ArrayCmp
{
       public static <T extends Comparable & Serializable> T CopmlComp(T[] a)
       {
              ...
       }
}
```

Взаємодія параметризованого коду і віртуальної машини

Віртуальна машина не працює з об'єктами параметризованих типів. Всі об'єкти, що надходять у віртуальну машину мають бути звичайними класами. Тому, компілятор на етапі компіляції перетворює всі параметризовані типи у «сирий» тип (універсальний тип без змінних типу, що коректно працює з довільним типом шляхом використання приведень до конкретного типу та інших маніпуляцій). Ім'я цього типу співпадає з іменем параметризованого типу з видаленими параметрами типу. Змінні типу в середині класів і методів замінюються на перший обмежуючий тип в списку обмежень, або на тип `Object`, якщо обмеження відсутні.

Розглянемо приклад параметризованого класу і результату його «підчистки» компілятором. Параметризований клас `Pair`:

```
public class Pair<T>
{
      public Pair (T objFirst, T objSec)
      { first = objFirst; second = objSec;}
      public T getFirst()  {return first; }
      public T getSecond() {return second;}
      public void setFirst(T newVal)  { first = newVal;}
      public void setSecond(T newVal) { second = newVal;}

      private T first;
      private T second;
}
```

Клас `Pair` після «підчистки» компілятором:

```
public class Pair
{
      public Pair (Object objFirst, Object objSec)
      { first = objFirst; second = objSec;}
      public Object getFirst()  {return first ;}
      public Object getSecond() {return second;}
      public void setFirst(Object newVal)  { first = newVal;}
      public void setSecond(Object newVal) { second = newVal;}

      private Object first;
      private Object second;
}
```

Оскільки `T` – необмежена змінна типу, то вона замінюється на клас `Object`.

Якщо б наш параметризований клас `Pair` мав би обмежуючі типи `T extends Comparable & Serializable`, то компілятор замінив би `T` на `Comparable` і де необхідно вставив би приведення до `Serializable`. Тому для підвищення ефективності виконання коду необхідно вставляти пусті інтерфейси (інтерфейси без методів) у кінець списку обмежень. Розглянемо приклад згаданого класу `Pair`.

Параметризований клас `Pair`:

```
public class Pair< T extends Comparable & Serializable >
{
      public Pair (T objFirst, T objSec)
      { first = objFirst; second = objSec;}
      public T getFirst()  {return first; }
      public T getSecond() {return second;}
```

```
        public void setFirst(T newVal)  { first = newVal; }
        public void setSecond(T newVal) { second = newVal;}
        public void sort()
        {
              if (first.compareTo(second) > 0)
              {
                    T tmp = first;
                    first = second;
                    second = tmp;
              }
        }

        private T first;
        private T second;
  }
```

Клас `Pair` після «підчистки» компілятором:

```
  public class Pair implements Serializable
  {
        public Pair (Comparable objFirst, Comparable objSec)
        { first = objFirst; second = objSec;}
        public Comparable getFirst()  {return first ;}
        public Comparable getSecond() {return second;}
        public void setFirst(Comparable newVal)  { first = newVal;}
        public void setSecond(Comparable newVal) { second = newVal;}
        public void sort()
        {
              if (first.compareTo(second) > 0)
              {
                    Comparable tmp = first;
                    first = second;
                    second = tmp;
              }
        }

        private Comparable first;
        private Comparable second;
  }
```

Незалежно від того скільки типів було створено в програмі на базі параметризованого типу всі вони будуть «підчищені» до одного «сирого» типу. Така «підчистка» дозволяє позбутися так званого «розбухання шаблонного коду», яка притаманна C++ і призводить до збільшення розміру програми з збільшенням кількості класів, що побудовані на основі шаблонів, оскільки всі вони зберігаються в пам'яті. З іншої сторони така підчистка має ще одну перевагу – сумісність з першими версіями Java, які не мали параметризованих типів, а використовували замість них «сирі» типи.

Трансляція коду при звертанні до параметризованих методів і полів

Коли програма викликає параметризований метод з параметризованим типом, що повертається, то реально викликатиметься «підчищений» компілятором метод. Цей метод повертатиме тип `Object` або перший обмежуючий тип із списку обмежень. Оскільки в програмі очікується, що метод поверне вказаний програмістом при написанні програми тип результату, то компілятор додатково вставляє інструкцію приведення типу результату роботи «сирого» методу до необхідного типу. Таке саме приведення типу додається при звертанні ззовні до загальнодоступних полів об'єкту, що мають параметризований тип.

Бувають ситуації, коли «підчистка» типів перетинається з поліморфізмом і не дає змоги здійснити поліморфний виклик методу. Така ситуація може виникнути, наприклад, тоді, коли підклас успадковує суперклас, що базується на параметризованому класі і у підкласі перевизначається метод суперкласу, що має параметризований тип параметра. В результаті підчистки параметризований тип параметра буде замінений на тип `Object`, або на перший обмежуючий тип в списку обмежень. У цій ситуації початковоперевизначений метод суперкласу стане новим методом сигнатура якого відома лише підкласу. Таким чином при виклику з-під посилання на базовий клас, що базується на параметризованому класі, перевизначеного методу викличеться «підчищений» метод з «підчищеним» типом параметру, а не перевизначений метод. Щоб захистити поліморфізм у цій ситуації компілятор генерує *метод-міст* у підкласі, який має таку ж сигнатуру, що й «підчищений» метод у суперкласі, і в тілі цього методу генерує код для виклику необхідного методу підкласу.

Розглянемо код, що ілюструє цю ситуацію. Початковий код до проведення компіляції має наступний вигляд:

```
public class Pair<T>
{
      public Pair (T objFirst, T objSec)
      { first = objFirst; second = objSec;}
      public T getFirst()  {return first ;}
      public T getSecond() {return second;}
      public void setFirst(T newVal)  { first = newVal;}
      public void setSecond(T newVal) { second = newVal;}

      private T first;
      private T second;
}

class DateInterval extends Pair<Date>
{
      . . .
      // перевизначення методу суперкласу
      public void setSecond(Date second)
      {
```

```
            if (second.compareTo(getFirst()) >= 0)
                  super.setSecond(second);
      }
}
.......
DateInterval interval = new DateInterval(...);
Pair<Date> pair = interval;
pair.setSecond(aDate);
```

При компіляції компілятор «підчистить» код до такого вигляду:

```
public class Pair
{
      public Pair (Object objFirst, Object T objSec)
      { first = objFirst; second = objSec;}
      public Object getFirst()  {return first ;}
      public Object getSecond() {return second;}
      public void setFirst(Object newVal)  { first = newVal;}
      public void setSecond(Object newVal) { second = newVal;}

      private Object first;
      private Object second;
}

class DateInterval extends Pair
{
      . . .
// абсолютно новий метод, що не має нічого спільного з методом суперкласу
      public void setSecond(Date second)
      {
            if (second.compareTo(getFirst()) >= 0)
                  super.setSecond(second);
      }
}
.......
DateInterval interval = new DateInterval(...);
Pair pair = interval;
pair.setSecond(aDate);
```

Після підчищання у нас з'явилося 2 методи `setSecond` у підкласі і суперкласі, що не мають між собою нічого спільного, але це має бути один і той самий поліморфний перевизначений метод. Оскільки класу `Pair` відомий лише метод з сигнатурою `void setSecond(Object)`, то у рядку `pair.setSecond(aDate)` викличеться саме цей метод. Щоб зберегти поліморфізм компілятор змушений згенерувати у класі `DateInterval` метод з «підчищеною» сигнатурою у якому здійснюватиметься виклик методу з сигнатурою `void setSecond(Date)`:

```
public void setSecond(Object second)
{
// виклик метода з сигнатурою void setSecond(Date) класу DateInterval
      setSecond((Date) second);
}
```

Тонкощі параметризованого програмування

Наявність «підчищення» компілятором параметризованого коду накладає багато обмежень і особливостей написання такого коду мовою Java.

1. При параметризованому програмуванні параметри типу не можуть приймати примітивні типи. Це обмеження пов'язане з тим, що при «підчищенні» змінні типу в середині класів і методів замінюються на перший обмежуючий тип із списку обмежень, або на тип `Object`, якщо обмеження відсутні. Тому при необхідності використання примітивних типів їх слід замінити на типи-обгортки.

2. Оператори дослідження типів під час виконання програми працюють лише з «сирими» типами. У віртуальній машині об'єкти завжди мають визначений непараметризований тип. Тому, всі динамічні опитування типів породжують тільки «сирий» тип. Тож запис

```
if (obj instanceof Pair<Int>)
```

еквівалентний запису

```
if (obj instanceof Pair).
```

При приведенні типів, так само перевіряється лише приналежність об'єктів до однакових «сирих» типів:

```
Pair<String> str = (Pair<String>) message; // OK
Pair<String> str = (Pair<File>) fIn;       // OK
```

Аналогічно метод `getClass()`, що використовується при рефлексії завжди повертає «сирий» тип.

3. Генерувати і перехоплювати об'єкти виключення параметризованого класу заборонено. Не можна використовувати змінну типу в конструкції `catch`. Наприклад, наступний код компілюватися не буде:

```
public static <T extends Throwable> void doWork (Class<T> t)
{
      try
      {
            // обробка даних
```

```
        }
        catch(T ex)      // помилка
        {
              // обробка виключення
        }
}
```

Однак можна використовувати змінні типу в специфікаціях виключень:

```
public static <T extends Throwable> void doWork (T t) throws T
{
        try
        {
              // обробка даних
        }

        catch(Throwable realCause)
        {
              t.initCause(realCause);
              throw t;
        }
}
```

4. Заборонено використання масивів параметризованих типів. Якби це можна було робити, то після «підчистки» типом масиву став би «сирий» тип і масиву можна було б присвоювати значення з різними значеннями змінних типу, що було б некоректно. Тож запис типу:

```
Pair<String>[] arr = new Pair<String>[10];
```

є некоректним. Якщо ж все-таки є потреба в зберіганні об'єктів параметризованих типів у масиві, то можна скористатися класом `ArrayList`, який зробить це безпечно і ефективно.

5. Не можна створювати екземпляри змінних типу. Не можна застосовувати змінні типу у виразах подібним на: `new T(…)`, `new T[…]` або `T.class`. Проблема знову ж таки полягає у «підчищенні», яке може перетворити `T` у `Object`, об'єкт якого створювати нема потреби.

6. Не можна застосовувати змінні типу у статичних членах класу, оскільки в протилежному випадку після «підчищення» можна було б створити лише один екземпляр члену «сирого» класу для всіх класів, що походять від параметризованого класу. Тобто у статичних членах класу слід вказувати окремо від класу параметризований тип:

```
class SomeClass<T> {
      public static <T> void someMethod(T obj) {
```

```
            ...
      }
}
```

Причому цей тип може бути відмінним від типу параметру класу:

```
class SomeClass<T> {
      public static <E> void someMethod(E obj) {
            ...
      }
}
```

7. Слід остерігатися конфліктів після «підчищання». Для цього не слід допускати умов, що можуть створити конфлікти після «підчищання» параметризованих типів, наприклад, коли існуватимуть два одинакові методи в параметризованому класі і у суперкласі цього класу, що прийматимуть тип `Object` та параметризований тип. Після «підчищання» ці методи будуть ідентичними та конфліктуватимуть.

Також слід дотримуватися правила: *для підтримання трансляції підчищанням накладається обмеження, що клас або змінна типу не можуть одночасно бути підтипами двох інтерфейсних типів, що представляють собою різні параметризації одного і того ж інтерфейсу.* Дотримання даного правила вберігає від конфлікту з синтезованими методами-мостами, що не можуть одночасно реалізувати кілька методів з однаковими сигнатурами для різних параметризованих типів. Тобто наступний код є некоректним:

```
class Calendar implements Comparable<Calendar> {…}
class GeorgianCalendar extends Calendar implements Comparable
<GeorgianCalendar> {…}
```

бо одночасне існування методів:

```
public int compareTo(Object other) {return compareTo((Calendar)
other);}
public int compareTo(Object other) {return
compareTo((GeorgianCalendar) other);}
```

у класі `GeorgianCalendar` є неможливим.

Проте непараметризований код є цілком коректним:

```
class Calendar implements Comparable {…}
class GeorgianCalendar extends Calendar implements Comparable {…}
```

Правила спадкування параметризованих типів

1. *Всі класи, що утворені з одного і того ж параметризованого класу з використанням різних значень змінних типів є незалежними* навіть якщо між цими типами є залежність спадкування. Тобто наступний код є некоректним:

```
class SupClass {…}
class SubClass extends SupClass {}
Pair<SubClass> subObj = new Pair<SubClass>(…);
// помилка. subObj і supObj посилаються на незалежні об'єкти
Pair<SupClass> supObj = subObj;
```

2. *Завжди можна перетворити параметризований клас у «сирий» клас*, при роботі з яким захист від некоректного коду є значно слабшим, що дозволяє здійснювати небезпечні присвоєння об'єктів параметризованого класу об'єктам «сирого» класу. Проте у цьому випадку можна зробити помилки, які генеруватимуть виключення на етапі виконання програми:

```
Pair<SubClass> subObj = new Pair<SubClass>(…);
Pair rawObj = subObj;               // OK
rawObj.setFirst(new File(…));       // лише попередження при компіляції
```

3. *Параметризовані класи можуть розширювати або реалізовувати інші параметризовані класи*. В цьому відношенні вони не відрізняються від звичайних класів. Наприклад, `ArrayList<T>` реалізує інтерфейс `List<T>`. Це значить, що `ArrayList<SubClass>` можна перетворити у `List<SubClass>`. Але `ArrayList<SubClass>` це не `ArrayList<SupClass>` і не `List<SupClass>`, де `SubClass` – підклас суперкласу `SupClass`.

Підстановочні типи

Підстановочні типи були введені у мову Java для збільшення гнучкості жорсткої існуючої системи параметризованих типів. На відміну від неї підстановочні типи дозволяють враховувати залежності між типами, що виступають параметрами для параметризованих типів. Це в свою чергу дозволяє застосовувати обмеження для параметрів, що підставляються замість параметризованих типів. Завдяки цьому підвищується надійність параметризованого коду, полегшується робота з ним та розділяється використання безпечних методів доступу і небезпечних модифікуючих методів. Підстановочні типи застосовуються у вигляді параметру типу, що передається у трикутних дужках при утворені реального типу з параметризованого типу, наприклад, у методі `main`.

Підстановочні типи дозволяють реалізувати:

1. обмеження підтипу;
2. обмеження супертипу;
3. необмежені підстановки.

Обмеження підтипу – дозволяє позначити будь-який параметризований тип, чий параметр типу є типом або підтипом вказаного у параметрі типу, що дозволяє одержувати результати роботи методів параметризованого типу, але не передавати параметри методам, що приймають параметри параметризованого типу. Це відбувається тому, що компілятор не здатний вивести з "?" конкретний тип параметру, але гарантує, що цей тип буде типом або підтипом вказаного у параметрі типу.

Обмеження супертипу – дозволяє позначити будь-який параметризований тип, чий параметр типу є класом або суперкласом вказаного у параметрі типу, що дозволяє передавати параметри методам, що приймають параметри параметризованого типу. Це відбувається тому, що компілятор хоч і не здатний вивести з "?" конкретний тип параметру, що передається у метод, але він може безпечно привести передане значення до будь-якого з супертипів. При одержанні результатів роботи методів параметризованого типу нема ніякої гарантії стосовно типу результату, тому результат роботи можна присвоїти лише типу `Object`.

Синтаксис встановлення обмеження підтипу для підстановок має такий вигляд:

```
НазваПараметризованогоТипу <? extends НазваТипуОбмеження>
```

Синтаксис встановлення обмеження супертипу для підстановок має такий вигляд:

```
НазваПараметризованогоТипу <? super НазваТипуОбмеження >
```

Приклад використання обмеження підтипу для підстановок:

```
public static void main ()
{     …
      Pair <? extends List> lst = new Pair<ArrayList<Integer>>();}
```

Синтаксис *необмежених підстановок* має такий вигляд:

```
НазваПараметризованогоТипу <?>
```

Цей запис означає, що параметризований тип `НазваПараметризованогоТипу` може приймати параметр будь-якого типу. Але **сама по собі підстановка "?" не є змінною типу, тому ми не можемо використати "?" як тип**.

Таким чином ми не можемо написати:

```
public static void someMethod(Pair<?> p)
{
      ? t = p.otherMethod();
      …
}
```

Щоб вийти з цієї ситуації застосовують техніку, що називається *захоплення підстановок*. Вона полягає у оголошенні допоміжного параметризованого методу (метод `someMethod` не є узагальненим методом, оскільки він має фіксований параметр типу `Pair<?>`) , який буде викликатися з методу `someMethod`:

```
public static void someMethod(Pair<?> p)
{
      HelperMethod(p);
}

public static <T> void HelperMethod (Pair<T> p)
{
      T t = p.otherMethod();
      …
}
```

В цьому випадку кажуть, що параметр `T` методу `HelperMethod` захоплює підстановку.

Захоплення підстановки не є необхідним для реалізації всіх методів, що використовують необмежену підстановку. Воно не потрібне при реалізації методів, яким для роботи не потрібно знати реальний тип підстановки, наприклад:

```
public static boolean hasNulls (Pair<?> p)
{
      return p.getFirst() == null || p.getSecond() == null;
}
```

Висновок

У лекції розглянуто питання створення і використання параметризованих класів і методів, встановлення обмежень для змінних типів, взаємодії параметризованого коду та віртуальної машини, трансляція коду при звертанні до параметризованих методів і полів, правила спадкування параметризованих типів та особливості застосування підстановочних типів.

Частина 2.

Основи Python

13. Вступ до Python

План

1. Різновиди Python.
2. Початок роботи з Python.
3. Структура програми Python.
4. Типи даних, оголошеня змінних та операції над ними.
5. Оператори.
6. List comprehension.
7. Dict comprehension.

Різновиди Python

Python (найчастіше вживане прочитання — «Па́йтон», запозичено назву з британського шоу Монті Пайтон) — інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. Розроблена в 1990 році Гвідо ван Россумом. В мові програмування Python підтримується кілька парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована.

Є кілька різновидів Python (їх скрипти можуть бути не сумісні між собою):

- CPython – еталонна реалізація Python мовою C.
- Jython – реалізація мови python на Java. Використовує JVM в своїй роботі.
- Iron Python – реалізація мови python на C#. Використовує віртуальну машину C# (CLR) в своїй роботі. Дозволяє працювати з .Net напряму.

Початок роботи з Python

Робота з Python починається з встановлення інтерпретатора Python. Крім нього для розробки програмного забезпечення мовою Python потрібно встановити менеджер пакетів та модулів Pip та середовище для розробки (IDE). Середовища є спеціалізовані, універсальні, а також дистрибутив Python встановлює IDE IDLE з мінімальним функціоналом. Переважно для розробки використовуються спеціалізоване IDE PyCharm, або уінверсальні MS Visual Studio Code або Eclipse.

Виконання скриптів python краще здійснювати у віртуальному середовищі Python, яке можна довільно конфігурувати під ваші потреби і воно буде впливати лише на вашу аплікацію. Для роботи з віртуальним середовищем необхідно спочатку його встановити за допомогою pip, активувати, виконати програму, а після використання деактивувати.

Встановлення Python

Для встановлення Python на систему під керуванням ОС Windows та Mac слід завантажити потрібну версію дистрибутива з https://www.python.org/ та встановити її.

На ОС Linux Python3 переважно є встановлений наперед, тому перед встановленням слід первірити чи python3 є встановлений, а вже потім пробувати його встановлювати. Якщо Python не встановлений, тоді слід встановити його виконавши команди з-під суперкористувача (root) у терміналі спочатку “sudo apt update”, а потім “sudo apt install python3”.

PIP

PIP – це менеджер пакетів Python. Він дозволяє завантажувати і встановлювати різноманітні додаткові бібліотеки з стандартного репозиторію пакетів – Python Package Index, з Git репозиторіїв, файлів з переліком залежностей requirements.txt тощо.

Для встановлення PIP на системі попередньо має бути встановлений Python. Для встановлення PIP на різних ОС треба виконати наступні команди:

- Linux/Mac – виконати: python -m pip install
- Windows – виконати: py -m pip install

Детальніше робота з PIP описана тут.

Віртуальне середовище

Віртуальне середовище Python забезпечує ізольоване середовище виконання програм для однієї чи більше програм, яке включає специфічні для програми налаштування Python, підключені бібліотеки специфічні для кожної з програм та інші речі. Налаштування віртуального середовища для програми відбувається згідно інструкції https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/.

Встановлення IDE

Встановлення IDE відбуваєстья в залежності від інструкції по встановленню специтфічної для кожного IDE.

Створення і відлагодження проектів

В загальному випадку слід відкрити IDE, вибрати тип проекту Python, активувати віртуальне середовище для проекту, підключити віртуальне середовище до проекту у налаштуваннях IDE вказуючи шлях до нього як до інтерпретатора python, який використовуватиметься для проекту, почати писати і відлагоджувати код. Додатково при розробці програмних проектів на Python застосовуються статичні аналізатори, наприклад, Pylint, які перевіряють проект на різноманітні семантичні і стилістичні помилки, та фреймворки для виконання юніт тестів, наприклад, pytest.

Структура і запуск програми Python

Python програми – це набір *.py файлів. Кожен файл – це окремий модуль. Модулі підключаються за допомогою ключового слова `import` після якого вказується назва файлу без розширення py. Щоб доступитися з одного модуля до іншого вони мають бути в одному каталозі. Якщо модуль лежить в іншому каталозі, то щоб до нього доступитися цей каталог має бути оформлений як пакет. Для цього достатньо у нього помістити порожній файл __init__.py. При підключені пакету виконується вміст файлу __init__.py, який може містити визначення імен і інші необхідні для роботи з пакетом конфігураційні дії. Детальніше модулі і пакети будуть розглянуті пізніше.

Коментарі

Python має лише рядкові коментарі. Коментарем у Python є текст після символа ‘#’:

```
# Comment
```

Форматування коду

Код у Python виділяється у блоки за допомогою 4-ох пробілів відносно попереднього блоку або одного символа табуляції. Пробіли і табуляцію не можна змішувати.

Запуск на виконання програми мовою Python

Для запуску на виконання програми мовою Python слід виконати в командному рядку: python.exe <file name>.py. Запустивши інтерпретатор Python.exe, можна вводити з командного рядка програму по-рядково і зразу отримувати результат виконання.

Типи даних, оголошеня змінних та операції над ними

Python підтримує наступні типи даних.

Таблиця 13.1.

Типи даних Python

Текстовий тип:	`str`
Числові типи:	`int, float, complex`
Послідовності:	`list, tuple, range`
Типи-відповідності (Mapping type):	`dict`
Множини:	`set, frozenset`
Булівські типи:	`bool`
Бінарні типи:	`bytes, bytearray, memoryview`
Ніякий тип (None Type):	`NoneType`

Детальніше типи і операції над ними описані тут: https://docs.python.org/3/library/stdtypes.html#built-in-typ. Для визначення типу змінної слід виконати команду: `type(<змінна>)`

Оголошення змінних

Змінна може бути оголошена в будь-якому місці і має бути обов'язково проініціалізована. Тип змінної визначається значенням, яким вона ініціалізована.

Таблиця 13.2.

Способи оголошення змінних

Приклад оголошення змінної	**Тип оголошеної змінної**
`x = "Slava Ukraini"`	`str`
`x = 5`	`int`
`x = 3.14`	`float`
`x = 1j`	`complex`
`x = ["One", "two", "three"]`	`list`
`x = ("One", "two", "three")`	`tuple` (кортеж: незмінний, гетерогенний (може містити елементи різних типів), впорядкований, з дублюваннями тип даних)
`x = range(6) range(0, 6) range(0, 6, 2)`	`range` (діапазон: список елементів в певному діапазоні з певним кроком у

Приклад оголошення змінної	Тип оголошеної змінної
	форматі: початкове значення, кінцеве значення, крок).
`x = {"name": "Ivan", "age": 20}`	`dict`
`x = {"One", "two", "three"}`	`set`
`x = frozenset({"One", "two", "three"})`	`frozenset`
`x = True або False`	`bool`
`x = b"Hello World"`	`bytes`
`x = bytearray(10)`	`bytearray`
`x = memoryview(bytes(16))`	`memoryview`
`x = None`	`NoneType`

Для зміни типу змінної після ініціалізації слід використовувати оператор приведення типу: `x = int(1)`.

Рядки

Букви у рядку можуть займати різну кількість байт в залежності від кодування, яке використовується. З рядками (тип str) можна робити наступні операції:

- Оголошувати багаторядкові рядки між """:

```
a = """ Multiline
        string"""
```

- Отримувати частини тексту вказуючи діапазон індексів букв:

```
b = "Hello World!"
print(b[3:7]) # виведеться 'lo W'
print(b[-1]) # виведеться '!'
print(b[0:-2]) # виведеться "Hello Worl"
print(b[::-1]) # виведеться ревертований рядок '!dlroW olleH'
```

- Викликати методи обробки рядків, оскільки тип str є класом.

- Здійснювати конкатенацію рядків (не рекомендовано, так як призводить до втрат пам’яті):

```
b = "Hello" + "World!"
```

- Здійснювати форматування рядків:

```
val1 = 1
val2 = 2
val3 = 33.1
text = "Val1 = {0}, Val2 = {1}, Val3 = {2}"
print(text.format(val1, val2, val3))
```

- Використовувати f-рядки:

```
val1 = 1
```

```
val2 = 2
val3 = 33.1
text = f"Val1 = {val1}, Val2 = {val2}, Val3 = {val3}"
print(text)
```

Списки

Списки також використовуються для заміни масивів, так як масивів у Python немає. Основні операції над списками:

- Оголошення списку:

```
x = [] # порожній список
x = list() # порожній список
x = ["One", "two", "three"] # ініціалізований список
```

- Доступ до елементів списку відбувається по індексу. Індексація починається з 0 :

```
x[1]
```

- Присвоєння:

```
x[1] = "New value"
```

- Визначення розміру списку:

```
len(x)
```

Таблиця 13.3.

Основні методи модифікації списків

Метод	**Призначення**
`append()`	Додати елемент(и) в кінець списку
`clear()`	Очистити список
`copy()`	Отримати копію списку
`count()`	Підрахунок кількості елементів з заданим значенням
`extend()`	Додати елементи до кінця списку, або будь що, що можна проітерувати.
`index()`	Повертає індекс першлго значення заданого параметром
`insert()`	Додати елемент в задану позицію
`pop()`	Повертає значення заданого парамтром індексу елемента і видаляє його зі списку
`remove()`	Видаляє елементи з вказаним значенням зі списку
`reverse()`	Розташовує елементи списку в зворотному порядку
`sort()`	Сортує список

Словники

Словник – це набір пар елементів ключ-значення. Ключі мають бути унікальні. Оголошення словника:

```
x = {} # порожній словник
x = dict() # порожній словник
x = {
"name": "Ivan",
 "age": 20
}
```

- Доступ до елементу словника за ключем:

```
x["age"]
```

- Отримання списку ключів:

```
x.keys()
```

- Модифікація значень:

```
x["age"] = 33
x.update({"name": "Petro"})
```

- Додавання елементів:

```
x["height"] = 180
x.update({"weight": 82})
```

- Видалення елементів:

```
x.pop("weight")
del x["height"]
```

- Видалення змінної словника:

```
del x
```

Таблиця 13.4.

Основні методи модифікації словників

Метод	**Призначення**
`clear()`	Видаляє усі елементи словника
`copy()`	Повертає копію словника
`fromkeys()`	Повертає новий словник на базі вказаних ключів
`get()`	Повертає значення визначеного ключа
`items()`	Повертає список, що містить tuple (кортеж) для кожної пари ключ-значення

Метод	Призначення
`keys()`	Повертає список ключів
`pop()`	Видаляє елемент з вказаним ключем і повертає його значення
`popitem()`	Видаляє останню додану пару ключ-значення
`setdefault()`	Повертає значення вказаного ключа. Якщо ключа не існує, то додає ключ з вказаним значенням
`update()`	Оновлює словник вказаними значеннями
`values()`	Повертає список значень словника

Словники і списки зручно використовувати при роботі з json і серіалізації даних.

Таблиця 13.5.

Математичні операції

Назва операції	Оператор
Додавання	+
Віднімання	-
Множення	*
Ділення	/
Цілочисельне ділення	//
Піднесення до степеня	**
Остача від ділення	%

Оператори

Умовний оператор if-else

Синтаксис умовного оператора if-else:

```
if <умова>:
     [оператор]
{elif <умова>:}
     [оператор]
[else:]
     [оператор]
```

Таблиця 13.6.

Умови

Тип умови	**Приклад**
Рівність	a == b
Не рівність	a != b
Менше	a < b
Менше або рівне	a<= b
Більше	a > b
Більше або рівне	a >= b

Приклад:

```
if a<b:
  c=1
elif a>b:
  c=2
else:
  c=3
```

Якщо гілка не має містити операцій, то можна вказати порожній оператор `pass`:

```
if a<b:
  c=1
else:
  pass
```

Таблиця 13.7.

Логічні оператори

Оператор	**Приклад**
або	or
і	and
ні	not

```
if a<b and b>0:
  c=1 # ця змінна буде доступна також і після виходу з if
else:
  c=2
```

Скорочені форми запису (умовні вирази):

```
if a > b: c=2
c=2 if a > b else c=1
c=2 if a > b else c=1 if a == b else c=3
```

Цикл while

Синтаксис циклу while:

```
while <умова>:
    <оператори>
```

Приклад:

```
i = 1
while i < 5:
  i += 1
```

Цикл for

Синтаксис циклу for:

```
for x in <ітератор>:
  <оператори>
[else
  <оператори>]
```

Цикл for у python має деякі відмінності в порівнянні з іншими мовами програмування. Розглянемо принцип його роботи. Змінній `x` почергово присвоюються елементи, що знаходяться у ітераторі, і для кожного з них виконуються оператори тіла циклу. Після завершення виконання циклу виконується блок операторів після `else`.

Приклади застосуваня циклу for:

1. Цикл for використовуєтсья для ітерування по послідовностях, таких як список, кортеж, словник, множина, рядок.

```
nums = ["one", "two", "three"]
for x in nums:
  print(x)

for x in "Hello":
  print(x)
```

2. Ітерування по діапазону значень (аналог звичайного циклу for).

```
for x in range(5): # створюється послідовність 0,1,2,3,4
  print(x)

for x in range(0, 5): # створюється послідовність 0,1,2,3,4
  print(x)

for x in range(0, 5, 2): # створюється послідовність 0,2,4
  print(x)
```

Перед початком виконання циклу `range` створює набір всіх значень, необхідних для циклу, тому це займає більше часу, ніж коли значення генеруються лише для окремої ітерації. Тому, доцільніше використовувати не `range`, а генератор:

```
def my_range(start, end, step):
  while start <= end:
   yield start
   start += step

for x in my_range(1, 10, 0.5):
    print(x)
```

У цьому прикладі з'являється оператор `yield`, який використовується для реалізації генераторів. Він перериває виконання функції і повертає своє поточне значення з функції. При наступному виклику функції вона почне виконуватися з наступного оператора після `yield`.

Оператори переривання потоку виконання

Python 3 має 2 оператори переривання потоку виконання це – `break` і `continue`. Принцип їх роботи аналогічний до принципу роботи цих операторів у мові C/C++.

List comprehension

List comprehension забезпечує скорочений синтаксис створення сисків на базі існуючих списків.
Ситаксис:

```
<список> = [<змінна> for <змінна> in <iterable>]
<список> = [<змінна> for <змінна> in <iterable> if <умова>]
```

Приклад:

а) Класичний підхід

```
original_lst = ["one", "two", "three", "four", "five"]
new_lst = []

for x in original_lst:
  if "o" in x:
    new_lst.append(x)
```

б) З застосуванням list comprehension

```
original_lst = ["one", "two", "three", "four", "five"]
new_lst = [x for x in original_lst if "o" in x]
```

Dict comprehension

Dict comprehension забезпечує скорочений синтаксис створення словників на базі типів, що підтримують ітератор. Існує багато способів застосування dict comprehension, тому можливі деякі відмінності в синтаксисі його застосування.

Ситаксис:

```
<словник> = {<ключ>: <значення> for <елемент> in <iterable>}
<словник> = {<ключ>: <значення> for <ключ>, <значення>
in <iterable>}
<словник> = {<ключ>: <значення> for <ключ>, <значення>
in <iterable> if <умова>}
```

Приклад 1

а) Класичний підхід

```
values = ["one", "two", "three", "four", "five"]
keys = [1, 2, 3, 4, 5]
new_dict = {}

for k, v in zip(keys, values):
```

б) З застосуванням dict comprehension

```
values = ["one", "two", "three", "four", "five"]
keys = [1, 2, 3, 4, 5]
new_dict = {k: v for k, v in zip(keys, values)}
```

Приклад 2

а) Класичний підхід

```
original_dict = {"Oleh": 180, "Ihor": 182, "Mykola":
175, "Petro": 178, "Ivan": 170}
new_dict = {}

for k, v in original_dict.items():
  if v > 175:
     new_dict[k] = v
```

б) З застосуванням dict comprehension

```
original_dict = {"Oleh": 180, "Ihor": 182, "Mykola":
175, "Petro": 178, "Ivan": 170}
new_dict = {k: v for k, v in original_dict.items() if v > 175}
```

Висновок

У лекції розглянуто налаштування середовища розробки програм мовою Python, структуру програми, основні типи даних, основні операції над даними та оператори мови Python.

14. Розширені можливості процедурного програмування у Python

План

1. Функції.
2. Виключні ситуації.
3. Введення даних з клавіатури.
4. Вивід даних на екран.
5. Файли.
6. Читання з файлів.
7. Запис у файли.
8. Оператор with.
9. Потік виконання.
10. Модулі.
11. Пакети.

Функції

Функції у мові python не відрізняються за своєю суттю від функцій C/C++. Синтаксис оголошення функцій:

```
def function_name({параметри}):
  [оператори]
```

Приклади оголошення функцій:

```
def my_output():
  print("Hello world")

def my_output(txt1, txt2, delimitator = " "):
  """
     Outputs concatenated string
     :param txt1: the first text string
     :param txt2: the second text string
     :param delimitator: Delimitator. Space by default.
     :return: concatenated string
  """
  merged_text = delimitator.join((txt1, txt2))
  return merged_text
```

Приклади виклику функцій:

```
my_output("Hello ", "world")
my_output(txt1 = "Hello ", txt2 = "world") # Передача параметрів за назвою параметра
```

Функції з довільної кількістю параметрів

У Python функції можуть мати довільну кількість параметрів. У цьому випадку їм можна передавати неіменовані або іменовані параметри, або їх комбінацію. Неіменовані параметри довільної кількості передаються за допомогою конструкції:

```
*args
```

При цьому всі передані аргументи поміщаються у артумент `args`, який є списком, що містить значення переданих аргументів.

Приклад оголошення функції з довільною кількістю неіменованих аргументів

```
def my_output(*args):
  """
     Outputs concatenated string
     :param args: the tuple of text strings
     :return: concatenated string
  """
  merged_text = str()
  for arg in args:
    merged_text = merged_text + " " + arg
  return merged_text
```

Приклад виклику функції з довільною кількістю неіменованих аргументів

```
 my_output("Hello ", "world")
```

Функції з довільною кількістю іменованих параметрів

Іншим способом передачі довільної кількості аргументів є використання іменованих параметрів, де при виклику аргументи передаються як пари значень – назва аргументу і його значення. При цьому використовується наступна конструкція:

```
**kwargs
```

Приклад оголошення функції з довільною кількістю іменованих аргументів

```
def my_output(**kwargs):
  """
     Outputs concatenated string
     :param kwargs: the dictionary of text strings
     :return: concatenated string
  """
  merged_text = str()
  for arg in kwargs.keys():
    merged_text = merged_text + " " + kwargs[arg]
  return merged_text
```

Приклад виклику функції з довільною кількістю іменованих аргументів

```
my_output(txt1 = "Hello", txt2 = "world", ... txtN = "ZZZZ")
```

Функція без тіла

Функція у Python не може зовсім не мати тіла. Якщо функція не має, тіла то слід вказати ключове слово `pass` як тіло такої функції.

```
def my_output(**kwargs):
  pass
```

Повернення кількох значень з функції

У Python можна передати з функції кілька результатів через оператор `return` через кому. У цьому випадку при виклику функції її результат треба буде присвоїти кільком змінним. Якщо функція повертає результат, який не використовується у програмі, то щоб уникнути оголошення змінної, яка ніде не буде використовуватися, застосовують символ “_” для таких результатів.

```
def few_output():
  return 1, 2
```

Приклад виклику функції

```
a, b = few_output() # a = 1, b=2
a = few_output() # a = (1, 2)
a, _ = few_output() # a = 1, другий результат ігноруємо
```

Виключні ситуації

Мова Python має вбудований механізм обробки виключних ситуацій. Обробка виключних ситуацій забезпечується блоками try-except-finally.

Синтаксис:

```
  try:
       <блок коду,  що може згенерувати виключення>
```

```
except <клас_виключення> as <посилання>:
     <блок коду обробника виключень>
else:
     <блок коду,  що виконується, якщо виключення не було
згенероване>
finally:
     <блок коду, який завжди виконується>
```

Приклад:

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Not a valid number.")
```

Блоки `try` є обов'язковим. Блок `try` містить код, який може згенерувати виключну ситуацію. Блок `except` не є обов'язковим, за умови, що визначено блок `finally`. Він містить код обробки виключної ситуації. Він може приймати перелік класів-виключень при генерації об'єктів яких буде виконане тіло даного блоку або бути порожнім:

```
# обробляє усі виключення. Доступу до об'єкту-виключення
немає.
except:

# обробляє виключення типу ValueError. Доступу до об'єкту-
виключення немає.
except ValueError:

# обробляє виключення типу RuntimeError, TypeError,
NameError. Доступу до об'єкту-виключення немає.
except (RuntimeError, TypeError, NameError):
```

Якщо `except` не містить жодного класу, то він буде реагувати на всі виключення, але доступитися до них буде неможливо через відсутність посилання на об'єкт класу-виключення.

Щоб доступитися до об'єкту-виключення слід вказати посилання на нього за допомогою наступного синтаксису:

```
except <клас-виключення> as <посилання>:
```

Приклад:

```
except ValueError as e:
```

У даному прикладі до об'єкту-виключення можна дістатися за допомогою посилання e. Об'єкт-виключення прийнято називати e, ex, exc або err.

Блоків except може не бути, бути один або бути багато. Якщо блоків багато, то кожен блок роблять таким, щоб він обробляв певний тип чи типи виключень. Блок except без параметрів, якщо використовується разом з іншими блоками except з параметрами, прийнято ставити в кінці послідовності блоків except, щоб він обробляв усі виключення які не були оброблені попередніми блоками except з параметрами.

При необхідності блок except може мати необов'язковий блок else, який виконується, якщо виключення даного типу не було згенероване і блок except не виконувався.

Блок finally виконується завжди, якщо він є присутній. Цей блок може бути відсутнім, якщо присутній блок except.

Введення даних з клавіатури

Зчитування рядка зі стандартного пристрою введення sys.stdin (клавіатура) в мові Python здійснюється за допомогою функції

```
input([prompt])
```

Необов'язковий параметр prompt, призначений для вказання запрошення до введення, та буде виведений на стандартний пристрій виведення sys.stdout (екран).

Функція повертає введений користувачем рядок після натискання клаваші Enter.

Приклад використання:

```
змінна = input([prompt])
```

Оскільки функція повертає текстовий рядок, то щоб отримати результат іншого типу його треба явно привести до потрібного типу. Наприклад, щоб отримати результат типу int і присвоїти його змінній a треба зробити наступний виклик:

```
a = int(input("Enter a number"))
```

Вивід даних на екран

Виведення на стандартний пристрій виведення sys.stdout (екран) можна здійснити функцією print(). Вона приймає наступні параметри:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

де:

*objects – послідовність об'єктів розділених комами (змінні, рядки, константи), значення яких потрібно вивести

`sep=' '` – роздільник, який функція ставитиме між об'єктами, що виводяться (за замовчуванням – пробіл). Для виводу спеціальних символів використовують Escape-послідовності, наприклад:
\' – одинарна лапка
\" – подвійна лапка
\? – знак питання
\\ – зворотний слеш
\n – новий рядок
\t – горизонтальна табуляція
\v – вертикальна табуляція

`end='\n'` – символ, що ставиться в кінці рядка (за замовчуванням – символ кінця рядка)

`file=sys.stdout` – виведення в файл. Об'єкт file повинен бути об'єктом з методом write(str). print() можна використовувати тільки для текстових файлів.

`flush=False` – примусове очищення буфера виводу (за замовчуванням – не здійснюється, оскільки зазвичай визначається файлом).

Приклад:

```
a=b=3
print(a, b)
print("a+b=", a+b)
```

Файли

Ключовою функцією для роботи з файлами є функція `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)`. Вона повертає дескриптор відкритого файлу або `None`. Параметри функції:

- `file` – шлях до файлу
- `mode` – режим відкривання файлу. Може приймати наступні значення та їх комбінації.

Таблиця 14.1.

Режими відкривання файлу

Параметр	Значення
'r'	Відкрити для читання (за замовчуванням)
'w'	Відкрити для запису, очистивши попередньо файл, якщо файл існує
'x'	Відкрити для ексклюзивного створення, якщо файл уже існує, то функція завершується невдачею
'a'	Відкрити для запису, дописуючи в кінець файлу, якщо він існує

'b'	Бінарний режим
't'	Текстовий режим (за замовчуванням)
'+'	Відкрити для оновлення (читання та запис)

Параметри r,w,x,a можуть комбінуватися з параметрами b і t. Параметр + додається до інших параметрів за потреби.

- `buffering` — необов'язкове ціле число, яке використовується для встановлення політики буферизації. Значення 0 вимикає буферизацію (використовується лише в бінарному режимі); значення 1 активує буферизацію рядків (використовується лише в текстовому режимі); ціле число > 1, вкаує розмір у байтах буфера фрагментів фіксованого розміру. При значеннях 0 і 1 розмір буферу вказується окремо згідно повної документації цієї функції. Якщо параметр рівний -1, то у більшості випадків розмір буферу визначається автоматично.
- `encoding` – задає назву кодування тексту.
- `errors` – необов'язковий рядок, який визначає, як мають оброблятися помилки кодування та декодування. Не використовується в бінарному режимі.
- `newline` – визначає, як аналізувати символи нового рядка з потоку. Це може бути None, '', '\n', '\r' і '\r\n'.
- Якщо `closefd` має значення False і вказано дескриптор файлу, а не ім'я файлу, базовий дескриптор файлу залишатиметься відкритим, коли файл буде закрито. Якщо вказано назву файлу, то closefd має мати значення True (за замовчуванням); інакше виникне помилка.
- `opener` – посилання на користувацьку функцію, яка буде викликана для відкривання файлу, замість стандартної. Функція приймає параметри (file, flags) і повертає дескриптор відкритого файлу.

Приклад використання:

```
f = open('file-path', 'w', encoding="utf-8")
```

Після завершення роботи з файлом слід викликати метод `close` об'єкту-дескриптора файлу:

```
f.close()
```

Читання/запис відбуваєтсья за допомогою методів read/write та їх похідними об'єкту-дескриптора файлу.

Приклад читання з файлу:

```
fname = 'somefile.txt'
```

```
try:
     f = open(fname, 'r', encoding="utf-8")
     for line in f:
         print(line, end='')
except FileNotFoundError:
    print(f"File {fname} not found.  Aborting")
    sys.exit(1)
except OSError:
    print(f"OS error occurred trying to open {fname}")
    sys.exit(1)
except Exception as err:
    print(f"Unexpected error opening {fname} is",repr(err))
    sys.exit(1)
finally:
     if f:
          f.close()
```

Приклад запису у файл:

```
text = 'text…'
fname = 'somefile.txt'

try:
     f = open('somefile.txt', 'w', encoding="utf-8")
     f.write(text)
except FileNotFoundError:
    print(f"File {fname} not found.  Aborting")
    sys.exit(1)
except OSError:
    print(f"OS error occurred trying to open {fname}")
    sys.exit(1)
except Exception as err:
    print(f"Unexpected error opening {fname} is",repr(err))
    sys.exit(1)
finally:
     f.close()
```

Читання з файлів

Читання з файлів здійснюється за допомогою методу `read` об'єкту-файлу. Для читання однобайтних текстових рядків достатньо викликати метод `read` (для читання всього файлу чи певної кількості байт, кількість яких передається аргументом методу), або методу `readline` (для по-рядкового читання з файлу). Для читання даних інших типів вони мають бути записані як байтові послідовності, які вичитуються методом `read`

після чого приводяться до відповідного типу. Для цього можна використати модуль `struct`, який призначений для полегшення інтерпретації байт як запакованих бінарних даних. Він перетворює значення Python на структури C, представлені як байтові об'єкти Python. Детальніше даний модуль описаний тут. Тож для розпаковування послідовності байт у певний тип слід використати метод `unpack` класу `struct`:

```
struct.unpack(format, buffer)
```

де `format` – визначає тип даних, які розпаковуються, а `buffer` – буфер, що містить послідовність байт, які треба розпакувати. Розмір буферу має відповідати типу даних, що розпаковується. Метод завжди повертає результат типу `tuple`, навіть якщо він містить лише одне значення. Тож наступний код вичитує одне бінарне число типу `double` з файлу і записує його у змінну `res`:

```
f = open('somefile.bin', 'wb')
res = struct.unpack('d', f.read())[0]
f.close()
```

Якщо файл має багато значень, а нам слід вичитати лише одне, тоді слід вказати кількість байт, що необхідно прочитати, наприклад,

```
f = open('somefile.bin', 'wb')
res = struct.unpack('d', f.read(struct.calcsize('d')))[0]
f.close()
```

Запис у файли

Запис у файл здійснюється за допомогою методу `write` об'єкту-файлу. Для запису однобайтних текстових рядків достатньо їх передати у метод `write`. При запису двійкових даних їх необхідно спочатку перетворити у послідовність байт. Для цього можна використати приведення до типу даних `bytearray`, метод `to_bytes` типу даних (наприклад, `int.to_bytes(var)`), або використати модуль `struct`. Тож, для запису бінарних даних їх спочатку треба запакувати у об'єкт, який являє собою послідовність байт та записати цю послідовність у файл. Для цього використаємо метод `pack` класу `struct`:

```
struct.pack(format, v1, v2, ...)
```

де `format` – визначає тип даних, які запаковуються, а значення `v1`, `v2`, … - послідовність даних, які слід запакувати у бінарну структуру. Тож наступний код:

```
data = 2.0
res = struct.pack('d', data)
```

запакує дійсне число 2.0 у форматі подвійної точності (цей формат заданий аргументом `'d'`) у бінарну структуру, яка готова до зберігання у бінарному файлі:

```
data = 2.0
res = struct.pack('d', data)
f = open('somefile.bin', 'wb')
f.write(res)
f.close()
```

Оператор `with`

Оператор `with` використовується для автоматизації процесів закриття ресурсу і коректної обробки виключних ситуацій (аналог оператора try-з-ресурсами у Java). Наприклад, автоматичне закриття файлу, чи з'єднання після завершення роботи з ним, а також, при виникненні виключень. Таким ресурсом може бути будь-який об'єкт, клас якого містить визначені методи `__enter__` та `__exit__`, які дозволяють належним чином керувати ресурсами під час входу в блок `with`, виходу з нього та обробки виключних ситуацій. Такий об'єкт в термінах оператора `with` називається менеджером контексту. Детальний опис оператора `with` є у PEP-343.

Синтаксис:

```
with EXPRESSION as VAR:
    BLOCK
```

де:

- `EXPRESSION` – вираз, який продукує об'єкт-менеджер контексту, або власне об'єкт-менеджер контексту;
- `VAR` – посилання на об'єкт, який перебуває під контролем менеджера контексту;
- `BLOCK` – блок коду, який має бути виконаний з використанням `TARGET` будучи обгорнутим при цьому конструкцією `try-except-finally`.

Конструкція "`as VAR`" є опціональною.
Ця конструкція семантично еквівалентна наступній:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    VAR = value
    BLOCK
except:
    hit_except = True
```

```
        if not exit(manager, *sys.exc_info()):
            raise
    finally:
        if not hit_except:
            exit(manager, None, None, None)
```

Приклад читання з файлу з використанням оператора `with`:

```
fname = 'somefile.txt'

try:
     with open(fname, 'r', encoding="utf-8") as f:
     for line in f:
         print(line, end='')
except FileNotFoundError:
    print(f"File {fname} not found.  Aborting")
    sys.exit(1)
except OSError:
    print(f"OS error occurred trying to open {fname}")
    sys.exit(1)
except Exception as err:
    print(f"Unexpected error opening {fname} is",repr(err))
    sys.exit(1)
# Наступні операції виконуються оператором with автоматично
#finally:
#    f.close()
```

Потік виконання

В загальному випадку Python виконує код у файлі зверху вниз. Але якщо у файлі знаходиться конструкція ”`__name__ == "__main__":`”. то можливі наступні 2 випадки:

1. Якщо запуск програми відбувся шляхом передачі інтерпретатору файлу, який містить main функцію і файл містить конструкцію, що наведена нижче, то виконання програми почнеться з main функції (якщо точніше, то виконається код у гілці true опреатора `if __name__ == "__main__":`):

```
def main():
    print("Hello World!")

if __name__ == "__main__":
    main()
```

2. Якщо цей же ж файл підключається в складі бібліотеки, то виконання main функція не буде запущена без явного виклику в коді програми.

При запуску файлу значення змінної __name__ рівне __main__. При імпортуванні коду файлу вона міститиме назву файлу.

Приклад файлу з скриптом сортування бульбашкою

```
def bubble_sort(array):
     n = len(array)

     for i in range(n):
        # Create a flag that will allow the function to
        # terminate early if there's nothing left to sort
        already_sorted = True

        # Start looking at each item of the list one by one,
        # comparing it with its adjacent value. With each
        # iteration, the portion of the array that you look at
        # shrinks because the remaining items have already been
        # sorted.
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                # If the item you're looking at is greater than
                # its adjacent value, then swap them
                array[j], array[j + 1] = array[j + 1], array[j]

                # Since you had to swap two elements,
                # set the `already_sorted` flag to `False` so e
                # algorithm doesn't finish prematurely
                already_sorted = False

        # If there were no swaps during the last iteration,
        # the array is already sorted, and you can terminate
        if already_sorted:
            break

    return array

def main():
    arr = [1, 10, 3, 4, 6, 7, 2, 8, 9]
    bubble_sort(arr)

if __name__ == "__main__":
    main()
```

Модулі

Модулем у Python називається файл з розширенням *.py. Ці файли можуть містити звичайні скрипти, змінні, функції, класи і їх комбінації. Python дозволяє структурувати код програм у різні модулі та доступатися до класів, функцій і змінних, які у них знаходяться з інших модулів. Для цього використовуються два оператори – `import` та `from-import`.

Оператор `import`

Оператор `import` дозволяє імпортувати модуль повністю, та доступатися до нього через назву модуля. Вона може бути вказана у будь-якому місці програми перед звертанням до елементів, які у ній містяться, але зазвичай її вказують на початку модуля. Для звертання до елементів модуля треба вказати назву модуля і після крапки вказати до якого елементу ви хочете звернутися.

Синтаксис

```
import назва_модуля
назва_модуля.елемент_модуля
```

Приклад

Файл my_print_module.py

```
def hello_print():
     print("Hello")
```

Файл printer.py

```
import my_print_module
my_print_module.hello_print ()
```

Для зручності оригінальну назву модуля можна змінити на псевдонім і доступатися до елементів модуля за допомогою псевдоніма.

Синтаксис

```
import оригінальна_назва_модуля as псевдонім
```

Приклад

```
import my_print_module as my_printer
```

Оператор `from-import`

Конструкція `from-import` дозволяє імпортувати модуль частково, вказавши який саме елемент має бути імпортовано. У цьому випадку доступатися до імпортованого елементу можна напряму не вказуючи назву модуля. Дана конструкція може бути вказана у будь-якому місці програми перед звертанням до елементів, які у ній містяться, але зазвичай її вказують на початку модуля. Вказуючи * після `import` імпортуватиметься весь вміст модуля.

Синтаксис

```
from назва_модуля import елемент_модуля
елемент_модуля
```

Приклад

Файл my_print_module.py

```
def hello_print():
     print("Hello")
```

Файл printer.py

```
from my_print_module import hello_print
hello_print()
```

Файл printer_all.py

```
from my_print_module import *
hello_print()
```

Для зручності оригінальну назву елемента модуля можна змінити на псевдонім і доступатися до нього за допомогою псевдоніма.

Синтаксис

```
from назва_модуля import елемент as псевдонім_елементу
```

Приклад

```
from my_print_module import hello_print as prnt
prnt()
```

Типи модулів

Модулі у Python можуть бути користувацькі, які створив користувач у свої програмі, і вбудовані (включно з вбудованими у інтерпретатор, наприклад модуль `sys`), які належать різним вбудованим пакетам Python.

Коли користувач імпортує модуль, скажімо, `my_print_module` Python здійснює його пошук наступним чином. Спочатку модуль шукається серед вбудованих модулів. Вони перелічені у `sys.builtin_module_names`. Якщо його не знайдено серед вбудованих модулів, то відбувається пошук у файлі my_print_module.py. Пошук цього файлу здійснюється серед списку директорій, які містяться у змінній `sys.path`. Змінна `sys.path` в свою чергу міть список дерикорій які знаходяться у:

- Поточній директорії скрипта, який шукає модуль або у поточній директорії, якщо скрипт міститься не у файлі;
- У змінній середовища `PYTHONPATH`, яка містить список директорій, який відповідає змінній `PATH` середовища з якого був запущений на виконання Python;
- Шляху за замовчуванням, який залежить від інстальованого пакету Python та включає каталог site-packages, який управляється модулем site; також на цей шлях може впливати наявність віртуального середовища викоання і інші фактори.

Деякі стандартні вбудовані модулі залежать від ОС на якій інстальовано Python, тому будуть недоступні на інших операційних системах.

Для пришвидшення пошуку модулів Python здійснює компіляцію усіх модулів, що завантажуються і розміщує компільований код у папці __pycache__ у файлах з назвою назва_модуля.формат_файлу.pyc. Форма файлу дозволяє розрізнити яким саме

компілятором Python був створений даний файл, що дозволяє виконувати програму за допомогою різних версій Python.

Для одержання списку усіх доступних (визначених) елементів (імен) у модулі треба викликати функцію `dir()`і передати ій параметром назву модуля для якого треба відобразити інформацію, наприклад, `dir(my_print_module)`.

Пакети

Пакети призначені для того, щоб розділити модулі по просторах імен. Фізично пакет – це папка, яка містить модулі і файл __init__.py, наявність якого визначає даний каталог модулем, а не звичайною папкою. Цей файл може бути порожнім, а може містити код ініціалізації пакету, який буде виконуватися при підключені пакету, а також змінну `__all__`, яка має містити перелік модулів, які будуть імпортовані конструкцією імпортування усіх пакетів: `from назва_пакету import *`.

Розглянемо приклад організації пакетів на диску:

```
Root_package/              кореневий пакет
    __init__.py
    Subpack_A/             пакет Subpack_A
        __init__.py
        a.py
        b.py
    Subpack_B/             пакет Subpack_B
        __init__.py
        a.py
        b.py
    Subpack_C/             пакет Subpack_C
        __init__.py
        a.py
        b.py
    Folder_D/              папка Folder_D
        file1.py
        file2.py
```

Якщо модулі знаходяться в одному пакеті, то імпорт модулів відбувається звичайним чином, без вказання назви пакета як розглядалося вище.

Якщо імпортується модуль, який знаходиться у підключеному зовнішньому пакеті, тоді при його імпорті слід вказати перед назвою модуля назву пакета до якого він належить. Так само адресуються модулі, які належать до одного кореневого модуля.

Наприклад імпорт модуля `a` з `Subpack_A` з модуля `d`, який не належить розглянутій ієрархії пакетів

```
import Root_package.Subpack_A.a
```

або

```
from Root_package.Subpack_A.a import *
```

Так само слід імпортувати модуль `a` з `Subpack_A` з модуля `b` з `Subpack_B`.

Розглянуті вище способи звернення до модулів у пакетах належать до абсолютної адресації пакетів і модулів. Крім неї Python підтримує також і відносну адресацію за допомогою крапок і синтаксису `from-import`, де одна крапка одначає поточний паке, дві крапки – батьківський пакет, 3 і більше відповідно вищі рівні ієрархії.

Наприклад, припустимо, що ми знаходимося у модулі `b` з `Subpack_B`, тоді:

```
from . import a                # імпорт a з Subpack_B
from .. import Subpack_A # імпорт цілого пакету Subpack_A
from ..Subpack_A import a     # імпорт a з Subpack_A
```

Зверніть увагу, що відносний імпорт базується на назві поточного модуля. Оскільки назва головного модуля завжди "`__main__`", то модулі, призначені для використання у вигляді головного модуля програми Python, повинні завжди використовувати абсолютний імпорт.

Пакети підтримують ще один спеціальний атрибут, `__path__`. Він ініціалізується як список, що містить назву каталогу, що містить __init__.py пакета, перед тим, як буде виконано код у цьому файлі. Цю змінну можна змінити, що вплине на майбутній пошук модулів і підпакетів, що містяться в пакеті.

Хоча ця функція не часто потрібна, її можна використовувати для розширення набору модулів, які містяться в пакеті.

Починаючи з Python 3.3 додано поняття *namespace package* – це папка з *.py файлами, що не містить __init__.py файлу. Він може фізично знаходитися в будь-якому місці. Детальніше про різницю між двома типами пакетів можна прочитати тут.

Висновок

У лекції розглянуто питання роботи з файлами, опрацювання виключних ситуацій, потік виконання, та особливості організації програми у вигляді модулів та пакетів.

15. Об’єктно-Орієнтоване Програмування у Python

План

1. Класи.
2. Спадкування.

Класи

Клас оголошується за допомогою ключового слова `class` після якого йде назва класу. Клас може містити:

- дані, які належать класу (статичні дані-члени класу);
- дані, які належать об’єкту класу;
- методи, які належать класу (статична методи);

- методи, які належать об’єкту класу.

Члени класу є лише публічні, проте Python забезпечує механізми, які дозволяють організувуати області видимості близькі за своєю суттю до protected і private. Це робиться шляхом використання нижнього підкреслення у назві членів класу. Одинарне нижнє підкреслення перед назвою члену класу робить за своїми властивостями схожим на захищений член класу, а подвійне – схожим на приватний член класу. Всі члени класу, що йому належать мають відступ у розмірі одного табулятора, або 4-ох пробілів від початку оголошення класу.

Статичні члени-дані класу оголошуються в класі як назва змінної і її початкове значення.

Дані, які належать об’єкту класу оголошуються в конструкторі з використанням ключового слова `self`, яке є посиланням на об’єкт класу:

```
self.<назва_змінної> = <початкове значення>
```

Статичні методи класу оголошуються в класі за правилами оголошення функцій.
Не статичні методи оголошуються в класі за правилами оголошення функцій, перший параметр якої є обов’язково `self`:

```
def <назва_методу>(self, <параметри>):
     тіло методу
```

Роль конструктора відіграє метод `__init__(self, <параметри>)`.

Доступ до статичних членів класу відбуваєтсья за допомогою назви класу:

```
<назва класу>.<назва статичного члену класу>.
```

Доступ до не статичних членів класу відбуваєтсья за допомогою назви об’єкту:

```
<назва об’єкту>.<назва не статичного члену класу>.
```

Параметри `self` передавати у метод при виклику не потрібно. Він передається неявно як і у інших мовах програмування.

Якщо клас не містить ніяких членів, то він має мати замість них ключову слово `pass`.

```
class <назва класу>:
    pass
```

Видалення окпремих членів класу або об’єкту загалом здійснюється за допомогою оператора `del`:

```
del <назва_об’єкту.назва_члену_даних> # видалення властивості
del <назва_об’єкту> # видалення об’єкту
```

Приклад:

```
class Pet:
    number = 0 # статична публічна змінна-член класу
    _average_height = 0 # статична захищена змінна-член класу
    def __init__(self, name, age, height):
        self.__name = name # не статична приватна змінна-член класу
        self.__age = age
        self.__height = height
        self._height_in_inch = height * 2.54 # не статична захищена змінна-член класу
        Pet._average_height = (Pet._average_height + height) / 2
        Pet.number = Pet.number + 1

    def __private_member(self, ...):
        """ приватний метод """
        <тіло>

    def _protected_member(self, ...):
        """ захищений метод """
        <тіло>

    def public_member(self, ...):
        """ загальнодоступний метод """
        <тіло>

    @staticmethod
    def public_static_member(...):
        """ загальнодоступний статичний метод """
        <тіло>

my_pet = Pet("Fluffy", 3, 15) # створення об’єкту my_pet
print(Pet.number) # звернення до статичних членів-даних
my_pet.public_member() # виклик не статичного методу
Pet.public_static_member() # виклик статичного методу
del my_pet # видалення об’єкту
```

Спадкування

Одинарне спадкування

Спадкування призначене для розширення функціональності існуючих класів шляхом утворення нових класів на базі вже існуючих. У Python усі класи спадкуються неявно від класу `object`. Python дозволяє реалізовувати як одинарне так і множинне спадкування. Для реалізації спадкування класи, які слід успадкувати вказуються у круглих дужках через кому після назви класу, який оголошується:

```
class <назва_класу> (<базовий_клас_1>, <базовий_клас_2>, ...):
    <тіло класу>
```

При одинарному спадкуванні похідний клас спадкує один базовий клас. Наприклад:

```
class Animal:
     def __init__(self, breed, age, height):
        self.__breed = breed # приватна змінна-член класу
        self.__age = age
        self.__height = height

     def get_breed(self):
        return self.__breed

     def get_age(self):
        return self.__age

     def get_height(self):
        return self.__heihgt

class Dog(Animal):
     def __init__(self, breed, age, height, voice):
        super().__init__(breed, age, height)
        self.__voice = voice

     def voice(self):
        print(self.__voice)
```

Явне звернення до членів будь-якого базового класу в межах ієрархії спадкування виконуєтсья за допомогою назви базового класу та його :

```
<назва_базового_класу>.<член_класу>.
```

Для виклику конструктора базового класу у тілі конструктора похідного класу слід викликати функцію `super()`, який повертає проксі об'єкт базового класу, та явно викликати з-під нього конструктор базового класу. Також функцію `super()` слід використовувати для звертання до інших членів базового класу.

Слід мати на увазі, що конструктор базового класу автоматично НЕ викликається з-під конструктора похідного класу як у більшості інших об'єктно-орієнтованих мов програмування. Його треба викликати самостійно, додавши виклик `super().__init__()` у метод `__init__()` похідного класу.

Множинне спадкування

Особливістю множинного спадкування є наявність кількох класів від яких одночасно відбувається спадкування. Тут можливі 2 випадки: Класи, що спадкуються від незалежних між собою класів та множинне спадкування від залежних між собою класів.

Множинне спадкування від незалежних між собою класів

При множинному спадкуванню від незалежних між собою класів усі базові класи по кожній гілці спадкування мають унікальних предків.

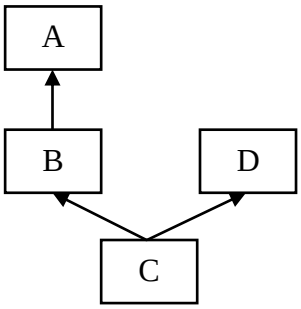

Рис. 15.1. Варіант множинного спадкування від незалежних між собою класів.

У звязку з тим, що конструктор (метод `__init()__`) у Python не викликає автоматично конструкторів базового класу, то їх слід викликати самостійно.

Якщо базовий клас не містить полів, які слід ініціалізовувати, то виклик конструктора цього базового класу можна пропустити, проте це не рекомендується робити.

Якщо методи `__init()__` різних класів приймають різні параметри, то:

1. Конструктори усіх класів необхідно реалізувати таким чином, щоб вони додатково приймали параметром словник параметрів у форматі ключ-значення (`**kwargs`). У цей словник попадатимуть усі параметри, які передані конструктору, але не згадані явно перед `**kwargs`.
2. Викликати конструктори базових класів для усіх класів в ієрархії. Це необхідно для коректної роботи механізму MRO (розглядається далі).

Приклад:

```
class Animal:
    def __init__(self, breed, age, height, **kwargs):
        super().__init__(**kwargs) # обов’язково має бути,
оскільки клас object спадкується 2 рази у ієрархії.
        self.__breed = breed
        self.__age = age
        self.__height = height
        print(f"Animal breed={self.__breed}, age={self.__age},
height={height}")

    def get_breed(self):
        return self.__breed

    def get_age(self):
        return self.__age

    def get_height(self):
        return self.__heihgt


class PetInfo:
    def __init__(self, address, vaccination, **kwargs):
        super().__init__(**kwargs) # обов’язково має бути,
оскільки клас object спадкується 2 рази у ієрархії.
        self.__address = address
        self.__vaccination = vaccination
        print(f"PetInfo address={self.__address},
vaccination={self.__vaccination}")

    def get_address(self):
        return self.__address

    def get_vaccination(self):
        return self.__vaccination

class DogPet(Animal, PetInfo):
    def __init__(self, breed, age, height, voice, address,
vaccination):
        super().__init__(breed=breed, age=age, height=height,
address=address, vaccination=vaccination)
        self.__voice = voice
        print(f"DogPet voice={self.__voice}")

    def voice(self):
```

```
        print(self.__voice)

x=DogPet("breed", 15, 10, "Bow", "address", True)
```

Вивід на екран:

```
PetInfo address=address, vaccination=True
Animal breed=breed, age=15, height=10
DogPet voice=Bow
```

Множинне спадкування від залежних між собою класів

При множинному спадкуванню від залежних між собою класів принаймі два базові класи по різних гілках спадкування мають принаймі одного спільного предка.

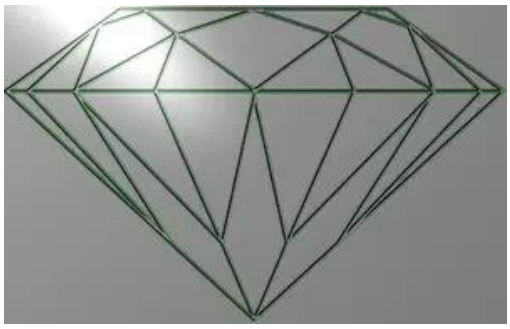

Рис.15.2. «Діамант смерті»

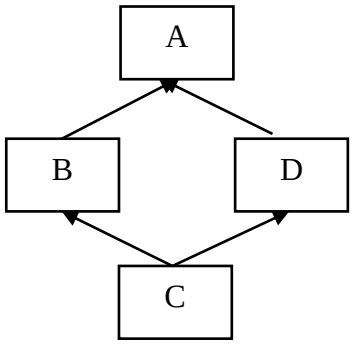

Рис. 15.3. Варіант множинного спадкування від залежних між собою класів.

Для коректної побудови усіх об’єктів при даному виді множинного спадкування для усіх класів слід задати метод `__init__()`, в якому здійснити виклик контруктора базового класу, викликавши `super().__init__()`.

Приклад:

```
class A:
     def __init__(self):
          super().__init__() # Можна опустити оскільки клас
object спадкується 1 раз у ієрархії через цей клас
          print("__init__() of A called")
```

```
class B(A):
     def __init__(self):
          super().__init__()
          print("__init__() of B called")

class C(A):
     def __init__(self):
          super().__init__()
          print("__init__() of C called")


     def foo(self):
          print("foo() of C called")


class D(B,C):
     def __init__(self):
          super().__init__()
          print("__init__() of D called")

x=D()
```

Вивід на екран:

```
__init__() of A called
__init__() of C called
__init__() of B called
__init__() of D called
```

Порядок виведення методів

Порядок виведення методів (Method Resolution Order - MRO) – це спосіб за допомогою якого деревовидна структура ієрархії спадкування класів представляється лінійно у вигляді списку. Він дозволяє Python з'ясувати, з якого предка потрібно викликати метод, якщо він не виявлений безпосередньо в класі-нащадку. Якщо в кожного нащадка є лише один предок, то завдання тривіальне. Відбувається висхідний пошук у всій ієрархії. Якщо ж використовується множинне спадкування, то можна зіткнутися зі специфічними проблемами для вирішення яких розробили алгоритм MRO C3. Суть алгоритму – отримати лінійний список класів, які задають послідовність пошуку методів. Даний список зберігається у полі `__mro__` класу. Наприклад, для попередньої ієрархії класів `D.__mro__` містить список (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>). Тобто, при виклику методу `foo()` його пошук спочатку

відбуватиметься у класі D, потім у класі B і аж потім у класі C. Якщо він не буде знайдений, то згенерується помилка `Attribute Error`.

Бувають випадки коли MRO неможливо побудувати. Наприклад, при наступному оголошенні:

```
class C(A, B): pass
class D(B, A): pass
class E(C, D): pass
```

У даному випадку конфлікт залишається нерозв'язаним, оскільки в оголошені класу C клас A стоїть перед B, а у оголошенні класу D – навпаки. Для розв'язання цієї колізії є 2 способи:

1. Переглянути структуру класів, оскільки скоріш за все у ній є помилка. У даному прикладі достатньо буде поміняти послідовність класів A і B місцями у одному з класів C або D.
2. Створити метаклас перевизначивши метод `mro` класу `type`, який повертатиме потрібну послідовність класів і застосувати створений метаклас при оголошенні класу E.

```
class MetaMRO(type):
    def mro(cls):
        return (cls, A, B, C, D, object)

class E(C, D, metaclass = MetaMRO): pass
```

У Python до версії 2.2, класи неявно не спадкували клас `object` і MRO був простішим: пошук відбувався у всіх батьківських класах зліва напарво на максимальну глибину.

Класи-домішки

Домішки або Mixin – це шаблон проектування, в якому деякий метод базового класу використовує метод, який не визначається у цьому класі. Цей метод призначений для реалізації іншим базовим класом. Клас-домішка або mixin class – це клас, який використовується у цьому шаблоні, надаючи функціональні можливості (методи), але не призначений для самостійного використання у вигляді об'єктів класу. В ідеальному випадку класи-домішки не мають власної ієрархії спадкування і не мають полів, а мають лише методи.

У Python немає ніякого спеціального синтаксису для підтримки класів-домішок, тому їх легко сплутати зі звичайними класами, але при цьому між ними є велика різниця. Для того щоб позначити, що клас є класом-домішкою, використовують суфікс `Mixin` в кінці назви класу.

Класи-домішки використовуються при реалізації множинного спадкування і не лише у Python. Проте у Python реалізація множинного спадкування з використаням класів-домішок має свою специфіку пов'язану з MRO – класи-домішки мають бути вказані на

початку списку базових класів, інакше методи класу-домішки можуть бути перекриті методами інших базових класів, що не є класами-домішками.

Розглянемо приклад множинного спадкування з використанням класу-домішки.

```
class Entity:
    def __init__(self, pos_x, pos_y):
        self.pos_x = pos_x
        self.pos_y = pos_y

class SquareMixin:
    def add_size(self, size_x):
        self.size_x = size_x
        self.size_y = size_x

    def perimeter(self):
        return self.size_x * 4

    def square(self):
        return self.size_x * self.size_x

class SquareEntity(SquareMixin, Entity):
    def print_square(self):
        print(f'Square size = {self.square()}')

--------------------------------------------------
square = SquareEntity(5, 4)
square.add_size(20)
square.print_square()
```

Тут похідний клас `SquareEntity()` отримує від класу-домішки `SquareMixin()` методи додавання розміру квадрата, а також обчислення його периметра та площі. Ця поведінка спрощує дерево спадкування `SquareEntity()`, що дозволяє використовувати клас `Entity()` як батьківський для інших фігур без необхідності успадковувати методи, які не потрібні (наприклад, для кола).

Якщо необхідно явно передати параметр size при ініціалізації класу `SquareEntity()` з прикладу вище, необхідно скористатися функцією `super()`.

```
class Entity:
    def __init__(self, pos_x, pos_y):
        self.pos_x = pos_x
        self.pos_y = pos_y

class SquareMixin:
    def __init__(self, size, **kwargs):
```

```
        super().__init__(**kwargs)
        self.size_x = size
        self.size_y = size

    def perimeter(self):
        return self.size_x * 4

    def square(self):
        return self.size_x * self.size_x


class SquareEntity(SquareMixin, Entity):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def print_square(self):
        print(f'Square size = {self.square()}')


# Всі параметри передаються як пара ключ-значення
square = SquareEntity(pos_x=5, pos_y=4, size=20)
square.print_square()
```

Висновок

У лекції розглянуто особливості реалізації об’єктно-орієнтованого програмування у Python – класи, об’єкти, методи, поля, принципи організації інкапсуляції, одинарне і множинне спадкування, класи-домішки та їх роль при реалізації множинного спадкування.

Список рекомендованої літератури

1. Хоростманн К. Java. Библиотека профессионала, том 1. Основы. 11-е издание / К. Хоростманн; пер. с англ. – К.: Диалектика, 2019. – 864 с.: ил.
2. Хоростманн К. Java. Библиотека профессионала, том 2. Расширенные средства программирования 11-е издание / К. Хоростманн, Г. Корнелл; пер. с англ. – К.: Диалектика, 2020. – 864 с.: ил.
3. Эккель Б. Философия Java. Библиотека программиста. 4-е издание. – СПб: Питер, 2012. – 640 с.: ил. – (Серия «Библиотека программиста»)
4. Шилдт Г. Java: руководство для начинающих. 7-е издание. – М.: ООО «И.Д. Вильямс», 2019. – 816 с.: ил.
5. Шилдт Г. Java. Полное руководство. Том 1. 10-е издание. – К.: Диалектика, 2020. – 730 с.: ил.
6. Сьерра К., Бэйтс Б. Изучаем Java. / Кэтти Сьерра, Берт Бэйтс. – М.: Эксмо, 2012. – 720 с.: ил.
7. Хабибуллин И. Java 7 в подлиннике. – СПб: BHV, 2012. – 768 с.
8. Васильев А. Самоучитель Java с примерами и программами (+ CD). – СПб: Наука и техника, 2011. – 352 с.
9. Java SE Documentation at a Glance [електронний ресурс]. – Режим доступу до документації: http://www.oracle.com/technetwork/java/javase/documentation/index.html
10. Java SE 8: Lambda Quick Start [електронний ресурс]. – Режим доступу: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
11. Oracle Java SE Support Roadmap [електронний ресурс]. – Режим доступу: https://www.oracle.com/java/technologies/java-se-support-roadmap.html
12. Java Versions and Features [електронний ресурс]. – Режим доступу: https://www.marcobehler.com/guides/a-guide-to-java-versions-and-features
13. Java Version History and Features [електронний ресурс]. – Режим доступу: https://howtodoinjava.com/java-version-wise-features-history
14. Python Documentation [електронний ресурс]. – Режим доступу: https://docs.python.org/3/contents.html
15. Multiple Inheritance Overview [електронний ресурс]. – Режим доступу: https://realpython.com/python-super/#multiple-inheritance-overview

НАВЧАЛЬНЕ ВИДАННЯ

Олексів Максим Васильович

КРОСПЛАТФОРМНІ ЗАСОБИ ПРОГРАМУВАННЯ

КОНСПЕКТ ЛЕКЦІЙ

для студентів Інституту комп'ютерних технологій, автоматики та метрології

Редактор

Комп'ютерне верстання

Здано у видавництво . Підписано до друку
Формат 70x100/16. Папір офсетний. Друк на різографі
Умовн. друк. арк. Обл..-вид. арк..
Тираж прим. Зам..

Видавництво Національного університету "Львівська політехніка"
Реєстраційне свідоцтво ДК №751 від 27.12.2001 р.

Поліграфічний центр Видавництва
Національного університету "Львівська політехніка"

Вул. Ф. Колесси, 2. Львів, 79000