Лабораторна робота №5

Введення-виведення. Exceptions. Interfaces

Теоретичні відомості

Інтерфейс у мові програмування Java визначається як абстрактний тип, що використовується для визначення поведінки класу. Інтерфейс в Java — це схема поведінки. Інтерфейс Java містить статичні константи та абстрактні методи.

Interfaces

Що таке інтерфейси в Java?

Інтерфейс — це механізм, за допомогою якого укладається договір між двома сторонами: постачальником сервісу та класами, які хочуть, щоб їхні об’єкти можна було використовувати із цим сервісом.

Розглянемо сервіс, який опрацьовує послідовності цілих чисел, обчислюючи середнє арифметичне перших n значень:

```
public static double average(IntSequence seq, int n) {
    // ...
}
```

Такі послідовності можуть мати різні форми. Наведемо декілька прикладів:

- Послідовність цілих чисел, наданих користувачем
- Послідовність випадкових цілих чисел
- Послідовність простих чисел
- Послідовність елементів у цілочисельному масиві
- Послідовність кодових точок у рядку
- Послідовність цифр числа

Ми хочемо реалізувати єдиний механізм для роботи з усіма цими типами послідовностей. Спочатку з’ясуємо, що є спільного між цілочисельними послідовностями. Як мінімум, для роботи з послідовністю потрібно два методи:

- Перевірити, чи є наступний елемент
- Отримати наступний елемент

Щоб оголосити інтерфейс, ви надаєте заголовки методів, наприклад, так:

```
public interface IntSequence {
    boolean hasNext();
    int next();
}
```

Вам не потрібно реалізовувати ці методи, але ви можете вказати їх реалізацію за замовчуванням, можливість створення "Методів за замовчуванням" з'явились у Java 8 та буде розглянута далі. Якщо не надано жодної реалізації, ми говоримо, що метод є *абстрактним*.

Усі методи інтерфейсу автоматично є `public`. Тому нема потреби оголошувати `hasNext()` та `next()` як `public`. Деякі програмісти роблять це все одно для більшої наочності.

Методів, що описані в інтерфейсі, достатньо для реалізації методу обчислення середнього значення:

```
public static double average(IntSequence seq, int n) {
    int count = 0;
    double sum = 0;
    while (seq.hasNext() && count < n) {
        count++;
        sum += seq.next();
    }
    return count == 0 ? 0 : sum / count;
}
```

Реалізація інтерфейсу

Тепер подивімося на іншу сторону медалі: класи, які хочуть використовувати метод average. Їм потрібно реалізувати інтерфейс IntSequence. Ось такий клас:

```
public class SquareSequence implements IntSequence {
    private int i;
    public boolean hasNext() {
        return true;
    }
    public int next() {
        i++;
        return i * i;
    }
}
```

Існує нескінченна кількість квадратів натуральних чисел, і об'єкт цього класу видає їх усі, по одному. Ключове слово `implements` вказує на те, що клас `SquareSequence` має намір відповідати інтерфейсу `IntSequence`.

TIP

Клас, що реалізує інтерфейс, повинен оголосити методи інтерфейсу як public. Інакше, за замовчуванням, вони будуть доступні з пакета. Проте, оскільки інтерфейс вимагає загальнодоступного доступу, компілятор повідомить про помилку.

Цей код отримує середнє значення перших 100 квадратів:

```
var squares = new SquareSequence();
double avg = average(squares, 100);
```

Існує багато класів, які можуть реалізовувати інтерфейс IntSequence. Наприклад, цей клас повертає скінченну послідовність, а саме цифри натурального числа, починаючи з найменшої:

```
public class DigitSequence implements IntSequence {
    private int number;
    public DigitSequence(int n) {
        number = n;
    }
    public boolean hasNext() {
        // Цей і наступний методи оголошені в інтерфейсі
        return number != 0;
    }
    public int next() {
        int result = number % 10;
        number /= 10;
        return result;
    }
    public int rest() {
        // Цей метод не оголошений в інтерфейсі
        return number;
    }
}
```

Об'єкт `new DigitSequence(1729)` повертає цифри 9 2 7 1 до того, як `hasNext()` поверне `false`.

NOTE

У класах `SquareSequence` та `DigitSequence` реалізовано усі методи інтерфейсу `IntSequence`. Якщо клас реалізує тільки частину методів, то він повинен бути оголошений з модифікатором `abstract`

Перетворення до типу інтерфейсу

Цей фрагмент коду обчислює середнє значення послідовності цифр:

```
IntSequence digits = new DigitSequence(1729);
double avg = average(digits, 100);
```

```
// тут є всього 4 значення у послідовності
```

Змінна `digits` має тип `IntSequence`, а не `DigitSequence`. Змінна типу `IntSequence` посилається на об'єкт деякого класу, який реалізує інтерфейс `IntSequence`. Змінній завжди можна присвоїти об'єкт, тип якого реалізує інтерфейс, або передати її методу, що очікує такий інтерфейс.

Визначення. Тип S є супертипом типу T (підтипу), коли будь-яке значення підтипу може бути присвоєне змінній супертипу без перетворення. Наприклад, інтерфейс `IntSequence` є супертипом класу `DigitSequence`.

NOTE	Попри те, що можна оголошувати змінні інтерфейсного типу, ви ніколи не можете мати об'єкт, тип якого є інтерфейсом. Усі об'єкти є екземплярами класів.

Приведення типу та оператор `instanceof`

Іноді вам потрібно зворотне перетворення — від супертипу до підтипу. Тоді ви використовуєте приведення типу. Наприклад, якщо ви знаєте, що об'єкт, який зберігається в `IntSequence`, насправді є `DigitSequence`, можна виконати перетворення типу таким чином:

```
IntSequence sequence = ...;
DigitSequence digits = (DigitSequence) sequence;
System.out.println(digits.rest());
```

У цьому сценарії приведення було необхідним, оскільки rest є методомDigitSequence, а не IntSequence.

Ви можете привести об'єкт лише до його фактичного класу або одного з його супертипів. Якщо ви помилитеся, виникне помилка під час компіляції або виняток приведення класу:

```
String digitString = (String) sequence;
// Це не буде працювати — IntSequence не є супертипом для String
SquareSequence squares = (SquareSequence) sequence;
// Може спрацювати, кидає a ClassCastException якщо ні
```

Щоб уникнути винятку, можна спочатку перевірити, чи об'єкт має потрібний тип, використовуючи оператор instanceof. Вираз `object instanceof Type` повертає `true`, якщо об'єкт є екземпляром класу, який має `Type` як супертип. Рекомендується виконувати цю перевірку перед використанням приведення типів.

```
if (sequence instanceof DigitSequence) {
    DigitSequence digits = (DigitSequence) sequence;
    ...
}
```

Оператор instanceof є null-безпечним: Вираз `obj instanceof Type` є хибним, якщо `obj` дорівнює `null`. Адже `null` не може бути посиланням на об'єкт будь-якого типу.

Успадкування (наслідування) інтерфейсів

Інтерфейс може розширювати інший, вимагаючи або надаючи додаткові методи на додаток до початкових. Наприклад, `Closeable` - це інтерфейс із єдиним методом:

```
public interface Closeable {
    void close();
}
```

Інтерфейс `Channel` розширює цей інтерфейс:

```
public interface Channel extends Closeable {
    boolean isOpen();
}
```

Клас, що реалізує інтерфейс `Channel`, повинен надавати обидва методи, а його об'єкти можуть бути конвертовані в обидва типи інтерфейсів.

Реалізація декількох інтерфейсів

Клас може реалізовувати будь-яку кількість інтерфейсів. Наприклад, клас `FileSequence`, який читає цілі числа з файлу, може реалізувати інтерфейс `Closeable` на додаток до `IntSequence`:

```
public class FileSequence implements IntSequence, Closeable {
    //...
}
```

Тоді клас `FileSequence` має як `IntSequence`, так і `Closeable` як супертипи.

Константи

Будь-яка змінна, визначена в інтерфейсі, автоматично стає `public static final`. Наприклад, інтерфейс `SwingConstants` визначає константи для напрямків за компасом:

```
public interface SwingConstants {
    int NORTH = 1;
    int NORTH_EAST = 2;
    int EAST = 3;
    // ...
}
```

Ви можете звертатися до них за кваліфікатором SwingConstants.NORTH. Якщо ваш клас

вирішив реалізувати інтерфейс SwingConstants, ви можете опустити кваліфікаторSwingConstants і просто написати NORTH. Однак це не є загальноприйнятою ідіомою. Набагато краще використовувати enums для наборів констант

NOTE	В інтерфейсі не може бути змінних. Інтерфейс визначає поведінку, а не стан об’єкта.

static, default та private методи

У попередніх версіях Java всі методи інтерфейсу були повинні бути абстрактними — тобто без тіла. Зараз ви можете додавати три типи методів з конкретною реалізацією: статичні (static), за замовчуванням (default) та приватні (private) методи.

Статичні (static) методи

Ніколи не було технічної причини, чому інтерфейс не міг би мати статичних методів, але вони не вписувалися в уявлення про інтерфейси як про абстрактні специфікації. Зараз це мислення змінилося. Зокрема, фабричні методи мають багато сенсу в інтерфейсах. Наприклад, інтерфейс IntSequence може мати статичний метод digitsOf, який генерує послідовність цифр заданого цілого числа:

```
IntSequence digits = IntSequence.digitsOf(2014);
```

Метод повертає екземпляр деякого класу, що реалізує інтерфейс IntSequence, але тому, хто його викликає, не обов’язково знати, який саме.

```
public interface IntSequence {
    //...
    static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }
}
```

Default методи

Ви можете надати реалізацію за замовчуванням для будь-якого методу інтерфейсу. Ви повинні позначити такий метод модифікатором `default`.

```
public interface IntSequence {
    default boolean hasNext() {
        return true; // За замовчуванням, послідовності нескінчені
    }

    int next();
}
```

Клас, що реалізує цей інтерфейс, може перевизначити метод `hasNext` або успадкувати реалізацію за замовчуванням.

Важливим застосуванням методів за замовчуванням є еволюція інтерфейсів. Розглянемо для прикладу інтерфейс Collection, який є частиною Java вже багато років. Припустимо, що колись ви створили клас

```
public class Bag implements Collection {}
```

Пізніше, у Java 8, до інтерфейсу було додано метод `stream`. Припустимо, що метод `stream` не був би методом за замовчуванням. Тоді клас `Bag` більше не компілюється, оскільки він не реалізує новий метод. Додавання методу не за замовчуванням до інтерфейсу є несумісним з вихідними текстами. Але припустимо, що ви не перекомпілюєте клас, а просто використовуєте старий JAR-файл, який його містить. Клас все одно завантажиться, навіть з відсутнім методом. Програми можуть створювати екземпляри Bag, і нічого поганого не станеться (додавання методу до інтерфейсу є бінарно сумісним). Однак, якщо програма викликає метод `stream` в екземплярі Bag, виникне помилка `AbstractMethodError`.

Зробивши метод методом за замовчуванням, ми розв'яжемо обидві проблеми. Клас Bag знову скомпілюється. А якщо клас завантажується без перекомпіляції та викликається метод `stream` в екземплярі Bag, викликається метод `Collection.stream`.

Приватні (private) методи

Методи в інтерфейсі можуть бути приватними. Приватний (private) метод може бути статичним або методом екземпляра, але він не може бути методом за замовчуванням, оскільки такі методи можна перевизначити. Оскільки приватні методи можуть бути використані лише в методах самого інтерфейсу, їх використання обмежується тим, що вони є допоміжними методами для інших методів інтерфейсу. Наприклад, нехай клас IntSequence надає методи:

```
static of(int a)
static of(int a, int b)
static of(int a, int b, int c)
```

Потім кожен із цих методів може викликати допоміжний метод

```
private static IntSequence makeFiniteSequence(int... values) { ... }
```

Exceptions - винятки (виключення)

У багатьох програмах робота з непередбачуваними ситуаціями може бути складнішою, ніж реалізація сценаріїв "щасливого дня". Як і більшість сучасних мов програмування, Java має надійний механізм обробки винятків для передачі керування від місця збою до компетентного обробника.

Ключовими моментами щодо винятків є наступні: 1. Коли ви генеруєте виняток, управління передається найближчому обробнику винятків. 2. У Java виняткові ситуації, що перевіряються, відстежуються компілятором. 3. Для обробки винятків використовується конструкція try/catch. 4. Інструкція try-with-resources автоматично закриває ресурси після нормального виконання або при виникненні винятку. 5. Використовуйте конструкцію try/finally для обробки інших дій, які мають відбутися незалежно від того, чи відбулося нормальне виконання, чи трапився виняток. 6. Ви можете перехопити та прокинути виняток або зв'язати його з іншим винятком.

Обробка винятків

Що повинен робити метод, коли він стикається із ситуацією, в якій він не може виконати свій контракт? Традиційною відповіддю було те, що метод повинен повернути деякий код помилки. Але це обтяжливо для програміста, який викликає метод. Він зобов'язаний перевірити наявність помилок, і якщо він не може з ними впоратися, то повернути код помилки об'єкта, що його викликав. Не дивно, що програмісти не завжди перевіряли та поширювали коди повернення, і помилки залишалися невиявленими, спричиняючи хаос пізніше.

Замість того щоб коди помилок спливали вгору по ланцюжку викликів методів, Java підтримує обробку винятків, коли метод може сигналізувати про серйозну проблему, "згенерувавши" виняток. Один з методів у ланцюжку викликів, але не обов'язково той метод, що безпосередньо викликається, відповідає за обробку винятку, "перехоплюючи" його. Фундаментальною перевагою обробки винятків є те, що вона розділяє процеси виявлення та обробки помилок.

Кидання винятків (Exceptions)

Метод може опинитися в ситуації, коли він не може виконати поставлену задачу. Можливо, відсутній необхідний ресурс, або він був викликаний з неузгодженими аргументами. У такому випадку найкраще згенерувати виняток. Припустимо, ви реалізуєте метод, який повертає випадкове ціле число між двома границями:

```
public static int randInt(int low, int high) {
    return low + (int) (Math.random() * (high - low + 1));
}
```

Що станеться, якщо хтось викличе `randInt(10, 5)`? Намагатися виправити це, ймовірно, не найкраща ідея, оскільки користувач міг переплутати декілька способів виклику. Натомість згенеруйте відповідний виняток:

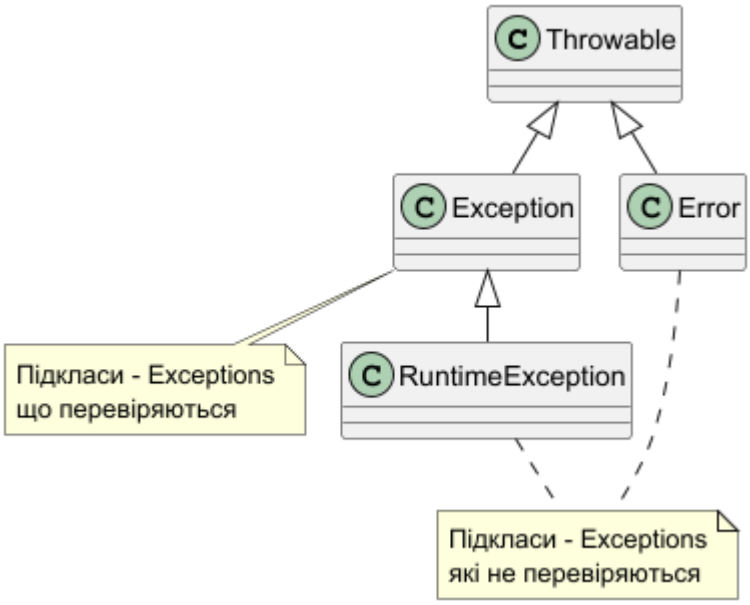
```
if (low > high)
    throw new IllegalArgumentException("low should be <= high but low is %d and high
is %d".formatted(low, high));
```

Коли виконується оператор `throw`, звичайний потік виконання негайно переривається. Метод `randInt` припиняє виконання і не повертає значення тому, хто його викликав. Замість

цього керування передається обробнику

Ієрархія Exceptions

Усі винятки є підкласами класу Throwable. Підкласи класу Error — це винятки, які генеруються, коли трапляється щось фатальне, із чим програма не може впоратися, наприклад, вичерпання пам'яті. З помилками не так багато можна зробити, окрім як повідомити користувачеві, що щось пішло не так.



Винятки, про які повідомляється програміст, є підкласами класу Exception. Ці винятки поділяються на дві категорії:

- Винятки, які не перевіряються (Unchecked) є підкласами класу RuntimeException.
- Всі інші винятки є винятками, що перевіряються (Checked). Їх також називають контрольованими винятками. Програмісти повинні або перехоплювати такі винятки, або оголошувати їх у заголовку методу. Компілятор перевіряє правильність обробки цих винятків

Винятки, що перевіряються, використовуються в ситуаціях, коли слід очікувати збою. Найпоширенішою причиною збоїв є уведення та виведення даних. Файли можуть бути пошкоджені, а мережеві з'єднання — розірвані. Ряд класів винятків розширюють IOException, і вам слід використовувати відповідний клас для повідомлення про будь-які помилки, з якими ви зіткнулися. Наприклад, коли файл, який мав би бути там, виявляється відсутнім, згенеруйте виняток FileNotFoundException.

Винятки, що не перевіряються, указують на логічні помилки, спричинені програмістами, а не неминучими зовнішніми ризиками. Наприклад, виняток NullPointerException не перевіряється. Практично будь-який метод може згенерувати такий виняток, і програмістам не варто витрачати час на його перехоплення. Натомість вони повинні в першу чергу переконатися, що жодне нульове посилання не буде розіменоване.

Чому така різниця? Причина в тому, що можна перевірити, чи є рядок дійсним цілим числом перед викликом Integer.parseInt, але неможливо дізнатися, чи можна завантажити

клас, доки ви не спробуєте його завантажити.

Java API надає багато класів винятків, таких як IOException, IllegalArgumentException тощо. Однак, якщо жоден зі стандартних класів винятків не підходить для вашої мети, ви можете створити власний, розширивши Exception, RuntimeException або інший наявний клас винятків.

Оголошення винятків, що перевіряються

Будь-який метод, який може згенерувати контрольований виняток, повинен бути оголошений у заголовку методу з оператором throws:

```
public void write(Object obj, String filename) throws IOException,
ReflectiveOperationException
```

Треба пам'ятати, що винятки, які може згенерувати метод через інструкцію `throw` або через те, що він викликає інший метод з оператором `throws`, треба вказувати у розділі `throws`. Ви також можете об'єднати винятки у спільний суперклас. Чи є це гарною ідеєю, залежить від винятків. Наприклад, якщо метод може генерувати декілька підкласів винятків `IOException`, то має сенс покрити їх усі в класі `throws IOException`. Але якщо винятки не пов'язані між собою, не варто об'єднувати їх у `throws Exception` - це суперечить меті перевірки винятків.

NOTE Деякі програмісти вважають, що соромно визнавати, що метод може згенерувати виняток. Чи не краще було б обробити його замість цього? Насправді все навпаки. Ви повинні дозволити кожному винятку знайти свій шлях до компетентного обробника. Золоте правило винятків звучить так: "Викинь рано, злови пізно".

При перевизначенні метода, він не може генерувати більше перевірених винятків, ніж ті, що оголошені в методі суперкласу. Наприклад, якщо ви розширите метод `write` наведений вище, перевизначений метод може згенерувати меншу кількість винятків:

```
public void write(Object obj, String filename)
    throws FileNotFoundException
```

NOTE Якщо метод суперкласу не має оператора throws, то жоден перевизначений метод не може згенерувати контрольований виняток.

Перехоплення винятків

Щоб перехопити виняток, налаштуйте блок try. У найпростішому вигляді це виглядає так:

```
try {
    // statements
} catch (ExceptionClass ex) {
    // handler
```

```
}
```

Якщо під час виконання операторів у блоці try виникає виняток заданого класу, управління передається обробнику. Змінна винятку (ex у нашому прикладі) посилається на об'єкт винятку, який обробник може перевірити за бажанням. Існує дві модифікації цієї базової структури. Ви можете мати декілька обробників для різних класів винятків:

```
try {
    // statements
} catch (ExceptionClass1 ex) {
    // handler1
} catch (ExceptionClass2 ex) {
    // handler2
} catch (ExceptionClass3 ex) {
    // handler3
}
```

Розділи перехоплення узгоджуються зверху вниз, тому найбільш специфічні класи винятків мають бути першими. Крім того, ви можете використовувати один обробник для декількох класів винятків:

```
try {
    // statements
} catch (ExceptionClass1 | ExceptionClass2 | ExceptionClass3 ex) {
    // handler
}
```

У цьому випадку обробник може викликати лише ті методи на змінній винятку, які належать до всіх класів винятків.

Оператор try-з-ресурсами

Однією з проблем, пов'язаних з обробкою винятків, є управління ресурсами. Припустимо, ви записуєте у файл і закриваєте його, коли закінчите:

```
String[] lines = ...;
var out = new PrintWriter("output.txt");
for (String line : lines) {
    out.println(line.toLowerCase());
}
out.close();
```

Цей код таїть у собі приховану небезпеку. Якщо будь-який метод генерує виняток, виклик out.close() ніколи не відбувається. Це погано. Вихідні дані можуть бути втрачені, або, якщо виняток спрацьовує багато разів, у системі можуть закінчитися дескриптори файлів. Подолати таку проблему може спеціальна форма оператора try. Ви можете вказати ресурси

в заголовку оператора try. Ресурс повинен належати класу, що реалізує інтерфейс AutoCloseable. Оголосити змінні можна в заголовку блоку try:

```
String[] lines = ...;
try (var out = new PrintWriter("output.txt")) {
    for (String line : lines)
    out.println(line.toLowerCase());
}
```

Крім того, ви можете використати раніше оголошені змінні, якщо вони final, або effectively final, в заголовку:

```
String[] lines = ...;
var out = new PrintWriter("output.txt");
try (out) {
    for (String line : lines)
    out.println(line.toLowerCase());
}
```

Коли блок try завершує роботу, або через те, що його кінець досягається нормально, або через те, що генерується виняток, викликаються методи закриття об'єктів ресурсу. Наприклад:

```
try (var out = new PrintWriter("output.txt")) {
    for (String line : lines) {
        out.println(line.toLowerCase());
    }
} // out.close() буде викликано тут
```

Можна оголосити кілька ресурсів, розділених крапкою з комою. Ось приклад з двома оголошеннями ресурсів:

```
try (var in = new Scanner(Path.of("words.txt"));
        var out = new PrintWriter("output.txt")) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

Ресурси закриваються в порядку, зворотному їх ініціалізації, тобто out.close() викликається перед in.close(). Припустимо, що конструктор PrintWriter генерує виняток. Тепер in вже ініціалізовано, а out – ні. Оператор try робить правильні речі: викликає in.close() і розповсюджує виняток. Деякі близькі методи можуть генерувати винятки. Якщо це станеться, коли блок try завершиться нормально, виняток буде передано абоненту. Однак, якби було згенеровано інший виняток, що призвело б до виклику близьких методів ресурсів, і один з них генерував виняток, цей виняток, швидше за все, був би менш

важливим, ніж початковий. У цій ситуації початковий виняток генерується повторно, а винятки виклику close перехоплюються та прикріплюються як "придушені" винятки. Це дуже корисний механізм, який було б нудно виконувати вручну. Коли ви перехоплюєте первинний виняток, ви можете отримати вторинні винятки, викликавши метод getSuppressed:

```
try {
    // ...
} catch (IOException ex) {
    Throwable[] secondaryExceptions = ex.getSuppressed();
    // ...
}
```

Якщо ви хочете реалізувати такий механізм самостійно в тій рідкісній ситуації, коли ви не можете використовувати оператор try-with-resources, викличте ex.addSuppressed(secondaryException). Оператор try-with-resources може за бажанням мати пункти catch, які перехоплюють будь-які винятки в інструкції.

Розділ finally

Як ви бачили, оператор try-with-resources автоматично закриває ресурси, незалежно від того, чи відбувається виняток, чи ні. Іноді потрібно очистити щось, що не підлягає автозакриттю. У цьому випадку використовуйте розділ finally:

```
try {
    // Do work
} finally {
    // Clean up
}
```

Розділ finally виконується, коли блок try добігає кінця, або успішно, або через виняток. Ви хочете переконатися, що ці дії виконуються незалежно від того, які винятки можуть бути викликані. Вам слід уникати генерування винятку в блоці finally. Якщо тіло блоку try було припинено через виняток, воно маскується винятком у блоці finally. Механізм придушення, який ви бачили в попередньому розділі, працює лише для операторів try-with-resources.

Аналогічно, розділ finally не повинен містити оператора return. Якщо тіло блоку try також має оператор return, той, що в блоці finally, замінює значення, що повертається. Розглянемо такий приклад:

```
public static int parseInt(String s) {
    try {
        return Integer.parseInt(s);
    } finally {
        return 0; // Error
    }
```

```
}
```

Це виглядає так, ніби у виклику `parseInt("42")`, тіло блоку try повертає ціле число 42. Однак пункт `finally` виконується до того, як метод фактично повертає і змушує метод повертати 0, ігноруючи початкове значення, що повертається. І може статись ще гірше. Розглянемо виклик `parseInt("zero")`. Метод `Integer.parseInt` генерує виняток `NumberFormatException`. Потім виконується пункт `finally`, а оператор return поглинає виняток!

Тіло розділу `finally` призначене для очищення ресурсів. Не розміщуйте оператори, які змінюють потік керування (`return`, `throw`, `break`, `continue`), всередині розділу `finally`.

Введення-виведення: Input/Output Streams, Readers та Writers

У Java API джерело, з якого можна зчитувати байти, називається вхідним потоком (InputStream). Байти можуть надходити з файлу, мережевого з'єднання або масиву в пам'яті. Аналогічно, кінцевим пунктом призначення для байтів є вихідний потік (OutputStream). На противагу цьому, Readers та Writers споживають і продукують послідовності символів.

Отримання потоків

Найпростіший спосіб отримати потік з файлу — це статичні методи:

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

Тут path є екземпляром класу `Path`. Він описує шлях у файловій системі. Якщо у вас є URL-адреса, ви можете прочитати її вміст із потоку вхідних даних, який повертає метод openStream класу URL:

```
var url = new URL("http://berkut.mk.ua/index.html");
InputStream in = url.openStream();
```

Клас `ByteArrayInputStream` дозволяє читати з масиву байтів.

```
byte[] bytes = ...;
var in = new ByteArrayInputStream(bytes);
// Read from in
```

І навпаки, щоб відправити вихідні дані в байтовий масив, використовуйте `ByteArrayOutputStream`:

```
var out = new ByteArrayOutputStream();
// Write to out
```

```
byte[] bytes = out.toByteArray();
```

Читання байтів

Клас InputStream має метод зчитування одного байта:

```
InputStream in = ...;
int b = in.read();
```

Цей метод або повертає байт як ціле число від 0 до 255, або повертає -1, якщо досягнуто кінця уведення.

Примітка. Тип `byte` Java має значення від -128 до 127. Ви можете перетворити повернуте значення в байт після того, як переконаєтеся, що воно не дорівнює -1.

Частіше ви захочете зчитувати байти масово. Найзручнішим методом є метод `readAllBytes`, який просто зчитує всі байти з потоку в байтовий масив:

```
byte[] bytes = in.readAllBytes();
```

Якщо ви хочете прочитати всі байти з файлу, викличте зручний метод:

```
byte[] bytes = Files.readAllBytes(path);
```

Запис байтів

Методи запису OutputStream можуть записувати окремі байти та байтові масиви.

```
OutputStream out = ...;
int b = ...;
out.write(b);
byte[] bytes = ...;
out.write(bytes);
out.write(bytes, start, length);
```

Коли ви закінчите писати потік, ви повинні закрити його, щоб зафіксувати будь-який буферизований результат. Найкраще це зробити за допомогою оператора try-with-resources:

```
try (OutputStream out = ...) {
    out.write(bytes);
}
```

Якщо вам потрібно скопіювати вхідний потік у вихідний, використовуйте метод InputStream.transferTo:

```
try (InputStream in = ...; OutputStream out = ...) {
    in.transferTo(out);
}
```

Обидва потоки потрібно закрити після виклику transferTo. Найкраще використовувати оператор try-with-resources, як у прикладі з кодом. Щоб записати файл до OutputStream, викличте

```
Files.copy(path, out);
```

І навпаки, щоб зберегти InputStream у файл, викличте

```
Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
```

Кодування символів

Вхідні та вихідні потоки призначені для послідовностей байтів, але в багатьох випадках ви працюватимете з текстом, тобто послідовностями символів. Тому має значення, як символи закодовані в байти. У Java використовується стандарт Unicode для символів. Кожен символ або «code point» має 21-бітове ціле число. Існують різні кодування символів — методи впаковування цих 21-бітових чисел у байти. Найбільш поширеним кодуванням є UTF-8, яке кодує кожний code point Юнікоду в послідовність від одного до чотирьох байт. UTF-8 має ту перевагу, що символи традиційного набору символів ASCII, який містить усі символи, що використовуються в англійській мові, займають лише один байт кожен.

Іншим поширеним кодуванням є UTF-16, яке кодує кожен code point Юнікоду в одне або два 16-бітні значення. Це кодування, яке використовується в рядках Java.

Клас StandardCharsets має статичні змінні типу Charset для кодувань символів, які повинна підтримувати кожна віртуальна машина Java:

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

Уведення тексту

Для читання тексту, що вводиться, використовуйте клас Reader. Ви можете отримати Reader з будь-якого вхідного потоку за допомогою адаптера InputStreamReader:

```
InputStream inStream = ...;
```

```
var in = new InputStreamReader(inStream, charset);
```

Якщо ви хочете обробляти вхідні дані по одному символу UTF-16 за раз, ви можете викликати метод зчитування:

```
int ch = in.read();
```

Метод повертає символ з кодом від 0 до 65536 або -1 як результат уведення. Це не дуже зручно. Ось кілька альтернативних варіантів. Якщо маєте короткий текстовий файл, ви можете прочитати його у вигляді рядка ось так:

```
String content = Files.readString(path, charset);
```

Але якщо ви хочете, щоб файл був зчитаний, як послідовність рядків, викличте

```
List<String> lines = Files.readAllLines(path, charset);
```

Примітка. Тип List (список, як і інші колекції, будемо розглядати пізніше)

Щоб прочитати числа або слова з файлу, скористайтеся сканером

```
var in = new Scanner(path, StandardCharsets.UTF_8);
while (in.hasNextDouble()) {
    double value = in.nextDouble();
    ...
}
```

Якщо ваші вхідні дані не надходять з файлу, оберніть InputStream у BufferedReader:

```
try (var reader = new BufferedReader(new InputStreamReader(url.openStream()))) {
    List<String> lines = reader.lines().toList();
    ...
}
```

BufferedReader зчитує вхідні дані фрагментами для ефективності. (Як не дивно, це не можуть інші читачі.) Він має метод readLine для читання цілого рядка. Якщо метод запитує Reader і ви хочете, щоб він читав з файлу, викличте `Files.newBufferedReader(path, charset)`.

Текстове виведення

Для виведення тексту використовуйте `Writer`. За допомогою методу `write` можна записувати рядки. Ви можете перетворити будь-який вихідний потік на `Writer`:

```
OutputStream outStream = ...;
var out = new OutputStreamWriter(outStream, charset);
out.write(str);
```

Щоб отримати Writer для файлу, скористайтеся командою

```
Writer out = Files.newBufferedWriter(path, charset);
```

Зручніше використовувати PrintWriter, який має print, println і printf, які ви завжди використовували з System.out. За допомогою цих методів можна друкувати числа та використовувати форматоване виведення. Якщо ви записуєте у файл, створіть PrintWriter так:

```
var out = new PrintWriter(Files.newBufferedWriter(path, charset));
```

Якщо ви пишете в інший потік, використовуйте

```
var out = new PrintWriter(new OutputStreamWriter(outStream, charset));
```

NOTE	System.out є екземпляром PrintStream, а не PrintWriter. Це пережиток з найдавніших часів Java. Однак, методи print, println і printf працюють однаково для класів PrintStream і PrintWriter, використовуючи кодування символів для перетворення символів в байти.

Серіалізація

Серіалізація — механізм перетворення об’єкта на послідовність байтів, які можуть бути відправлені кудись в інше місце або збережені на диску, а також для відновлення об’єкта з цих байтів.

Серіалізація є важливим інструментом для розподіленої обробки, коли об’єкти передаються з однієї віртуальної машини на іншу. Вона також використовується для перемикання та балансування навантаження, коли серіалізовані об’єкти можна перемістити на інший сервер. Якщо ви працюєте із серверним програмним забезпеченням, вам часто потрібно буде ввімкнути серіалізацію класів.

Інтерфейс Serializable

Для того, щоб об’єкт був серіалізований, тобто перетворений на послідовність байтів, він повинен бути екземпляром класу, який реалізує інтерфейс `Serializable`. Це маркерний інтерфейс, тобто інтерфейс без методів.

Наприклад, щоб зробити об’єкти `Employee` як `Serializable`, клас потрібно оголосити як

```
public class Employee implements Serializable {
    private String name;
    private double salary;
    // ...
}
```

Для класу доцільно реалізовувати серіалізований інтерфейс, якщо всі змінні екземпляра мають примітив або тип переліку, або містять посилання на серіалізовані об'єкти. Багато класів у стандартній бібліотеці можна серіалізувати. Масиви та класи колекції, можна серіалізувати за умови, що їхні елементи також є `Serializable`.

У випадку з класом `Employee`, та й узагалі з більшістю класів, проблем немає. Щоб серіалізувати об'єкти, вам потрібен `ObjectOutputStream`, який будується з іншим `OutputStream` всередині, що отримує фактичні байти.

```
var out = new ObjectOutputStream(Files.newOutputStream(path));
```

Тепер можна викликати метод `writeObject`:

```
var peter = new Employee("Peter", 75000);
var paul = new Manager("Paul", 120000);
out.writeObject(peter);
out.writeObject(paul);
```

Щоб прочитати об'єкти, треба отримати `ObjectInputStream`:

```
var in = new ObjectInputStream(Files.newInputStream(path));
```

Зчитувати об'єкти треба в тому ж порядку, в якому вони були записані, використовуючи метод `readObject`.

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

Коли записується об'єкт, зберігається ім'я класу, а також імена та значення всіх змінних-екземплярів. Якщо значення змінної-екземпляра належить до примітивного типу, воно зберігається як двійкові дані. Якщо це об'єкт, він знову записується методом `writeObject`. Коли об'єкт зчитується, процес змінюється на протилежний. Ім'я класу, імена та значення змінних екземпляра зчитуються, а об'єкт відновлюється. Є лише одна заковика. Припустимо, що було два посилання на один і той же об'єкт. Наприклад, у кожного співробітника є посилання на свого начальника:

```
var peter = new Employee("Peter", 75000);
var paul = new Manager("Barney", 120000);
```

```
var mary = new Manager("Mary", 160000);
peter.setBoss(mary);
paul.setBoss(mary);
out.writeObject(peter);
out.writeObject(paul);
```

При читанні цих двох об'єктів обидва повинні мати одного й того ж начальника, а не два посилання на однакові, але різні об'єкти. Для цього кожен об'єкт отримує порядковий номер при збереженні. Коли ви передаєте посилання на об'єкт у метод `writeObject`, `ObjectOutputStream` перевіряє, чи було посилання на об'єкт записано раніше. У цьому випадку він просто записує серійний номер і не дублює вміст об'єкта. Таким же чином, `ObjectInputStream` запам'ятовує всі об'єкти, з якими він стикався. При зчитуванні посилання на повторюваний об'єкт він просто дає посилання на раніше прочитаний об'єкт.

NOTE	Якщо суперклас серіалізованого класу не є серіалізованим, він повинен мати доступний конструктор без аргументів.

Розглянемо такий приклад:

```
class Person // Not serializable
class Employee extends Person implements Serializable
```

Коли об'єкт `Employee` десеріалізується, змінні його екземпляра зчитуються з потоку уведення об'єкта, але змінні екземпляра `Person` задаються конструктором `Person`.

Змінні об'єктів із модифікатором transient

Деякі змінні екземпляра не повинні бути серіалізовані, наприклад, з'єднання з базою даних, які не мають сенсу при відновленні об'єкта. Крім того, коли об'єкт зберігає кеш значень, можливо, краще скинути кеш і переобчислити його, а не зберігати. Щоб запобігти серіалізації змінної екземпляра, просто позначте її модифікатором `transient`. Завжди позначайте змінні екземпляра як тимчасові, якщо вони містять екземпляри класів, які не можна серіалізувати. Тимчасові змінні екземпляра пропускаються, коли об'єкти серіалізуються.

Методи readObject і writeObject

У рідкісних випадках потрібно підправити механізм серіалізації. Серіалізований клас може додати будь-яку бажану дію до стандартної поведінки читання та запису, визначаючи методи з такими сигнатурами

```
@Serial private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
@Serial private void writeObject(ObjectOutputStream out)
    throws IOException
```

Після цього заголовки об'єктів продовжують записуватися як зазвичай, але поля змінних екземпляра більше не серіалізуються автоматично. Натомість ці методи називаються. Зверніть увагу на анотацію `@Serial`. Методи налаштування серіалізації не належать до інтерфейсів. Тому ви не можете використовувати анотацію `@Override` для того, щоб компілятор перевіряв оголошення методів. Анотація `@Serial` призначена для того, щоб увімкнути таку саму перевірку методів серіалізації. До Java 21 компілятор javac не робить такої перевірки, але це може статися в майбутньому. Деякі IDE перевіряють анотацію.

Використання json-формату для серіалізації/десеріалізації

Дуже часто в сучасних програмах використовують формат JSON для серіалізації/десеріалізації замість звичайних механізмів Java. Розглянемо основні механізми перетворення об'єктів у формат json і навпаки.

Підключення за допомогою Maven

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.13.2</version>
    <scope>compile</scope>
</dependency>
```

Базові налаштування

Create a Gson instance:

```
Gson gson = new Gson();
```

Pretty printing / custom config:

```
Gson gson = new GsonBuilder()
        .setPrettyPrinting()
        .serializeNulls()
        .create();
```

Серіалізація (Object → JSON)

```
String json = gson.toJson(obj);
```

Приклад:

```
String json = gson.toJson(new User("Eugeny", 55));
```

Десеріалізація (JSON → Object)**

```
User user = gson.fromJson(jsonString, User.class);
```

Колекції & Generics

Use `TypeToken`:

```
Type type = new TypeToken<List<User>>(){}.getType();
List<User> users = gson.fromJson(jsonString, type);
```

Завдання

1. Використовуючи наведений нижче інтерфейс, реалізувати на його основі класи, що виконують операції уведення-виведення об’єктів та масивів у файли вказаних форматів:

```
public interface Repository<T> {
    void outputArray(T[] t, File file);
    void outputArray(T[] t, String fileName);

    T[] readArray(File file);
    T[] readArray(String fileName);
}
```

Примітка. Для реалізації цього інтерфейсу для уведення-виведення об’єктів типу `Dog` можна спочатку оголосити інтерфейс `DogRepository`, успадкувавши його від Repository, наприклад так:

```
public interface DogRepository extends Repository<Dog> {
    // ... визначення default методів
    // ... додаткові методи, якщо потрібні
}
```

а потім створити класи-реалізації цього інтерфейсу для роботи з файлами відповідних форматів. Наприклад, так:

```
public class DogRepositoryTextImpl implements DogRepository {
    // ... реалізація методів запису та читання з файлами текстового формату
}

public class DogRepositoryBinaryImpl implements DogRepository {
    // ... реалізація методів запису та читання з файлами двійкового формату
}
```

2. На основі програми з лабораторної роботи №2 доповніть класи предметної області, додавши до них нові властивості.
3. Доповніть програму функціями запису масивів об’єктів у файли текстового та бінарного форматів та читання інформації із цих файлів.
4. Додайте відповідні пункти меню та їхню обробку у View, контролер, сервіс(и).

Варіанти завдань

Варіант 1

Student: id, ПІБ, Дата народження, Адреса, Телефон, Факультет, Група, Рейтинг

Вивести:

a. Список студентів заданого факультету;
b. Список студентів, які народилися після заданого року;
c. Список студентів указаної групи, телефон яких має вказаний код оператора.
d. Список студентів із найбільшим рейтингом

Варіант 2

Customer: id, Прізвище, Ім’я, По батькові, Адреса, Номер кредитної картки, Баланс рахунку — кількість грошей на ньому (може бути як додатнім, так і від’ємним), Кількість покупок

Вивести:

a. Список покупців з указаним іменем;
b. Список покупців, у яких номер кредитної картки знаходиться в заданому інтервалі;
c. Список покупців, які мають заборгованість (від’ємний баланс на карті)
d. Список покупців, що зробили найбільшу кількість покупок

Варіант 3

Patient: id, Прізвище, Ім’я, По батькові, Адреса, Телефон, Номер медичної карти, Діагноз, Дата останнього візиту

Вивести:

a. Список пацієнтів, які мають указаний діагноз;
b. Список пацієнтів, номер медичної карти у яких знаходиться в заданому інтервалі;
c. Список пацієнтів, номер телефона яких починається з указаної цифри
d. Список пацієнтів, останній візит яких був максимально давно від поточної дати

Варіант 4

Abiturient: id, Прізвище, Ім’я, По батькові, Адреса, Телефон, Середній бал, Кількість заяв

Вивести:

a. Список абітурієнтів з указаним іменем;
b. Список абітурієнтів, середній бал у яких вище заданого;
c. Список абітурієнтів, прізвище яких починається з указаної літери, а номер телефона закінчується на вказані цифри;
d. Список абітурієнтів із найбільшою кількістю заяв.

Варіант 5

Book: id, Назва, Автор, Видавництво, Кількість сторінок, Рік видання, Ціна.

Вивести:

a. Список книг заданого автора;
b. Список книг, що видані заданим видавництвом;
c. Список книг, що були надруковані до заданого року та мають як мінімум указану кількість сторінок;
d. Список найдорожчих книг.

Варіант 6

House: id, Номер квартири, Площа, Поверх, Кількість кімнат, Вулиця, Оцінкова вартість

Вивести:

a. Список квартир, які мають задану кількість кімнат;
b. Список квартир, які мають задану кількість кімнат та розташовані на поверсі, який знаходиться в заданому проміжку;
c. Список квартир, які мають площу, що перевищує задану та розташовані на вказаній вулиці;
d. Список найдешевших квартир.

Варіант 7

Phone: id, Прізвище, Ім’я, По батькові, Номер рахунку, Час міських розмов, Час міжміських розмов, Баланс рахунку (залишок грошей)

Вивести:

a. Список абонентів, у яких час міських розмов перевищує заданий;
b. Список абонентів, які користувалися міжміським зв’язком;
c. Список абонентів, чий номер рахунку знаходиться у вказаному діапазоні;
d. Список абонентів, у яких баланс рахунку найбільший.

Варіант 8

Car: id, Модель, Рік випуску, Ціна, Реєстраційний номер, Пробіг

Вивести:

a. Список автомобілів заданої моделі;
b. Список автомобілів заданої моделі, які експлуатуються більше n років;
c. Список автомобілів заданого року випуску, ціна яких більше вказаної;
d. Список автомобілів з найменшим пробігом.

Варіант 9

Product: id, Найменування, Виробник, Ціна, Термін зберігання, Кількість, Одиниця вимірювання.

Вивести:

a. Список товарів для заданого найменування;
b. Список товарів для заданого найменування, ціна яких не перевищує задану;
c. Список товарів указаного виробника, термін зберігання яких більше заданого;
d. Список товарів, що вимірюються у кілограмах та мають найбільшу кількість.

Варіант 10

Train: id, Пункт призначення, Номер поїзда, Час відправлення, Число місць (загальних, купе, плацкарт, люкс), Час у дорозі

Вивести:

a. Список поїздів, які прямують до заданого пункту призначення;

b. Список поїздів, які прямують до заданого пункту призначення та відправляються після заданої години;

c. Список поїздів, які відправляються до заданого пункту призначення та мають загальні місця;

d. Список поїздів із найбільшим часом у дорозі.

Варіант 11

Bus: id, Номер автобуса, Номер маршруту, Марка, Рік початку експлуатації, Пробіг, Кількість пасажирів.

Вивести:

a. Список автобусів для заданого номера маршруту;

b. Список автобусів, які експлуатуються більше за заданий термін;

c. Список автобусів указаної марки, пробіг у яких більший за задану відстань;

d. Список автобусів, що вміщують найбільшу кількість пасажирів.

Варіант 12

Plane: id, Пункт призначення, Номер рейсу, Тип літака, Час вильоту, Дні тижня, Кількість місць.

Вивести:

a. Список рейсів для заданого пункту призначення;

b. Список рейсів, що виконуються заданим типом літака;

c. Список рейсів для заданого дня тижня, час вильоту для яких більший за заданий;

d. Список рейсів, на які кількість місць найменша.

Варіант 13

Dinnerware: id, Найменування, Виробник, Ціна, Матеріал, Кількість, Колір, Рік виробництва.

Вивести:

a. Список посуду заданого кольору;
b. Список посуду заданого найменування, ціна яких не перевищує задану;
c. Список посуду з указаного матеріалу заданого виробника;
d. Список найстарішого посуду.

Варіант 14

Drink: id, Найменування, Виробник, Ціна, Тип, Упаковка, Склад напою, Калорійність.

Вивести:

a. Список напоїв заданого виробника;
b. Список напоїв заданого найменування, ціна яких не перевищує задану;
c. Список напоїв указаного типу, в обраній упаковці;
d. Список напоїв з найбільшою калорійністю.

Варіант 15

Textile: id, Найменування виробу, Колір, Матеріал, Ціна, Категорія, Виробник, Товщина.

Вивести:

a. Список текстильних виробів заданого найменування та вказаного кольору;
b. Список текстильних виробів обраного виробника, ціна яких знаходиться у вказаних межах;
c. Список текстильних виробів указаного матеріалу та категорії;
d. Список найтонкіших текстильних виробів.