Екзамен ООП 1 семестр

Типи та структури даних

- **Які типи даних підтримуються в Java? В чому особливості цих типів даних? Порівняння з C++.**

Примітивні типи даних в Java
Java має 8 примітивних типів даних:

1. **Цілі числа**:
    - byte (8 біт): -128 до 127
    - short (16 біт): -32,768 до 32,767
    - int (32 біт): -2,147,483,648 до 2,147,483,647
    - long (64 біт): -2^{63} до $2^{63}-1$
2. **Числа з плаваючою крапкою**:
    - float (32 біт): менша точність
    - double (64 біт): вища точність
3. **Логічний тип**:
    - boolean: приймає значення true або false.
4. **Символьний тип**:
    - char (16 біт): зберігає один символ у кодуванні Unicode (на відміну від C++).

Особливості Java:

- **Розмір примітивів фіксований**: він не залежить від платформи, на відміну від C++, де розмір типів може змінюватися в залежності від архітектури.
- В Java **немає unsigned типів**, окрім char.
- Примітиви зберігаються в стеку, що забезпечує швидкий доступ.

Об'єктні типи даних в Java
Об'єктні типи (reference types) включають:

- **Класи** (class)
- **Інтерфейси** (interface)
- **Масиви** (array)
- **Перерахування** (enum)

Особливості Java:

- Всі об'єкти в Java зберігаються у **кучі (heap)**, а змінні містять лише посилання на ці об'єкти.
- Об'єктні типи завжди використовуються з оператором new, окрім рядків (String), які мають спеціальну підтримку.
- Для примітивів існують обгорткові класи (наприклад, Integer, Double), які дозволяють працювати з ними як з об'єктами.

Порівняння з C++

Особливість	Java	C++
Розмір примітивних типів	Фіксований (незалежно від платформи).	Залежить від компілятора та платформи.
Управління пам'яттю	Автоматичне (Garbage Collector).	Ручне (оператори new, delete) або розумні вказівники.

Особливість	Java	C++
Логічний тип	Має чіткий boolean.	Тип bool може бути використаний як ціле число (0 або 1).
Символьний тип	char (16-біт, підтримка Unicode).	char (8-біт, ASCII за замовчуванням).
Робота з unsigned	Немає unsigned типів (окрім char).	Підтримуються типи unsigned (unsigned int, unsigned char тощо).
Масиви	Масиви є об'єктами (мають методи, наприклад, .length).	Масиви — це набори пам’яті, без явного об’єктного функціоналу.
Посилання та вказівники	Підтримуються тільки посилання (немає вказівників, доступу до адрес пам'яті).	Підтримуються вказівники, посилання, і явна арифметика адрес.
Умовні оператори	boolean строго вимагається в умовах (не можна передати int).	Будь-який тип може використовуватись в умовах (0 — false, інше — true).

- **Які типи можна використовувати для зберігання послідовності однотипних об'єктів в Java? Які операції підтримуються для таких типів? Навести приклади з власного коду.**

1. Масиви (Arrays)
- Фіксований розмір, швидкий доступ до елементів за індексом.
- Елементи повинні бути одного типу.
- Створюється оператором new або через ініціалізацію.

Операції:
- Доступ до елементів за індексом (array[i]).
- Зміна значення елемента.
- Визначення довжини (array.length).

Приклад:
```
int[] numbers = {1, 2, 3, 4, 5};
numbers[2] = 10; // Зміна третього елемента
System.out.println(numbers[2]); // 10
System.out.println(numbers.length); // 5
```

---

2. Колекції (Collections)
Колекції надають більшу гнучкість, ніж масиви. Основні класи для зберігання послідовностей:

a) ArrayList (частина java.util):
- Масив змінного розміру.
- Елементи можуть додаватися, видалятися.
- Підтримує динамічне зростання.

Операції:
- Додавання елементів (add()).
- Видалення елементів (remove()).
- Доступ до елементів (get()).
- Зміна елементів (set()).
- Визначення розміру (size()).

Приклад:

```
import java.util.ArrayList;

ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Cherry");
fruits.set(1, "Blueberry"); // Зміна другого елемента
System.out.println(fruits.get(1)); // Blueberry
System.out.println(fruits.size()); // 3
fruits.remove(0); // Видалення першого елемента
System.out.println(fruits); // [Blueberry, Cherry]
```

b) LinkedList:

- Список на основі вузлів (подвійно зв'язаний список).
- Кращий для частих операцій вставки та видалення елементів.
- Схожий API, як у ArrayList.

Приклад:

```
import java.util.LinkedList;

LinkedList<Integer> numbers = new LinkedList<>();
numbers.add(10);
numbers.add(20);
numbers.addFirst(5); // Додати елемент на початок
System.out.println(numbers); // [5, 10, 20]
numbers.removeLast(); // Видалити останній елемент
System.out.println(numbers); // [5, 10]
```

---

3. Множини (Set)

- Унікальні елементи (без дублікатів).
- Використовують для зберігання послідовності однотипних об'єктів без повторів.

HashSet:

- Використовує хешування для швидкого доступу.
- Порядок елементів не гарантується.

Приклад:

```
import java.util.HashSet;

HashSet<String> cities = new HashSet<>();
cities.add("New York");
cities.add("Los Angeles");
cities.add("New York"); // Дублікат, не буде доданий
System.out.println(cities); // [New York, Los Angeles]
```

---

4. Масиви-обгортки (List.of)

- Статичний список, створений із заданих елементів.
- Непідтримує зміну розміру.

Приклад:

```
import java.util.List;

List<String> colors = List.of("Red", "Green", "Blue");
System.out.println(colors); // [Red, Green, Blue]
// colors.add("Yellow"); // Помилка, список незмінний
```

Порівняння

Тип	Динамічність	Доступ за індексом	Унікальність елементів	Швидкість доступу	Гнучкість
Масиви	Фіксована	Так	Ні	Висока	Низька
ArrayList	Динамічна	Так	Ні	Середня	Висока
LinkedList	Динамічна	Ні	Ні	Низька	Висока
HashSet	Динамічна	Ні	Так	Висока	Висока

- **Які типи можна використовувати для зберігання інформації про належність об'єкта до певної групи чи множини об'єктів в Java? Які операції підтримуються для таких типів? Навести приклади з власного коду.**

Для зберігання інформації про належність об'єкта до певної групи чи множини об'єктів у Java використовуються **колекції**, орієнтовані на множини. Основними з них є:

---

1. Set (java.util.Set)
Множина (Set) забезпечує зберігання унікальних елементів без дублікатів.
Основні реалізації Set:

- **HashSet**:
  - Використовує хешування для забезпечення швидкого доступу.
  - Порядок елементів не гарантується.
- **LinkedHashSet**:
  - Зберігає порядок вставки елементів.
  - Підходить, якщо потрібен визначений порядок і унікальність.
- **TreeSet**:
  - Елементи сортуються у природному порядку (або за допомогою компаратора).
  - Працює повільніше, ніж HashSet.

Операції:

- **Додавання**: add(element)
- **Видалення**: remove(element)
- **Перевірка наявності**: contains(element)
- **Об'єднання**: addAll(otherSet)
- **Перетин**: retainAll(otherSet)
- **Різниця**: removeAll(otherSet)

Приклад:

```java
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<String> group = new HashSet<>();

        // Додавання елементів
        group.add("Alice");
        group.add("Bob");
        group.add("Charlie");
        group.add("Alice"); // Дублікат ігнорується

        // Перевірка наявності
        System.out.println(group.contains("Alice")); // true
        System.out.println(group.contains("David")); // false

        // Видалення
        group.remove("Charlie");
        System.out.println(group); // [Alice, Bob]
    }
}
```

---

2. Множини через булеві масиви
Якщо кількість об'єктів відома та зафіксована, можна використовувати булеві масиви для відображення належності.
Операції:

- Індекси масиву відображають об'єкти.

- true означає належність до множини, false — ні.

Приклад:

```
public class Main {
    public static void main(String[] args) {
        boolean[] group = new boolean[5]; // Множина для 5 об'єктів (0-4)

        // Додавання до множини
        group[1] = true; // Додаємо об'єкт 1
        group[3] = true; // Додаємо об'єкт 3

        // Перевірка
        System.out.println(group[1]); // true (об'єкт 1 належить множині)
        System.out.println(group[2]); // false (об'єкт 2 не належить множині)

        // Видалення
        group[1] = false;
        System.out.println(group[1]); // false
    }
}
```

---

3. Бітові множини (BitSet)

BitSet — це ефективна структура для зберігання великих множин із булевими значеннями.

Операції:

- **Додавання**: set(index)
- **Видалення**: clear(index)
- **Перевірка**: get(index)
- **Об'єднання**: or(otherBitSet)
- **Перетин**: and(otherBitSet)

Приклад:

```
import java.util.BitSet;

public class Main {
    public static void main(String[] args) {
        BitSet group = new BitSet();

        // Додавання до множини
        group.set(1); // Додаємо елемент 1
        group.set(3); // Додаємо елемент 3

        // Перевірка
        System.out.println(group.get(1)); // true
        System.out.println(group.get(2)); // false

        // Видалення
        group.clear(1);
        System.out.println(group.get(1)); // false
    }
}
```

---

4. Перерахування (EnumSet)

Для зберігання об'єктів з обмеженого набору (енумерації).

Операції:

- Створення множини: EnumSet.of(...)
- Додавання: add(element)
- Видалення: remove(element)

Приклад:

```java
import java.util.EnumSet;

enum Days {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY
}

public class Main {
    public static void main(String[] args) {
        EnumSet<Days> workDays = EnumSet.of(Days.MONDAY, Days.TUESDAY);

        // Додавання
        workDays.add(Days.WEDNESDAY);

        // Перевірка
        System.out.println(workDays.contains(Days.MONDAY)); // true

        // Видалення
        workDays.remove(Days.TUESDAY);
        System.out.println(workDays); // [MONDAY, WEDNESDAY]
    }
}
```

Порівняння підходів

Тип	Переваги	Недоліки
HashSet	Швидка перевірка, динамічний розмір	Не зберігає порядок
LinkedHashSet	Зберігає порядок вставки	Трохи повільніше за HashSet
TreeSet	Сортує елементи	Повільніше за інші Set реалізації
BitSet	Ефективне використання пам'яті	Лише для числових індексів
EnumSet	Зручний для елементів енума	Лише для об'єктів enum

- **Які типи можна використовувати для зберігання відображення з одного типу в інший в Java? Які операції підтримуються для таких типів? Навести приклади з власного коду.**

Для зберігання **відображення** (відповідностей) з одного типу в інший у Java використовуються **мапи (Maps)**. Основний інтерфейс для цього — java.util.Map.

Основні реалізації Map

1. **HashMap**
   - Зберігає ключі та значення.
   - Використовує хешування для швидкого доступу до елементів.
   - Порядок елементів не гарантується.
2. **LinkedHashMap**
   - Зберігає порядок вставки елементів.
   - Підходить, якщо потрібна і швидкість, і порядок.
3. **TreeMap**
   - Зберігає ключі у відсортованому порядку (за природним порядком або компаратором).
   - Працює повільніше, але забезпечує впорядкованість.
4. **EnumMap**
   - Оптимізована для використання ключів типу enum.

Операції, які підтримуються

Для всіх реалізацій Map:

- **Додавання або заміна**: put(key, value)
- **Отримання значення за ключем**: get(key)
- **Видалення пари ключ-значення**: remove(key)
- **Перевірка наявності ключа/значення**: containsKey(key), containsValue(value)
- **Розмір мапи**: size()
- **Очистка мапи**: clear()
- **Отримання ключів, значень, пар**:
    - keySet() — всі ключі.
    - values() — всі значення.
    - entrySet() — всі пари ключ-значення.

Приклади з коду

```
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        // Створення мапи
        HashMap<Integer, String> map = new HashMap<>();

        // Додавання елементів
        map.put(1, "Alice");
        map.put(2, "Bob");
        map.put(3, "Charlie");

        // Отримання значення
        System.out.println(map.get(2)); // Bob

        // Перевірка наявності ключа
        System.out.println(map.containsKey(1)); // true

        // Видалення елемента
        map.remove(3);

        // Вивід усіх пар ключ-значення
        for (var entry : map.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
        // Результат:
        // 1 -> Alice
        // 2 -> Bob
    }
}
```

```
import java.util.LinkedHashMap;

public class Main {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> map = new LinkedHashMap<>();

        // Додавання елементів
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Порядок зберігається
        System.out.println(map); // {Alice=25, Bob=30, Charlie=35}
    }
}
```

```java
import java.util.TreeMap;

public class Main {
    public static void main(String[] args) {
        TreeMap<String, Integer> map = new TreeMap<>();

        // Додавання елементів
        map.put("Charlie", 35);
        map.put("Alice", 25);
        map.put("Bob", 30);

        // Сортування за ключами
        System.out.println(map); // {Alice=25, Bob=30, Charlie=35}
    }
}
```

```java
import java.util.EnumMap;

enum Color {
    RED, GREEN, BLUE
}

public class Main {
    public static void main(String[] args) {
        EnumMap<Color, String> map = new EnumMap<>(Color.class);

        // Додавання елементів
        map.put(Color.RED, "Stop");
        map.put(Color.GREEN, "Go");
        map.put(Color.BLUE, "Relax");

        // Отримання значення
        System.out.println(map.get(Color.GREEN)); // Go
    }
}
```

Порівняння реалізацій Map

Тип	Переваги	Недоліки
HashMap	Швидкий доступ (O(1)).	Порядок ключів не зберігається.
LinkedHashMap	Зберігає порядок вставки.	Працює трохи повільніше за HashMap.
TreeMap	Сортує ключі за природним порядком.	Повільніша (O(log n)).
EnumMap	Дуже швидка, оптимізована для enum ключів.	Можна використовувати лише enum.

Коли використовувати:

- **HashMap**: Для швидкого доступу без збереження порядку.
- **LinkedHashMap**: Для збереження порядку вставки.
- **TreeMap**: Для відсортованих даних.
- **EnumMap**: Для мап із ключами типу enum.

- **Що таке колекції в Java? Які класи для підтримки колекцій є в стандартній бібліотеці? Як вони між собою пов'язані? Навести приклади з власного коду.**

Колекції в Java — це **структури даних**, які використовуються для зберігання, організації й маніпуляцій із групами об'єктів. Вони є частиною стандартної бібліотеки Java (java.util) і забезпечують універсальні способи роботи з динамічними даними.
Колекції базуються на **інтерфейсах**, які визначають загальні операції, такі як додавання, видалення, сортування тощо. Їхні реалізації пропонують різні способи зберігання та обробки даних.

Основні інтерфейси колекцій

1. **Collection**
   - Базовий інтерфейс для всіх колекцій (списки, множини, черги).
2. **List**
   - Зберігає елементи у вигляді впорядкованого списку.
   - Дозволяє дублікати.
   - Реалізації: ArrayList, LinkedList, Vector.
3. **Set**
   - Зберігає унікальні елементи (без дублікатів).
   - Реалізації: HashSet, LinkedHashSet, TreeSet.
4. **Queue**
   - Використовується для обробки даних за принципом FIFO (перший увійшов — перший вийшов).
   - Реалізації: PriorityQueue, ArrayDeque.
5. **Map**
   - Зберігає пари ключ-значення (асоціативний масив).
   - Реалізації: HashMap, TreeMap, LinkedHashMap.

Класи для підтримки колекцій

1. List

- **ArrayList**: Динамічний масив, швидкий доступ за індексом.
- **LinkedList**: Список на основі вузлів, ефективний для вставки/видалення.
- **Vector**: Схожий на ArrayList, але синхронізований.

2. Set

- **HashSet**: Використовує хешування, не гарантує порядку.
- **LinkedHashSet**: Зберігає порядок вставки.
- **TreeSet**: Сортує елементи.

3. Queue

- **PriorityQueue**: Черга із пріоритетами (за замовчуванням — природний порядок).
- **ArrayDeque**: Двостороння черга (Deque).

4. Map

- **HashMap**: Використовує хешування для зберігання пар ключ-значення.
- **TreeMap**: Сортує ключі.
- **LinkedHashMap**: Зберігає порядок вставки.

Структура і зв'язок класів

```
Collection
├── List
|   ├── ArrayList
|   ├── LinkedList
|   └── Vector
├── Set
|   ├── HashSet
|   ├── LinkedHashSet
|   └── TreeSet
└── Queue
    ├── PriorityQueue
    └── ArrayDeque
Map (не є підінтерфейсом Collection)
├── HashMap
├── TreeMap
└── LinkedHashMap
```

Приклади коду

1. Робота зі списком (List)

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        // Виведення списку
        for (String name : list) {
            System.out.println(name);
        }

        // Отримання елемента за індексом
        System.out.println("Другий елемент: " + list.get(1));

        // Видалення елемента
        list.remove("Alice");
        System.out.println(list); // [Bob, Charlie]
    }
}
```

2. Робота з множиною (Set)

```
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<Integer> set = new HashSet<>();
        set.add(10);
        set.add(20);
        set.add(30);
        set.add(10); // Дублікат ігнорується

        System.out.println(set); // [10, 20, 30]
        System.out.println("Містить 20? " + set.contains(20)); // true
    }
}
```

3. Робота з чергою (Queue)

```
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> queue = new PriorityQueue<>();
        queue.add(30);
        queue.add(10);
        queue.add(20);

        // Виведення з черги за пріоритетом
        while (!queue.isEmpty()) {
            System.out.println(queue.poll());
        }
        // Результат: 10, 20, 30
    }
}
```

4. Робота з мапою (Map)

```java
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Виведення значень за ключами
        System.out.println("Вік Bob: " + map.get("Bob"));

        // Виведення всіх пар
        for (var entry : map.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Висновок
Колекції Java пропонують різні класи для зберігання й обробки даних. Їхній вибір залежить від вимог до продуктивності, збереження порядку й унікальності.
Наприклад:

- ArrayList — для роботи зі списками з частим доступом за індексом.
- HashSet — для зберігання унікальних елементів.
- PriorityQueue — для роботи з чергами пріоритетів.
- HashMap — для зберігання пар ключ-значення.

- **Яким чином можна обходити елементи колекцій в Java? Навести приклади з власного коду.**

Обхід елементів колекцій у Java можна здійснювати кількома способами залежно від типу колекції та вимог до коду. Основні підходи:

---

1. Обхід за допомогою циклу for-each
Цей спосіб підходить для колекцій, які реалізують інтерфейс Iterable.
Приклад:

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        // Обхід елементів
        for (String name : list) {
            System.out.println(name);
        }
    }
}
```

---

2. Обхід за допомогою Iterator
Iterator — це об'єкт, що дозволяє послідовно отримувати елементи колекції. Цей метод дозволяє безпечно видаляти елементи під час обходу.
Основні методи Iterator:

- hasNext() — перевіряє, чи є наступний елемент.
- next() — повертає наступний елемент.
- remove() — видаляє поточний елемент.

Приклад:

```java
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        Iterator<String> iterator = list.iterator();

        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);

            // Видалення елемента під час обходу
            if (name.equals("Bob")) {
                iterator.remove();
            }
        }

        System.out.println(list); // [Alice, Charlie]
    }
}
```

---

3. Обхід за допомогою індексу

Підходить для колекцій, які підтримують доступ за індексом (наприклад, List).

Приклад:

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        for (int i = 0; i < list.size(); i++) {
            System.out.println("Елемент " + i + ": " + list.get(i));
        }
    }
}
```

---

4. Обхід за допомогою forEach (Java 8+)

Функціональний стиль обходу із використанням лямбда-виразів або методів.

Приклад:

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        // Використання лямбда-виразу
        list.forEach(name -> System.out.println(name));

        // Використання посилання на метод
        list.forEach(System.out::println);
    }
}
```

---

5. Обхід за допомогою Stream API (Java 8+)

Stream API дозволяє здійснювати обхід і обробку елементів у функціональному стилі.

Приклад:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        // Обробка елементів потоку
        list.stream()
            .filter(name -> name.startsWith("A")) // Фільтруємо елементи
            .forEach(System.out::println); // Виводимо "Alice"
    }
}
```

---

6. Обхід елементів Map

Для мапи (Map) існує кілька способів обходу, залежно від того, чи потрібно обійти ключі, значення або пари ключ-значення.

Обхід ключів:

```
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        for (String key : map.keySet()) {
            System.out.println("Ключ: " + key);
        }
    }
}
```

Обхід значень:

```
for (Integer value : map.values()) {
    System.out.println("Значення: " + value);
}
```

Обхід пар ключ-значення:

```
for (var entry : map.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

Порівняння методів

Метод	Переваги	Недоліки
for-each	Простота, зручність.	Не можна змінювати колекцію під час обходу.
Iterator	Дозволяє безпечне видалення.	Складніший, ніж for-each.
Індексний доступ	Прямий доступ до елементів.	Тільки для списків (List).
forEach	Лаконічний, функціональний.	Не дає прямого доступу до індексів.
Stream API	Потужна обробка даних.	Більш ресурсоємний.

- **Які засоби декларативної роботи з колекціями даних, перетворення даних тощо підтримуються в Java? Навести приклади з власного коду.**

У Java для декларативної роботи з колекціями даних і їх перетворення використовується **Stream API**, введений у версії Java 8. Stream API дозволяє працювати з колекціями у функціональному стилі, надаючи методи для обробки даних, фільтрації, сортування, групування тощо.

Основні можливості Stream API

1. **Створення стрімів**
    - Використовувати stream() або parallelStream() для створення стріму з колекції.
2. **Фільтрація**: метод filter().
3. **Перетворення даних**: методи map() і flatMap().
4. **Сортування**: метод sorted().
5. **Агрегація**: методи reduce(), count(), min(), max().
6. **Збір даних**: методи collect(), що повертають результати у вигляді колекції або іншого об'єкта.
7. **Групування і підрахунок**: за допомогою Collectors.groupingBy() і Collectors.counting().

Приклади з коду

1. Створення стріму

```
import java.util.List;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie");

        // Створення стріму
        Stream<String> stream = names.stream();

        // Виведення елементів
        stream.forEach(System.out::println);
    }
}
```

---

2. Фільтрація

```
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie", "Anna");

        // Фільтруємо імена, що починаються на "A"
        List<String> filteredNames = names.stream()
                .filter(name -> name.startsWith("A"))
                .collect(Collectors.toList());

        System.out.println(filteredNames); // [Alice, Anna]
    }
}
```

---

3. Перетворення даних

```java
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie");

        // Перетворення імен на великі літери
        List<String> upperCaseNames = names.stream()
                .map(String::toUpperCase)
                .collect(Collectors.toList());

        System.out.println(upperCaseNames); // [ALICE, BOB, CHARLIE]
    }
}
```

---

4. Сортування

```java
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Charlie", "Bob", "Alice");

        // Сортування в алфавітному порядку
        List<String> sortedNames = names.stream()
                .sorted()
                .toList();

        System.out.println(sortedNames); // [Alice, Bob, Charlie]
    }
}
```

---

5. Агрегація

```java
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        // Сума елементів
        int sum = numbers.stream()
                .reduce(0, Integer::sum);

        System.out.println("Сума: " + sum); // 15
    }
}
```

---

6. Групування

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie", "Anna", "Brian");

        // Групування за першим символом
        Map<Character, List<String>> grouped = names.stream()
                .collect(Collectors.groupingBy(name -> name.charAt(0)));

        System.out.println(grouped);
        // {A=[Alice, Anna], B=[Bob, Brian], C=[Charlie]}
    }
}
```

---

7. Підрахунок елементів

```java
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie", "Alice");

        // Підрахунок унікальних елементів
        long count = names.stream()
                .distinct()
                .count();

        System.out.println("Кількість унікальних імен: " + count); // 3
    }
}
```

---

8. Робота з паралельними стрімами

Для великих колекцій можна використовувати parallelStream() для обробки в кілька потоків.

```java
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        // Паралельна обробка
        numbers.parallelStream()
                .map(num -> num * num)
                .forEach(System.out::println);
    }
}
```

Ключові переваги Stream API

1. **Декларативність**: Код виглядає більш лаконічно та читається як опис процесу.
2. **Легкість обробки великих даних**: Можливість працювати з паралельними потоками.
3. **Комбінування операцій**: Методи стрімів можна комбінувати для досягнення складних трансформацій.

Stream API дозволяє ефективно працювати з колекціями, не змінюючи вихідні дані, оскільки всі операції є **немутуючими**.

- **В чому відмінність між масивами та списками в Java? Які операції підтримуються для таких типів? Навести приклади з власного коду.**

Відмінності між масивами та списками в Java

Характеристика	Масиви	Списки (List)
Розмір	Фіксований, задається під час створення.	Динамічний, може змінюватися.
Тип даних	Однорідні елементи одного типу.	Може містити об'єкти, включаючи null.
Методи роботи	Обмежений набір операцій.	Розширений набір методів (додавання, видалення, сортування тощо).
Реалізація	Частина мови (низькорівнева структура).	Частина Java Collections API (високий рівень абстракції).
Продуктивність	Працює швидше, оскільки немає додаткових перевірок.	Деякі операції можуть бути повільнішими через накладні витрати.
Інтерфейси/Класи	Не реалізує інтерфейси.	Реалізує інтерфейс List.

Операції з масивами

1. Створення та доступ до елементів

```java
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        // Доступ до елементів
        System.out.println("Перший елемент: " + numbers[0]);

        // Зміна значення
        numbers[0] = 10;

        // Виведення всіх елементів
        for (int number : numbers) {
            System.out.println(number);
        }
    }
}
```

2. Сортування масиву

```java
import java.util.Arrays;

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

        // Сортування масиву
        Arrays.sort(numbers);

        System.out.println(Arrays.toString(numbers)); // [1, 2, 3, 4, 5]
    }
}
```

3. Копіювання масиву

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] original = {1, 2, 3};
        int[] copy = Arrays.copyOf(original, original.length);

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

Операції зі списками

1. Створення та додавання елементів

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        // Додавання елементів
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        System.out.println(list); // [Alice, Bob, Charlie]
    }
}
```

2. Видалення елементів

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(List.of("Alice", "Bob", "Charlie"));

        // Видалення за індексом
        list.remove(1);

        // Видалення за значенням
        list.remove("Charlie");

        System.out.println(list); // [Alice]
    }
}
```

3. Сортування списку

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(List.of("Charlie", "Alice", "Bob"));

        // Сортування
        Collections.sort(list);

        System.out.println(list); // [Alice, Bob, Charlie]
    }
}
```

4. Перетворення масиву на список

```java
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        String[] array = {"Alice", "Bob", "Charlie"};

        // Перетворення на список
        List<String> list = Arrays.asList(array);

        System.out.println(list); // [Alice, Bob, Charlie]
    }
}
```

5. Перетворення списку на масив

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(List.of("Alice", "Bob", "Charlie"));

        // Перетворення на масив
        String[] array = list.toArray(new String[0]);

        System.out.println(Arrays.toString(array)); // [Alice, Bob, Charlie]
    }
}
```

Порівняння операцій

Операція	Масиви	Списки
Додавання елементів	Непідтримується (фіксований розмір).	Підтримується (add).
Видалення елементів	Непідтримується.	Підтримується (remove).
Сортування	Arrays.sort().	Collections.sort().
Зміна розміру	Неможливо.	Динамічний розмір.

Операція	Масиви	Списки
Ітерація	Цикл for/for-each.	Цикл for, for-each, Iterator.

Висновок

- **Масиви** підходять для роботи з фіксованою кількістю елементів, коли потрібна висока продуктивність.
- **Списки** є більш гнучкими, надають багато методів для роботи з динамічними даними, але мають трохи більші накладні витрати.

- **В чому відмінність між списками та множинами в Java? Які операції підтримуються для таких типів? Навести приклади з власного коду.**

Відмінність між списками (List) та множинами (Set) в Java

Характеристика	Список (List)	Множина (Set)
Дублікати	Дозволяє дублікати елементів.	Не дозволяє дублікати.
Порядок елементів	Зберігає порядок вставки елементів.	Порядок не гарантується (залежить від реалізації).
Індексований доступ	Доступ за індексом (get(index)).	Доступ за індексом недоступний.
Швидкодія	Ефективний для операцій вставки/отримання за індексом.	Ефективний для перевірки наявності елемента.
Реалізації	ArrayList, LinkedList, Vector.	HashSet, LinkedHashSet, TreeSet.

Операції, підтримувані списками (List)

1. **Додавання**: add(element), add(index, element).
2. **Видалення**: remove(element), remove(index).
3. **Доступ за індексом**: get(index).
4. **Оновлення елементів**: set(index, element).
5. **Пошук**: indexOf(element), lastIndexOf(element).
6. **Перевірка**: contains(element).
7. **Сортування**: Collections.sort(list).

Операції, підтримувані множинами (Set)

1. **Додавання**: add(element). Якщо елемент вже є, новий не додається.
2. **Видалення**: remove(element).
3. **Перевірка**: contains(element).
4. **Робота з перетинами**:
    - Об'єднання: addAll(collection).
    - Перетин: retainAll(collection).
    - Різниця: removeAll(collection).

Приклади з коду

1. Робота зі списком (List)

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        // Додавання елементів
        list.add("Alice");
        list.add("Bob");
        list.add("Alice"); // Дублікати дозволені

        // Доступ за індексом
        System.out.println("Елемент на індексі 1: " + list.get(1));

        // Видалення
        list.remove("Bob");

        // Сортування
        list.sort(String::compareTo);

        System.out.println(list); // [Alice, Alice]
    }
}
```

---

2. Робота з множиною (Set)

```java
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();

        // Додавання елементів
        set.add("Alice");
        set.add("Bob");
        set.add("Alice"); // Дублікати не додаються

        // Перевірка наявності
        System.out.println("Чи міститься 'Bob'? " + set.contains("Bob"));

        // Видалення
        set.remove("Alice");

        System.out.println(set); // [Bob]
    }
}
```

---

3. Об'єднання списку та множини

```java
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Alice");

        Set<String> set = new HashSet<>(list); // Унікальні елементи

        System.out.println("Список: " + list); // [Alice, Bob, Alice]
        System.out.println("Множина: " + set); // [Alice, Bob]
    }
}
```

---

4. Перетин множин

```java
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<Integer> set1 = new HashSet<>(Set.of(1, 2, 3, 4));
        Set<Integer> set2 = new HashSet<>(Set.of(3, 4, 5, 6));

        // Перетин
        set1.retainAll(set2);

        System.out.println("Перетин: " + set1); // [3, 4]
    }
}
```

Реалізації множини

1. **HashSet**:
    - Використовує хешування.
    - Порядок елементів не гарантується.
    - Швидкий доступ.
2. **LinkedHashSet**:
    - Зберігає порядок вставки.
    - Повільніше, ніж HashSet.
3. **TreeSet**:
    - Зберігає елементи в відсортованому порядку.
    - Реалізує інтерфейс NavigableSet.

Приклад використання TreeSet:

Висновки

- **Списки** підходять для збереження впорядкованих даних, у тому числі з дублікатами, коли важливий доступ за індексом.
- **Множини** використовуються для зберігання унікальних елементів і швидких операцій перевірки, об'єднання, перетину тощо.

Вибір залежить від вимог до структури даних.

- **В чому відмінність між списками та словниками/відображеннями/колекціями ключ-значення в Java? Які операції підтримуються для таких типів? Навести приклади з власного коду.**

Відмінність між списками (List) та словниками/відображеннями (Map) у Java

Характеристика	Список (List)	Словник/Відображення (Map)
Структура даних	Колекція однотипних елементів.	Колекція пар ключ -> значення.
Доступ до даних	За індексом (list.get(index)).	За ключем (map.get(key)).
Дублікати	Дублікати значень дозволені.	Дублікати ключів не дозволені; значення можуть дублюватись.
Порядок елементів	Залежить від реалізації (ArrayList, LinkedList).	Залежить від реалізації (HashMap, TreeMap).
Реалізації	ArrayList, LinkedList, Vector.	HashMap, LinkedHashMap, TreeMap.

Операції, підтримувані списками (List)

1. **Додавання**: add(element), add(index, element).
2. **Видалення**: remove(index), remove(element).
3. **Доступ за індексом**: get(index).
4. **Оновлення елементів**: set(index, element).
5. **Пошук**: indexOf(element), lastIndexOf(element).
6. **Сортування**: Collections.sort(list).

Операції, підтримувані словниками (Map)

1. **Додавання пари ключ-значення**: put(key, value).
2. **Отримання значення за ключем**: get(key).
3. **Видалення пари**: remove(key).
4. **Перевірка наявності**:
    - Ключа: containsKey(key).
    - Значення: containsValue(value).
5. **Отримання колекцій**:
    - Ключів: keySet().
    - Значень: values().
    - Пар ключ-значення: entrySet().
6. **Об'єднання карт**: putAll(map).

Приклади з коду

1. Робота зі списком (List)

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        // Додавання елементів
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        // Доступ за індексом
        System.out.println("Елемент на індексі 1: " + list.get(1)); // Bob

        // Видалення
        list.remove("Bob");

        System.out.println(list); // [Alice, Charlie]
    }
}
```

---

2. Робота зі словником (Map)

```java
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();

        // Додавання пар ключ-значення
        map.put(1, "Alice");
        map.put(2, "Bob");
        map.put(3, "Charlie");

        // Отримання значення за ключем
        System.out.println("Ключ 2 відповідає значенню: " + map.get(2)); // Bob

        // Видалення пари
        map.remove(1);

        // Перевірка наявності ключа
        System.out.println("Чи є ключ 3? " + map.containsKey(3)); // true

        System.out.println(map); // {2=Bob, 3=Charlie}
    }
}
```

---

3. Ітерація через словник

```java
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "Alice");
        map.put(2, "Bob");
        map.put(3, "Charlie");

        // Ітерація через ключі
        for (Integer key : map.keySet()) {
            System.out.println("Ключ: " + key + ", Значення: " + map.get(key));
        }

        // Ітерація через пари ключ-значення
        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println("Ключ: " + entry.getKey() + ", Значення: " + entry.get\
        }
    }
}
```

---

4. Порівняння списку та словника

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        // Список імен
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");

        // Словник номерів телефонів
        Map<String, String> phoneBook = new HashMap<>();
        phoneBook.put("Alice", "123-456");
        phoneBook.put("Bob", "789-012");

        // Виведення списку
        System.out.println("Список імен: " + names);

        // Виведення телефонів
        for (String name : names) {
            System.out.println("Телефон для " + name + ": " + phoneBook.get(name));
        }
    }
}
```

Реалізації словників

1. **HashMap**:
    - Використовує хешування для швидкого доступу.
    - Порядок пар не гарантується.
2. **LinkedHashMap**:
    - Зберігає порядок вставки.
    - Повільніше за HashMap.
3. **TreeMap**:
    - Зберігає ключі в відсортованому порядку.
    - Повільніше за HashMap і LinkedHashMap.

Висновки

- **Списки** (List) використовуються для зберігання впорядкованих елементів із можливістю дублювання.
- **Словники** (Map) потрібні для зберігання пар ключ-значення, забезпечуючи швидкий доступ до значень за унікальними ключами.

Вибір залежить від потреб: якщо дані впорядковані за індексом, використовується List, якщо необхідно працювати з ключами і значеннями — Map.

- **В чому відмінність між списками, стеками, чергами в Java? Які операції підтримуються для таких типів? Навести приклади з власного коду.**

Відмінність між списками, стеками та чергами в Java

Характеристика	Список (List)	Стек (Stack)	Черга (Queue)
Структура даних	Колекція елементів з доступом за індексом.	LIFO (Last In, First Out): останній доданий видаляється першим.	FIFO (First In, First Out): перший доданий видаляється першим.
Порядок елементів	Зберігає порядок вставки елементів.	Зберігає порядок елементів у стилі LIFO.	Зберігає порядок елементів у стилі FIFO.

Характеристика	Список (List)	Стек (Stack)	Черга (Queue)
Дублікати	Дублікати дозволені.	Дублікати дозволені.	Дублікати дозволені.
Основні операції	Додавання, видалення, доступ за індексом.	push(), pop(), peek().	offer(), poll(), peek().
Реалізації	ArrayList, LinkedList.	Stack (на основі Vector).	LinkedList, ArrayDeque, PriorityQueue.

Операції для кожного типу

1. Список (List)

- **Додавання**: add(element), add(index, element).
- **Видалення**: remove(element), remove(index).
- **Доступ за індексом**: get(index).
- **Оновлення**: set(index, element).
- **Сортування**: Collections.sort(list).

2. Стек (Stack)

- **Додавання елемента**: push(element).
- **Видалення верхнього елемента**: pop().
- **Перегляд верхнього елемента**: peek().
- **Перевірка наявності елемента**: search(element).

3. Черга (Queue)

- **Додавання елемента**: offer(element).
- **Видалення першого елемента**: poll() (або remove()).
- **Перегляд першого елемента**: peek() (або element()).
- **Додавання елемента в кінець черги**: add(element) (при успіху).

Приклади коду

1. Список (List)

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        // Додавання елементів
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        // Доступ за індексом
        System.out.println("Елемент на індексі 1: " + list.get(1)); // Bob

        // Видалення
        list.remove("Alice");

        System.out.println("Список: " + list); // [Bob, Charlie]
    }
}
```

---

2. Стек (Stack)

```java
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();

        // Додавання елементів (push)
        stack.push("Alice");
        stack.push("Bob");
        stack.push("Charlie");

        // Перегляд верхнього елемента
        System.out.println("Верхній елемент: " + stack.peek()); // Charlie

        // Видалення верхнього елемента (pop)
        System.out.println("Видалено: " + stack.pop()); // Charlie

        // Перевірка вмісту стека
        System.out.println("Стек: " + stack); // [Alice, Bob]
    }
}
```

---

3. Черга (Queue)

```java
import java.util.LinkedList;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // Додавання елементів (offer)
        queue.offer("Alice");
        queue.offer("Bob");
        queue.offer("Charlie");

        // Перегляд першого елемента
        System.out.println("Перший елемент: " + queue.peek()); // Alice

        // Видалення першого елемента (poll)
        System.out.println("Видалено: " + queue.poll()); // Alice

        // Перевірка вмісту черги
        System.out.println("Черга: " + queue); // [Bob, Charlie]
    }
}
```

---

4. Декілька реалізацій черги

- **Пріоритетна черга**:

```java
import java.util.PriorityQueue;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        Queue<Integer> priorityQueue = new PriorityQueue<>();

        priorityQueue.offer(10);
        priorityQueue.offer(5);
        priorityQueue.offer(15);

        System.out.println("Перший елемент: " + priorityQueue.peek()); // 5 (найменши

        // Видалення найменшого елемента
        priorityQueue.poll();

        System.out.println("Черга: " + priorityQueue); // [10, 15]
    }
}
```

- **Deque як черга**:

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();

        deque.offerFirst("Alice"); // Додає на початок
        deque.offerLast("Bob");   // Додає в кінець

        System.out.println("Deque: " + deque); // [Alice, Bob]

        // Видалення з початку
        deque.pollFirst();

        System.out.println("Deque після видалення: " + deque); // [Bob]
    }
}
```

Ключові моменти

1. **Список (List)** використовується для роботи з впорядкованими елементами з доступом за індексом.
2. **Стек (Stack)** застосовується для задач, де важлива послідовність LIFO (наприклад, обробка викликів функцій).
3. **Черга (Queue)** потрібна для обробки елементів у порядку FIFO (наприклад, задачі на черговість або обробку подій).

- ## В чому відмінність між множинами та словниками/відображеннями/колекціями ключ-значення в Java? Які операції підтримуються для таких типів? Навести приклади з власного коду.

Відмінність між множинами (Set) та словниками/відображеннями (Map) в Java

Характеристика	Множина (Set)	Словник/Відображення (Map)
Структура даних	Набір унікальних елементів.	Набір пар ключ-значення.
Дублікати	Дублікати не дозволені.	Ключі не можуть дублюватися; значення можуть.
Порядок елементів	Залежить від реалізації (HashSet, TreeSet, LinkedHashSet).	Залежить від реалізації (HashMap, TreeMap, LinkedHashMap).
Доступ до даних	Через ітерацію або перевірку наявності елемента (contains).	Через ключ (map.get(key)).
Основні операції	Додавання, видалення, перевірка наявності.	Додавання, видалення, доступ за ключем.
Реалізації	HashSet, TreeSet, LinkedHashSet.	HashMap, TreeMap, LinkedHashMap.

Операції, підтримувані множинами (Set)

1. **Додавання елементів**: add(element).
2. **Видалення елементів**: remove(element).
3. **Перевірка наявності**: contains(element).
4. **Операції над множинами** (залежить від бібліотек):
    - Перетин: retainAll().
    - Об'єднання: addAll().
    - Різниця: removeAll().

Операції, підтримувані словниками (Map)

1. **Додавання пари ключ-значення**: put(key, value).
2. **Отримання значення за ключем**: get(key).
3. **Видалення за ключем**: remove(key).
4. **Перевірка наявності**:
    - Ключа: containsKey(key).
    - Значення: containsValue(value).
5. **Отримання колекцій**:
    - Ключів: keySet().
    - Значень: values().
    - Пар ключ-значення: entrySet().

Приклади з коду

1. Множина (Set)

```
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();

        // Додавання елементів
        set.add("Alice");
        set.add("Bob");
        set.add("Charlie");

        // Додавання дубліката (ігнорується)
        set.add("Alice");

        // Перевірка наявності
        System.out.println("Чи є 'Alice': " + set.contains("Alice")); // true

        // Видалення
        set.remove("Bob");

        System.out.println("Множина: " + set); // [Alice, Charlie]
    }
}
```

---

2. Словник (Map)

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();

        // Додавання пар ключ-значення
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Отримання значення за ключем
        System.out.println("Вік Alice: " + map.get("Alice")); // 25

        // Видалення пари за ключем
        map.remove("Bob");

        // Перевірка наявності ключа
        System.out.println("Чи є 'Charlie': " + map.containsKey("Charlie")); // true

        // Перегляд усіх ключів
        System.out.println("Ключі: " + map.keySet()); // [Alice, Charlie]
    }
}
```

---

3. Взаємозв'язок множин та словників

Множина може бути отримана зі словника у вигляді ключів (keySet) або пар (entrySet):

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);

        // Отримання множини ключів
        Set<String> keys = map.keySet();
        System.out.println("Ключі: " + keys); // [Alice, Bob]

        // Отримання множини пар ключ-значення
        Set<Map.Entry<String, Integer>> entries = map.entrySet();
        for (Map.Entry<String, Integer> entry : entries) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

---

4. Операції над множинами

```java
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<Integer> set1 = new HashSet<>();
        set1.add(1);
        set1.add(2);
        set1.add(3);

        Set<Integer> set2 = new HashSet<>();
        set2.add(3);
        set2.add(4);
        set2.add(5);

        // Перетин
        Set<Integer> intersection = new HashSet<>(set1);
        intersection.retainAll(set2);
        System.out.println("Перетин: " + intersection); // [3]

        // Об'єднання
        Set<Integer> union = new HashSet<>(set1);
        union.addAll(set2);
        System.out.println("Об'єднання: " + union); // [1, 2, 3, 4, 5]

        // Різниця
        Set<Integer> difference = new HashSet<>(set1);
        difference.removeAll(set2);
        System.out.println("Різниця: " + difference); // [1, 2]
    }
}
```

Ключові моменти

1. **Множини (Set)**: для роботи з унікальними елементами. Їх основна мета — забезпечити відсутність дублікатів.
2. **Словники (Map)**: для роботи з парами ключ-значення, забезпечуючи швидкий доступ до даних за ключем.
3. **Множина (Set) може бути використана для отримання ключів зі словника**, але вона не зберігає зв'язок із відповідними значеннями.

Вибір структури даних залежить від задачі: якщо потрібні унікальні елементи — використовується Set, якщо потрібен зв'язок між ключем і значенням — використовується Map.

Парадигма ООП

- **Як працює інкапсуляція в Java? Які модифікатори доступу підтримуються для полів та методів? Порівняння з C++.**

Інкапсуляція — це принцип ООП, який передбачає приховування деталей реалізації об'єкта та надання доступу до даних через визначений інтерфейс. Це досягається за допомогою:

1. **Модифікаторів доступу** для контролю видимості полів і методів.
2. **Геттерів і сеттерів** для управління доступом до полів.

Модифікатори доступу в Java

Java підтримує чотири рівні доступу:

Модифікатор	Ключове слово	Видимість
Публічний	public	Доступний усюди (в межах будь-якого класу чи пакета).
Приватний	private	Доступний тільки в межах того ж класу.
Захищений	protected	Доступний у тому ж класі, у класах того ж пакета, а також у спадкоємцях навіть з інших пакетів.
Пакетний	(відсутній)	Доступний у межах того ж пакета (за замовчуванням).

Порівняння з C++

Аспект	Java	C++
Модифікатори доступу	public, protected, private, (пакетний).	public, protected, private.
Підхід до інкапсуляції	Використовуються геттери й сеттери.	Прямий доступ до полів може бути обмежений через модифікатори доступу.
Підтримка friend	Немає friend класів/функцій.	Можна оголосити клас або функцію "другом" для доступу до приватних полів.
Рівень доступу за замовчуванням	Пакетний (відсутній модифікатор).	private для членів класу, public для структур.

Приклад інкапсуляції в Java

```java
public class Person {
    // Приватні поля
    private String name;
    private int age;

    // Конструктор
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Геттери
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Сеттер з перевіркою
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println("Вік має бути позитивним!");
        }
    }

    // Метод для відображення інформації
    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

Приклад інкапсуляції в C++

```cpp
#include <iostream>
#include <string>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    // Конструктор
    Person(string name, int age) : name(name), age(age) {}

    // Геттери
    string getName() const {
        return name;
    }

    int getAge() const {
        return age;
    }

    // Сеттер
    void setAge(int age) {
        if (age > 0) {
            this->age = age;
        } else {
            cout << "Вік має бути позитивним!" << endl;
        }
    }

    // Метод для відображення інформації
    void displayInfo() const {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person person("Alice", 30);

    // Доступ до полів через методи
    person.displayInfo();

    // Оновлення віку через сеттер
    person.setAge(35);
    cout << "Оновлений вік: " << person.getAge() << endl;

    return 0;
}
```

Ключові моменти інкапсуляції

1. **Java** більше спирається на стандартизоване використання геттерів/сеттерів, часто генерує їх автоматично (наприклад, через IDE або Lombok).
2. **C++** дозволяє ширшу кастомізацію через friend та інші механізми.
3. Обидві мови дозволяють обмежувати доступ до полів і методів, але Java зазвичай додає більше обмежень для забезпечення кращої безпеки даних.

- ## Як працює успадкування та поліморфізм в Java? Порівняння з C++.

Успадкування (Inheritance)

Успадкування в Java дозволяє створювати нові класи на основі існуючих. Новий клас (дочірній або похідний) отримує поля й методи базового (батьківського) класу, що сприяє повторному використанню коду.

- **Ключове слово:** extends для класів, implements для інтерфейсів.
- **Особливості в Java:**

1. Java підтримує лише **одинарне успадкування класів** (один базовий клас), але дозволяє реалізовувати кілька інтерфейсів.
2. Для доступу до методів або конструктора батьківського класу використовується ключове слово super.
3. Конструктор базового класу автоматично викликається перед конструктором похідного класу.

Поліморфізм (Polymorphism)

Поліморфізм дозволяє одному і тому ж методу виконувати різну поведінку залежно від об'єкта, з яким він викликається. Є два види поліморфізму:

1. **Компільований час** (перевантаження методів — *method overloading*).
2. **Час виконання** (перевизначення методів — *method overriding*).

Приклад успадкування та поліморфізму в Java

Успадкування

```
// Батьківський клас
class Animal {
    public void speak() {
        System.out.println("Animal makes a sound");
    }
}

// Дочірній клас
class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Dog barks");
    }
}
```

Поліморфізм

```
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog(); // Поліморфізм: базовий клас посилається на об'єкт до

        myAnimal.speak(); // "Animal makes a sound"
        myDog.speak();    // "Dog barks" (перевизначений метод)
    }
}
```

Порівняння успадкування та поліморфізму в Java та C++

Особливість	Java	C++
Тип успадкування	Тільки **одинарне успадкування** класів (можливе кілька інтерфейсів).	Підтримує **множинне успадкування** класів і віртуальне успадкування для уникнення конфліктів.
Ключові слова	extends для класів, implements для інтерфейсів.	Немає спеціальних ключових слів для реалізації множинного успадкування.
Поліморфізм часу виконання	Вимагає використання ключового слова @Override для перевизначення методів.	Поліморфізм реалізується за допомогою віртуальних методів (virtual).
Виклик базового класу	super.methodName() або super() для конструктора.	BaseClass::methodName() або автоматично при виклику конструктора.
Підтримка інтерфейсів	Підтримує через implements (клас може реалізовувати кілька інтерфейсів).	Інтерфейси можна реалізувати через абстрактні базові класи.
Множинне успадкування	Відсутнє для класів (уникнення діамантової проблеми).	Підтримується, але може виникнути діамантова проблема.

Приклад успадкування та поліморфізму в C++

Успадкування

```cpp
#include <iostream>
using namespace std;

// Базовий клас
class Animal {
public:
    virtual void speak() {
        cout << "Animal makes a sound" << endl;
    }
};

// Дочірній клас
class Dog : public Animal {
public:
    void speak() override {
        cout << "Dog barks" << endl;
    }
};
```

Поліморфізм

```cpp
int main() {
    Animal* myAnimal = new Animal();
    Animal* myDog = new Dog(); // Поліморфізм через віртуальні функції

    myAnimal->speak(); // "Animal makes a sound"
    myDog->speak();    // "Dog barks"

    delete myAnimal;
    delete myDog;

    return 0;
}
```

Ключові моменти

1. **Java** обмежує успадкування одного класу, що спрощує управління кодом, але підтримує реалізацію кількох інтерфейсів.
2. У **C++** є більша гнучкість з множинним успадкуванням, але це може призводити до конфліктів.
3. Обидві мови підтримують поліморфізм через методи базового класу, які перевизначаються в дочірньому класі. У Java для цього завжди використовується ключове слово @Override, тоді як у C++ використовується virtual.

- **Чи підтримує Java множинне успадкування? приватне успадкування? віртуальне успадкування? успадкування інтерфейсу? успадкування реалізації? Порівняння з C++.**

1. Множинне успадкування

- **Java** не підтримує множинне успадкування класів через можливість виникнення **діамантової проблеми**.
- Множинне успадкування підтримується через **інтерфейси**. Клас може реалізовувати кілька інтерфейсів за допомогою ключового слова implements.

Приклад:

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class C implements A, B {
    @Override
    public void methodA() {
        System.out.println("Method A");
    }

    @Override
    public void methodB() {
        System.out.println("Method B");
    }
}
```

2. Приватне успадкування

- **Java** не підтримує приватне успадкування. Успадкування в Java завжди публічне або пакетне. Доступність членів залежить від модифікаторів доступу:
    - Поля й методи з модифікатором private не успадковуються.
    - Поля й методи з модифікатором protected доступні дочірнім класам.

3. Віртуальне успадкування

- У **Java** всі методи є **віртуальними** за замовчуванням (можуть бути перевизначені у дочірньому класі). Це виключає необхідність додаткових ключових слів, як у C++ (virtual, override).

4. Успадкування інтерфейсу

- У **Java** інтерфейси є основним засобом забезпечення множинного успадкування. Інтерфейс визначає лише контракт (методи без реалізації) до Java 8. Починаючи з Java 8, інтерфейси можуть мати **реалізацію за замовчуванням** (default methods).

Приклад:

```
interface Printable {
    default void print() {
        System.out.println("Printing...");
    }
}
```

5. Успадкування реалізації

- **Java** дозволяє часткове успадкування реалізації через:
    - **Абстрактні класи** (abstract class), які можуть мати як реалізовані, так і абстрактні методи.
    - **Default methods** в інтерфейсах.

Приклад абстрактного класу:

```
abstract class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }

    abstract void makeSound();
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks.");
    }
}
```

Порівняння з C++

Особливість	Java	C++
Множинне успадкування	Не підтримується для класів. Реалізується через інтерфейси (implements).	Підтримується (може викликати діамантову проблему).
Приватне успадкування	Не підтримується.	Підтримується (class B : private A).
Віртуальне успадкування	Всі методи віртуальні за замовчуванням.	Використовується через ключове слово virtual.
Успадкування інтерфейсу	Підтримується через інтерфейси (interface).	Інтерфейси реалізуються через абстрактні базові класи.
Успадкування реалізації	Через абстрактні класи та default methods в інтерфейсах.	Реалізується через базові класи.

Ключові моменти

1. **Java** усуває складнощі, пов'язані з множинним успадкуванням класів, і вирішує їх через інтерфейси.
2. **C++** надає повну гнучкість із множинним, приватним і віртуальним успадкуванням, але це може спричиняти труднощі з керуванням кодом (наприклад, **діамантова проблема**).
3. У **Java** немає прямої підтримки приватного успадкування — доступ обмежується через модифікатори доступу.

- **Чи підтримує Java віртуальні методи? методи що не є віртуальними? абстрактні класи? абстрактні методи? Чи бувають класи, від яких не можна успадковуватись? класи, які вимагають успадкування від себе? Порівняння з C++.**

Особливості методів і класів у Java

1. Віртуальні методи

- **У Java всі методи не є статичними або приватними — віртуальні за замовчуванням.**
- Це означає, що методи базового класу можуть бути перевизначені (overridden) у дочірніх класах.
- Якщо метод не потрібно перевизначати, його можна зробити фінальним (final), щоб заборонити перевизначення.

Приклад:

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}
```

---

2. Методи, що не є віртуальними

- У Java методи можуть бути невіртуальними, якщо:
    1. Вони оголошені як private. Приватні методи не доступні для перевизначення.

2. Вони оголошені як static. Статичні методи належать класу, а не конкретному об'єкту, тому не можуть бути перевизначені.
3. Вони оголошені як final. Такі методи не можна перевизначати в дочірніх класах.

Приклад:

```
class Parent {
    private void privateMethod() {
        System.out.println("This is a private method");
    }

    static void staticMethod() {
        System.out.println("This is a static method");
    }

    final void finalMethod() {
        System.out.println("This is a final method");
    }
}
```

---

3. Абстрактні класи та методи

- **Абстрактний клас** (abstract class) — це клас, який може містити абстрактні (незавершені) методи та звичайні методи з реалізацією. Абстрактний клас не можна інстанціювати.
- **Абстрактний метод** — це метод без реалізації, який дочірній клас має обов’язково перевизначити.

Приклад:

```
abstract class Animal {
    abstract void makeSound(); // Абстрактний метод

    void eat() { // Метод з реалізацією
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}
```

---

4. Класи, від яких не можна успадковувати

- У Java клас можна зробити фінальним (final), що забороняє успадкування від нього.
- Наприклад:

```
final class Utility {
    void performTask() {
        System.out.println("Performing task...");
    }
}
```

Спроба успадкування від такого класу викличе помилку:

```
class ExtendedUtility extends Utility { // Помилка
}
```

---

5. Класи, які вимагають успадкування від себе

- У Java **абстрактний клас** є типом класу, від якого потрібно успадковувати, оскільки його не можна інстанціювати напряму.

Приклад:

```java
abstract class BaseClass {
    abstract void doSomething();
}

class DerivedClass extends BaseClass {
    @Override
    void doSomething() {
        System.out.println("Implementation in derived class");
    }
}
```

Порівняння з C++

Особливість	Java	C++
Віртуальні методи	Усі методи (не private, static чи final) є віртуальними за замовчуванням.	Методи є невіртуальними за замовчуванням. Для віртуальних методів використовується virtual.
Методи, що не є віртуальними	Методи private, static або final не можна перевизначати.	Будь-який метод може бути невіртуальним за замовчуванням, якщо явно не використано virtual.
Абстрактні класи	Підтримуються через ключове слово abstract.	Підтримуються через класи, що містять хоча б один чисто віртуальний метод (= 0).
Абстрактні методи	Методи без реалізації вказуються як abstract.	Реалізуються через чисто віртуальні методи (virtual void method() = 0;).
Класи, від яких не можна успадковувати	Використовується ключове слово final.	Використовується ключове слово final (з C++11).
Класи, які вимагають успадкування	Будь-який абстрактний клас.	Клас із чисто віртуальними методами або захищеним/приватним конструктором.

Ключові моменти

1. У **Java** механізм віртуальних методів є простішим і не потребує додаткових ключових слів (virtual, override).
2. **C++** надає більше контролю через virtual, final, та інші ключові слова, але це може ускладнювати розуміння коду.
3. Обидві мови підтримують абстрактні класи та методи, але реалізація в Java є строгою і зрозумілою через обов'язковість ключового слова abstract.
4. **C++** дає можливість створювати класи, які "вимагають" успадкування, використовуючи різні конструкції (наприклад, чисто віртуальні методи).

Керування пам’яттю

- **Що таке керування пам'яттю? Як працює керування пам'яттю в Java? Порівняння з C++.**

Керування пам’яттю — це процес розподілу, використання та звільнення пам’яті програмою під час її виконання. Це включає:

1. **Розподіл пам’яті** для змінних, об’єктів і структур даних.
2. **Звільнення пам’яті** після завершення її використання, щоб уникнути витоків пам’яті.
3. **Оптимізацію використання пам’яті** для підвищення продуктивності програми.

Керування пам’яттю в Java

У Java керування пам’яттю автоматизоване. Основні компоненти:

1. **Купа (Heap Memory):**
   - Використовується для зберігання об’єктів і класів.
   - Пам’ять динамічно розподіляється під час виконання.
2. **Стека (Stack Memory):**
   - Використовується для зберігання локальних змінних, методів і викликів функцій.
   - Пам’ять автоматично звільняється, коли метод закінчує виконання.
3. **Методична область (Method Area):**
   - Зберігає мета-інформацію про класи, методи й константи.
4. **GC (Garbage Collector):**
   - Відповідає за автоматичне звільнення пам’яті. Перевіряє об’єкти, які більше не використовуються, і видаляє їх.

Особливості:

- Пам’ять під об’єкти розподіляється автоматично за допомогою оператора new.
- Пам’ять звільняється автоматично завдяки **збірнику сміття (Garbage Collector)**.
- Розробник не контролює процес звільнення пам’яті напряму.

Приклад:

```
class Main {
    public static void main(String[] args) {
        String str = new String("Hello, Java!"); // Пам’ять для об’єкта розподіляєтьс
        str = null; // Об’єкт стає кандидатом для видалення Garbage Collector
    }
}
```

Керування пам’яттю в C++

У C++ керування пам’яттям є ручним або напівавтоматичним. Основні компоненти:

1. **Купа (Heap):**
   - Використовується для динамічного розподілу пам’яті через new та delete.
2. **Стека (Stack):**
   - Використовується для зберігання локальних змінних і викликів функцій.
3. **Ручне звільнення пам’яті:**
   - Програміст зобов’язаний вручну викликати delete для звільнення пам’яті, виділеної через new.

Особливості:

- Пам’ять під об’єкти виділяється вручну.
- Якщо програміст забуває звільнити пам’ять, виникають **витоки пам’яті (memory leaks)**.
- Використання розумних вказівників (std::unique_ptr, std::shared_ptr) з C++11 значно полегшує керування пам’яттю.

Приклад:

```cpp
#include <iostream>
using namespace std;

int main() {
    int* ptr = new int(10); // Виділення пам'яті вручну
    cout << *ptr << endl;
    delete ptr; // Звільнення пам'яті вручну
    return 0;
}
```

Порівняння керування пам’яттю в Java та C++

Особливість	Java	C++
Розподіл пам’яті	Автоматичний (new для об’єктів).	Ручний (new для об’єктів), також може бути автоматичний для локальних змінних.
Звільнення пам’яті	Автоматичний за допомогою Garbage Collector.	Ручний (вимагає виклику delete), може використовувати розумні вказівники для автоматизації.
Витоки пам’яті	Рідкісні (основна причина — посилання на об’єкти, які більше не використовуються).	Можливі, якщо програміст забуває звільнити пам’ять.
Контроль над звільненням	Немає прямого контролю.	Програміст повністю контролює, але це збільшує ризик помилок.
Продуктивність	Додаткові накладні витрати через роботу Garbage Collector.	Швидше, якщо програміст ефективно керує пам’яттю.
Розумні вказівники	Відсутні (автоматичне керування через GC).	Підтримуються (std::unique_ptr, std::shared_ptr) для безпечного керування пам’яттю.

Ключові моменти

1. У **Java** керування пам’яттю автоматизоване, що зменшує ризики витоків пам’яті, але може додати затримки через роботу збірника сміття.
2. У **C++** програміст має більше контролю над пам’яттю, але це також створює ризики, якщо пам’яттю керують неефективно.
3. **Java** підходить для додатків, де критично важлива стабільність керування пам’яттю, а **C++** — для додатків з високими вимогами до продуктивності.

- ## Чи можна в Java дізнатись, коли відбулось звільнення пам'яті певного об'єкта? Порівняння з C++.

У Java прямого механізму для відстеження моменту звільнення пам’яті для конкретного об’єкта немає. Це пов’язано з тим, що звільнення пам’яті управляється автоматично **Garbage Collector (GC)**, і програміст не має контролю над його виконанням.
Проте, Java надає спеціальний метод finalize(), який можна перевизначити у класі. Цей метод викликається перед тим, як об’єкт стає кандидатом на видалення збирачем сміття. **Однак finalize() вважається застарілим починаючи з Java 9 і не рекомендується до використання через непередбачуваність його виклику.**

Приклад:

```java
class Demo {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Object is being garbage collected");
    }
}

public class Main {
    public static void main(String[] args) {
        Demo obj = new Demo();
        obj = null; // Об'єкт стає недосяжним
        System.gc(); // Запит на запуск GC (не гарантує виконання)
    }
}
```

Особливості:

1. Метод System.gc() лише рекомендує JVM запустити збирач сміття, але це не гарантується.
2. Звільнення пам’яті відбувається асинхронно, і точний момент запуску GC програміст не контролює.

Як це працює в C++?

У C++ звільнення пам’яті для об’єктів із динамічним розподілом виконується вручну за допомогою оператора delete. Це дає програмісту повний контроль над моментом звільнення пам’яті.

Приклад:

```cpp
#include <iostream>
using namespace std;

class Demo {
public:
    Demo() {
        cout << "Object created" << endl;
    }
    ~Demo() {
        cout << "Object destroyed" << endl;
    }
};

int main() {
    Demo* obj = new Demo(); // Динамічне створення об'єкта
    delete obj; // Ручне звільнення пам'яті
    return 0;
}
```

Особливості:

1. Деструктор (~Demo) викликається автоматично під час звільнення пам’яті через delete.
2. У разі локальних об’єктів деструктор викликається автоматично при виході з області видимості.

Порівняння Java та C++

Особливість	Java	C++
Контроль звільнення пам’яті	Немає прямого контролю. Звільнення пам’яті керується Garbage Collector.	Програміст визначає момент звільнення пам’яті вручну через delete.
Виклик деструктора	Використовується метод finalize(), але його виконання не гарантується і він застарілий.	Деструктор (~ClassName) викликається автоматично під час звільнення пам’яті.

Особливість	Java	C++
Момент звільнення пам’яті	Невизначений, залежить від роботи збирача сміття.	Визначається програмістом або системою (для локальних змінних).
Гарантоване звільнення пам’яті	Немає (об’єкт може залишатися в пам’яті до запуску GC).	Є (деструктор завжди викликається під час звільнення пам’яті або виходу з області видимості).
Витоки пам’яті	Малоймовірні (GC автоматично звільняє невикористовувані об’єкти).	Можливі, якщо забути викликати delete для динамічних об’єктів.

Ключові моменти

1. У **Java** програміст не знає точного моменту звільнення пам’яті, але автоматичне управління пам’яттю знижує ризики витоків.
2. У **C++** деструктори дозволяють передбачити і контролювати звільнення пам’яті, але це створює додаткову відповідальність для програміста.
3. Для сучасного C++ використання розумних вказівників (std::unique_ptr, std::shared_ptr) спрощує керування пам’яттю і наближає його до автоматизації, схожої на Java.

- **Яким чином виконується звільнення ресурсів в Java? Чи підтримується гарантоване звільнення ресурсів, незалежно від поведінки програми? Порівняння з C++.**

У Java звільнення ресурсів реалізовано через спеціальний механізм **try-with-resources**, який автоматично закриває ресурси після завершення роботи. Це забезпечує **гарантоване звільнення ресурсів**, незалежно від того, як поводиться програма (наприклад, чи виникла помилка під час виконання).

Основні підходи до звільнення ресурсів у Java:

Автоматичне звільнення через try-with-resources (Java 7 і новіші версії):

- Ресурси, які потребують звільнення (наприклад, файли, сокети, потоки), повинні реалізовувати інтерфейс AutoCloseable (або його попередник Closeable).
- При виході з блоку try, метод close() викликається автоматично.

Приклад:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ручне звільнення ресурсів:

- До появи try-with-resources ресурси закривали вручну через блок finally.

Приклад:

```java
public class Main {
    public static void main(String[] args) {
        BufferedReader br = null;
        try {
            br = new BufferedReader(new FileReader("example.txt"));
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (br != null) br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Garbage Collector (GC):

- Звільнення пам’яті автоматизоване, але **GC не закриває ресурси**. Для зовнішніх ресурсів (файли, мережеві з’єднання тощо) потрібно явне закриття.

Гарантоване звільнення ресурсів у Java

Механізм try-with-resources забезпечує гарантоване закриття ресурсів, навіть якщо в середині блоку try виникла помилка. Це є важливою перевагою для уникнення витоків ресурсів, таких як файлові дескриптори, з’єднання з базами даних або сокети.

Звільнення ресурсів у C++

У C++ звільнення ресурсів виконується через деструктори. Коли об’єкт виходить за межі області видимості або звільняється вручну (для динамічно створених об’єктів), його деструктор викликається автоматично.

Основні підходи до звільнення ресурсів у C++:

RAII (Resource Acquisition Is Initialization):

- Концепція RAII забезпечує автоматичне звільнення ресурсів завдяки тому, що деструктори викликаються під час знищення об’єкта.

Приклад:

```cpp
#include <iostream>
#include <fstream>
using namespace std;

class FileHandler {
private:
    ifstream file;
public:
    FileHandler(const string& filename) {
        file.open(filename);
        if (!file.is_open()) {
            throw runtime_error("Cannot open file");
        }
    }
    ~FileHandler() {
        if (file.is_open()) {
            file.close();
            cout << "File closed" << endl;
        }
    }
};

int main() {
    try {
        FileHandler fh("example.txt");
        // Робота з файлом
    } catch (const exception& e) {
        cerr << e.what() << endl;
    }
    return 0;
}
```

Ручне звільнення ресурсів:

- Для динамічно створених об’єктів ресурси звільняються вручну через виклик delete.

Приклад:

```cpp
int* ptr = new int(10);
delete ptr; // Ручне звільнення пам’яті
```

Порівняння Java та C++

Особливість	Java	C++
Автоматичне звільнення пам'яті	Автоматичне звільнення через Garbage Collector (але GC не закриває зовнішні ресурси).	Деструктори автоматично викликаються при виході об’єкта за область видимості.
Звільнення ресурсів	Try-with-resources (AutoCloseable / Closeable) забезпечує гарантоване звільнення.	RAII забезпечує автоматичне звільнення ресурсів через деструктори.
Контроль програміста	Відсутній контроль над GC. Try-with-resources дає частковий контроль для зовнішніх ресурсів.	Програміст має повний контроль через деструктори.
Складність коду	Простий код із try-with-resources.	Може бути складнішим, якщо потрібно писати власні деструктори.

Особливість	Java	C++
Гарантія звільнення ресурсів	Try-with-resources гарантує звільнення ресурсів.	Деструктори гарантують звільнення ресурсів при правильній реалізації.

Ключові моменти

1. **Java**:
   - Гарантоване звільнення ресурсів реалізоване через try-with-resources.
   - Зовнішні ресурси потребують явного звільнення, Garbage Collector їх не закриває.
2. **C++**:
   - RAII і деструктори забезпечують гарантоване звільнення ресурсів.
   - Ручне керування ресурсами може створювати ризик витоків, якщо деструктори не реалізовані належним чином.
3. **Головна відмінність**:
   - У Java програміст більше покладається на автоматизацію, тоді як у C++ більше контролю, але й більше відповідальності.

Стилі програмування

- **Чи підтримує Java структурний (процедурний) стиль програмування? Якщо так - яким чином, якщо ні - чому? Порівняння з C++.**

Java є **об'єктно-орієнтованою мовою програмування** (ООП), яка за своєю природою орієнтована на роботу з об’єктами та класами. Однак, структурний (процедурний) стиль програмування частково підтримується через можливість створювати **статичні методи** та використовувати їх для роботи без об'єктів.

Як у Java можна використовувати процедурний стиль?

1. **Використання статичних методів:** У Java можна визначати методи як static, і вони можуть викликатися без створення об'єктів. Це дозволяє імітувати процедурний стиль програмування.

Приклад:

```
public class ProceduralExample {
    public static void main(String[] args) {
        int result = add(5, 3);
        System.out.println("Result: " + result);
    }

    public static int add(int a, int b) {
        return a + b;
    }
}
```

2. У цьому прикладі всі дії виконуються через статичні методи, що нагадує процедурний підхід.
3. **Відсутність глобальних змінних:** На відміну від C++, у Java всі змінні мають належати класу. Це обмежує використання чисто процедурного стилю, оскільки неможливо створювати глобальні змінні поза межами класів.
4. **Головний метод main:** У Java програма починається зі статичного методу main, що також узгоджується з процедурним підходом.
5. **Функціональність через статичні утилітні класи:** Багато бібліотек у Java (наприклад, java.lang.Math) надають набір статичних методів, що дозволяє виконувати процедурні операції.

Чому Java не є повністю процедурною мовою?

1. **Програмна модель:**
   - Усе в Java має бути оголошене в класах. Це ускладнює реалізацію чисто процедурного стилю.
2. **Відсутність глобального простору імен:**
   - У Java немає глобальних функцій або змінних, оскільки все має належати певному класу.
3. **Природна орієнтація на ООП:**
   - Java була створена як мова для ООП. Її концепції (успадкування, поліморфізм, інкапсуляція) не сумісні з чисто процедурним підходом.

Як це працює в C++?

C++ підтримує як об'єктно-орієнтоване, так і структурне програмування, що дає більшу гнучкість.

Процедурний стиль у C++:

- У C++ можна писати програми в чисто процедурному стилі, як це робиться в C.
- Функції можуть існувати поза межами класів, а глобальні змінні доступні в усьому коді.

Приклад:

```cpp
#include <iostream>
using namespace std;

int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(5, 3);
    cout << "Result: " << result << endl;
    return 0;
}
```

Взаємодія стилів:

- У C++ можна комбінувати структурний і об'єктно-орієнтований стиль у межах однієї програми.

Порівняння Java та C++

Особливість	Java	C++
Можливість процедурного стилю	Обмежена. Процедурний стиль реалізується через статичні методи всередині класів.	Повна підтримка процедурного стилю, аналогічно до C.
Глобальні змінні та функції	Немає глобальних функцій або змінних (усе має бути в класах).	Є глобальні змінні та функції.
Обов’язковість класів	Усе має бути визначене в класах.	Класи необов’язкові для процедурних програм.
Гнучкість стилю програмування	Чітка орієнтація на ООП.	Підтримує як структурний, так і об'єктно-орієнтований підходи.

Ключові моменти

- У **Java** процедурний стиль можливий, але обмежений через необхідність використання класів.
- У **C++** процедурний стиль повністю підтримується завдяки можливості писати функції поза класами.
- Вибір стилю програмування в Java часто обумовлений її орієнтацією на об’єктно-орієнтоване програмування, тоді як C++ дозволяє більше гнучкості.

• Чи підтримує Java функціональний стиль програмування? Якщо так - яким чином, якщо ні - чому? Порівняння з C++.

Java, починаючи з версії 8, частково підтримує функціональний стиль програмування. Вона залишається об'єктно-орієнтованою мовою, але включає інструменти для роботи з функціональними конструкціями, такими як **лямбда-вирази**, **функціональні інтерфейси** і **стріми (Streams)**.

Як Java підтримує функціональний стиль?

1. **Лямбда-вирази:**
    - Це функції, які можна передавати як аргументи, повертати як результат і використовувати для визначення поведінки об'єктів.
    - Вони надають синтаксичний цукор для реалізації анонімних класів.

Приклад:

```java
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<Integer, Integer> square = x -> x * x;
        System.out.println(square.apply(5)); // Виведе 25
    }
}
```

2. **Функціональні інтерфейси:**

- Інтерфейси з єдиним абстрактним методом (SAM) можуть використовуватися для передачі функцій.
- Java надає стандартні функціональні інтерфейси в пакеті java.util.function, наприклад:
    - Function<T, R> – функція з аргументом і результатом.
    - Consumer<T> – функція, яка приймає аргумент і нічого не повертає.
    - Supplier<T> – функція, яка не приймає аргументів, але повертає результат.

3. **API для роботи зі стрімами (Streams):**

- Streams API дозволяє виконувати операції над колекціями у функціональному стилі (фільтрація, мапінг, скорочення).

Приклад:

```java
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        numbers.stream()
               .filter(n -> n % 2 == 0)
               .map(n -> n * n)
               .forEach(System.out::println); // Виведе 4 і 16
    }
}
```

4. **Функції як об’єкти:**

- Завдяки функціональним інтерфейсам і лямбда-виразам Java дозволяє використовувати функції як об'єкти.

5. **Посилання на методи:**

- Це синтаксичний засіб, що дозволяє передавати існуючі методи або конструктори як функції.

Приклад:

```java
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        names.forEach(System.out::println); // Посилання на метод println
    }
}
```

Чому Java не є повністю функціональною?

1. **Мутованість об’єктів:**
    - Java дозволяє змінювати стан об’єктів, що суперечить принципам чистого функціонального програмування.
2. **Відсутність першокласних функцій:**
    - У Java функції не є повноцінними об’єктами, і їх не можна оголошувати поза класами.
3. **Використання класів:**
    - Усе має бути частиною класу, навіть якщо це функціональний стиль.

4. **Відсутність каррування та часткового застосування:**
    - Java не надає вбудованих механізмів для каррування функцій (перетворення функцій з кількома аргументами у функції з одним аргументом).

Функціональний стиль у C++

C++ також підтримує функціональне програмування завдяки:

1. **Лямбда-виразам:**
    - Як і в Java, C++ дозволяє створювати анонімні функції.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    std::for_each(numbers.begin(), numbers.end(), [](int n) {
        if (n % 2 == 0) std::cout << n * n << std::endl;
    });
    return 0;
}
```

2. **Функціональні бібліотеки:**
    - У C++ є стандартна бібліотека <functional> для роботи з функціями як об’єктами (std::function, std::bind).
3. **Шаблони:**
    - Шаблони функцій і класів дозволяють створювати узагальнені функціональні рішення.
4. **Чисті функції та незмінюваність:**
    - C++ дозволяє створювати чисті функції, але це залежить від дисципліни програміста, а не від самої мови.

Порівняння функціонального стилю в Java та C++

Особливість	Java	C++
Лямбда-вирази	Підтримуються (з Java 8).	Підтримуються (з C++11).
Функції як об’єкти	Використовуються через функціональні інтерфейси (Function, Consumer).	Використовуються через std::function або лямбда-функції.
Каррування	Не підтримується.	Можна реалізувати вручну або через бібліотеки.
Стрім-парадигма	Streams API для роботи з потоками даних.	Стандартні алгоритми STL (std::transform, std::filter).
Чистота функцій	Немає строгих вимог до чистоти функцій.	Залежить від програміста, є більша свобода в реалізації.
Взаємодія з ООП	Функції належать до класів (через функціональні інтерфейси).	Функції можуть існувати незалежно від класів.

Висновки

- **Java** підтримує функціональний стиль через лямбда-вирази, функціональні інтерфейси та Streams API, але залишається об'єктно-орієнтованою мовою.
- **C++** має більше гнучкості завдяки підтримці як процедурного, так і функціонального програмування, а також можливості працювати з функціями незалежно від класів.
- Java надає більш прості інструменти для інтеграції функціонального стилю в ООП-код, тоді як C++ дає більше свободи, але вимагає більше зусиль для досягнення подібного результату.

- **Чи підтримує Java generic programming? Якщо так - яким чином, якщо ні - чому? Порівняння з C++.**

Так, Java підтримує **узагальнене програмування (generic programming)**. Generics були введені у версії Java 5 і дозволяють створювати класи, інтерфейси та методи, які працюють з різними типами даних, забезпечуючи типобезпеку на рівні компіляції.

Як працюють Generics у Java?

1. **Узагальнені класи:** Узагальнені класи дозволяють створювати класи, які працюють з типами, визначеними при створенні об’єкта.

Приклад:

```
public class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setItem("Hello");
        System.out.println(stringBox.getItem()); // Виведе "Hello"

        Box<Integer> intBox = new Box<>();
        intBox.setItem(123);
        System.out.println(intBox.getItem()); // Виведе 123
    }
}
```

2. **Узагальнені методи:** Методи можуть бути узагальненими, навіть якщо клас не є generic.

Приклад:

```
public class Main {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3};
        String[] strArray = {"A", "B", "C"};

        printArray(intArray); // Виведе 1 2 3
        printArray(strArray); // Виведе A B C
    }
}
```

3. **Узагальнені інтерфейси:** Інтерфейси також можуть бути generic.

Приклад:

```
public interface Pair<K, V> {
    K getKey();
    V getValue();
}

public class KeyValue<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    public KeyValue(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}
```

4. **Обмеження (bounded types):** Generics у Java можуть обмежувати типи через ключове слово extends.

Приклад:

```
public class Main {
    public static <T extends Number> double sum(T a, T b) {
        return a.doubleValue() + b.doubleValue();
    }

    public static void main(String[] args) {
        System.out.println(sum(5, 10));       // Виведе 15.0
        System.out.println(sum(2.5, 3.5));    // Виведе 6.0
    }
}
```

5. **Wildcards (заміщення):** Generics у Java підтримують wildcard ? для створення універсальних методів, які працюють з узагальненими типами.

Приклад:

```
public static void printList(List<?> list) {
    for (Object item : list) {
        System.out.println(item);
    }
}
```

Особливості Generics у Java

1. **Типобезпека:** Generics дозволяють уникати помилок типізації на рівні компіляції.
2. **Стираність типів (type erasure):** У Java generics реалізуються через **стираність типів**, що означає, що інформація про тип видаляється під час компіляції. Наприклад, List<String> і List<Integer> у байт-коді виглядають однаково, як List.

Це обмежує можливість використовувати generics із примітивними типами (наприклад, List<int> недопустимий, потрібно використовувати List<Integer>).

3. **Обмеження через відсутність рефлексії:** Через стираність типів неможливо отримати тип параметра generic під час виконання.

Generics у C++

C++ реалізує узагальнене програмування через **шаблони (templates)**. Шаблони в C++ працюють на рівні компіляції та забезпечують більш потужну та гнучку реалізацію generics у порівнянні з Java.

1. **Шаблони функцій:**

```cpp
#include <iostream>
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add(5, 10) << std::endl; // Виведе 15
    std::cout << add(2.5, 3.5) << std::endl; // Виведе 6.0
    return 0;
}
```

2. **Шаблони класів:**

```cpp
#include <iostream>
template <typename T>
class Box {
private:
    T item;
public:
    void setItem(T item) { this->item = item; }
    T getItem() { return item; }
};

int main() {
    Box<int> intBox;
    intBox.setItem(123);
    std::cout << intBox.getItem() << std::endl;

    Box<std::string> stringBox;
    stringBox.setItem("Hello");
    std::cout << stringBox.getItem() << std::endl;

    return 0;
}
```

3. **Шаблони із спеціалізацією:** C++ дозволяє створювати спеціалізовані версії шаблонів для певних типів.
4. **Підтримка примітивних типів:** На відміну від Java, у C++ шаблони працюють із примітивними типами, такими як int або double, без потреби в обгортках.

Порівняння Java та C++ у контексті узагальненого програмування

Особливість	Java	C++
Механізм реалізації	Generics із type erasure (стираність типів).	Шаблони з компіляцією для кожного типу.
Примітивні типи	Не підтримуються (потрібні обгортки, напр. Integer).	Підтримуються безпосередньо.
Типобезпека	Типобезпека на рівні компіляції.	Типобезпека на рівні компіляції.
Рефлексія	Не підтримується для generics.	Немає потреби у рефлексії для шаблонів.
Спеціалізація	Непідтримується.	Підтримується.
Гнучкість	Більш обмежена через стираність типів.	Більш гнучка, але може ускладнити відлагодження.
Область використання	Орієнтована на спрощення роботи з колекціями.	Використовується для широкого кола завдань.

Висновок

Java підтримує узагальнене програмування через Generics, забезпечуючи типобезпеку та спрощення роботи з колекціями. Однак реалізація через стираність типів створює певні обмеження.

C++ надає потужніший та гнучкіший механізм generics через шаблони, що дозволяє працювати з будь-якими типами, включаючи примітивні, і забезпечує ширші можливості налаштування.

Текстові рядки

- **Як працювати з текстовими рядками в Java? Які типи для цього використовуються? Як створювати змінні, що містять текстові рядки? Навести приклади з власного коду.**

1. **Типи для роботи з рядками:**

- **String**:
  - Це **незмінний (immutable)** клас, що означає, що після створення об'єкта його вміст не можна змінити.
  - Використовується для роботи зі статичними текстами.
- **StringBuilder**:
  - Це **змінний (mutable)** клас, який дозволяє змінювати рядок без створення нового об'єкта.
  - Краще підходить для операцій з частим додаванням або видаленням символів.
- **StringBuffer**:
  - Подібний до **StringBuilder**, але є **потокобезпечним (thread-safe)**.
  - Використовується рідше через нижчу продуктивність у порівнянні з StringBuilder.

---

2. Створення змінних з текстовими рядками:

Варіанти створення String:

1. **Літерали рядків**: Рекомендується використовувати літерали для ефективного використання пам'яті.

```
String greeting = "Hello, World!";
```

За допомогою конструктора:

```
String anotherGreeting = new String("Hello, Java!");
```

Приклад використання StringBuilder:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(", Java!");  // Додає до рядка
System.out.println(sb.toString());  // Виведе: Hello, Java!
```

---

3. Приклади з власного коду:

Просте створення та використання String:

```
public class Main {
    public static void main(String[] args) {
        String name = "Anna";
        String greeting = "Hello, " + name + "!";
        System.out.println(greeting); // Виведе: Hello, Anna!

        // Довжина рядка
        int length = greeting.length();
        System.out.println("Довжина рядка: " + length);

        // Перевірка на порожній рядок
        boolean isEmpty = greeting.isEmpty();
        System.out.println("Рядок порожній? " + isEmpty);
    }
}
```

Використання StringBuilder для зміни тексту:

```
public class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Java");

        // Додавання тексту
        sb.append(" is awesome!");
        System.out.println(sb);  // Виведе: Java is awesome!

        // Вставка тексту
        sb.insert(5, " programming");
        System.out.println(sb);  // Виведе: Java programming is awesome!

        // Видалення частини тексту
        sb.delete(5, 16);
        System.out.println(sb);  // Виведе: Java is awesome!

        // Заміна тексту
        sb.replace(8, 15, "amazing");
        System.out.println(sb);  // Виведе: Java is amazing!
    }
}
```

---

4. Ключові методи класу **String**:

Метод	Опис	Приклад використання
length()	Повертає довжину рядка	"Hello".length(); → 5
charAt(index)	Повертає символ за індексом	"Hello".charAt(1); → 'e'

Метод	Опис	Приклад використання
substring(start, end)	Повертає підрядок	"Hello".substring(0, 2); → "He"
toLowerCase()	Перетворює рядок у нижній регістр	"HELLO".toLowerCase(); → "hello"
toUpperCase()	Перетворює рядок у верхній регістр	"hello".toUpperCase(); → "HELLO"
trim()	Видаляє пробіли на початку і в кінці рядка	" Hello ".trim(); → "Hello"
equals()	Порівнює рядки на рівність	"Hello".equals("hello"); → false

---

В залежності від задачі ви можете використовувати String для простих операцій, а StringBuilder — для складних.

- **Які операції над текстовими рядками підримує Java? Навести приклади з власного коду.**

Java підтримує широкий набір операцій для роботи з текстовими рядками. Нижче наведені основні операції з прикладами коду:

---

1. **Конкатенація рядків**

Об'єднання двох або більше рядків.

```
public class Main {
    public static void main(String[] args) {
        String firstName = "John";
        String lastName = "Doe";
        String fullName = firstName + " " + lastName;  // Конкатенація за допомогою оператора +
        System.out.println("Повне ім'я: " + fullName);  // Виведе: Повне ім'я: John Doe
    }
}
```

---

2. **Порівняння рядків**

- **equals()**: порівнює вміст рядків.
- **equalsIgnoreCase()**: порівнює рядки без урахування регістру.
- **compareTo()**: лексикографічне порівняння.

```
public class Main {
    public static void main(String[] args) {
        String str1 = "Java";
        String str2 = "java";

        System.out.println(str1.equals(str2));          // Виведе: false
        System.out.println(str1.equalsIgnoreCase(str2));// Виведе: true
        System.out.println(str1.compareTo(str2));       // Виведе: -32 (ASCII-різниця між 'J' і 'j')
    }
}
```

---

3. **Операції з індексами**

- **charAt()**: отримання символа за індексом.
- **indexOf()**: знаходження першого входження підрядка.
- **lastIndexOf()**: знаходження останнього входження підрядка.

```
public class Main {
    public static void main(String[] args) {
        String text = "Java programming";

        System.out.println("Символ за індексом 5: " + text.charAt(5)); // Виведе: p
        System.out.println("Індекс 'program': " + text.indexOf("program")); // Виведе: 5
        System.out.println("Останній індекс 'a': " + text.lastIndexOf('a')); // Виведе: 13
    }
}
```

---

4. **Видалення пробілів**

- **trim()**: видаляє пробіли на початку та в кінці рядка.

```
public class Main {
    public static void main(String[] args) {
        String text = "   Hello, Java!   ";
        System.out.println("Оригінал: [" + text + "]");
        System.out.println("Очищений: [" + text.trim() + "]");  // Виведе: [Hello, Java!]
    }
}
```

---

5. **Зміна регістру**

- **toUpperCase()**: змінює всі символи на великі.
- **toLowerCase()**: змінює всі символи на малі.

```
public class Main {
    public static void main(String[] args) {
        String text = "Java Programming";
        System.out.println("Верхній регістр: " + text.toUpperCase()); // Виведе: JAVA PROGRAMMING
        System.out.println("Нижній регістр: " + text.toLowerCase()); // Виведе: java programming
    }
}
```

---

6. **Виділення підрядків**

- **substring()**: повертає частину рядка.

```java
public class Main {
    public static void main(String[] args) {
        String text = "Java programming";
        System.out.println("Підрядок: " + text.substring(5, 16)); // Виведе: programming
    }
}
```

---

7. **Розділення рядка**

- **split()**: розбиває рядок на частини за вказаним роздільником.

```java
public class Main {
    public static void main(String[] args) {
        String text = "apple,banana,orange";
        String[] fruits = text.split(",");

        for (String fruit : fruits) {
            System.out.println(fruit);
        }
        // Виведе:
        // apple
        // banana
        // orange
    }
}
```

---

8. **Заміна символів**

- **replace()**: замінює всі входження символа або підрядка.
- **replaceAll()**: замінює за регулярним виразом.

```java
public class Main {
    public static void main(String[] args) {
        String text = "Java is cool!";
        System.out.println(text.replace("cool", "awesome")); // Виведе: Java is awesome!

        String text2 = "123-456-789";
        System.out.println(text2.replaceAll("-", ":")); // Виведе: 123:456:789
    }
}
```

---

9. **Перевірка початку або кінця рядка**

- **startsWith()**: перевіряє, чи починається рядок з підрядка.

- **endsWith()**: перевіряє, чи закінчується рядок підрядком.

```
public class Main {
    public static void main(String[] args) {
        String text = "Java programming";

        System.out.println(text.startsWith("Java")); // Виведе: true
        System.out.println(text.endsWith("ing"));   // Виведе: true
    }
}
```

---

10. **Перевірка на порожній рядок**

- **isEmpty()**: перевіряє, чи рядок порожній.
- **isBlank()**: перевіряє, чи рядок містить лише пробіли (з Java 11).

```
public class Main {
    public static void main(String[] args) {
        String text1 = "";
        String text2 = "   ";

        System.out.println(text1.isEmpty()); // Виведе: true
        System.out.println(text2.isEmpty()); // Виведе: false
        System.out.println(text2.isBlank()); // Виведе: true (працює з Java 11)
    }
}
```

Висновок

Java забезпечує багатий набір методів для роботи з текстовими рядками. Для більш складних операцій, таких як додавання, видалення чи заміна, зручніше використовувати **StringBuilder** або **StringBuffer**, які дозволяють працювати з рядками ефективніше.

- **Чи можна в Java будувати один текстовий рядок як конкатенацію декількох інших рядків? Якщо можна - як це рекомендовано робити, якщо не можна - чому? Навести приклади з власного коду.**

1. **Конкатенація за допомогою оператора** +

```
public class Main {
    public static void main(String[] args) {
        String part1 = "Hello";
        String part2 = "World";
        String result = part1 + ", " + part2 + "!";
        System.out.println(result); // Виведе: Hello, World!
    }
}
```

Недоліки:

- Якщо використовувати оператор + в циклах, кожна операція створює новий об’єкт рядка через незмінність (immutable) об’єктів класу String. Це може призвести до зниження продуктивності.

---

2. **Використання класу** StringBuilder

Цей підхід рекомендується для частих операцій конкатенації, особливо в циклах, тому що StringBuilder є змінним і не створює нові об’єкти на кожну операцію.

```
public class Main {
    public static void main(String[] args) {
        StringBuilder builder = new StringBuilder();
        builder.append("Hello");
        builder.append(", ");
        builder.append("World!");

        System.out.println(builder.toString()); // Виведе: Hello, World!
    }
}
```

Особливості:

- Ефективний для великих обсягів текстових операцій.
- Не є потокобезпечним, тому для багатопотокового середовища краще використовувати StringBuffer.

---

3. **Використання класу** StringBuffer

Аналогічний до StringBuilder, але підтримує потокобезпечність (thread-safe). Використовується, коли потрібно забезпечити коректну роботу в багатопотоковому середовищі.

Приклад:

```
public class Main {
    public static void main(String[] args) {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Hello");
        buffer.append(", ");
        buffer.append("World!");

        System.out.println(buffer.toString()); // Виведе: Hello, World!
    }
}
```

Недоліки:

- Менш продуктивний у порівнянні з StringBuilder у однопотоковому середовищі.

---

4. **Використання методу** String.join()

Зручний для об’єднання кількох рядків через роздільник.

Приклад:

```
public class Main {
    public static void main(String[] args) {
        String part1 = "Hello";
        String part2 = "World";
        String result = String.join(", ", part1, part2);
        System.out.println(result); // Виведе: Hello, World
    }
}
```

Особливості:

- Рекомендований для об’єднання кількох рядків із фіксованим роздільником.
- Зручний для читабельності коду.

---

5. **Використання** String.format()

Підходить для створення форматованих рядків.

Приклад:

```java
public class Main {
    public static void main(String[] args) {
        String name = "John";
        int age = 25;
        String result = String.format("My name is %s and I am %d years old.", name, age);
        System.out.println(result); // Виведе: My name is John and I am 25 years old.
    }
}
```

Особливості:

- Зручний для читабельного створення складних рядків.
- Використовує синтаксис подібний до printf у C.

---

6. **Порівняння підходів**

Підхід	Продуктивність	Зручність у коді	Переваги
Оператор +	Низька	Висока	Простий для коротких конкатенацій.
StringBuilder	Висока	Середня	Ефективний для великої кількості операцій.
StringBuffer	Середня	Середня	Безпечний для багатопотокового середовища.
String.join()	Висока	Висока	Зручно для об’єднання з роздільником.
String.format()	Середня	Висока	Добре підходить для форматованих рядків.

Рекомендація:

- Короткі рядки: використовуйте оператор + або String.format().
- Багаторазова конкатенація: використовуйте StringBuilder.
- Багатопотокове середовище: використовуйте StringBuffer.
- Об'єднання через роздільник: використовуйте String.join().

- **Чи можна в Java перетворювати числа на їх текстове подання? Як це правильно робити? Навести приклади з власного коду.**

Основні способи перетворення чисел на текст:

---

1. **Використання методу** String.valueOf()

Цей метод приймає число як аргумент і повертає його текстове подання.

```java
public class Main {
    public static void main(String[] args) {
        int number = 123;
        double pi = 3.14159;

        String strNumber = String.valueOf(number);
        String strPi = String.valueOf(pi);

        System.out.println("Ціле число: " + strNumber); // Виведе: Ціле число: 123
        System.out.println("Число з плаваючою точкою: " + strPi); // Виведе: Число з плаваючою точкою: 3.14159
    }
}
```

---

2. **Використання конкатенації з порожнім рядком**

Додаючи число до рядка, Java автоматично перетворює його на текст.

```java
public class Main {
    public static void main(String[] args) {
        int number = 456;

        String strNumber = number + ""; // Перетворення через конкатенацію
        System.out.println("Число: " + strNumber); // Виведе: Число: 456
    }
}
```

---

3. **Використання класу Integer.toString() або Double.toString()**

Ці методи спеціально створені для перетворення чисел у текст.

```java
public class Main {
    public static void main(String[] args) {
        int number = 789;
        double pi = 3.14;

        String strNumber = Integer.toString(number);
        String strPi = Double.toString(pi);

        System.out.println("Ціле число: " + strNumber); // Виведе: Ціле число: 789
        System.out.println("Число з плаваючою точкою: " + strPi); // Виведе: Число з плаваючою точкою: 3.14
    }
}
```

---

4. **Використання класу String.format()**

Цей підхід дозволяє форматувати число перед перетворенням у текст.

```java
public class Main {
    public static void main(String[] args) {
        double pi = 3.14159265359;

        String formattedPi = String.format("%.2f", pi); // Форматування до двох знаків після коми
        System.out.println("Форматоване число: " + formattedPi); // Виведе: Форматоване число: 3.14
    }
}
```

---

5. **Використання класу DecimalFormat**

Для більш складного форматування чисел.

```java
import java.text.DecimalFormat;

public class Main {
    public static void main(String[] args) {
        double salary = 1234567.89;

        DecimalFormat df = new DecimalFormat("#,###.00");
        String formattedSalary = df.format(salary);

        System.out.println("Форматована зарплата: " + formattedSalary); // Виведе: Форматована зарплата: 1,234,567.89
    }
}
```

---

6. **Використання класу NumberFormat**

Цей клас дозволяє форматувати числа відповідно до локалі.

```java
import java.text.NumberFormat;
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
        double amount = 1234567.89;

        NumberFormat nf = NumberFormat.getInstance(Locale.US);
        String formattedAmount = nf.format(amount);

        System.out.println("Форматована сума: " + formattedAmount); // Виведе: Форматована сума: 1,234,567.89
    }
}
```

---

7. **Використання класу BigDecimal**

Якщо потрібно точне подання числа, особливо для фінансових обчислень.

```java
import java.math.BigDecimal;

public class Main {
    public static void main(String[] args) {
        BigDecimal bigDecimal = new BigDecimal("12345.6789");

        String strBigDecimal = bigDecimal.toString();
        System.out.println("BigDecimal у текстовому вигляді: " + strBigDecimal); // Виведе: BigDecimal у текстовому
        //вигляді: 12345.6789
    }
}
```

Висновок

Різні методи підходять для різних задач:

- **Прості перетворення**: String.valueOf(), конкатенація, Integer.toString().
- **Форматовані перетворення**: String.format(), DecimalFormat, NumberFormat.
- **Точне подання чисел**: BigDecimal.

Для більшості базових завдань String.valueOf() є найбільш зручним і універсальним рішенням.

- **Що таке регулярні вирази в Java? Як з ними працювати? Для чого вони потрібні? Навести приклади з власного коду.**

Що таке регулярні вирази в Java?

Регулярні вирази (regular expressions або regex) — це шаблони для пошуку, зіставлення та обробки тексту. В Java регулярні вирази реалізовані через класи з пакету **java.util.regex**, серед яких найпоширенішими є:

- **Pattern** — представляє регулярний вираз.
- **Matcher** — використовується для роботи з текстом, щоб знайти збіги для шаблону.

Регулярні вирази потрібні для:

- Перевірки правильності введення (наприклад, email, номер телефону).
- Пошуку і заміни тексту.
- Парсингу тексту або виділення його частин.

Як працювати з регулярними виразами в Java?

---

*1. **Створення шаблону***

Для створення шаблону використовується клас **Pattern**:

```
Pattern pattern = Pattern.compile("шаблон");
```

---

2. Пошук збігів

Для пошуку збігів у тексті використовується клас **Matcher**:

```
Matcher matcher = pattern.matcher("текст");
```

---

3. Виконання операцій

- **matches()**: перевіряє, чи весь рядок відповідає шаблону.
- **find()**: знаходить збіги у тексті.
- **group()**: повертає знайдений збіг.
- **replaceAll()**: замінює всі збіги.

Приклади використання регулярних виразів

---

*1. **Перевірка email-адреси***

```java
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args) {
        String email = "example@test.com";
        String regex = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,6}$";

        boolean isValid = Pattern.matches(regex, email);
        System.out.println("Чи є email валідним? " + isValid); // Виведе: Чи є email валідним? true
    }
}
```

---

*2. **Пошук усіх чисел у тексті***

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args) {
        String text = "У рядку є числа: 123, 456 і 789.";
        Pattern pattern = Pattern.compile("\\d+"); // \\d+ – знаходить послідовності цифр
        Matcher matcher = pattern.matcher(text);

        System.out.println("Знайдені числа:");
        while (matcher.find()) {
            System.out.println(matcher.group()); // Виведе: 123, 456, 789
        }
    }
}
```

---

*3. **Заміна всіх пробілів на коми***

```java
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args) {
        String text = "Java is fun!";
        String result = text.replaceAll("\\s", ","); // \\s – пробіли
        System.out.println("Результат: " + result); // Виведе: Java,is,fun!
    }
}
```

---

4. ***Розділення рядка за шаблоном***

```java
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args) {
        String text = "apple,banana;cherry.orange";
        String[] fruits = text.split("[,;.]+"); // [,;.]+ – будь-який із символів: , ; . один або більше разів

        System.out.println("Розділені слова:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
        // Виведе:
        // apple
        // banana
        // cherry
        // orange
    }
}
```

---

5. ***Виділення підрядків з тексту***

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args) {
        String text = "Дата народження: 1990-05-25, номер паспорта: AB123456.";
        Pattern pattern = Pattern.compile("\\d{4}-\\d{2}-\\d{2}"); // Шукає дати у форматі YYYY-MM-DD
        Matcher matcher = pattern.matcher(text);

        if (matcher.find()) {
            System.out.println("Знайдена дата: " + matcher.group()); // Виведе: Знайдена дата: 1990-05-25
        }
    }
}
```

Особливі символи у регулярних виразах

Символ	Опис
.	Будь-який символ, крім нового рядка.
*	0 або більше повторень.
+	1 або більше повторень.
?	0 або 1 повторення.
\\d	Будь-яка цифра (0-9).
\\D	Будь-який символ, крім цифри.
\\w	Будь-яка буква, цифра або _.
\\W	Будь-який символ, крім \\w.
\\s	Будь-який пробільний символ.
^	Початок рядка.
$	Кінець рядка.
[abc]	Один із символів: a, b або c.
[a-z]	Будь-яка літера від a до z.
[^abc]	Будь-який символ, крім a, b, c.

Висновок

Регулярні вирази — це потужний інструмент для роботи з текстом у Java. Їх можна використовувати для перевірки введення, парсингу тексту, пошуку або заміни підрядків. Для складніших завдань, наприклад, роботи з великими текстовими даними, регулярні вирази значно полегшують і скорочують код.

Консоль та файли

- **Яким чином можна вивести дані про об'єкт в консоль? В файл? Навести приклади з власного коду.**
    1. **Вивід у консоль**:
       У Java для виводу в консоль використовується System.out.println(). Якщо ми хочемо реалізувати універсальне представлення об’єкта у вигляді рядка, зазвичай перевизначаємо метод toString() класу Object.
    2. **Вивід у файл:**
       Для виводу у файл у Java зазвичай використовують FileWriter, PrintWriter або інші класи потоків введення/виведення.

Приклад коду:

```java
import java.io.*;

class Point {
    private double x;
    private double y;

    public Point(double xCoord, double yCoord) {
        this.x = xCoord;
        this.y = yCoord;
    }

    // Перевизначаємо toString() для зручного перетворення об’єкта у рядок
    @Override
    public String toString() {
        return "Point(" + x + ", " + y + ")";
    }
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point(3.14, 2.71);

        // Вивід у консоль
        System.out.println("Вивід у консоль: " + p);

        // Вивід у файл
        File file = new File("points.txt");
        try (PrintWriter writer = new PrintWriter(new FileWriter(file, true))) {
            writer.println(p); // метод println викликає p.toString() автоматично
        } catch (IOException e) {
            System.err.println("Помилка роботи з файлом: " + e.getMessage());
        }
    }
}
```

- **Як працювати з файлами в Java? Які класи для цього використовуються? Навести приклади з власного коду.**

Основні класи для роботи з файлами в Java

1. **File**: Клас java.io.File використовується для представлення шляху до файлу або директорії. Він не виконує безпосереднє читання/запис даних, а лише описує сам файл (його розташування, назву тощо).
2. **Класи для запису у файл:**
    - FileWriter: Низькорівневий символьний потік для запису даних у файл.
    - BufferedWriter: Обгортка над FileWriter, яка забезпечує буферизований запис, що покращує продуктивність.
    - PrintWriter: Зручний для форматованого виводу, надає методи println(), printf(). Часто використовується з FileWriter.
3. **Класи для читання з файлу:**
    - FileReader: Низькорівневий символьний потік для читання символів з файлу.
    - BufferedReader: Обгортка над FileReader, що забезпечує буферизований доступ та надає зручний метод readLine() для читання рядків.
4. **Альтернативні/Новіші можливості:**

- Files (клас з пакету java.nio.file): Містить статичні методи для читання і запису файлів повністю або рядково.
- Path та Paths: З класу java.nio.file для більш гнучкої та платформонезалежної роботи з шляхами до файлів.

Запис у файл

Нижче наведено приклад, який демонструє запис у файл використовуючи FileWriter та PrintWriter. Код також ілюструє використання конструкції try-with-resources, щоб автоматично закривати потоки:

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class WriteExample {
    public static void main(String[] args) {
        File file = new File("output.txt");

        // Використання try-with-resources гарантує автоматичне закриття ресурсів
        try (PrintWriter writer = new PrintWriter(new FileWriter(file, true))) {
            writer.println("Це перший рядок у файлі.");
            writer.println("Це другий рядок.");
            System.out.println("Дані успішно записано у файл: " + file.getAbsolutePath());
        } catch (IOException e) {
            System.err.println("Помилка під час запису у файл: " + e.getMessage());
        }
    }
}
```

Коментарі:

- Конструктор FileWriter(file, true) відкриває файл у режимі додавання. Якщо флаг true не вказано, файл буде перезаписано.
- PrintWriter використовується для зручності: println() автоматично додає символ нового рядка, а також дає змогу форматувати вивід.
- Всі ресурси будуть закриті автоматично після завершення блоку try.

Читання з файлу

Нижче наведено приклад читання вмісту файлу пострядково за допомогою BufferedReader:

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadExample {
    public static void main(String[] args) {
        File file = new File("output.txt");

        // Використовуємо try-with-resources для безпечного закриття потоків
        try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println("Прочитано з файлу: " + line);
            }
        } catch (IOException e) {
            System.err.println("Помилка під час читання файлу: " + e.getMessage());
        }
    }
}
```

Коментарі:

- BufferedReader надає метод readLine(), який дозволяє читати файл пострядково.
- Якщо файл відсутній або недоступний для читання, буде згенеровано IOException, яка обробляється у блоці catch.

Альтернативний підхід з використанням Files та Path (Java NIO)

Якщо ми хочемо прочитати всі рядки файлу за один раз у список, можна скористатися класом Files:

```java
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.List;

public class ReadAllLinesExample {
    public static void main(String[] args) {
        Path path = Paths.get("output.txt");

        try {
            List<String> allLines = Files.readAllLines(path);
            for (String line : allLines) {
                System.out.println("Рядок з файлу: " + line);
            }
        } catch (IOException e) {
            System.err.println("Помилка під час читання файлу: " + e.getMessage());
        }
    }
}
```

Коментарі:

- Files.readAllLines(path) повертає List<String> усіх рядків файлу.
- Цей спосіб зручний, коли файл відносно невеликий, оскільки вся його інформація зчитується одразу у пам’ять.

- **Як працювати з консоллю в Java? Як виводити дані, як вводити дані? Навести приклади з власного коду.**

Вивід даних у консоль

Для виводу тексту в консоль в Java використовується стандартний вихідний потік System.out. Найпоширеніші методи:

- System.out.print(...) – виводить дані без переходу на новий рядок.
- System.out.println(...) – виводить дані з переходом на новий рядок.
- System.out.printf(...) – надає можливість форматованого виводу, подібно до printf у мовах на зразок C/C++.

Приклад виводу в консоль:

```java
public class ConsoleOutputExample {
    public static void main(String[] args) {
        System.out.println("Це рядок з переходом на новий рядок");
        System.out.print("Це рядок без переходу на новий рядок");
        System.out.println(" - продовження тексту в тому ж рядку");

        int x = 42;
        double pi = 3.14159;
        System.out.printf("Число x = %d, а число pi ≈ %.2f%n", x, pi);
        // %d – формат для цілих чисел, %.2f – формат з двома знаками після коми для чисел з плаваючою крапкою
        // %n – перехід на новий рядок незалежно від платформи
    }
}
```

Введення даних з консолі

Для введення даних з консолі зазвичай використовується клас Scanner з пакету java.util. Об’єкт Scanner, створений з System.in як джерела даних, дозволяє читати строки, цілі числа, числа з плаваючою крапкою тощо.

Основні методи Scanner:

- nextLine() – читає весь рядок тексту (до символа нової строки).
- next() – читає одне слово (до роздільника, наприклад, пробілу).
- nextInt() – читає наступне введене число як int.
- nextDouble() – читає наступне введене число як double.
- hasNextInt(), hasNextDouble() та інші методи – дозволяють перевірити, чи є в потоці наступне значення відповідного типу.

Приклад введення з консолі:

```java
import java.util.Scanner;

public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Введіть своє ім'я: ");
        String name = scanner.nextLine(); // читаємо весь рядок введеного тексту

        System.out.print("Введіть свій вік: ");
        int age = scanner.nextInt(); // читаємо ціле число

        // Виведемо зворотній зв'язок у консоль
        System.out.println("Привіт, " + name + "! Тобі " + age + " років.");

        scanner.close(); // закриваємо сканер
    }
}
```

Коментарі:

- scanner.close() бажано викликати після завершення роботи з введенням, але для System.in закриття не завжди обов’язкове.
- Якщо між nextInt() (або іншим next-методом) та nextLine() виникають проблеми з пропуском рядка, можна перед nextLine() викликати додатковий scanner.nextLine() для "очистки" буфера.

Альтернативний спосіб введення

Починаючи з Java 6, також доступний клас Console (через System.console()), який дозволяє читати пароль без відображення його на екрані. Однак System.console() поверне null, якщо програма не запущена в "реальній" консолі (наприклад, при запуску з IDE).

Приклад використання Console:

```java
public class ConsoleExample {
    public static void main(String[] args) {
        java.io.Console console = System.console();
        if (console == null) {
            System.out.println("Консоль недоступна (можливо запущено з IDE).");
            return;
        }

        String userName = console.readLine("Введіть ім'я: ");
        char[] password = console.readPassword("Введіть пароль: ");

        System.out.println("Вітаємо, " + userName + "! Ваш пароль містить " + password.length + " символів.");

        // З міркувань безпеки після використання масив password можна обнулити:
        java.util.Arrays.fill(password, ' ');
    }
}
```

Коментарі:

- readLine() та readPassword() дозволяють більш гнучко працювати з введенням, причому readPassword() не відображає введені символи на екрані.
- Застосування System.console() частіше буде корисним при написанні консольних додатків, які запускаються з командного рядка, а не з середовища розробки.

Класи та методи

- **Які стандартні методи містять будь-які об'єкти в Java? Чи треба визначати ці методи для кожного класу? Навести приклади з власного коду.**

Стандартні методи класу Object

Усі класи в Java неявно наслідуються від класу Object, тому кожен об'єкт містить принаймні такі методи:

1. **toString()**
    - Повертає рядкове представлення об’єкта.
    - За замовчуванням повертає рядок виду: ІмʼяКласу@ХешКодУШістнадцятковійФормі.
    - Зазвичай перевизначається для надання більш інформативного опису об'єкта.
2. **equals(Object obj)**
    - Порівнює цей об'єкт з іншим на рівність.
    - За замовчуванням перевіряє, чи посилання на об'єкти співпадають.
    - Рекомендується перевизначати, якщо є потреба порівнювати об'єкти за їх значенням (наприклад, для сутностей з однаковими даними).
3. **hashCode()**
    - Повертає хеш-код цілого типу для об’єкта.

- За замовчуванням базується на адресі об’єкта в пам’яті.
- Якщо перевизначено equals(), бажано також перевизначити hashCode() для узгодженості, особливо якщо об’єкти будуть використовуватись у структурах даних на зразок HashMap, HashSet.

4. **getClass()**
   - Повертає об’єкт типу Class<?> який представляє клас даного об’єкта.
5. **notify(), notifyAll(), wait()**
   - Методи для синхронізації потоків.
   - Використовуються у блоках синхронізації для управління доступом до ресурсів між різними потоками.
6. **finalize()** (Застарілий, не рекомендується використовувати)
   - Викликався збирачем сміття (GC) перед видаленням об’єкта з пам’яті.
   - Починаючи з Java 9, позначений застарілим, оскільки не є надійним способом керування ресурсами. Рекомендується застосовувати механізми try-with-resources та інші керовані ресурси.

Чи треба визначати ці методи для кожного класу?

- **Не обов’язково.** Кожен клас автоматично успадковує ці методи з Object, і якщо вам підходить їх стандартна реалізація, то додатково визначати чи перевизначати нічого не потрібно.
- **Коли варто перевизначати:**
  - toString(): Якщо ви хочете, щоб ваш клас повертав зрозумілий текстовий опис об’єкта, а не стандартне Ім’яКласу@ХешКод.
  - equals() та hashCode(): Якщо ваші об’єкти порівнюються за значеннями полів, а не за посиланнями. Це особливо важливо для коректної роботи з колекціями, як-от HashSet чи HashMap, де логіка унікальності чи пошуку об’єкта базується на equals() і hashCode().
- getClass(), notify(), notifyAll(), wait() зазвичай не перевизначаються. Вони мають семантику, зв’язану з внутрішнім устроєм JVM та роботою зі потоками, або ж призначені для отримання інформації про клас.

Приклад коду

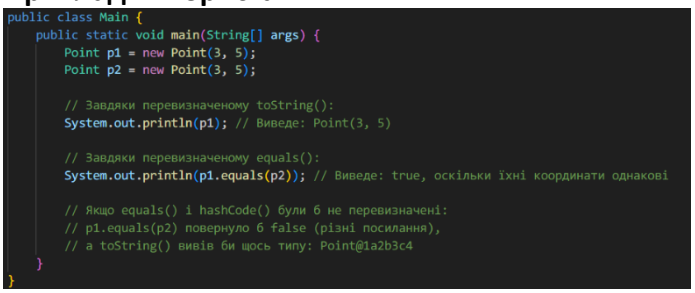
Приклад класу без перевизначення методів:

```java
public class SimplePoint {
    private int x;
    private int y;

    public SimplePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Тут жодні методи не перевизначено, використовуються унаслідувані з Object
    // equals() порівнюватиме за посиланнями, toString() поверне SimplePoint@hexcode
}
```

Приклад класу з перевизначенням toString(), equals() та hashCode():

```java
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Перевизначаємо toString() для більш інформативного представлення
    @Override
    public String toString() {
        return "Point(" + x + ", " + y + ")";
    }

    // Перевизначаємо equals() для логіки порівняння двох точок за їх координатами
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // однакові посилання
        if (obj == null || getClass() != obj.getClass()) return false;
        // Переконуємось, що об'єкт належить тому ж класу

        Point other = (Point) obj;
        return this.x == other.x && this.y == other.y;
    }

    // Перевизначаємо hashCode() узгоджено з equals()
    @Override
    public int hashCode() {
        int result = 17;
        result = 31 * result + x;
        result = 31 * result + y;
        return result;
    }

    // Інші методи, якщо потрібно
}
```

Приклад використання:



```java
public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(3, 5);
        Point p2 = new Point(3, 5);

        // Завдяки перевизначеному toString():
        System.out.println(p1); // Виведе: Point(3, 5)

        // Завдяки перевизначеному equals():
        System.out.println(p1.equals(p2)); // Виведе: true, оскільки їхні координати однакові

        // Якщо equals() і hashCode() були б не перевизначені:
        // p1.equals(p2) повернуло б false (різні посилання),
        // а toString() вивів би щось типу: Point@1a2b3c4
    }
}
```

- **Як іменуються класи, методи, змінні в Java? Як уникати конфліктів імен? Порівняння з C++.**

Стандарти іменування в Java

1. **Іменування класів і інтерфейсів:**
    - Використовується **PascalCase**: кожне слово в назві починається з великої літери. Наприклад: MyClass, CustomerAccount, ListProcessor.
    - Класи зазвичай — це іменники або іменникові фрази.
    - Інтерфейси можуть інколи мати прикметникову форму (наприклад, Comparable, Runnable).
2. **Іменування методів:**
    - Використовується **camelCase**: перше слово в нижньому регістрі, наступні слова починаються з великої літери.
    - Імена методів зазвичай є дієсловами або фразами, що починаються з дієслова, оскільки методи виконують дію.
      Наприклад: calculateSum(), printResults(), getUserName().
3. **Іменування змінних (локальних змінних та полів класів):**
    - Використовується **camelCase**, починаючи з малої літери.
      Наприклад: userName, totalCount, maxValue.
    - Імена змінних зазвичай відображають те, що зберігається в змінній, і є іменниками або фразами.
    - Константи зазвичай позначаються великими літерами з підкресленнями як роздільниками слів.
      Наприклад: MAX_SIZE, DEFAULT_TIMEOUT.
4. **Іменування пакетів:**
    - Імена пакетів — лише малі літери. В якості роздільників використовують крапку.
      Наприклад: com.example.myapp.utilities.
    - Часто імена пакетів починаються з реверсованого доменного імені компанії, щоб уникнути конфліктів.

Уникнення конфліктів імен

В Java для уникнення конфліктів імен широко використовується механізм пакетів. Пакет виступає логічним простором імен. Якщо два класи мають однакові назви, але належать до різних пакетів, повного конфлікту не буде: можна використовувати повну назву з пакетом. Наприклад, com.example.utils.DateUtils та org.company.date.DateUtils — дві різні повні назви. Якщо виникає необхідність використовувати обидва класи з однаковим ім’ям в одному файлі, можна:

- Використовувати повну кваліфікацію, тобто:
  com.example.utils.DateUtils myDateUtils = new com.example.utils.DateUtils();
- Змінити імпорт або перейменувати власні класи, щоб уникнути плутанини.

В цілому пакети в Java виконують функцію простору імен, забезпечуючи контекст для унікалізації імен класів.

Порівняння з C++

Нотація іменування:

- В C++ також існують загальноприйняті стилі, проте мова сама по собі не надає офіційних рекомендацій. Організації та проекти нерідко мають власні керівництва.
- Поширені варіанти в C++: PascalCase для класів, camelCase для методів і змінних або інколи використовують підкреслення, наприклад snake_case, залежно від стилю, прийнятого в конкретному проекті.
- C++ менш жорсткий у цьому питанні, розробники можуть обрати свій підхід. В Java конвенції більш стандартні та усталені.

Простори імен (namespaces) vs пакети (packages):

- В C++ уникнення конфліктів імен здійснюється за допомогою namespace.
- Подібно до пакетів у Java, namespace у C++ дозволяє групувати класи та функції і уникати конфліктів.
- Проте пакети у Java і namespace у C++ організовані по-різному:
    - Java пакети мають прямий зв'язок з файловою системою — кожен пакет відповідає директорії.
    - В C++ namespace не вимагає такої жорсткої відповідності файловій структурі, вони логічні та не прив'язані жорстко до директорій.

Унікальність імен:

- Java заохочує унікальність іменування пакетів на основі доменних імен, що допомагає уникати глобальних конфліктів, особливо при використанні сторонніх бібліотек.
- В C++ розробнику доводиться самостійно турбуватися про унікальність назв, часто використовуючи префікси або власні простори імен.

- **Яким чином правильно порівнювати об'єкти в Java? Чи треба для порівняння об'єктів визначати якісь власні методи? Якщо такі методи визначено, чи треба визначати якісь інші методи? Навести приклади з власного коду.**

Порівняння об'єктів у Java

В Java об'єкти можна порівнювати двома основними способами:

1. **Порівняння посилань (оператор ==):**
   Оператор == перевіряє, чи два посилання вказують на один і той самий об'єкт у пам'яті. Це порівняння на рівні адреси, а не на рівні "значення" об'єктів.
2. **Порівняння за вмістом (equals()):**
   Метод equals() визначений у класі Object. За замовчуванням він поводиться так само, як == (порівнює посилання), але більшість стандартних класів (наприклад, String) перевизначають його так, щоб порівнювати логічний вміст об'єктів.

Чи треба для порівняння об'єктів визначати власні методи?

Якщо ви розробляєте власний клас і хочете порівнювати його екземпляри за логічним змістом (значеннями полів), а не за посиланнями, то:

- Вам треба **перевизначити метод equals()**.
- Якщо ви перевизначили equals(), ви **також повинні перевизначити hashCode()**, аби ці два методи були узгоджені. Це вимога, закріплена в контракті між equals() і hashCode() в Java.

Чому це важливо?

Багато структур даних у Java (наприклад, HashSet, HashMap) використовують hashCode() для пошуку об'єктів. Якщо дві логічно рівні (за equals()) сутності мають різні хеш-коди, це призведе до непередбачуваної поведінки структур даних.

Контракт між equals() і hashCode()

Контракт, продиктований документацією Java, визначає, що:

1. Якщо a.equals(b) == true, тоді a.hashCode() має дорівнювати b.hashCode().

2. Якщо a.equals(b) == false, тоді hashCode() може бути різним (не зобов’язано, але бажано, щоб було різним задля рівномірного розподілу хешів).
3. При кожному виклику hashCode() для одного й того самого об’єкта результат повинен бути стабільним, поки стан об’єкта не змінюється.

Приклад коду з власного класу

Припустимо, ми маємо клас Point, який містить координати x та y. Ми хочемо вважати дві точки рівними, якщо вони мають однакові координати.

```java
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Перевизначуємо equals(), щоб порівняти логічний зміст
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // Перевірка на те ж саме посилання
        if (obj == null || getClass() != obj.getClass()) return false;
        // Перевірка на null та однаковий клас

        Point other = (Point) obj;
        // Порівнюємо поля
        return this.x == other.x && this.y == other.y;
    }

    // Перевизначуємо hashCode() згідно з логікою equals()
    @Override
    public int hashCode() {
        int result = 17;    // початкове просте число
        result = 31 * result + x; // множимо результат на 31 та додаємо x
        result = 31 * result + y; // множимо результат на 31 та додаємо y
        return result;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(3, 5);
        Point p2 = new Point(3, 5);
        Point p3 = new Point(4, 6);

        System.out.println(p1.equals(p2)); // true, оскільки поля однакові
        System.out.println(p1.equals(p3)); // false, поля різні

        // Використання у HashSet:
        java.util.HashSet<Point> set = new java.util.HashSet<>();
        set.add(p1);
        set.add(p2);
        // p2 не буде додано вдруге, оскільки p1.equals(p2) == true і hashCode() однаковий
        System.out.println(set.size()); // 1

        // Якби ми не перевизначили equals() і hashCode(),
        // p1 та p2 вважалися б різними об’єктами, і size() був би 2.
    }
}
```

Чи потрібно визначати інші методи?

- При порівнянні **об’єктів за значенням** головне — правильно перевизначити equals() та hashCode().
- Якщо ви хочете, щоб ваш клас можна було порівнювати у відношенні порядку (наприклад, щоб об’єкти можна було сортувати), то слід реалізувати інтерфейс Comparable<T> і визначити метод compareTo(T o). Це вже інше завдання — воно відповідає за порівняння не на рівність, а за визначення, який об’єкт більше, менше чи дорівнює іншому об’єкту за певним критерієм.

• Які модифікатори можна застосовувати до класів, полів, методів в Java? Порівняння з C++.

Модифікатори доступу в Java

В Java існує чотири рівні доступу, які контролюють видимість класів, полів та методів:

1. **public**
    - Доступний усюди, де тільки можливо.
    - Клас, оголошений як public, доступний поза межами свого пакету.
    - Поля та методи public доступні для використання з будь-якого іншого класу (якщо цей клас видимий).
2. **protected**

    - Доступний у тому ж пакеті та в підкласах (навіть якщо підклас знаходиться в іншому пакеті).
    - Не може бути застосований до топ-левел класу, але може бути застосований до внутрішніх (inner) класів, методів, полів.
3. **(package-private)** (за замовчуванням, коли не вказано жодного модифікатора)
    - Якщо не вказати явно жодного модифікатора доступу для класу, методу або поля, воно буде доступне в межах того ж пакету.
    - Такий доступ ще називають "default package" або просто "default".
4. **private**
    - Видимість лише в межах класу, де оголошено член (поле/метод).
    - Не може бути застосований до топ-левел класів (лише до внутрішніх класів, полів і методів).

Інші модифікатори в Java

Окрім модифікаторів доступу, існують й інші:

1. **static**
    - Застосовується до методів, полів, вкладених (inner) класів.
    - Статичні елементи належать класу в цілому, а не конкретному екземпляру.
    - Поля static ініціалізуються при завантаженні класу, методи static можна викликати без створення об’єкта.
2. **final**
    - Застосовується до класів, методів і полів.
    - Клас final неможливо успадкувати.
    - Метод final неможливо перевизначити в підкласі.
    - Поле final можна присвоїти один раз (під час оголошення чи в конструкторі) — після цього його значення змінити неможливо.
3. **abstract**
    - Застосовується до класів і методів.
    - Абстрактний клас не можна створити напряму (не можна створити його екземпляр), він призначений для успадкування.
    - Абстрактний метод не має тіла і повинен бути реалізований у підкласах.
4. **synchronized**
    - Застосовується до методів і блоків коду.
    - Гарантує взаємну виключність доступу в багатопотоковому середовищі.
5. **native**
    - Застосовується до методів, що реалізовані не на Java, а, наприклад, на C чи C++ через JNI.
6. **strictfp**
    - Застосовується до класів, інтерфейсів та методів.
    - Гарантує однакову точність обчислень з плаваючою крапкою незалежно від платформи.

Модифікатори у C++ (порівняння)

В C++ немає ключових слів public, protected, private для топ-левел функцій у вигляді модифікаторів доступу перед визначенням, як у Java. Проте вони є для класів і структур, але працюють по-іншому:

- **public, protected, private**:
  В C++ вони застосовуються до класів/структур, але за замовчуванням для класу всі члени — private, а для struct — public.
  У Java за замовчуванням для класів верхнього рівня доступ package-private, а для членів класу без модифікатора — теж package-private.
- C++ не має окремого поняття package-private. Видимість контролюється переважно на рівні класу та простору імен (namespace).

- **static в C++**:
  - static для методів класу схожий за сенсом: метод стає "методом класу".
  - static для змінних класу також означає, що змінна є "спільною" для всіх екземплярів.
  - Проте у файлах, оголошення static перед глобальною змінною чи функцією обмежує їх видимість до поточного перекладного блоку (translation unit).
- **final у C++11 і пізніше**:
  - C++11 запровадив ключове слово final для класів і методів, яке схоже за логікою до Java.
  - До C++11 такої можливості не було стандартними засобами.
- **abstract**:
  В C++ немає ключового слова abstract, але є поняття чисто віртуальних функцій (pure virtual functions). Клас з хоча б однією чисто віртуальною функцією — це абстрактний клас, його не можна створити. Це схоже на abstract методи в Java, але реалізується синтаксично через = 0 для віртуальних методів.
- **synchronized та інші модифікатори поточності**: В C++ немає прямого аналога synchronized. Синхронізація досягається через інші механізми (std::mutex, std::lock_guard тощо). Таким чином, потоки і синхронізація вирішуються більш низькорівнево в C++.
- **strictfp, native**:
  В C++ немає аналогів strictfp чи native.
  native в Java — це специфіка взаємодії з не-Java кодом. В C++ уся мова "рідна", і немає потреби у спеціальному ключовому слові для взаємодії з "некооригінальним кодом".
  Щодо strictfp, в C++ точність операцій з плаваючою крапкою визначається стандартами IEEE та конкретною реалізацією компілятора. Немає окремого модифікатора, щоб примусово узгодити точність.

• Чи підтримує Java конструктори? деструктори? copy constructors? move constructors? default constructors? виклик конструктора того ж класу? виклик конструктора батьківського класу? Порівняння з C++.

Нижче наведено докладну відповідь на питання щодо підтримки конструкторів, деструкторів, копіювальних та переміщувальних конструкторів, а також механізмів виклику конструкторів у Java. Відповідь також містить порівняння з C++, щоб висвітлити відмінності та подібності між цими двома мовами програмування. Кожен розділ супроводжується прикладами коду для кращого розуміння.

---

1. Конструктори в Java
Підтримка конструкторів:

- **Конструктори** в Java повністю підтримуються.
- Вони використовуються для ініціалізації об'єктів під час їх створення.
- Конструктор має таку ж назву, як і клас, і не має типу повернення.

2. Деструктори в Java
Чи підтримує Java деструктори?

- **Java не має деструкторів** у тому сенсі, як їх розуміють у C++.
- Натомість, Java використовує **Garbage Collection (GC)** для автоматичного управління пам'яттю, тобто об'єкти звільняються автоматично, коли на них більше немає посилань.
- Існує метод finalize(), який можна перевизначити для виконання очищувальних дій перед збиранням сміття, але **він застарів і не рекомендується до використання**. Починаючи з Java 9, метод finalize() позначено як застарілий.

Альтернативи деструкторам:

- **try-with-resources**: Використовується для автоматичного закриття ресурсів, таких як потоки вводу/виводу.
- **Інтерфейс AutoCloseable**: Дозволяє визначити метод close(), який буде викликаний автоматично при закритті ресурсу.

3. Copy Constructors (Копіювальні конструктори) в Java

Чи підтримує Java копіювальні конструктори?

- **Java не має вбудованої підтримки копіювальних конструкторів** як у C++.
- Однак, ви можете реалізувати копіювання об'єктів вручну, створюючи власні конструктори, які приймають об'єкт того ж класу і копіюють його поля.

4. Move Constructors (Переміщувальні конструктори) в Java

Чи підтримує Java переміщувальні конструктори?

- **Java не підтримує переміщувальні конструктори** як у C++ (наприклад, конструктори з rvalue-ссилками Point(Point&&)).
- Оскільки Java використовує **Garbage Collection**, концепція переміщення ресурсів не застосовується так само, як у C++.
- Проте, ви можете використовувати **Reference Variables** і **Immutable Objects** для ефективного управління ресурсами.

Альтернативи:

- Використання **Immutable Objects**: Об'єкти, стан яких не змінюється після створення, що дозволяє безпечно передавати посилання на них.
- Використання **Builder Pattern**: Для створення об'єктів зі складними конструкторами.

5. Default Constructors (Конструктори за замовчуванням) в Java

Що таке конструктори за замовчуванням?

- **Конструктор за замовчуванням** – це конструктор без параметрів, який автоматично генерується компілятором, якщо в класі **не визначено жодного іншого конструктора**.

Особливості:

- Якщо ви визначаєте хоча б один конструктор (з параметрами або без), компілятор **не створює** конструктор за замовчуванням автоматично.
- Якщо вам потрібен конструктор за замовчуванням поруч із іншими конструкторами, його слід визначити вручну.

6. Виклик конструктора того ж класу (Constructor Chaining) в Java

Механізм:

- **Constructor Chaining** дозволяє викликати один конструктор з іншого конструктора в тому ж класі.
- Виконується за допомогою ключового слова this().
- Виклик this() **повинен бути першою** інструкцією в конструкторі.

7. Виклик конструктора батьківського класу (Superclass Constructor) в Java

Механізм:

- Для виклику конструктора **батьківського класу** використовується ключове слово super().
- Виклик super() **повинен бути першою** інструкцією в конструкторі підкласу.
- Якщо super() не викликається явно, компілятор автоматично додає виклик до **конструктора за замовчуванням** батьківського класу.

8. Порівняння з C++

Особливість	**Java**	**C++**
Конструктори	Повністю підтримуються. Включають конструктори з параметрами, конструктори за замовчуванням.	Повністю підтримуються. Підтримка конструктора копіювання та переміщення (C++11 і вище).

Особливість	Java	C++
Деструктори	**Не підтримуються** в класичному розумінні. Використовується GC та finalize() (застарілий).	Підтримуються. Використовуються для звільнення ресурсів, визначаються за допомогою тильди ~.
Copy Constructors	**Не підтримуються** як в C++. Можна реалізувати вручну.	Підтримуються. Використовуються для глибокого або поверхневого копіювання об'єктів.
Move Constructors	**Не підтримуються**.	Підтримуються (C++11 і вище). Використовуються для переміщення ресурсів, оптимізуючи продуктивність.
Default Constructors	Автоматично генерується, якщо не визначено жодного конструктора.	Автоматично генерується, якщо не визначено інших конструкторів.
Constructor Chaining (виклик конструктора того ж класу)	Підтримується за допомогою this().	Підтримується за допомогою делегування конструкторів (C++11 і вище).
Виклик конструктора батьківського класу	Виконується через super().	Виконується через список ініціалізації (ініціалізатори базових класів після двокрапки :).

Додаткові Порівняння:

- **Управління ресурсами:**
  - **Java:** Автоматичне управління пам'яттю через GC. Використання try-with-resources для управління іншими ресурсами.
  - **C++:** Розробник відповідає за управління ресурсами. Використання RAII (Resource Acquisition Is Initialization) для автоматичного управління життєвим циклом ресурсів через деструктори.
- **Перегрузка конструкторів:**
  - **Java:** Підтримується перегрузка конструкторів (багато конструкторів з різними параметрами).
  - **C++:** Підтримується перегрузка конструкторів, а також **delegating constructors** (C++11 і вище).

Чи можна в Java визначати клас в середині іншого класу? в середині методу? Якщо можна - які особливості таких класів, якщо не можна - чому? Порівняння з C++.

1. Внутрішні класи в Java

1.1. Визначення класу всередині іншого класу

У Java можливе визначення класу всередині іншого класу. Такий клас називається **внутрішнім класом** (nested class). Внутрішні класи поділяються на кілька типів:

1. **Статичний внутрішній клас (static nested class)**
2. **Нестатичний внутрішній клас (inner class)**
3. **Локальний клас (local class)** — визначається всередині методу
4. **Анонімний клас (anonymous class)** — визначається без імені, зазвичай при ініціалізації об'єктів

1.2. Статичний внутрішній клас

Особливості:

- Визначається зі статичним модифікатором static.

- Не має доступу до нестатичних членів зовнішнього класу без посилання на його екземпляр.
- Може мати свої власні статичні члени.

1.3. Нестатичний внутрішній клас (Inner Class)

Особливості:

- Не має модифікатора static.
- Має доступ до всіх членів зовнішнього класу, включаючи приватні.
- Для створення екземпляру внутрішнього класу потрібен екземпляр зовнішнього класу.

1.4. Локальний клас (Local Class)

Особливості:

- Визначається всередині методу.
- Може мати тільки локальні змінні, що є кінцевими або ефективно кінцевими.
- Не може мати модифікатор доступу (public, private, protected).

1.5. Анонімний клас (Anonymous Class)

Особливості:

- Визначається без імені під час створення об'єкта.
- Використовується для реалізації інтерфейсів або наслідування класів на льоту.
- Переважно використовується для одноразових потреб, наприклад, у обробниках подій або колбэках.

2. Особливості внутрішніх класів у Java

2.1. Доступ до членів зовнішнього класу

- **Статичні внутрішні класи** мають доступ тільки до статичних членів зовнішнього класу.
- **Нестатичні внутрішні класи** мають доступ до всіх членів зовнішнього класу, включаючи приватні.

2.2. Життєвий цикл

- **Статичні внутрішні класи** існують незалежно від екземплярів зовнішнього класу.
- **Нестатичні внутрішні класи** прив’язані до конкретного екземпляру зовнішнього класу.

2.3. Взаємодія з C++

У C++ внутрішні класи також підтримуються, але їх функціональність дещо відрізняється:

- **У C++** внутрішні класи не мають автоматичного доступу до приватних членів зовнішнього класу. Для доступу потрібно явно оголосити їх як друзі (friend).
- **У Java** внутрішні класи автоматично мають доступ до всіх членів зовнішнього класу, незалежно від модифікаторів доступу.

3. Порівняння з C++

3.1. Визначення класів всередині класів

- **Java:**
  - Підтримує статичні та нестатичні внутрішні класи.
  - Внутрішні класи можуть мати доступ до приватних членів зовнішнього класу.
  - Визначення класів всередині методів (локальні та анонімні класи).
- **C++:**
  - Підтримує внутрішні класи (nested classes), але вони не мають автоматичного доступу до приватних членів зовнішнього класу.
  - Немає концепції локальних чи анонімних класів всередині методів.
  - Для доступу до приватних членів потрібно використовувати friend.

3.2. Доступ до членів зовнішнього класу

- **Java:** Внутрішні класи автоматично мають доступ до приватних та інших членів зовнішнього класу.
- **C++:** Внутрішні класи не мають доступу до приватних членів зовнішнього класу, якщо не оголошені як друзі.

3.3. Створення екземплярів внутрішніх класів

- **Java:** Для нестатичних внутрішніх класів потрібен екземпляр зовнішнього класу.
- **C++:** Внутрішні класи створюються незалежно від екземплярів зовнішнього класу.

3.4. Локальні та анонімні класи

- **Java:** Підтримує локальні класи (внутрішні класи всередині методів) та анонімні класи.
- **C++:** Не підтримує локальні чи анонімні класи всередині методів. У C++ функціональні можливості, подібні до анонімних класів, досягаються через лямбда-вирази.

5. Порівняння з C++

5.1. Визначення внутрішніх класів

- **Java:**
  - Підтримує різні типи внутрішніх класів: статичні, нестатичні, локальні та анонімні.
  - Внутрішні класи мають автоматичний доступ до приватних членів зовнішнього класу.
- **C++:**
  - Підтримує лише внутрішні класи (nested classes).
  - Внутрішні класи не мають доступу до приватних членів зовнішнього класу, якщо не оголошені як друзі.
  - Не підтримує локальні чи анонімні класи.

5.2. Створення екземплярів внутрішніх класів

- **Java:**
  - Створення нестатичних внутрішніх класів вимагає екземпляру зовнішнього класу.
  - Статичні внутрішні класи створюються незалежно від екземпляру зовнішнього класу.
- **C++:**
  - Внутрішні класи створюються незалежно від екземпляру зовнішнього класу.
  - Не існує поняття статичних внутрішніх класів; всі внутрішні класи в C++ є подібними до нестатичних.

5.3. Доступ до членів зовнішнього класу

- **Java:**
  - Внутрішні класи мають повний доступ до всіх членів зовнішнього класу.
- **C++:**
  - Внутрішні класи не мають доступу до приватних членів зовнішнього класу, якщо не оголошені як друзі.

5.4. Локальні та анонімні класи

- **Java:**
  - Підтримує локальні та анонімні класи всередині методів.
- **C++:**
  - Не підтримує локальні чи анонімні класи всередині методів. Подібні функціональні можливості реалізуються через лямбда-вирази.

• Як працюють generics в Java? Як вони підтримуються під час компіляції, під час виконання? Які параметри можуть приймати generic класи та методи? Порівняння з C++.

1. Вступ до Generics в Java

Generics (узагальнення) були введені в Java 5 для забезпечення більшої типобезпеки та гнучкості при роботі з різними типами даних без необхідності виконувати явне приведення типів (casting). Вони дозволяють визначати класи, інтерфейси та методи з параметрами типів, що забезпечує повторне використання коду та зменшує кількість помилок під час виконання.

Приклади використання Generics:

- Колекції (List<T>, Map<K, V>)
- Класи-обгортки (Optional<T>)

- Власні generic класи та методи

---

2. Як працюють Generics в Java

Generics дозволяють визначати параметризовані типи, де T (типове параметрування) може бути будь-яким об'єктним типом. Це забезпечує:

- **Типобезпеку:** Помилки типу виявляються під час компіляції, а не під час виконання.
- **Повторне використання коду:** Один і той же код може працювати з різними типами даних.
- **Уникнення приведення типів:** Зменшується кількість необхідних приведень типів, що робить код чистішим і менш схильним до помилок.

3. Підтримка Generics під час компіляції

3.1. Type Erasure (Видалення типів)

Java реалізує generics за допомогою механізму **type erasure**. Це означає, що інформація про параметри типів видаляється під час компіляції, і generics реалізуються через використання об'єктів базового типу Object або специфічних обмежень (bounded types).

3.2. Переваги Type Erasure

- **Зворотна сумісність:** Generics були введені у Java 5, а type erasure дозволяє використовувати generic-класи та методи з кодом, написаним до введення generics.
- **Менший розмір байт-коду:** Оскільки інформація про типи видаляється, байт-код залишається компактним.

3.3. Недоліки Type Erasure

- **Втрата інформації про типи:** Після компіляції неможливо отримати інформацію про конкретні параметри типів через reflection.
- **Обмеження:** Неможливо створювати масиви generic-типів (new T[]), використовувати instanceof з generic-типами (instanceof List<String>), або мати статичні змінні generic-типів.

---

4. Підтримка Generics під час виконання

Оскільки generics реалізуються через type erasure, під час виконання JVM не має інформації про параметри типів. Це означає, що всі generic-типи в процесі виконання представлені їх верхніми обмеженнями (наприклад, Object або конкретними класами, якщо встановлені bounded types).

Приклад:

```
List<String> stringList = new ArrayList<>();
System.out.println(stringList.getClass() == new ArrayList<String>().getClass()); // true
```

Обидва списки мають однаковий клас (ArrayList) під час виконання, незалежно від параметра типу.

---

5. Параметри, які можуть приймати Generic класи та методи

5.1. Типові параметри (Type Parameters)

- **Однобуквені імена:** Зазвичай використовуються однобуквені імена для типових параметрів:
  - E – Element (елемент)
  - T – Type (тип)
  - K – Key (ключ)
  - V – Value (значення)
  - N – Number (число)

Приклад:

```
public class Box<T> {
   // ...
}
```

5.2. Bounded Type Parameters (Обмежені типи)

Generics можуть мати обмеження, які визначають, які типи можна використовувати як параметри.

- **Upper Bounded:** Вказують, що тип повинен бути спадкоємцем певного класу або реалізовувати інтерфейс.

Синтаксис:

```
public <T extends Number> void process(T number) {
  // ...
}
```

- **Lower Bounded:** Використовуються для вказівки нижньої межі типу, але обмежені дельті.

Синтаксис:

```
public <T super Integer> void addNumbers(List<T> list) {
  // ...
}
```

Примітка: У Java прямої підтримки lower bounded типів у параметрах типів немає, але можна використовувати wildcards (? super T).

5.3. Wildcards (Джокери)

Wildcards дозволяють визначати невизначені типи, які можуть бути будь-якими типами або типами з певними обмеженнями.

- **Upper Bounded Wildcards:**
- List<? extends Number> numbers;

Цей список може містити об'єкти будь-якого типу, що наслідують Number (наприклад, Integer, Double).

- **Lower Bounded Wildcards:**
- List<? super Integer> integers;

Цей список може містити об'єкти типу Integer або будь-якого батьківського класу (Number, Object).

- **Unbounded Wildcards:**
- List<?> unknownList;

Цей список може містити об'єкти будь-якого типу.

5.4. Generic Methods (Generic методи)

Методи можуть бути параметризовані незалежно від класів, в яких вони знаходяться.

Синтаксис:

```
public <T> void printArray(T[] array) {
  for (T element : array) {
    System.out.println(element);
  }
}
```

6. Порівняння Generics в Java з Templates в C++

Generics в Java та Templates в C++ мають схожі цілі — забезпечити параметризовані типи для повторного використання коду, але реалізуються вони різними способами та мають свої особливості.

6.1. Реалізація

- **Java Generics:**
  - Реалізуються через **type erasure**.
  - Генерується один байт-код для всіх параметрів типів.
  - Відсутня інформація про типи під час виконання.
- **C++ Templates:**
  - Реалізуються через **template instantiation**.
  - Для кожного унікального параметра типу генерується окремий код.

    - Інформація про типи зберігається під час виконання.

6.2. Типобезпека

- **Java:**
    - Типобезпека забезпечується під час компіляції через type erasure.
    - Некоректні приведення типів виявляються під час компіляції.
- **C++:**
    - Типобезпека також забезпечується під час компіляції через шаблони.
    - Помилки типу можуть призводити до складних повідомлень про помилки, оскільки генерується код для кожного типу.

6.3. Обмеження

- **Java:**
    - Немає підтримки арифметичних операцій з параметрами типів.
    - Неможливо створювати масиви generic-типів.
    - Обмежені можливості з частковими спеціалізаціями.
- **C++:**
    - Підтримує повний спектр операцій з шаблонами, включаючи часткові та повні спеціалізації.
    - Можливість створювати масиви та інші структури даних з шаблонними типами.
    - Підтримує метапрограмування з використанням шаблонів.

6.4. Використання

- **Java:**
    - Generics більше орієнтовані на колекції та загальні утиліти.
    - Простіше синтаксису, але менш гнучкі.
- **C++:**
    - Templates використовуються для широкого спектру завдань, включаючи створення контейнерів, алгоритмів, полімерних функцій та метапрограмування.
    - Більш складні та гнучкі, але потребують глибшого розуміння.

- **Чи можна в Java під час виконання програми отримати доступ до інформації про класи, методи тощо? використовувати цю інформацію для маніпулювання об'єктами? Порівняння з C++.**

Так, у Java під час виконання програми можна отримати доступ до інформації про класи, методи та інші елементи за допомогою механізму, відомого як **Reflection** (рефлексія). Reflection дозволяє програмі досліджувати та змінювати свою структуру та поведінку динамічно під час виконання. Це включає можливість отримувати метадані про класи, створювати екземпляри об'єктів, викликати методи та доступатися до полів класів навіть якщо вони мають обмежений рівень доступу.

1. Reflection в Java

Основні можливості Reflection:

- **Отримання інформації про класи:** Дізнаватись про ім'я класу, батьківські класи, інтерфейси, які реалізує клас, модифікатори доступу тощо.
- **Доступ до конструкторів:** Отримувати інформацію про конструктори класу, створювати нові екземпляри об'єктів за допомогою конструкторів.
- **Взаємодія з полями:** Отримувати інформацію про поля класу, читати та змінювати значення полів навіть якщо вони приватні.
- **Виклик методів:** Викликати методи класу динамічно, включаючи приватні методи.
- **Маніпулювання масивами:** Створювати та змінювати масиви динамічно.

Основні класи та інтерфейси для роботи з Reflection:

- java.lang.Class: Основний клас для отримання метаданих про інші класи.
- java.lang.reflect.Method: Представляє метод класу.
- java.lang.reflect.Field: Представляє поле класу.

- java.lang.reflect.Constructor: Представляє конструктор класу.
- java.lang.reflect.Modifier: Допоміжний клас для роботи з модифікаторами доступу.

2. Використання Reflection для маніпулювання об'єктами

Reflection дозволяє виконувати наступні дії:

- **Динамічне створення об'єктів:** Використовуючи класи Constructor та метод newInstance(), можна створювати нові екземпляри об'єктів без знання їхнього типу на етапі компіляції.
- **Доступ до полів:** Можна читати та змінювати значення полів об'єкта, включаючи приватні поля, використовуючи методи get() та set() класу Field.
- **Виклик методів:** Метод invoke() класу Method дозволяє викликати методи об'єкта динамічно, передаючи необхідні параметри.
- **Отримання інформації про анотації:** Reflection дозволяє досліджувати анотації, що застосовані до класів, методів, полів тощо.

3. Переваги та Недоліки Reflection

Переваги:

- **Гнучкість:** Дозволяє створювати більш гнучкі та загальні бібліотеки, фреймворки та інструменти, які можуть працювати з різними типами даних динамічно.
- **Динамічна конфігурація:** Можливість налаштовувати поведінку програми на основі метаданих, наприклад, для серіалізації/десеріалізації об'єктів.

Недоліки:

- **Продуктивність:** Використання Reflection може бути повільнішим порівняно з прямим доступом до класів та методів через те, що виконується додаткова обробка під час виконання.
- **Безпека:** Reflection може порушити інкапсуляцію, дозволяючи доступ до приватних членів класів, що може стати джерелом потенційних вразливостей.
- **Складність коду:** Код, що використовує Reflection, зазвичай менш зрозумілий та важче підтримується.

4. Reflection в Java vs. C++

Java:

- **Вбудована підтримка:** Java має вбудовану підтримку Reflection через API java.lang.reflect, що дозволяє досліджувати та маніпулювати класами під час виконання.
- **Безпека:** Java забезпечує певний рівень безпеки при використанні Reflection, наприклад, через політики безпеки (SecurityManager), які можуть обмежувати доступ до певних операцій Reflection.
- **Type Erasure:** Через механізм type erasure інформація про generic-типи не зберігається під час виконання, що обмежує деякі можливості Reflection з generics.

C++:

- **Відсутність вбудованого Reflection:** C++ не має вбудованої підтримки Reflection у стандарті. Хоча деякі компілятори та бібліотеки надають обмежені можливості Reflection, вони не є частиною стандарту мови.
- **RTTI (Run-Time Type Information):** C++ підтримує обмежену інформацію про типи через механізм RTTI, який дозволяє використовувати динамічне приведення типів (dynamic_cast), але це значно менш потужно порівняно з Java Reflection.
- **Шаблони (Templates):** Хоча не є прямим аналогом Reflection, шаблони в C++ дозволяють створювати генерований під час компіляції код, який може працювати з різними типами даних, але без динамічного доступу до метаданих.
- **Можливості Метапрограмування:** C++ надає потужні можливості метапрограмування через шаблони, що дозволяє виконувати деякі задачі, подібні до того, що робить Reflection, але це відбувається на етапі компіляції, а не виконання.

Порівняння:

- **Динамічність:** Java Reflection дозволяє динамічно досліджувати та маніпулювати класами під час виконання, тоді як C++ має обмежені можливості для подібних дій через відсутність стандартного Reflection.
- **Безпека та Інкапсуляція:** Java надає механізми для контролю доступу при використанні Reflection, тоді як в C++ можливості доступу до внутрішніх членів класів через Reflection не підтримуються стандартними засобами.
- **Продуктивність:** У Java використання Reflection може впливати на продуктивність, тоді як в C++ відсутність Reflection означає відсутність таких накладних витрат, але й обмеженість у динамічних операціях.

5. Використання Reflection у Java

Reflection широко використовується в таких сферах як:

- **Фреймворки та бібліотеки:** Наприклад, фреймворки для залежностей (Spring), ORM (Hibernate) використовують Reflection для автоматичного створення об'єктів, впровадження залежностей та мапінгу полів.
- **Серіалізація/Десеріалізація:** Бібліотеки для серіалізації використовують Reflection для автоматичного перетворення об'єктів у різні формати (JSON, XML тощо) та назад.
- **Тестування:** Інструменти для тестування можуть використовувати Reflection для доступу до приватних методів та полів класів, що дозволяє проводити більш глибоке тестування.
- **Реалізація плагінів:** Reflection дозволяє динамічно завантажувати та використовувати класи, що дозволяє створювати плагіни або модулі, які можуть бути додані без зміни основного коду програми.

6. Обмеження Reflection у Java

- **Type Erasure:** Через type erasure неможливо отримати інформацію про параметри generic-типів під час виконання, що обмежує деякі можливості.
- **Безпека:** Використання Reflection може порушити інкапсуляцію, тому в деяких середовищах (наприклад, в аплетах) доступ до Reflection може бути обмежений політиками безпеки.
- **Складність та Помилки:** Код, що використовує Reflection, менш зрозумілий та більш схильний до помилок, особливо пов'язаних з неправильними іменами класів, методів або полів.

7. Альтернативи та Доповнення до Reflection

- **Annotation Processing:** Використання анотацій разом з Reflection дозволяє створювати більш гнучкі та конфігуровані програми.
- **Dynamic Proxies:** Java надає механізм динамічних проксі-класів, який дозволяє створювати об'єкти, що реалізують певні інтерфейси, динамічно перехоплюючи виклики методів.
- **Third-Party Libraries:** Бібліотеки як Jackson для JSON обробки або Guava можуть використовувати Reflection для автоматизації різних задач, таких як серіалізація, валідація тощо.

Інше

- **Процес компіляції та запуску Java-коду. Основні етапи компіляції, інструменти що використовуються. Порівняння з C++.**

Процес компіляції та запуску Java-коду

Процес компіляції і запуску Java-коду складається з кількох основних етапів, які відрізняються від аналогічного процесу у C++.

Основні етапи компіляції та запуску Java-коду

*1. **Написання вихідного коду***

Код пишеться в текстовому файлі з розширенням .java. Наприклад, файл Main.java.

*2. **Компіляція***

Java використовує компілятор javac, який перетворює вихідний код .java у байт-код — проміжний машинозалежний формат. Результат компіляції зберігається у файлі .class.

Команда для компіляції:

```
javac Main.java
```

Після цього з’явиться файл Main.class, який містить байт-код.

*3. **Виконання байт-коду***

Для виконання байт-коду використовується віртуальна машина Java (JVM). JVM інтерпретує байт-код і перетворює його на машинний код, що виконується процесором.

Команда для запуску:

```
java Main
```

Основні інструменти Java для компіляції та виконання

1. javac — компілятор Java, що входить до складу JDK (Java Development Kit). Перетворює .java у .class.
2. java — інтерпретатор байт-коду, що входить до складу JVM (Java Virtual Machine).
3. JDK (Java Development Kit) — набір інструментів для розробки, який містить компілятор, JVM, стандартні бібліотеки та інші утиліти.
4. JRE (Java Runtime Environment) — середовище виконання Java, що включає JVM та бібліотеки для запуску вже зкомпільованого коду.

Процес компіляції в C++

У C++ процес компіляції і виконання має інший підхід:

1. Написання вихідного коду у файлах .cpp.
2. Компіляція:
    - Використовується компілятор, наприклад, g++.
    - Код компілюється безпосередньо у машинний код, який зберігається у виконуваному файлі (наприклад, .exe для Windows).
    - Під час компіляції C++ також виконується лінкування, яке об'єднує код із бібліотеками.

Команда для компіляції:

```
g++ -o program Main.cpp
```

3. **Запуск**:
    - Виконуваний файл запускається без додаткового інтерпретатора чи віртуальної машини.

Команда для запуску:

```
./program
```

Порівняння Java і C++ у контексті компіляції

Критерій	Java	C++
Формат результату	Байт-код у .class (проміжний код).	Машинний код у виконуваному файлі .exe, .out тощо.
Середовище виконання	Потребує JVM для запуску.	Виконуваний файл запускається без додаткових середовищ.
Портативність	Висока: байт-код працює на будь-якій платформі з JVM.	Залежить від платформи, для якої було зкомпільовано код.
Швидкість виконання	Трохи повільніше через використання JVM.	Вища: виконується безпосередньо в машинному коді.
Інструменти	javac, java.	g++, clang++.
Додатковий етап	Інтерпретація байт-коду JVM.	Лінкування під час компіляції.
Типи помилок	Помилки виконання можливі через динамічне завантаження класів.	Помилки лінкування можливі під час компіляції.

Переваги підходу Java

1. Портативність: однаковий .class файл може виконуватись на різних ОС за умови наявності JVM.
2. Безпека: Java-код виконується у контрольованому середовищі (JVM), що захищає систему від шкідливого коду.
3. Спрощення компіляції: немає явного етапу лінкування.

Переваги підходу C++

1. Швидкість виконання: через компіляцію у машинний код програми виконуються напряму.
2. Контроль над ресурсами: програміст має більше контролю над пам’яттю та системними ресурсами.
3. Немає залежності від віртуальної машини: скомпільований файл готовий до запуску.

Висновок

Java забезпечує високу портативність і безпеку за рахунок проміжного байт-коду та JVM, що робить її ідеальною для кросплатформного програмування. Натомість C++ орієнтований на максимальну продуктивність і контроль над системою, але вимагає більше зусиль для забезпечення сумісності між платформами.

- **Що таке JAR, JDK, JIT, JRE, JVM? В чому між ними відмінність? Чи існують аналогічні технології в C++?**

Основні поняття в екосистемі Java

---

1. JAR (Java Archive)

Що це?

- JAR — це архівний файл, який містить компільований байт-код Java (файли .class), ресурси (наприклад, зображення, конфігураційні файли) і метадані.
- Це зручний формат для поширення Java-додатків або бібліотек.

Особливості:

- Створюється за допомогою інструменту jar, який входить до складу JDK.
- Його можна запустити за допомогою команди:

```
java -jar MyApplication.jar
```

Аналог у C++:

- У C++ не існує прямого аналога, але .dll (Windows) або .so (Linux) можуть виконувати схожу роль у вигляді динамічно підключених бібліотек.

---

2. JDK (Java Development Kit)

Що це?

- JDK — це набір інструментів для розробки Java-додатків, що включає:
    - Компілятор javac.
    - Інструмент для створення JAR-файлів.
    - Додаткові утиліти (наприклад, javadoc, jdb для налагодження).
    - Java Runtime Environment (JRE).

Особливості:

- Без JDK розробка Java-додатків неможлива.
- Зазвичай використовується програмістами для створення та компіляції програм.

Аналог у C++:

- Інструменти, такі як GCC (GNU Compiler Collection), Clang, або SDK (Software Development Kit) для певної платформи.

---

3. JIT (Just-In-Time компілятор)

Що це?

- JIT — це частина JVM, яка динамічно компілює байт-код Java у машинний код під час виконання програми.
- Завдяки цьому:
    - Програма виконується швидше, оскільки не всі інструкції інтерпретуються.
    - JIT оптимізує код під конкретну платформу.

Особливості:

- Комбінація інтерпретації (для початкової швидкості) і компіляції (для продуктивності).
- Працює лише під час виконання програми.

Аналог у C++:

- У C++ такого механізму немає, оскільки код компілюється одразу в машинний перед виконанням. Проте в деяких динамічних середовищах, наприклад LLVM, JIT може бути використаний.

---

4. JRE (Java Runtime Environment)

Що це?

- JRE — це середовище виконання Java-програм, яке включає:
    - JVM (Java Virtual Machine).
    - Стандартні бібліотеки Java.
    - Класи для виконання додатків.

Особливості:

- JRE дозволяє запускати Java-додатки, але не включає компілятор.
- Призначене для користувачів кінцевих програм.

Аналог у C++:

- Виконувані файли C++ не потребують додаткового середовища, але стандартні бібліотеки (наприклад, libstdc++) можна розглядати як схожий компонент.

---

5. JVM (Java Virtual Machine)

Що це?

- JVM — це віртуальна машина, яка виконує байт-код Java.

- Забезпечує платформну незалежність, дозволяючи одному і тому ж байт-коду виконуватись на будь-якій системі з JVM.

Особливості:

- Інтерпретує байт-код або компілює його в машинний код за допомогою JIT.
- Реалізує управління пам'яттю (збирання сміття).

Аналог у C++:

- У C++ немає прямого аналога, оскільки код компілюється безпосередньо у машинний. Однак такі середовища, як LLVM, можуть виконувати схожі функції для інших мов.

Різниця між термінами

Термін	Функція	Що включає
JAR	Упаковка Java-додатків або бібліотек.	Файли .class, ресурси, метадані.
JDK	Інструменти для розробки Java-додатків.	Компілятор, JRE, додаткові утиліти.
JIT	Динамічна компіляція байт-коду у машинний код під час виконання.	Частина JVM.
JRE	Середовище виконання Java-програм.	JVM, стандартні бібліотеки.
JVM	Віртуальна машина, яка виконує байт-код.	Інтерпретатор байт-коду, JIT, управління пам’яттю.

Чи існують аналогічні технології в C++?

- JAR: У C++ застосовуються статично або динамічно підключені бібліотеки (.a, .lib, .dll, .so).
- JDK: Аналогами є GCC, Clang або будь-який інший набір інструментів для компіляції C++.
- JIT: C++ зазвичай не використовує JIT, але LLVM може виконувати JIT-компіляцію для інших мов.
- JRE: У C++ немає аналогічного середовища виконання, оскільки виконуваний файл не залежить від додаткових шарів.
- JVM: У C++ код компілюється напряму у машинний, тому немає потреби в інтерпретаторі.

Висновок

Java забезпечує платформну незалежність і зручність використання через JAR, JDK, JIT, JRE і JVM. У C++ подібні концепції реалізуються по-іншому, з орієнтацією на продуктивність і прямий доступ до ресурсів.

• Які правила та рекомендації щодо розподілу коду на файли в Java? Чи може один файл містити декілька класів? Чи може один клас міститись в декількох файлах? Порівняння з C++.

Розподіл коду на файли в Java

У Java існують чіткі правила та рекомендації щодо організації коду по файлах. Це забезпечує структурування проєкту, легкість у підтримці та відповідність загальноприйнятим стандартам.

---

1. Основні правила

1.1. Один клас — один файл

- У Java рекомендується (але не обов'язково) розміщувати кожен клас у окремому файлі. Це полегшує читання і підтримку коду.
- Назва файлу повинна збігатися з назвою публічного класу в ньому, і файл повинен мати розширення .java.

Приклад:

```
// Файл: MyClass.java
public class MyClass {
    // Код класу
}
```

1.2. Публічний клас у файлі

- У одному файлі може бути кілька класів, але лише один клас може бути public.
- Якщо файл містить кілька класів, публічний клас повинен збігатися за назвою з іменем файлу.

Приклад:

```
// Файл: Main.java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}

class Helper {
    // Допоміжний клас
}
```

1.3. Пакети

- Класи організовуються в пакети (аналог директорій), щоб уникнути конфліктів імен і забезпечити логічну структуру проєкту.
- Якщо клас належить до пакету, це потрібно вказати у верхній частині файлу.

Приклад:

```java
package com.example.myapp;

public class MyClass {
    // Код класу
}
```

1.4. Імпорти

- Для використання класів із інших пакетів потрібно їх імпортувати:

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

---

2. Чи може один файл містити декілька класів?

- Може, але:
  - Лише один клас може бути public.
  - Інші класи мають бути без модифікатора доступу (default).
  - Це не рекомендується для великих проєктів, оскільки ускладнює підтримку.

---

3. Чи може один клас міститися в декількох файлах?

- Ні, один клас у Java не може бути розділений між кількома файлами.
- Це обумовлено тим, що компілятор Java очікує, що весь код класу буде в одному файлі, який відповідає імені класу.

Рекомендації щодо розподілу коду

1. Один клас — один файл: це стандартна практика, яка полегшує навігацію в проєкті.
2. Структуруйте пакети: групуйте класи за функціональністю (наприклад, models, services, utils).
3. Дотримуйтесь модифікаторів доступу:
   - Використовуйте public лише для класів, які повинні бути доступні з інших пакетів.
   - Класи для внутрішнього використання залишайте default (без модифікатора).

Розподіл коду у C++

У C++ немає таких жорстких правил, але є загальноприйняті практики.

1. Основні принципи

- Розділення на .h і .cpp:
  - Заголовковий файл (.h) містить оголошення класів, функцій і змінних.

- Файл реалізації (.cpp) містить визначення функцій і методів.

Приклад:

```cpp
// Файл: MyClass.h
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    void printMessage();
};

#endif
```

```cpp
// Файл: MyClass.cpp
#include "MyClass.h"
#include <iostream>

void MyClass::printMessage() {
    std::cout << "Hello, World!" << std::endl;
}
```

- Інкапсуляція реалізації: реалізація приховується в .cpp, тоді як інтерфейс доступний через .h.

---

2. Чи може один файл містити кілька класів?

- Може, без обмежень. Наприклад, кілька класів можуть бути оголошені в одному .h або реалізовані в одному .cpp.

3. Чи може один клас міститися в декількох файлах?

- Може, але лише за допомогою спеціальних технік, наприклад:
  - Використання часткової спеціалізації шаблонів.
  - Розділення оголошень і реалізацій за допомогою заголовкових файлів.

Порівняння Java та C++ щодо організації коду

Аспект	Java	C++
Один клас у кількох файлах	Неможливо	Можливо (наприклад, через .h і .cpp).
Кілька класів у файлі	Можливо, але лише один клас може бути public.	Можливо без обмежень.
Назва файлу	Повинна збігатися з назвою публічного класу.	Необов’язково.

Аспект	Java	C++
Пакети/модулі	Використовуються для групування класів.	Використовуються простори імен (namespace).
Компоновка	Всі класи компілюються в байт-код для JVM.	Компоновка виконується лінкером в машинний код.

Висновок

Java має чіткі правила щодо організації коду, що сприяють читабельності та стандартизації. У C++ підхід більш гнучкий, але вимагає від програміста більше відповідальності у підтримці структури.

- **Що таке серіалізація? Як реалізувати серіалізацію в Java? Навести приклади з власного коду.**

Серіалізація в Java

Серіалізація — це процес перетворення об'єкта в потік байтів, який можна зберігати у файлі, передавати мережею чи зберігати в базі даних. Десеріалізація — це зворотний процес, коли потік байтів відновлюється до об'єкта.

Навіщо потрібна серіалізація?

1. **Зберігання стану об'єкта:**
   - Збереження об'єктів у файли для майбутнього використання.
2. **Передача об'єктів:**
   - Відправка об'єктів через мережу (наприклад, у RMI чи через сокети).
3. **Клонування об'єктів:**
   - Створення глибокої копії об'єкта.

Реалізація серіалізації в Java

1. Вимоги до серіалізації

- Клас, об'єкти якого потрібно серіалізувати, має реалізувати інтерфейс java.io.Serializable.
- У класі можна використовувати модифікатор transient для полів, які не потрібно серіалізувати.

---

2. Основні методи

- **Серіалізація:** Використовується клас ObjectOutputStream для запису об'єкта в потік.
- **Десеріалізація:** Використовується клас ObjectInputStream для читання об'єкта з потоку.

Приклад: Серіалізація та десеріалізація

```java
import java.io.*;

// Клас, що реалізує Serializable
class User implements Serializable {
    private static final long serialVersionUID = 1L; // Рекомендується для підтримки сумісності версій
    private String name;
    private int age;

    // Поле, яке не буде серіалізоване
    private transient String password;

    public User(String name, int age, String password) {
        this.name = name;
        this.age = age;
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{name='" + name + "', age=" + age + ", password='" + password + "'}";
    }
}
```

```java
public class SerializationExample {
    public static void main(String[] args) {
        // Створення об'єкта
        User user = new User("Alice", 30, "secret123");

        // Серіалізація
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("user.ser"))) {
            oos.writeObject(user);
            System.out.println("Об'єкт серіалізовано.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Десеріалізація
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("user.ser"))) {
            User deserializedUser = (User) ois.readObject();
            System.out.println("Об'єкт десеріалізовано: " + deserializedUser);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Результат виконання:

```
Об'єкт серіалізовано.
Об'єкт десеріалізовано: User{name='Alice', age=30, password='null'}
```

Ключові моменти:

1. **transient поля:**
   - Поле password у прикладі не серіалізується, тому після десеріалізації воно буде null.
2. **serialVersionUID:**
   - Використовується для ідентифікації версії класу.
   - Якщо його не вказати, Java згенерує його автоматично. Але це може спричинити проблеми при оновленні класу.

Особливості серіалізації в Java

1. **Глибока серіалізація:**

    - Якщо клас має поле з типом іншого об'єкта, цей об'єкт теж повинен реалізувати Serializable.
2. **Не серіалізуються:**
    - Статичні поля.
    - Поля з модифікатором transient.
3. **Безпека:**
    - Серіалізовані дані можуть бути вразливими до атак, тому їх слід захищати, наприклад, шифруванням.

Порівняння з C++

У C++ немає вбудованого механізму серіалізації, але можна реалізувати його вручну. Зазвичай це робиться шляхом запису даних об'єкта у файл або використання бібліотек, таких як **Boost.Serialization**.

Аспект	Java	C++
Вбудована підтримка	Є через Serializable.	Немає, потрібна реалізація вручну.
Захист даних	Можна використовувати transient.	Реалізується вручну.
Глибока серіалізація	Автоматично через Serializable.	Потребує додаткових зусиль.
Механізми безпеки	Можливе шифрування та перевірка версій.	Реалізується програмістом.

Висновок

Серіалізація в Java — потужний і простий у використанні механізм для збереження стану об'єктів. Вона автоматизує багато аспектів, які в інших мовах потребують ручної роботи. Але для безпечного використання серіалізації потрібно враховувати всі її особливості, включаючи можливі вразливості та вплив на продуктивність.

- **Чи підтримує Java pointers? references? null pointers/references? void pointers/references? smart pointers/references? weak pointers/references? Порівняння з C++.**

Pointers та References у Java: Чи підтримуються?

Java суттєво відрізняється від C++ у роботі з вказівниками (pointers) та посиланнями (references). Вона створена для безпечної роботи з пам’яттю, що значно впливає на її модель роботи з цими конструкціями.

---

1. Чи підтримує Java pointers?

- **Вказівники** (pointers) у Java **не підтримуються**.
    - Java не дозволяє прямого доступу до пам’яті через адреси, як це робиться у C++.

- Мотивація: забезпечення **безпеки пам’яті** та виключення таких проблем, як **dangling pointers**, **buffer overflows**, або маніпуляції з необмеженим доступом до пам’яті.

---

2. Чи підтримує Java references?

- У Java використовуються **references** для об’єктів:
  - Reference — це спосіб звертання до об’єкта в пам’яті.
  - Всі **об’єкти** в Java використовуються за посиланнями, а **примітивні типи** працюють із копіями значень.

Приклад:

```
class Demo {
    int value;
}

public class ReferenceExample {
    public static void main(String[] args) {
        Demo obj1 = new Demo();
        obj1.value = 10;

        // Посилання на той самий об'єкт
        Demo obj2 = obj1;
        obj2.value = 20;

        System.out.println(obj1.value); // Виведе 20, бо obj1 і obj2 — це однаковий об'єкт.
    }
}
```

---

3. Null references у Java

- Java підтримує **null references**.
  - Reference може бути null, що означає відсутність об’єкта.
  - Операції з null-посиланням викликають **NullPointerException (NPE)**.

Приклад NPE:

```
String str = null;
System.out.println(str.length()); // Викличе NullPointerException
```

---

4. Void pointers/references

- У Java немає поняття **void pointers** (як у C++).
  - Кожне посилання в Java є типізованим і не може вказувати на "нічого".

- Java забезпечує **строгу типізацію**, що запобігає помилкам роботи з пам’яттю.

---

5. Smart pointers/references

Java не має прямого аналога **розумних вказівників (smart pointers)** з C++. Однак у Java є **Garbage Collector**, який автоматично управляє пам'яттю, звільняючи об'єкти, на які більше немає посилань.

Weak references у Java

- Java підтримує слабкі посилання через клас java.lang.ref.WeakReference.
  - Слабкі посилання дозволяють Garbage Collector видалити об'єкт, якщо на нього немає сильних посилань.

Приклад слабкого посилання:

```
import java.lang.ref.WeakReference;

public class WeakReferenceExample {
    public static void main(String[] args) {
        String strongRef = new String("Hello, World!");
        WeakReference<String> weakRef = new WeakReference<>(strongRef);

        System.out.println("Before GC: " + weakRef.get()); // Виведе "Hello, World!"

        strongRef = null; // Видаляємо сильне посилання
        System.gc();      // Запускаємо збірник сміття

        System.out.println("After GC: " + weakRef.get()); // Може вивести null
    }
}
```

Порівняння Java та C++ щодо pointers і references

Особливість	Java	C++
Pointers (вказівники)	Не підтримуються.	Підтримуються (включаючи void*, nullptr).
References (посилання)	Підтримуються для об’єктів (типізовані).	Підтримуються (як звичайні, так і const).
Null pointers/references	Підтримуються (null references, викликають NullPointerException).	nullptr — базовий механізм.
Void pointers	Не підтримуються.	Підтримуються через void*.
Smart pointers	Немає, є Garbage Collector.	Підтримуються через бібліотеки (std::unique_ptr, std::shared_ptr).
Weak pointers	WeakReference, для слабких посилань на об’єкти.	std::weak_ptr для weak pointers.

Особливість	Java	C++
Garbage Collection	Автоматично, через JVM.	Вручну або через std::shared_ptr.

Висновок

Java спрощує роботу з пам’яттю завдяки використанню **references** замість вказівників і автоматичному управлінню пам’яттю через **Garbage Collector**. Це значно знижує ризик помилок, пов’язаних із вказівниками, але зменшує контроль над пам’яттю порівняно з C++.

- **Як працює обробка помилок в Java? Порівняння з C++.**

Обробка помилок у Java

В Java використовується **механізм виключень (exceptions)** для обробки помилок, який базується на трьох ключових елементах: try, catch, finally. Цей підхід дозволяє відокремити основну логіку програми від коду, що обробляє помилки, забезпечуючи зрозумілість і зручність у використанні.

Основні елементи обробки помилок у Java

1. **try блок**
   - Код, який може викликати помилку, обгортається в блок try.
2. **catch блок**
   - Використовується для перехоплення та обробки виключень.
3. **finally блок**
   - Містить код, який виконується завжди, незалежно від того, виникло виключення чи ні (наприклад, закриття ресурсів).
4. **Створення власних виключень**
   - Можна створювати користувацькі класи виключень, які успадковують Exception або RuntimeException.
5. **Оператор throw**
   - Використовується для явного створення виключення.
6. **Оператор throws**
   - Додається до сигнатури методу, щоб вказати, які виключення він може викидати.

Класифікація виключень у Java

1. **Checked Exceptions** (перевіряються на етапі компіляції):
   - Повинні оброблятися через try-catch або декларуватися через throws.
   - Наприклад: IOException, SQLException.
2. **Unchecked Exceptions** (не перевіряються на етапі компіляції):
   - Наслідують клас RuntimeException.
   - Наприклад: NullPointerException, ArithmeticException.
3. **Errors**:
   - Наслідують клас Error. Ці виключення зазвичай сигналізують про критичні помилки (наприклад, OutOfMemoryError).

```java
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
            System.out.println("Результат: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Помилка: Ділення на нуль.");
        } finally {
            System.out.println("Завершення обробки.");
        }
    }

    public static int divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException("Ділення на нуль не дозволено.");
        }
        return a / b;
    }
}
```

Результат:

```
Помилка: Ділення на нуль.
Завершення обробки.
```

Порівняння обробки помилок у Java та C++

Особливість	Java	C++
Типи помилок	Використовує виключення (Exception, Error).	Використовує виключення (throw) і системні коди помилок.
Checked/Unchecked	Розрізняє Checked та Unchecked.	Всі виключення однакові, перевірки на етапі компіляції немає.
finally блок	Є finally, який виконується завжди.	В C++ немає прямого аналога finally, але є RAII для управління ресурсами.
Декларація виключень	Використовує throws у сигнатурі методу для Checked виключень.	Необов'язково декларувати, що метод може викидати виключення.
Коди помилок	Коди помилок не використовуються (виключно виключення).	Можна використовувати коди помилок або виключення.

Особливість	Java	C++
RAII (Resource Management)	Управління ресурсами через try-with-resources (з Java 7).	RAII — стандартна практика для автоматичного звільнення ресурсів.
Продуктивність	Використання виключень може впливати на продуктивність.	Виключення також впливають на продуктивність, але можна використовувати альтернативи.
Власні виключення	Легко створювати власні класи виключень.	Аналогічно, користувацькі класи можуть наслідувати від std::exception.

RAII vs try-with-resources

RAII (C++):

```cpp
#include <iostream>
#include <fstream>

void readFile() {
    std::ifstream file("example.txt");
    if (!file) {
        throw std::runtime_error("Помилка відкриття файлу");
    }
    // Файл автоматично закриється при виході з області видимості
}
```

try-with-resources (Java):

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TryWithResourcesExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Переваги обробки помилок у Java

1. Вища безпека коду завдяки Checked Exceptions.

2. Легкість в управлінні ресурсами (try-with-resources).
3. Уніфікована модель помилок через використання виключень.

Висновок

Обробка помилок у Java більш стандартизована та проста у використанні завдяки чіткій класифікації виключень і строгій типізації. У C++ механізм виключень є гнучкішим, але потребує більшої уваги програміста через відсутність строгих перевірок і ширшого спектру підходів до обробки помилок.