Колекції в Java

Java Collections Framework простими словами

List, Set, Map і Deque: як зберігати дані та обирати відповідну колекцію

---

1. Що таке колекція?

Колекція дозволяє зберігати разом кілька об’єктів і виконувати з ними спільні операції.
Наприклад, список імен:

```
List<String> names = new ArrayList<>();

names.add("Anna");
names.add("Bohdan");
names.add("Denys");
```

Тепер усі імена знаходяться в одному об’єкті `names`.
З колекціями можна:

- додавати елементи;
- отримувати елементи;
- шукати їх;
- видаляти їх;
- перебирати всі елементи.

Тип `String` усередині кутових дужок

```
List<String>
```

означає, що цей список зберігає рядки.

Головна ідея

Колекція потрібна, коли ми хочемо працювати не з одним об’єктом, а з групою об’єктів.

2. Основні види колекцій

У Java є кілька основних інтерфейсів.

List

Зберігає елементи в певному порядку. Повтори дозволені.
Приклад реалізації:

```
ArrayList
```

Set

Зберігає лише унікальні елементи.
Приклад реалізації:

```
HashSet
```

Map

Зберігає пари:

ключ ⟶ значення.

Приклад реалізації:

```
HashMap
```

Deque
Зручна для черг і стеків.
Приклад реалізації:

```
ArrayDeque
```

Важливо

`Map` стоїть дещо окремо.
Вона не успадковує `Collection`, оскільки зберігає не окремі елементи, а пари «ключ — значення».

3. List і ArrayList

List — упорядкований список елементів.
Він:

- зберігає порядок додавання;
- дозволяє звертатися до елементів за індексом;
- допускає однакові елементи.

Приклад:

```
List<String> names = new ArrayList<>();

names.add("Anna");
names.add("Bohdan");
names.add("Anna");
```

У списку буде три елементи.
Отримати елемент за індексом:

```
String first = names.get(0);
```

Перший елемент має індекс `0`.
Змінити елемент:

```
names.set(1, "Denys");
```

Видалити елемент:

```
names.remove(0);
```

Дізнатися кількість елементів:

```
int size = names.size();
```

Важливо

Для більшості звичайних списків гарним першим вибором є `ArrayList`.

4. Set і HashSet

Set використовується, коли елементи мають бути унікальними.
Приклад:

```
Set<String> languages = new HashSet<>();

languages.add("Java");
languages.add("Python");
languages.add("Java");
```

Хоча рядок "Java" додавався двічі, він зберігатиметься лише один раз.
Перевірити наявність елемента:

```
boolean exists =
    languages.contains("Java");
```

Видалити елемент:

```
languages.remove("Python");
```

Важливо пам’ятати:
HashSet не гарантує порядок елементів.
Якщо потрібно зберегти порядок додавання, можна використати:

```
LinkedHashSet
```

Якщо елементи мають зберігатися відсортованими:

```
TreeSet
```

5. Map і HashMap

Map зберігає пари:

$$\text{ключ} \longrightarrow \text{значення}.$$

Наприклад, можна пов’язати ім’я учня з кількістю його балів:

```
Map<String, Integer> scores =
    new HashMap<>();

scores.put("Anna", 10);
scores.put("Bohdan", 8);
scores.put("Denys", 9);
```

Отримати значення за ключем:

```
int annaScore = scores.get("Anna");
```

Перевірити наявність ключа:

```
boolean exists =
    scores.containsKey("Bohdan");
```

Видалити пару:

```
scores.remove("Denys");
```

Ключі в Map унікальні.
Якщо знову записати значення з тим самим ключем:

```
scores.put("Anna", 12);
```

старе значення `10` буде замінено на `12`.

Важливо

`HashMap` зручна, коли потрібно швидко знаходити значення за ключем.

6. Queue, Deque і ArrayDeque

Черга працює за принципом:

першим прийшов ⟶ першим вийшов.

Це називається **FIFO**:

First In, First Out.

У Java для звичайної черги зручно використовувати `ArrayDeque`.

```
Deque<String> tasks =
    new ArrayDeque<>();

tasks.offerLast("Task A");
tasks.offerLast("Task B");

String next = tasks.pollFirst();
```

У змінну `next` потрапить:

```
"Task A"
```

Подивитися перший елемент, не видаляючи його:

```
String first = tasks.peekFirst();
```

`Deque` також можна використовувати як стек.
Стек працює за принципом:

останнім прийшов ⟶ першим вийшов.

Це називається **LIFO**.

```
Deque<String> stack =
    new ArrayDeque<>();

stack.push("A");
stack.push("B");

String top = stack.pop();
```

Тут буде вилучено елемент `"B"`.

7. Як швидко обрати колекцію

Для більшості навчальних задач достатньо таких правил.

- Потрібен звичайний список з індексами:

  ```
  ArrayList
  ```

- Потрібні лише унікальні елементи:

```
HashSet
```

- Потрібно шукати значення за ключем:

```
HashMap
```

- Потрібно зберегти порядок додавання унікальних елементів:

```
LinkedHashSet
```

- Потрібно зберегти порядок додавання пар «ключ — значення»:

```
LinkedHashMap
```

- Елементи мають автоматично зберігатися у відсортованому порядку:

```
TreeSet
```

- Ключі карти мають бути відсортованими:

```
TreeMap
```

- Потрібна черга або стек:

```
ArrayDeque
```

8. Перебір елементів

Один із найпростіших способів пройти по всіх елементах списку — цикл `for-each`.

```
for (String name : names) {
    System.out.println(name);
}
```

Так само можна перебирати `Set`:

```
for (String language : languages) {
    System.out.println(language);
}
```

Для `Map` можна пройти по всіх парах:

```
for (Map.Entry<String, Integer> entry
        : scores.entrySet()) {

    System.out.println(
```

```
        entry.getKey()
        + ": "
        + entry.getValue()
    );
}
```

Тут:

- `entry.getKey()` повертає ключ;
- `entry.getValue()` повертає значення.

9. Сортування

Звичайний список можна відсортувати.
Наприклад:

```
List<String> names =
    new ArrayList<>();

names.add("Denys");
names.add("Anna");
names.add("Bohdan");

Collections.sort(names);
```

Після сортування елементи будуть розташовані в алфавітному порядку:

```
Anna
Bohdan
Denys
```

Для чисел використовується природний порядок від меншого до більшого.
Якщо потрібно автоматично зберігати унікальні елементи у відсортованому вигляді, можна використати `TreeSet`:

```
Set<Integer> numbers =
    new TreeSet<>();

numbers.add(5);
numbers.add(2);
numbers.add(8);
```

Порядок елементів буде таким:

```
2, 5, 8
```

10. Швидкість основних операцій

Під час вибору колекції корисно розуміти, наскільки швидко виконуються операції.
Позначення:

- $O(1)$ — операція зазвичай виконується приблизно за сталий час;
- $O(\log n)$ — час зростає повільно зі збільшенням кількості елементів;
- $O(n)$ — у гіршому випадку доводиться переглянути багато або всі елементи.

Колекція	Операція	Зазвичай
`ArrayList`	доступ за індексом	$O(1)$
`ArrayList`	пошук елемента	$O(n)$
`HashSet`	пошук / додавання	$O(1)^*$
`HashMap`	пошук / додавання за ключем	$O(1)^*$
`TreeSet`	пошук / додавання	$O(\log n)$
`TreeMap`	пошук / додавання	$O(\log n)$
`ArrayDeque`	додавання / видалення з країв	$O(1)$

* У середньому. В окремих випадках операція може виконуватися довше.

Важливо

Для перших програм не потрібно обирати колекцію лише за формулами складності.
Спочатку подумайте, **що саме** потрібно робити з даними.

11. equals() і hashCode()

Для `HashSet` і ключів `HashMap` важливо правильно визначати, коли два об'єкти вважаються однаковими.
Для цього використовуються методи:

```
equals()
hashCode()
```

Спрощено:
equals()
відповідає на запитання:

«чи вважаються об'єкти рівними?»

hashCode()
повертає спеціальне число, яке допомагає хеш-колекції швидше знайти об'єкт.
Головне правило:

Важливо

Якщо два об'єкти рівні за `equals()`, вони обов'язково повинні мати однаковий `hashCode()`.

Для стандартних типів Java, наприклад

```
String
Integer
Long
```

ці методи вже реалізовані правильно.
Під час створення власних класів ця тема вивчається докладніше.

12. Незмінні колекції

Іноді потрібно створити невелику колекцію, яку після створення не можна змінювати.
Наприклад:

```
List<String> colors =
    List.of("red", "green", "blue");
```

Такий список можна читати:

```
String first = colors.get(0);
```

Але не можна додати новий елемент:

```
colors.add("yellow"); // Помилка
```

Для карти:

```
Map<String, Integer> scores =
    Map.of(
        "Anna", 10,
        "Bohdan", 8
    );
```

Такі колекції зручні, коли набір даних має залишатися незмінним.

13. Поширені помилки

Помилка 1.
Очікувати певного порядку елементів у `HashSet` або `HashMap`.
Їхній порядок не гарантується.

Помилка 2.
Забувати, що індекси списку починаються з нуля.
Для списку з трьох елементів допустимі індекси:

$$0, \quad 1, \quad 2.$$

Помилка 3.
Очікувати, що `Set` збереже дублікати.

```
Set<String> set = new HashSet<>();

set.add("Java");
set.add("Java");
```

У наборі залишиться лише один елемент.

Помилка 4.
Забувати, що повторний запис у `Map` за тим самим ключем замінює старе значення.

```
scores.put("Anna", 10);
scores.put("Anna", 12);
```

Тепер значення для `"Anna"` дорівнює `12`.

Помилка 5.
Видаляти елементи звичайним способом із колекції безпосередньо всередині циклу `for-each`.
Для простого видалення за умовою зручніше:

```
names.removeIf(String::isBlank);
```

14. Коротка пам’ятка

ArrayList
Звичайний упорядкований список. Є індекси та дозволені повтори.
HashSet
Набір унікальних елементів. Порядок не гарантується.
HashMap
Зберігає пари «ключ — значення». Кожному ключу відповідає одне значення.
LinkedHashSet
Унікальні елементи зі збереженням порядку додавання.
LinkedHashMap
Пари «ключ — значення» зі збереженням порядку додавання.
TreeSet
Унікальні елементи у відсортованому порядку.
TreeMap
Пари «ключ — значення» з відсортованими ключами.
ArrayDeque
Зручна реалізація черги або стека.

Головний принцип вибору

Спочатку поставте собі три запитання.

1. Чи потрібен порядок елементів?
2. Чи можуть елементи повторюватися?
3. Як ми хочемо знаходити дані: за індексом, за значенням чи за ключем?

Після цього вибір зазвичай стає простим:

- список — `ArrayList`;
- унікальні елементи — `HashSet`;
- ключ і значення — `HashMap`;
- черга або стек — `ArrayDeque`.

Не потрібно обирати найскладнішу колекцію. Потрібно обирати ту, яка найкраще відповідає задачі.

Пам’ятка призначена для першого знайомства з колекціями Java. Приклади спеціально спрощено, щоб показати основні ідеї та найпоширеніші операції.