Коллекции в Java

Java Collections Framework простыми словами

List, Set, Map и Deque: как хранить данные и выбирать подходящую коллекцию

1. Что такое коллекция?

Коллекция позволяет хранить вместе несколько объектов и выполнять с ними общие операции.
Например, список имён:

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

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

Теперь все имена находятся в одном объекте `names`.
С коллекциями можно:

- добавлять элементы;
- получать элементы;
- искать их;
- удалять их;
- перебирать все элементы.

Тип `String` внутри угловых скобок

```
List<String>
```

означает, что этот список хранит строки.

Главная идея

Коллекция нужна, когда мы хотим работать не с одним объектом, а с группой объектов.

2. Основные виды коллекций

В Java есть несколько основных интерфейсов.

List

Хранит элементы по порядку. Повторы разрешены.
Пример реализации:

```
ArrayList
```

Set

Хранит только уникальные элементы.
Пример реализации:

```
HashSet
```

Map

Хранит пары:

ключ ⟶ значение.

Пример реализации:

```
HashMap
```

Deque
Удобна для очередей и стеков.
Пример реализации:

```
ArrayDeque
```

Важно

`Map` стоит немного отдельно.
Она не является наследником `Collection`, потому что хранит не отдельные элементы, а пары «ключ — значение».

3. List и ArrayList

List — упорядоченный список элементов.
Он:

- сохраняет порядок добавления;
- позволяет обращаться по индексу;
- допускает одинаковые элементы.

Пример:

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

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

В списке будет три элемента.
Получить элемент по индексу:

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

Первый элемент имеет индекс `0`.
Изменить элемент:

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

Удалить элемент:

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

Узнать количество элементов:

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

Важно

Для большинства обычных списков хорошим первым выбором является `ArrayList`.

4. Set и HashSet

Set используется, когда элементы должны быть уникальными.
Пример:

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

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

Хотя строка "Java" добавлялась дважды, она будет храниться только один раз.
Проверить наличие элемента:

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

Удалить элемент:

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

Важно помнить:
HashSet не гарантирует порядок элементов.
Если требуется сохранить порядок добавления, можно использовать:

```
LinkedHashSet
```

Если элементы должны храниться отсортированными:

```
TreeSet
```

5. Map и HashMap

Map хранит пары:

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

Например, можно связать имя ученика с его количеством баллов:

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

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

Получить значение по ключу:

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

Проверить наличие ключа:

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

Удалить пару:

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

Ключи в Map уникальны.
Если снова записать значение с тем же ключом:

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

старое значение `10` будет заменено на `12`.

Важно

`HashMap` удобна, когда нужно быстро находить значение по ключу.

6. Queue, Deque и ArrayDeque

Очередь работает по принципу:

первым пришёл ⟶ первым вышел.

Это называется **FIFO**:

First In, First Out.

В Java для обычной очереди удобно использовать `ArrayDeque`.

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

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

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

В переменную `next` попадёт:

```
"Task A"
```

Посмотреть первый элемент, не удаляя его:

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

`Deque` также можно использовать как стек.
Стек работает по принципу:

последним пришёл ⟶ первым вышел.

Это называется **LIFO**.

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

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

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

Здесь будет извлечён элемент `"B"`.

7. Как быстро выбрать коллекцию

Для большинства учебных задач достаточно следующих правил.

- Нужен обычный список с индексами:

  ```
  ArrayList
  ```

- Нужны только уникальные элементы:

```
HashSet
```

- Нужно искать значение по ключу:

```
HashMap
```

- Нужно сохранить порядок добавления уникальных элементов:

```
LinkedHashSet
```

- Нужно сохранить порядок добавления пар «ключ — значение»:

```
LinkedHashMap
```

- Элементы должны автоматически храниться в отсортированном порядке:

```
TreeSet
```

- Ключи карты должны быть отсортированы:

```
TreeMap
```

- Нужна очередь или стек:

```
ArrayDeque
```

8. Перебор элементов

Один из самых простых способов пройти по всем элементам списка — цикл `for-each`.

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

Так можно перебирать и `Set`:

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

Для `Map` можно пройти по всем парам:

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

    System.out.println(
```

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

Здесь:

- `entry.getKey()` возвращает ключ;
- `entry.getValue()` возвращает значение.

9. Сортировка

Обычный список можно отсортировать.
Например:

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

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

Collections.sort(names);
```

После сортировки элементы будут расположены в алфавитном порядке:

```
Anna
Bohdan
Denys
```

Для чисел используется естественный порядок от меньшего к большему.
Если нужно автоматически хранить уникальные элементы в отсортированном виде, можно использовать `TreeSet`:

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

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

Порядок элементов будет:

```
2, 5, 8
```

10. Скорость основных операций

При выборе коллекции важно понимать, насколько быстро выполняются операции.
Обозначения:

- $O(1)$ — операция обычно выполняется примерно за постоянное время;
- $O(\log n)$ — время растёт медленно при увеличении количества элементов;
- $O(n)$ — в худшем случае приходится просмотреть много или все элементы.

Коллекция	Операция	Обычно
`ArrayList`	доступ по индексу	$O(1)$
`ArrayList`	поиск элемента	$O(n)$
`HashSet`	поиск / добавление	$O(1)^*$
`HashMap`	поиск / добавление по ключу	$O(1)^*$
`TreeSet`	поиск / добавление	$O(\log n)$
`TreeMap`	поиск / добавление	$O(\log n)$
`ArrayDeque`	добавление / удаление с краёв	$O(1)$

* В среднем. В отдельных случаях операция может выполняться дольше.

Важно

Для первых программ не нужно выбирать коллекцию только по формулам сложности.
Сначала подумайте, **что именно** нужно делать с данными.

11. equals() и hashCode()

Для `HashSet` и ключей `HashMap` важно правильно определять, когда два объекта считаются одинаковыми.
Для этого используются методы:

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

Упрощённо:
equals()
отвечает на вопрос:

«считаются ли объекты равными?»

hashCode()
возвращает специальное число, которое помогает хеш-коллекции быстрее найти объект.
Главное правило:

Важно

Если два объекта равны по `equals()`, у них обязательно должен быть одинаковый `hashCode()`.

Для стандартных типов Java, например

```
String
Integer
Long
```

эти методы уже реализованы правильно.
При создании собственных классов эта тема изучается подробнее.

12. Неизменяемые коллекции

Иногда нужно создать небольшую коллекцию, которую после создания нельзя менять.
Например:

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

Такой список можно читать:

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

Но нельзя добавить новый элемент:

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

Для карты:

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

Такие коллекции удобны, когда набор данных должен оставаться неизменным.

13. Частые ошибки

Ошибка 1.
Ожидать определённый порядок элементов в `HashSet` или `HashMap`.
Их порядок не гарантирован.

Ошибка 2.
Забывать, что индексы списка начинаются с нуля.
Для списка из трёх элементов допустимы индексы:

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

Ошибка 3.
Ожидать, что `Set` сохранит дубликаты.

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

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

В наборе останется только один элемент.

Ошибка 4.
Забывать, что повторная запись в `Map` по тому же ключу заменяет старое значение.

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

Теперь значение для `"Anna"` равно `12`.

Ошибка 5.
Удалять элементы обычным способом из коллекции прямо внутри цикла `for-each`.
Для простого удаления по условию удобнее:

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

14. Короткая памятка

ArrayList
Обычный упорядоченный список. Есть индексы и разрешены повторы.
HashSet
Набор уникальных элементов. Порядок не гарантирован.
HashMap
Хранит пары «ключ — значение». У каждого ключа одно значение.
LinkedHashSet
Уникальные элементы с сохранением порядка добавления.
LinkedHashMap
Пары «ключ — значение» с сохранением порядка добавления.
TreeSet
Уникальные элементы в отсортированном порядке.
TreeMap
Пары «ключ — значение» с отсортированными ключами.
ArrayDeque
Удобная реализация очереди или стека.

Главный принцип выбора

Сначала задайте себе три вопроса.

1. Нужен ли порядок элементов?
2. Могут ли элементы повторяться?
3. Как мы хотим находить данные: по индексу, по значению или по ключу?

После этого выбор обычно становится простым:

- список — `ArrayList`;
- уникальные элементы — `HashSet`;
- ключ и значение — `HashMap`;
- очередь или стек — `ArrayDeque`.

Не нужно выбирать самую сложную коллекцию. Нужно выбирать ту, которая лучше всего соответствует задаче.

Памятка предназначена для первого знакомства с коллекциями Java. Примеры специально упрощены, чтобы показать основные идеи и наиболее распространённые операции.