Коллекции в Java: Подробный обзор

1. Интерфейсы коллекций

Иерархия коллекций:

- **Collection** (корневой интерфейс)
    - **List** (упорядоченные последовательности)
    - **Set** (уникальные элементы)
    - **Queue** (очереди)
- **Map** (пары ключ-значение, не наследует Collection)

1.1 List (списки)

Характеристики:

- Упорядоченная коллекция (сохраняет порядок добавления)
- Допускает дубликаты
- Доступ по индексу

Основные реализации:

1. **ArrayList**:
    - Динамический массив
    - Быстрый доступ по индексу (O(1))
    - Медленные вставка/удаление в середине (O(n))
    - Оптимален для частого чтения
2. **LinkedList**:
    - Двусвязный список
    - Быстрая вставка/удаление (O(1))
    - Медленный доступ по индексу (O(n))
    - Реализует также Deque
    - Оптимален для частых изменений

Пример:

```
List<String> arrayList = new ArrayList<>();
arrayList.add("A");
arrayList.add("B");
arrayList.add(1, "C"); // [A, C, B]
```

```
List<String> linkedList = new LinkedList<>();
linkedList.add("X");
linkedList.addFirst("Y"); // [Y, X]
```

1.2 Set (множества)

Характеристики:

- Хранит только уникальные элементы
- Не гарантирует порядок (кроме TreeSet)

Основные реализации:

1. **HashSet**:
    - Основан на HashMap
    - Хранит элементы в хэш-таблице
    - Самый быстрый доступ (O(1))
    - Порядок не гарантируется
2. **TreeSet**:
    - Основан на TreeMap (красно-черное дерево)
    - Хранит элементы отсортированными (по natural ordering или Comparator)
    - Доступ за O(log n)
    - Реализует NavigableSet
3. **LinkedHashSet**:
    - Сохраняет порядок вставки
    - Компромисс между HashSet и сохранением порядка

Пример:

```
Set<Integer> hashSet = new HashSet<>();
hashSet.add(5);
hashSet.add(3);
hashSet.add(5); // не добавится
// Порядок может быть [3, 5] или [5, 3]

Set<String> treeSet = new TreeSet<>();
treeSet.add("Banana");
treeSet.add("Apple");
// Гарантированный порядок [Apple, Banana]
```

1.3 Queue (очереди)

Характеристики:

- Обычно работают по принципу FIFO (кроме стэков)

- Могут иметь ограниченную емкость

Основные реализации:

1. **LinkedList**:
    - Реализует также List и Deque
    - Может использоваться как очередь или двусторонняя очередь
2. **PriorityQueue**:
    - Элементы обрабатываются по приоритету
    - Сортировка по natural ordering или Comparator
    - Не thread-safe
3. **ArrayDeque**:
    - Реализация двусторонней очереди на массиве
    - Быстрее LinkedList для большинства операций
    - Не поддерживает null

Пример:

```
Queue<String> queue = new LinkedList<>();
queue.offer("First");
queue.offer("Second");
queue.poll(); // вернет "First"

Queue<Integer> priorityQueue = new PriorityQueue<>();
priorityQueue.add(5);
priorityQueue.add(1);
priorityQueue.add(3);
priorityQueue.poll(); // вернет 1 (наименьший)
```

1.4 Map (ассоциативные массивы)

Характеристики:

- Пары ключ-значение
- Ключи уникальны
- Значения могут дублироваться

Основные реализации:

1. **HashMap**:
    - Хэш-таблица
    - Быстрый доступ O(1)
    - Порядок не гарантируется

- Позволяет один null-ключ

2. **TreeMap**:
    - Красно-черное дерево
    - Сортировка по ключам (natural ordering или Comparator)
    - Доступ за O(log n)
    - Реализует NavigableMap
3. **LinkedHashMap**:
    - Сохраняет порядок вставки или порядок доступа
    - Компромисс между HashMap и сохранением порядка

Пример:

```
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("Apple", 1);
hashMap.put("Banana", 2);
hashMap.put("Apple", 3); // заменит предыдущее значение

Map<Integer, String> treeMap = new TreeMap<>();
treeMap.put(3, "C");
treeMap.put(1, "A");
// Порядок ключей: 1, 3
```

2. Сравнение коллекций

2.1 ArrayList vs LinkedList

Критерий	ArrayList	LinkedList
Структура данных	Динамический массив	Двусвязный список
Доступ по индексу	O(1)	O(n)
Вставка в начало	O(n)	O(1)
Вставка в конец	O(1) (амортизированное)	O(1)
Вставка в середину	O(n)	O(n) (но быстрее ArrayList)
Удаление	O(n)	O(1) для первого/последнего
Память	Меньше (только данные)	Больше (ссылки next/prev)
Использование	Частое чтение, редкие изменения	Частые изменения, редкое чтение

2.2 HashMap vs TreeMap

Критерий	HashMap	TreeMap
Структура данных	Хэш-таблица	Красно-черное дерево

Порядок элементов	Не гарантируется	Отсортирован по ключам
Время доступа	O(1)	O(log n)
Null-ключи	Разрешен один	Запрещены (если нет Comparator)
Реализует	Map	NavigableMap, SortedMap
Использование	Быстрый доступ, порядок не важен	Нужна сортировка или диапазонные запросы

3. Итерация по коллекциям

3.1 for-each цикл

Самый простой способ итерации (работает для всех Iterable):

```
List<String> list = Arrays.asList("A", "B", "C");
for (String item : list) {
    System.out.println(item);
}
```

3.2 Итератор (Iterator)

Позволяет безопасно удалять элементы во время итерации:

```
Set<Integer> set = new HashSet<>(Set.of(1, 2, 3));
Iterator<Integer> iterator = set.iterator();
while (iterator.hasNext()) {
    Integer num = iterator.next();
    if (num % 2 == 0) {
        iterator.remove(); // безопасное удаление
    }
}
```

3.3 forEach() (Java 8+)

Функциональный стиль с лямбда-выражением:

```
Map<String, Integer> map = Map.of("A", 1, "B", 2, "C", 3);
map.forEach((key, value) ->
    System.out.println(key + ": " + value));
```

3.4 ListIterator (только для List)

Двунаправленный итератор с дополнительными методами:

```
List<String> list = new ArrayList<>(List.of("A", "B", "C"));
ListIterator<String> listIterator = list.listIterator();
while (listIterator.hasNext()) {
    String item = listIterator.next();
    if (item.equals("B")) {
        listIterator.set("B-New"); // замена элемента
        listIterator.add("B-Added"); // вставка перед следующим
```

```
    }
}
```

3.5 Итерация по Map

Несколько способов:

```
Map<String, Integer> map = new HashMap<>();
map.put("A", 1);
map.put("B", 2);

// 1. По entrySet (наиболее эффективный)
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}

// 2. По keySet
for (String key : map.keySet()) {
    System.out.println(key + ": " + map.get(key));
}

// 3. По values
for (Integer value : map.values()) {
    System.out.println(value);
}

// 4. С использованием Stream API
map.entrySet().stream()
   .forEach(entry -> System.out.println(entry));
```

Дополнительные возможности

Неизменяемые коллекции (Java 9+)

```
List<String> immutableList = List.of("A", "B", "C");
Set<Integer> immutableSet = Set.of(1, 2, 3);
Map<String, Integer> immutableMap = Map.of("A", 1, "B", 2);

// Попытка изменения вызовет UnsupportedOperationException
// immutableList.add("D"); // Ошибка!
```

Синхронизированные коллекции

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
Map<String, Integer> syncMap = Collections.synchronizedMap(new HashMap<>());

// При итерации нужно синхронизировать вручную
synchronized(syncList) {
    for (String item : syncList) {
        System.out.println(item);
    }
}
```

Утилитные методы Collections

```
List<Integer> list = new ArrayList<>(List.of(3, 1, 4, 1, 5));
```

```
Collections.sort(list); // сортировка
Collections.reverse(list); // обратный порядок
Collections.shuffle(list); // случайная перестановка
Collections.fill(list, 0); // заполнение одним значением
Collections.frequency(list, 1); // количество вхождений
Collections.max(list); // максимальный элемент
Collections.min(list); // минимальный элемент
```