



Коллекции. Часть II

Croc Java School

—

Итераторы



Итерация

Повторное применение операции.

Математическое определение:

$$y = f(x)$$

$$f_1(x) = f(x), f_2(x) = f(f_1(x)), f_3(x) = f(f_2(x)), \dots, f_n(x) = f(f_{n-1}(x))$$

$f_n(x)$ — n -ая итерация функции f .



Итерация в контексте коллекций

В контексте коллекций итерация — это повторное применение операции (или операций) ко всем элементам этой коллекции.

При этом порядок применения операции к элементам коллекции в общем случае не специфицируется: каждая коллекция может задавать свой порядок, но он не фиксируется конструкциями языка.

Например:

- Итерация элементов `List` производится в порядке следования элементов.
- Итерация `TreeSet/TreeMap` выполняется в порядке сортировки элементов/ключей.
- Порядок итерации `HashSet` не определен.



Итерация в порядке следования

Известный пример итерации в порядке следования реализуется средствами языка: перечисление элементов массива.

```
int[] array = new int[] {2, 5, 1, 7, 9};  
for (int i = 0; i < array.length; i++) {  
    System.out.println(array[i]);  
}
```



Итерация в порядке следования

Аналогично итерация выглядит в коллекциях с фиксированным порядком.

```
List<Integer> list = new ArrayList<>();  
// insert elements into the list...  
for (int i = 0; i < list.size(); i++) {  
    System.out.println(list.get(i));  
}
```



Если коллекция не определяет порядок элементов

Если элементы в коллекции хранятся не в фиксированном порядке (как это делает, например, `HashSet`), она не предоставляет метод получения по индексу:

`HashSet.get(i)` — такого метода нет.

В этом случае необходимы другие конструкции для перечисления всех элементов.

Iterator<E>



java.util.Iterator<E>

Для решения проблемы служит библиотечный интерфейс Iterator<E>.

```
public interface Iterator<E> {  
    // iteration has more elements  
    boolean hasNext();  
    // next element in the iteration  
    E next();  
    // ...  
}
```



java.util.Iterator<E>

Методы `hasNext` и `next` позволяют перечислить элементы без знания о внутренней структуре хранения элементов внутри коллекции.

```
Set<String> set = new HashSet<>();  
// insert elements into the set...  
Iterator<String> it = set.iterator();  
while (it.hasNext()) {  
    String element = it.next();  
    System.out.println(element);  
}
```



LineIterator

Итераторы можно использовать не только в коллекциях. Это более общий механизм. Например, можно реализовать итератор строк в произвольном тексте.

```
public class LineIterator implements Iterator<String> {
    private Scanner scanner;
    public LineIterator(String text) {
        scanner = new Scanner(text);
    }
    public boolean hasNext() {
        return scanner.hasNextLine();
    }
    public String next() {
        return scanner.nextLine();
    }
}
```



LineIterator

Практическая польза от LineIterator сомнительная, т.к. он дублирует функционал класса Scanner, но идею он демонстрирует.

```
Iterator<String> it = new LineIterator("1\n2\n3");
while (it.hasNext()) {
    String element = it.next();
    System.out.println(element);
}
```



java.util.Iterator<E>

Класс `Iterator` не совсем простой — он интегрирован в язык в виде конструкции `for each`.

```
for (Element e : collection) {  
    // do something with e  
}
```

Iterable<T>



java.lang.Iterable<T>

Для этого используется вспомогательный интерфейс `Iterable<T>`.

```
public interface Iterable<T> {  
    // iterator over elements of type T  
    Iterator<T> iterator();  
    // ...  
}
```



for each

Если класс реализует интерфейс `Iterable<T>`, то к нему применима конструкция языка `for each`.

Все коллекции реализуют `Iterable`.

```
public interface Collection<E> extends Iterable<E> {  
    // ...  
}
```




for each

По сути `for each` — “синтаксический сахар”.

Когда мы пишем:

```
Set<String> words = new HashSet<>();  
for (String word : words) {  
    System.out.println(word);  
}
```



for each

Компилятор видит:

```
Set<String> words = new HashSet<>();  
Iterator<String> it = words.iterator();  
while (it.hasNext()) {  
    String word = it.next();  
    System.out.println(word);  
}
```

ConcurrentModificationException
WTF?!



Изменение коллекции во время итерации

Вставка и удаление элементов в коллекции может изменить порядок итерации.
Изменение коллекции в процессе работы итератора “разрушает” его состояние.



ConcurrentModificationException

```
Set<Integer> numbers = new HashSet<>(Arrays.asList(5, 1, 2, 3, 4));  
// remove all even numbers from the set  
for (Integer number : numbers) {  
    if (number % 2 == 0) {  
        numbers.remove(number);  
    }  
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException  
    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1442)  
    at java.util.HashMap$KeyIterator.next(HashMap.java:1466)
```



Итерация и удаление

```
Set<Integer> numbers = new HashSet<>(Arrays.asList(5, 1, 2, 3, 4));
Set<Integer> remove = new HashSet<>();
// remove all even numbers from the set
for (Integer number : numbers) {
    if (number % 2 == 0) {
        remove.add(number);
    }
}
for (Integer number : remove) {
    numbers.remove(number);
}
```

Работает, но не очень удобно

—



Iterator.remove()

Интерфейс `Iterator<E>` предоставляет метод `remove()` для удаления текущего элемента без нарушения консистентности итерации.


```
public interface Iterator<E> {  
    default void remove() {  
        throw new UnsupportedOperationException("remove");  
    }  
}
```




Итерация и удаление

```
Set<Integer> numbers = new HashSet<>(Arrays.asList(5, 1, 2, 3, 4));  
// remove all even numbers from the set  
Iterator<Integer> it = numbers.iterator();  
while (it.hasNext()) {  
    if (it.next() % 2 == 0) {  
        it.remove();  
    }  
}
```

Это эффективный способ, но требует отказа от конструкции for each.

 функциональные
интерфейсы Java упрощают
жизнь, но об этом не сегодня



Итерация элементов Map

Подход 1. Перечислим все ключи (keySet) и для каждого запросим значение (get).

```
Map<String, Integer> wordCounts = countWords();  
for (String word : wordCounts.keySet()) {  
    int count = wordCounts.get(word);  
    System.out.println(word + ": " + count);  
}
```

Подход может быть не очень эффективным. Например, в случае с TreeMap сложность составит $O(n \times \lg(n))$.



Итерация элементов Map

Подход 2.

```
Map<String, Integer> wordCounts = countWords();  
for (Map.Entry<String, Integer> entry : wordCounts.entrySet()) {  
    String word = entry.getKey();  
    int count = entry.getValue();  
    System.out.println(word + ": " + count);  
}
```

К сожалению, в случае с TreeMap сложность по-прежнему остается

$O(n \times \lg(n))$, хотя могла быть линейной (но с использованием дополнительной памяти).

Компараторы



java.lang.Comparable<T>

Определяет естественный порядок (natural order) на объектах класса T.

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```



java.lang.Comparable<T>

Варианты возвращаемого значения:

```
int cmp = a.compareTo(b):
```

- $cmp < 0 \Rightarrow a < b$
- $cmp = 0 \Rightarrow a = b$
- $cmp > 0 \Rightarrow a > b$



Свойства отношения compareTo

Рефлексивность:

$$x.compareTo(x) = 0$$

Асимметричность:

$$x.compareTo(y) = -y.compareTo(x)$$

Транзитивность:

$$x.compareTo(y) < 0 \wedge y.compareTo(z) < 0 \Rightarrow x.compareTo(z) < 0$$



Согласованность с equals

```
x.compareTo(y) = 0  $\Rightarrow$  x.equals(y)
```

Сортировка



Сортировка с учетом естественного порядка

```
int[] array = new int[] {3, 2, 7, 1};  
Arrays.sort(array);
```

```
List<Integer> list = Arrays.asList(3, 2, 7, 1);  
Collections.sort(list);
```

Сортировка выполняется с учетом реализации Comparable.

**Что делать, если нужен
другой порядок**

—



java.util.Comparator<T>

Определяет произвольный порядок на объектах класса T.

Класс T не должен реализовывать интерфейс Comparator<T>.

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Семантика Comparator.compare(x, y) аналогична семантике Comparable.compareTo(y).

**Пример. Упорядочить точки на
плоскости по удаленности от
заданной**

—



Класс точки

```
public class Point {  
    private final double x;  
    private final double y;  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    public double getX() {  
        return x;  
    }  
    public double getY() {  
        return y;  
    }  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
}
```



Компаратор

```
public class DistanceComparator implements Comparator<Point> {  
    private final Point target;  
    public DistanceComparator(Point target) {  
        this.target = target;  
    }  
    private double squareDistance(Point p1, Point p2) {  
        return Math.pow(p1.getX() - p2.getX(), 2) + Math.pow(p1.getY() - p2.getY(),  
2);  
    }  
    @Override  
    public int compare(Point p1, Point p2) {  
        double d1 = squareDistance(p1, target);  
        double d2 = squareDistance(p2, target);  
        return Double.compare(d1, d2);  
    }  
}
```




Сортировка

```
List<Point> points = new ArrayList<>();  
points.add(new Point(0, 0));  
points.add(new Point(2, 2));  
points.add(new Point(1, 1));  
  
Collections.sort(points,  
    new DistanceComparator(new Point(3, 3)));  
  
System.out.println(points);  
  
Out:  
[(2.0, 2.0), (1.0, 1.0), (0.0, 0.0)]
```



Сортировка

Вместо

```
Collections.sort(points,  
    new DistanceComparator(new Point(3, 3)));
```

МОЖНО ИСПОЛЬЗОВАТЬ

```
points.sort(new DistanceComparator(new Point(3, 3)));
```



Сортировка в обратном порядке

```
Collections.reverseOrder()  
Collections.reverseOrder(comparator)  
или  
comparator.reversed()
```

Предыдущий пример, но более удаленные точки идут сначала:

```
Comparator<Point> comparator = new DistanceComparator(new Point(3, 3))  
points.sort(comparator.reversed());
```

Queue & Deque



LIFO & FIFO

Queue: FIFO — First In First Out

Добавляем в конец, читаем с начала.

Stack: LIFO — Last In First Out

Добавляем в начало, читаем с начала.

Deque: FIFO + LIFO



java.util.Queue<E>

	<i>Throws exception</i>	<i>Returns special value</i>
Insert	<code>add(e)</code>	<code>offer(e)</code>
Remove	<code>remove()</code>	<code>poll()</code>
Examine	<code>element()</code>	<code>peek()</code>



Реализации Queue

- ArrayDeque
- LinkedList
- PriorityQueue



java.util.Deque<E>

```
public interface Deque<E> extends Queue<E> {  
    // ...  
}
```

Реализации:

- ArrayDeque
- LinkedList



Deque как очередь

Queue Method	Equivalent Deque Method
<code>add(e)</code>	<code>addLast(e)</code>
<code>offer(e)</code>	<code>offerLast(e)</code>
<code>remove()</code>	<code>removeFirst()</code>
<code>poll()</code>	<code>pollFirst()</code>
<code>element()</code>	<code>getFirst()</code>
<code>peek()</code>	<code>peekFirst()</code>



Deque как стек

Stack Method	Equivalent Deque Method
<code>push(e)</code>	<code>addFirst(e)</code>
<code>pop()</code>	<code>removeFirst()</code>
<code>peek()</code>	<code>peekFirst()</code>



Пример работы с очередью

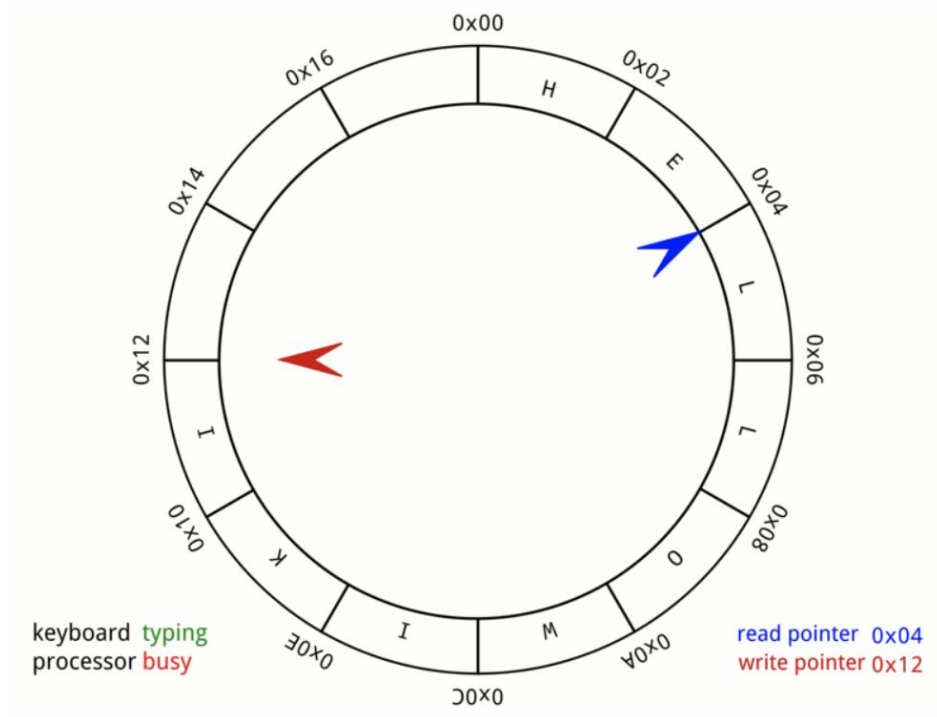
```
Queue<String> killQueue = new ArrayDeque<>();
killQueue.offer("Dragon Mother");
killQueue.offer("Cersei Lannister");
killQueue.offer("Jon Snow");
while (!killQueue.isEmpty()) {
    String nextKill = killQueue.poll();
    System.out.println(nextKill + " has been killed :(");
}
```

Out:

```
Dragon Mother has been killed :(
Cersei Lannister has been killed :(
Jon Snow has been killed :(
```

Circular Buffer 🥰❤️

Circular buffer как основа ArrayDeque





Устаревшие классы коллекций

Вместо `Vector` используем `ArrayList`.

Вместо `Hashtable` используем `HashMap`.

Вместо `Stack` используем `Deque (ArrayDeque)`.