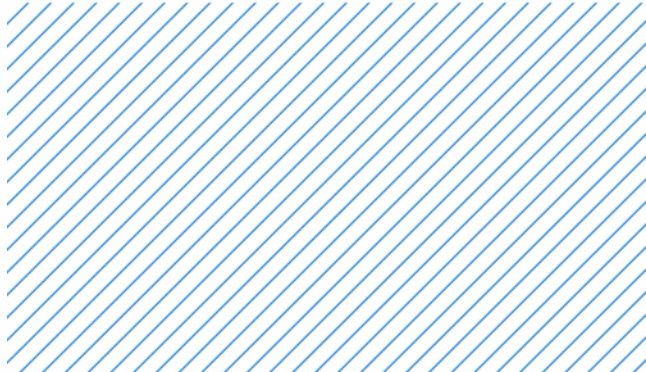
HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 12

## Коллекции

# Программирование на языке Java

Роман Гуров
ВШЭ БИ 2021



# Queue

**Queue** – интерфейс, соответствующий Абстрактному Типу Данных “Очередь”

Очередь обеспечивает порядок **FIFO** – **F**irst **I**n **F**irst **O**ut: «первым вошёл – первым вышел»

Queue похож на настоящую очередь:
элементы добавляются в конец и извлекаются из начала

```
public interface Queue<E> extends Collection<E> {

    boolean add(E value);
    boolean offer(E value);

    E remove();
    E poll();

    E element();
    E peek();
}
```

Методы можно разделить на три типа: добавить, вытащить и подсмотреть элемент

# Queue

```
public interface Queue<E> extends Collection<E> {

    boolean add(E value);
    boolean offer(E value);

    E remove();
    E poll();

    E element();
    E peek();
}
```

`add` и `offer` добавляют элемент в хвост очереди

В чем отличие?

Очередь может быть ограничена по своему максимальному размеру

В случае, когда добавляется элемент сверх этого лимита:

- `add` бросит исключение
- `offer` просто вернёт `false`

# Queue

```
public interface Queue<E> extends Collection<E> {

    boolean add(E value);
    boolean offer(E value);

    E remove();
    E poll();

    E element();
    E peek();
}
```

`remove` и `poll` извлекают элемент из начала очереди:
то есть, удаляют его и возвращают нам

Что, если в момент извлечения очередь пуста?

Поведение будет таким:

- `remove` бросит исключение
- `poll` просто вернёт `null`

# Queue

```java
public interface Queue<E> extends Collection<E> {

    boolean add(E value);
    boolean offer(E value);

    E remove();
    E poll();

    E element();
    E peek();
}
```

`element` и `peek` позволяют подсмотреть элемент в начале очереди без извлечения

Различие аналогично предыдущим

Если очередь пуста:

- `element` бросит исключение
- `peek` просто вернёт `null`

Получается, что `add`, `remove` и `element` в случае невозможности бросают исключение

А `offer`, `poll` и `peek` возвращают особое значение

То есть, вторые просто являются *смягчёнными* версиями первых

# Deque

**Deque** (читается: *дэк*) – **d**ouble-**e**nded **que**ue – двухсторонняя очередь

Расширение интерфейса `Queue`, позволяющее добавлять и извлекать элементы с обеих сторон

```
public interface Deque<E> extends Queue<E> {

    void addFirst(E var1);
    void addLast(E var1);

    boolean offerFirst(E var1);
    boolean offerLast(E var1);

    E removeFirst();
    E removeLast();

    E pollFirst();
    E pollLast();

    E getFirst();
    E getLast();

    E peekFirst();
    E peekLast();
}
```

Методы как у `Queue`, но у каждого есть две версии: для начала и для конца дека

# Deque

```
public interface Deque<E> extends Queue<E> {

    void addFirst(E var1);
    void addLast(E var1);

    boolean offerFirst(E var1);
    boolean offerLast(E var1);

    E removeFirst();
    E removeLast();

    E pollFirst();
    E pollLast();

    E getFirst();
    E getLast();

    E peekFirst();
    E peekLast();
}
```

Но ведь методы Queue назывались иначе, разве дек их не наследует?

Наследует. Поэтому получается,
что дек дублирует методы очереди под другими именами

| Метод Queue | Эквивалентный метод Deque |
|---|---|
| add(e) | addLast(e) |
| offer(e) | offerLast(e) |
| remove() | removeFirst() |
| poll() | pollFirst() |
| element() | getFirst() |
| peek() | peekFirst() |

Делать они должны одно и то же

# Имплементации Deque

Существуют две основные имплементации интерфейса `Deque` (а он является и `Queue`)

```
public class ArrayDeque<E> extends /* ... */ implements Deque<E>, /* ... */ {
```

`ArrayDeque` – дек на базе динамического массива, который хитро модифицирован, чтобы позволить быстро добавлять и удалять элементы не только в конец, но и в начало

```
public class LinkedList<E> extends /* ... */ implements List<E>, Deque<E>, /* ... */ {
```

А это нам уже знакомо: оказывается, связный список хорошо реализует интерфейсы как `List`, так и `Deque`

```
Deque<Integer> deque = new ArrayDeque<>();

deque.offerFirst(1);
deque.addFirst(2);
deque.offerLast(3);
deque.addLast(4);

Integer element;

while ((element = deque.pollFirst()) != null) {
    System.out.println(element);
}
```

```
2
1
3
4
```

# Имплементации Queue

Возникает вопрос: а есть ли `Queue`, который не `Deque`?

Есть -- например, `PriorityQueue` -- очередь с приоритетом

В такой очереди элементы можно вставлять в любом порядке,
при извлечении из очереди достаётся *минимальный* из всех элементов

Также, есть понятие блокирующей очереди -- интерфейса `BlockingQueue`

Именно она позволяет ограничивать свой максимальный размер

Её особенность -- есть специальные методы добавления/изъятия, которые при переполнении/нехватке блокируют исполнение программы до освобождения места/появления новых элементов

Особенность используется в многопоточных программах,
потому что в однопоточных состояние объекта изменяется только из одного потока,
а значит блокироваться в ожидании изменений бесполезно

# Set

**`Set`** – интерфейс, соответствующий АТД “Множество”

Представляет собой неупорядоченный набор элементов *без повторений*:
`Set` не может содержать двух элементов, *равных* друг другу

При попытке добавить в него дубликат методом `add`, в ответ вернётся `false`

```
public interface Set<E> extends Collection<E> {

    /* ... */

}
```

Интерфейс `Set` не добавляет никаких новых методов к интерфейсу `Collection`,
он служит как явное уточнение того, что должна быть реализована описанная семантика

# HashSet

Основная реализация интерфейса `Set` – `HashSet`

`HashSet` реализует множество на основе хэш-таблицы

```
Set<ComplexNumber> set = new HashSet<>();
set.add(new ComplexNumber(3, 3));
set.add(new ComplexNumber(3, 3));
set.add(new ComplexNumber(3, 3));
System.out.println(set.size());  // prints 1

set.remove(new ComplexNumber(3, 3));
System.out.println(set.size());  // prints 0
```

Особенность хэш-таблицы – операции добавления, удаления и проверки на наличие элемента все работают по времени за `O(1)` в среднем

# HashSet

```
Set<ComplexNumber> set = new HashSet<>();
set.add(new ComplexNumber(3, 3));
set.add(new ComplexNumber(3, 3));
set.add(new ComplexNumber(3, 3));
System.out.println(set.size());  // prints 1

set.remove(new ComplexNumber(3, 3));
System.out.println(set.size());  // prints 0
```

Использование `HashSet` подразумевает соблюдение некоторых требований

Для него то нам и необходима корректная реализация метода `hashCode()` у хранимых объектов

Но также нужен и корректный `equals()`

- `hashCode()` используется для первоначального выбора ячейки, в которой должен лежать элемент
  - `equals()` нужен для конкретной проверки, тот же элемент лежит в ячейке или нет

Именно тут важно выполнение следствия: (`equals == true => hashCode` равны)

Также, нельзя изменять (в плане `equals`) объекты, лежащие внутри хэш-таблицы
Это почти гарантированно приведёт к некорректной её работе

# HashSet

Что происходит при итерировании по хэш-таблице?

Оно, естественно, поддерживается, но порядок обхода, по сути, случайный

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

set.add("odin");
set.add("dva");
set.add("tri");

for (String str : set) {
    System.out.println(str);
}
```

```
odin
tri
dva
```

Если хочется сохранять порядок добавления – есть другая реализация `Set` – `LinkedHashSet`

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

set.add("odin");
set.add("dva");
set.add("tri");

for (String str : set) {
    System.out.println(str);
}
```

```
odin
dva
tri
```

`LinkedHashSet` представляет собой связный список с прикрученной рядом хэш-таблицей для быстрого поиска

# SortedSet

У `Set` есть наследник – интерфейс `SortedSet` – упорядоченное множество

Благодаря упорядоченности, обход элементов итератором происходит в порядке их возрастания

```
public interface SortedSet<E> extends Set<E> {

    SortedSet<E> subSet(E var1, E var2);

    SortedSet<E> headSet(E var1);

    SortedSet<E> tailSet(E var1);

    E first();

    E last();
}
```

Также, становятся доступны новые полезные операции:

- `first()` – возвращает первый элемент: то есть *минимальный* из всех
- `last()` – возвращает последний элемент: то есть *максимальный* из всех

# SortedSet

```
public interface SortedSet<E> extends Set<E> {

    SortedSet<E> subSet(E fromElement, E toElement);

    SortedSet<E> headSet(E toElement);

    SortedSet<E> tailSet(E fromElement);

    E first();

    E last();
}
```

Также, становятся доступны новые полезные операции:

`headSet()` возвращает подмножество всех элементов, меньше переданного

Это подмножество не является копией
В случае изменения состава оригинального множества, подмножество из `headSet()` увидит все эти изменения

`tailSet()` – то же самое, но элементы *больше либо равны* переданному

`subSet()` – пересечение `headSet()` и `tailSet()`

# TreeSet

Реализация интерфейса `SortedSet` – класс `TreeSet` – двоичное дерево поиска

```
SortedSet<String> words = new TreeSet<>();

words.add("apple");
words.add("banana");
words.add("carrot");
System.out.println(words);

SortedSet<String> head = words.headSet("carrot");
System.out.println(head);

head.clear();
System.out.println(words);
```

Как научить `TreeSet` правильно сравнивать ваши объекты?

# Компараторы

Как научить `TreeSet` правильно сравнивать ваши объекты?

Есть два варианта

1) Ваш объект реализует интерфейс `Comparable<T>`:

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

```
public final class Integer extends Number implements Comparable<Integer> {
    private final int value;

    public int compareTo(Integer anotherInteger) {
        return compare(this.value, anotherInteger.value);
    }

    public static int compare(int x, int y) {
        return x < y ? -1 : (x == y ? 0 : 1);
    }
}
```

Метод сравнения должен возвращать знак от `(this - other)`

Тогда, `TreeSet` сможет сравнивать объекты, вызывая его

# Компараторы

Как научить `TreeSet` правильно сравнивать ваши объекты?

2) Вы предоставляете в конструктор `TreeSet` свой компаратор:

```
public interface Comparator<T> {
    int compare(T a, T b);
}
```

Компаратор – функциональный интерфейс, описывающий ту самую функцию сравнения, но вне класса

Этим способом можно воспользоваться если вы не хотите/не можете реализовать метод `compareTo`, или если хочется поменять его обычное поведение

```
class ReverseComparator<T extends Comparable<T>> implements Comparator<T> {
    public int compare(T a, T b) {
        return -a.compareTo(b);
    }
}
```

Например, можем упорядочить элементы по убыванию

```
SortedSet<String> words = new TreeSet<>(new ReverseComparator<>());

words.add("apple");
words.add("banana");
words.add("carrot");
System.out.println(words);
```

```
[carrot, banana, apple]
```

# Пример полезности множеств

Используя `Set` можно очень легко очистить список от повторяющихся элементов:

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

words.add("apple");
words.add("apple");
words.add("apple");
words.add("apple");
words.add("carrot");
words.add("carrot");
words.add("banana");
words.add("carrot");
words.add("apple");
words.add("banana");
words.add("banana");

Set<String> set = new LinkedHashSet<>(words);

List<String> wordsWithoutDuplicates = new ArrayList<>(set);

System.out.println(wordsWithoutDuplicates);
```

```
[apple, carrot, banana]
```

Тут нам пригождается наличие у коллекций конструкторов, принимающих любого другого наследника `Collection`

Использование `LinkedHashSet` позволяет даже сохранить оригинальный порядок следования элементов

# Интерфейс Map

Интерфейс **`Map`** – соответствует АТД “Ассоциативный массив”

Также, называется «*словарь*» или «*отображение*»

В отличие от обычного массива, словарь позволяет индексировать свои объекты не только числом от `1` до `size()`, а произвольным другим объектом

Записи в словаре выглядят как пары объектов `(ключ, значение)`

Ключ – уникальный объект-идентификатор, по которому словарь умеет быстро найти соответствующее ему значение

То есть, ключ – и есть аналог индекса из обычного массива, каждому значению соответствует некоторый ключ

# Интерфейс Map

Интерфейс **`Map`** ввиду своей специфики *не* является коллекцией:

```
public interface Map<K, V> {

    int size();
    boolean isEmpty();

    boolean containsKey(Object var1);
    boolean containsValue(Object var1);

    V get(Object key);
    V put(K key, V value);

    V remove(Object key);
    void clear();

    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
}
```

- `containsKey()` – проверка на наличие пары с таким ключом
- `containsValue()` – проверка на наличие пары с таким значением

- `get()` – получает значение по ключу
- `put()` – добавляет пару ключ-значение в словарь

- `remove()` – удаление пары с данным ключом
- `clear()` – очистка словаря

Методы `put` и `remove` возвращают значение, которое было по данному ключу до выполнения операции, `put` перезапишет значение, если ключ уже занят

# Интерфейс Map

```
public interface Map<K, V> {

    int size();
    boolean isEmpty();

    boolean containsKey(Object var1);
    boolean containsValue(Object var1);

    V get(Object key);
    V put(K key, V value);

    V remove(Object key);
    void clear();

    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
}
```

Как итерироваться по Map, раз уж у него нет итераторов?

Есть специальные методы

- `keySet()` – возвращает множество всех ключей в словаре (они и так уникальны)
- `values()` – возвращает коллекцию всех значений в словаре (значения могут повторяться)
- `entrySet()` – возвращает множество всех записей словаря (то есть пар ключ-значение)

# Реализации Map

Map очень похож на `Set`, в котором каждому элементу подвесили ещё и некоторое значение

Реализации `Map` такие же, как и у сета: `HashMap`, `LinkedHashMap`, `TreeMap` (с интерфейсом `SortedMap`)

Рассмотрим пример использования и разных итерирований по мапе:

```
Map<String, Integer> word_counts = new HashMap<>();
word_counts.put("the", 300);
word_counts.put("a", 220);
word_counts.put("of", 300);
word_counts.put("and", 200);
word_counts.put("a", 100);

for (String key : word_counts.keySet()) {
    System.out.println(key);
}

for (Integer value : word_counts.values()) {
    System.out.println(value);
}

for (Map.Entry<String, Integer> entry : word_counts.entrySet()) {
    System.out.println(
        "Word '" + entry.getKey() +
        "' has occurred in the text " + entry.getValue() + " times"
    );
}
```

```
the
a
and
of
```

```
300
100
200
300
```

```
Word 'the' has occurred in the text 300 times
Word 'a' has occurred in the text 100 times
Word 'and' has occurred in the text 200 times
Word 'of' has occurred in the text 300 times
```

# Старые типы коллекций

В Java есть несколько устаревших типов коллекций:

- `Vector`
- `Stack`
- `Dictionary`
- `Hashtable`

Они существуют ещё со времён первой джавы, и их использование не рекомендуется

Для каждого из них придуман более предпочтительный аналог:

- `Vector -> ArrayList`
- `Stack -> Deque (?)`
- `Dictionary -> Map`
- `Hashtable -> HashMap`

# Класс Collections

В работе с коллекциями бывает полезен утилитный класс `Collections`

Этот класс содержит много удобных `static`-методов для работы с коллекциями, например для перемешивания и сортировки списков:

```
List<Integer> list = new ArrayList<>();
for (int i = 0; i < 10; ++i) {
    list.add(10 - i);
}
System.out.println(list);

Collections.shuffle(list);
System.out.println(list);

Collections.sort(list);
System.out.println(list);
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[9, 8, 2, 6, 1, 10, 7, 4, 5, 3]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Неизменяемые коллекции

В Collections также есть семейство функций, начинающихся со слова `unmodifiable`:

`unmodifiableList, unmodifiableMap, unmodifiableSet, unmodifiableSortedMap, unmodifiableSortedSet`

Смысл методов – сделать неизменяемую обёртку над коллекцией

```
Set<String> immutableSet = Collections.unmodifiableSet(set);

immutableSet.remove("apple");
// throws java.lang.UnsupportedOperationException
```

Попытка вызова изменяющих операций приведёт к бросанию исключения

# Collection и простые массивы

Интерфейс коллекции имеет метод `toArray`, позволяющий преобразовать содержимое коллекции в обычный массив

```
List<Integer> list = new ArrayList<>();

Object[] array1 = list.toArray();
```

Есть проблема – из-за ограничений дженериков, метод не может создать массив нужного типа, поэтому он создаёт `Object[]`

Чтобы получить массив правильного типа, придется создать его самому и передать в перегрузку `toArray()`:

```
Integer[] array2 = list.toArray(new Integer[0]);
```

Метод заполнит переданный массив, и при необходимости даже пересоздаст его с увеличенным размером

# Collection и простые массивы

Также есть способы превратить обычный массив в коллекцию

С этим поможет утилитный класс `Arrays`

Метод `Arrays.asList()` превращает обычный массив в список:

```
String[] array = {"A", "B", "C"};

Set<String> set1 = new HashSet<>(Arrays.asList(array));
```

Также, можно добавить элементы из массива в уже существующую коллекцию методом `Collections.addAll()`:

```
Set<String> set2 = new HashSet<>();
Collections.addAll(set2, array);
```

# Ещё о классе Arrays

Метод `Arrays.asList()` умеет принимать аргументы не только в виде массива

Это можно считать аналогом инициализации обычных массивов через фигурные скобки:

```
String[] array = {"A", "B", "C"};

Set<String> set1 = new HashSet<>(Arrays.asList(array));
```

```
Set<String> set2 = new HashSet<>(Arrays.asList("A", "B", "C"));
```

Ещё в `Arrays` есть полезный для обычных массивов метод `equals()`, сравнивающий два любых массива поэлементно:

```
Object[] array1 = {1, "test", 13.37};

Object[] array2 = {1, "bad_test", 13.37};

System.out.println(Arrays.equals(array1, array2));
```