Java Core
Collection framework & Generics

Recording

Collection framework

Introduction

Основные темы:

- Основные структуры данных
- Оценка сложности алгоритмов
- Иерархия и основные компоненты Collection framework
- List, Queue, Map, Set
- Stream API

Why we need Collections framework?

Collections framework – иерархия интерфейсов и классов, являющаяся частью JDK и предоставляющая возможность разработчикам использовать различные структуры данных.

Разработан Джошуа Блохом

Data structures

Какие структуры данных будем рассматривать:

- Массив
- Список
- Стек
- Очередь
- Хеш-таблица
- Множество
- Дерево

Big O notation

Временная сложность алгоритма это зависимость времени выполнения алгоритма от количества входных данных.

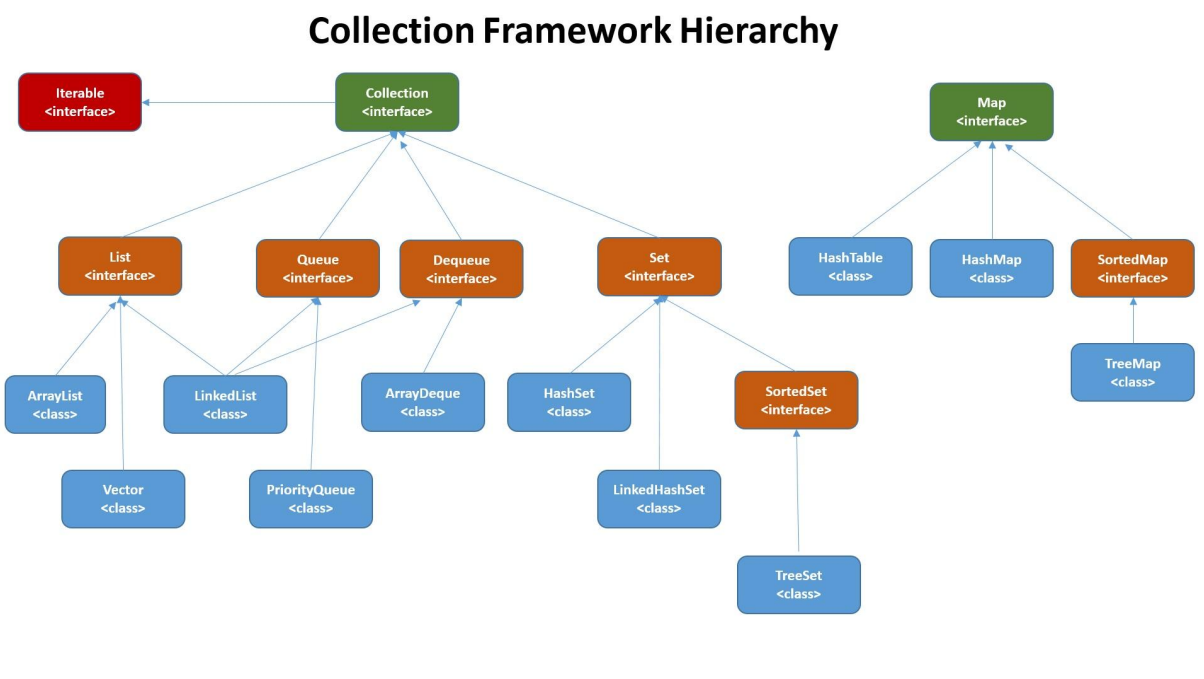
O – нотация (***O(f(n)***) определяет асимптотическое поведение функции, т.е. характер изменения функции при стремлении аргумента к определенному значению.

Big O notation

Порядки роста:

Операция	Порядок роста	Пример
Константный	1	Поиск по индексу в массиве
Логарифмический	logN	Бинарный поиск
Линейный	N	Цикл
Линейно-логарифмический	NlogN	Сортировка слиянием
Квадратичный	N^2	Двойной цикл
Кубический	N^3	Тройной цикл
Экспоненциальный	2^N	Двойная рекурсия

Collections hierarchy



Collection interface

Collection – основной интерфейс, от которого наследуются все коллекции (кроме Map). Представляет собой коллекцию объектов-элементов.

```
public interface Collection <E> extends java.lang.Iterable<E> {
```

Методы:

- int size();
- boolean isEmpty();
- boolean contains(Object o);
- Iterator<E> iterator();
- Object[] toArray();
- <T> T[] toArray(T[] a);
- boolean add(E e);
- boolean remove(Object o);
- boolean containsAll(Collection<?> c);
- boolean addAll(Collection<? extends E> c);
- boolean removeAll(Collection<?> c);
- boolean retainAll(Collection<?> c);

Методы с Java 8:

- default Spliterator<E> spliterator()
- default Stream<E> stream()
- default Stream<E> parallelStream()

Iterable & Iterator

Iterable – предоставляет возможность использование объекта в **for-each** выражении.

```
public interface Iterable<T> {

    Iterator<T> iterator();


    // since Java 8
    default void forEach(Consumer<? super T> action) {
        // impl
    }
    default Spliterator<T> spliterator() {
        // impl
    }
}
```

Iterator – позволяет поочередно получать элементы коллекции.

```
public interface Iterator<E> {

    boolean hasNext();

    E next();

    default void remove() {
        // impl
    }

    default void forEachRemaining(Consumer<? super E> action) {
        // impl
    }
}
```

Iterable & Iterator

Явное использование итератора:

```
List<String> strings = Arrays.asList("Hello", "Tinkoff");
Iterator<String> it = strings.iterator();
while (it.hasNext()){
    System.out.println(it.next());
}
```

Использование итератора в for-each loop

```
List<String> strings = Arrays.asList("Hello", "Tinkoff");
for (String str : strings){
    System.out.println(str);
}
```

Iterator

Удаление элементов

```
Collection<Integer> coll =
        new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
Iterator<Integer> iterator = coll.iterator();
while (iterator.hasNext()) {
    Integer next = iterator.next();
    if (next % 2 == 0) {
        coll.remove(next);
    }
}
System.out.println(coll);
```

```
Exception in thread "main" java.util.ConcurrentModificationException
```

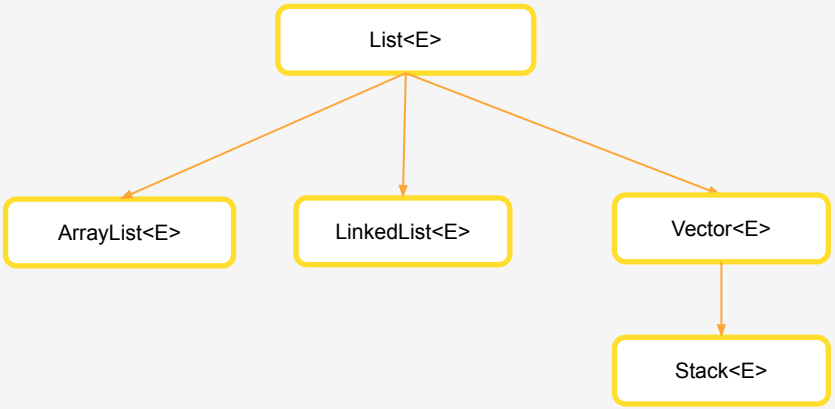
При использовании итератора, удалять элементы нужно методом iterator.remove()

```
Collection<Integer> coll =
        new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
Iterator<Integer> iterator = coll.iterator();
while (iterator.hasNext()) {
    Integer next = iterator.next();
    if (next % 2 == 0) {
        iterator.remove();
    }
}
System.out.println(coll); // 1, 3, 5
```

List

List – представляет собой упорядоченный список объектов.

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



Особенности:

- Объекты упорядочены в порядке вставки
- Доступ к элементам по индексу
- Позволяет хранить дубликаты

Методы:

- int indexOf(Object o) boolean isEmpty();
- int lastIndexOf(Object o);
- ListIterator<E> listIterator();
- E set(int index, E element);
- E get(int index);

Представители:

- **ArrayList** – массив объектов
- **LinkedList** – связный список
- **Stack** – реализация стека (синхронизирован)
- **Vector** – динамический массив (синхронизирован)

ArrayList

ArrayList – динамически расширяемый массив

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

Пример:

```
List<String> arr = new ArrayList<>(); // default capacity 10
List<String> arr50 = new ArrayList<>( initialCapacity: 50); // capacity 50


arr.add("Hello");
arr.add("Tinkoff");

String str = arr.get(0);
System.out.println(str); // Hello
```

ArrayList

ArrayList состоит из:

- Object[] elementData – массив с хранимыми объектами
- int size – размер массива (**size != capacity**)

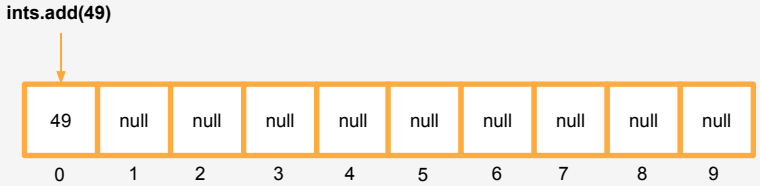
Создание ArrayList:

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

size: 0

Добавление элемента:

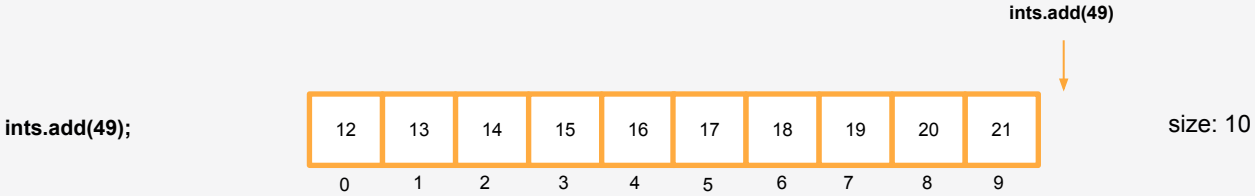
```
ints.add(49);
```



size: 1

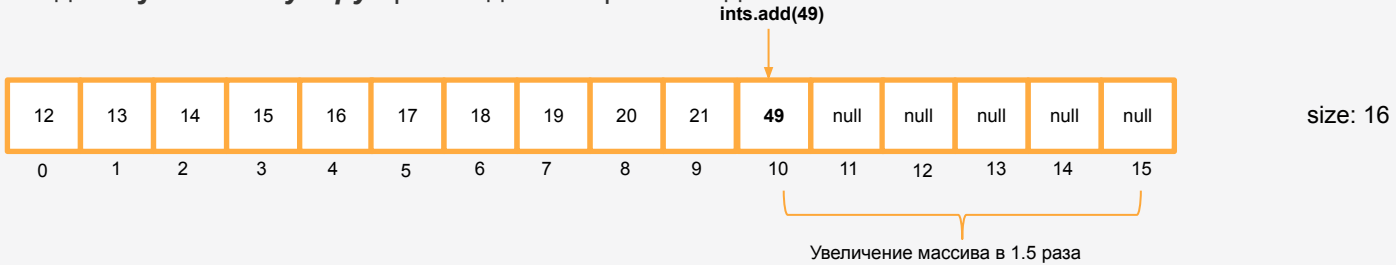
ArrayList

Добавление элемента в заполненный массив:



Метод **ensureCapacityInternal** проверяет заполняемость массива и при необходимости расширяет его

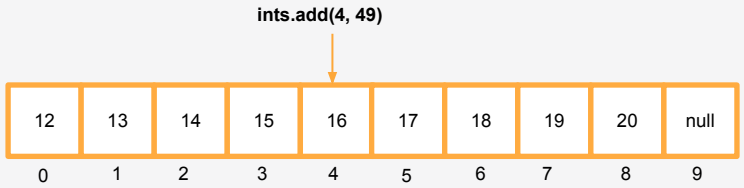
- Создается новый массив увеличенного размера
- Нативным методом **System.*arraycopy*** происходит копирование данных в новый массив



ArrayList

Добавление элемента в середину массива:

ints.add(4, 49);



size: 9

Перед добавлением методом **ensureCapacityInternal** проверяется заполняемость, если массив не заполнен, добавляется новый элемент и происходит сдвиг элементов в правую часть массива методом **System.*arraycopy***

ints.add(4, 49);

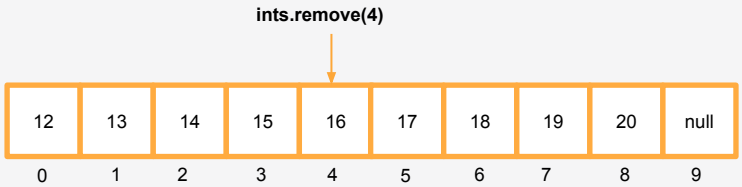
size: 10

Если вставка в середину выполняется в заполненный массив, то метод **System.*arraycopy*** вызовется дважды!

ArrayList

Удаление элемента из массива:

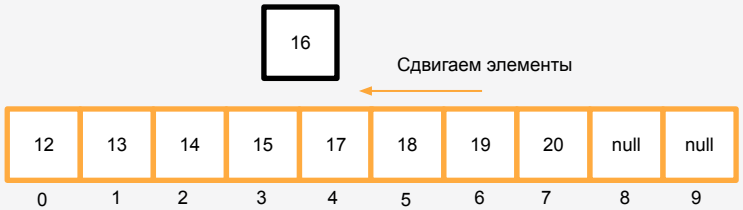
ints.remove(4);



size: 9

Рассчитывается количество элементов, которые необходимо сдвинуть после удаления и происходит сдвиг элементов копированием методом **System.*arraycopy***

ints.remove(4);



size: 8

Метод **trimToSize()** используется для уменьшения capacity массива

ArrayList

Оценка сложности основных операций:

Операция	Реализация	Сложность
Поиск/перебор элементов	Используем iterator()	O(n)
Поиск по индексу	get(int index)	O(1)
Вставка элемента	add(E element)	O(1), в худшем случае O(n)
Вставка элементу в середину	add(int index, E element)	O(n)
Удаление	remove(int index)	O(n)

LinkedList

LinkedList – двунаправленный связный список

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

Пример:

```
List<String> list = new LinkedList<>();

list.add("Hello");
list.add("Tinkoff");

String value = list.get(0);
System.out.println(value); // Hello
```

LinkedList

LinkedList состоит из:

- Node<E> first – указатель на первый элемент списка
- Node<E> last – указатель на последний элемент списка
- int size – размер списка

LinkedList представляет собой набор элементов или узлов (Node), каждый из которых имеет ссылку на следующий и предыдущий элемент (узел) списка

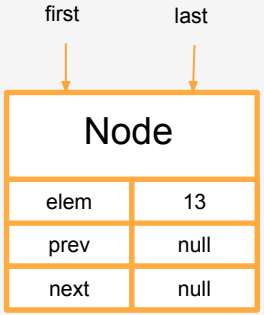
Реализация Node:

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

LinkedList

Добавление элементов в список:

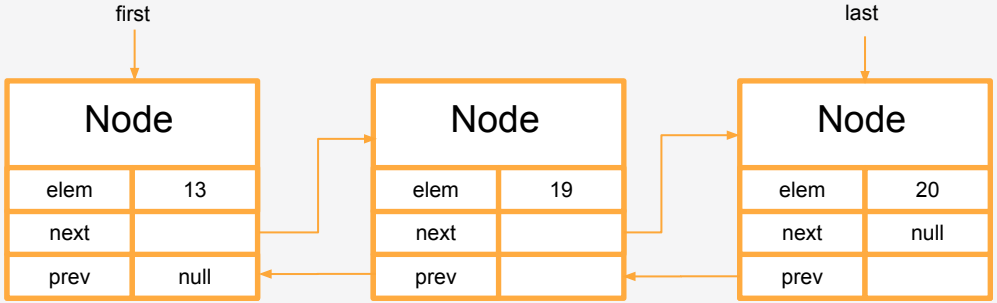
list.add(13)



size: 1

list.add(19)

list.add(20)



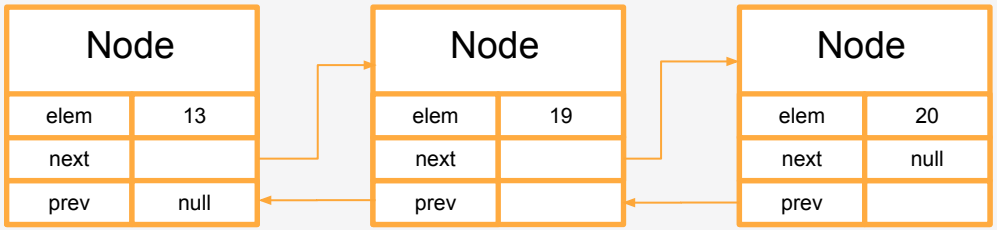
size: 3

Для быстрого добавления элементов в начало и конец списка используются методы **addFirst(E e)** и **addLast(E e)**

LinkedList

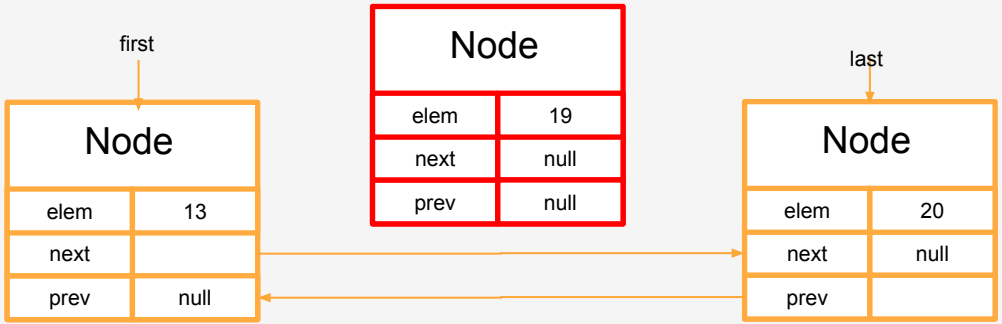
Итерация и поиск:

Итерация по списку или доступ к элементу осуществляется путем последовательного прохода по узлам списка с использованием указателей **prev** и **next**:



Удаление элемента:

list.remove(1)



Для быстрого удаления элементов в начале и конце списка используются методы **removeFirst()** и **removeLast()**

LinkedList

Оценка сложности основных операций:

Операция	Реализация	Сложность
Поиск/перебор элементов	Используем iterator()	O(n)
Поиск по индексу	get(int index)	O(n)
Вставка элемента	add(E element)	O(1)
Вставка элементу в середину	add(int index, E element)	O(n)
Удаление	remove(int index)	O(n)
Удаление из конца/начала	removeFirst(), removeLast()	O(1)

Vector, Stack

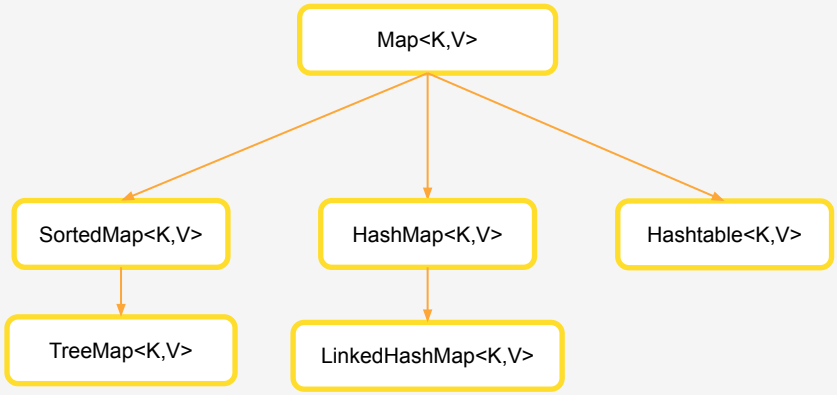
Редко используемые реализации списков:

- **Vector<E>** – динамический массив (методы synchronized)
- **Stack<E>** – реализация стека на основе динамического массива (методы synchronized)
  - public E push(E item)
  - public synchronized E pop()
  - public synchronized E peek()

Map

Map – множество элементов, хранящихся в формате ключ-значение

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



Особенности:

- Ключ – уникальный идентификатор
- Порядок элементов зависит от реализации
- Доступ к элементу осуществляется по ключу

Методы:

- boolean containsKey(Object key)
- boolean containsValue(Object value);
- V get(Object key)
- V put(K key, V value);
- V remove(Object key)
- Set<K> keySet()
- Collection<V> values()
- Set<Map.Entry<K, V>> entrySet()

HashMap

HashMap – хеш-таблица, реализованная на основе динамического массива

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
```

Не поддерживает порядок вставки элементов.

Пример:

```
Map<Integer, String> map = new HashMap<>();
map.put(123, "Hello");
map.put(1234, "Tinkoff");

System.out.println(map.get(123)); // Hello
System.out.println(map.containsKey(1234)); // true
System.out.println(map.containsValue("Hello")); // true
```

HashMap

HashMap состоит из:

- Node<K,V>[] table – содержимое хеш-таблицы
- Set<Map.Entry<K,V>> entrySet – кешированое множество элементов хеш-таблицы
- int size – размер хеш-таблицы
- final float loadFactor – порядок загруженности хеш-таблицы (default – **0.75**)
- int threshold – предельное кол-во элементов до увеличения хеш-таблицы

Node – внутренний класс HashMap, представляет элемент хеш-таблицы

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
```

HashMap

Как происходит добавление элемента методом **put (K key, V value)**?

- Методом int hash(Object key) вычисляется hashcode ключа
- Вычисляем на основе hashcode индекс массива для хранения элемента
- Создаем объект Node
- Сохраняем Node в указанной ячейке

Объект использующийся в качестве ключа должен соответствовать контракту hashCode и equals:

- Если два объекта равны (equals()), то у них обязательно должен быть одинаковый hashCode
- Если два объекта имеют одинаковый hashCode, то они не обязательно равны

Требования к ключу:

- Объект ключа не должен изменяться
- Желательно делать ключ immutable
- Не использовать массивы для ключа

HashMap

Создание:

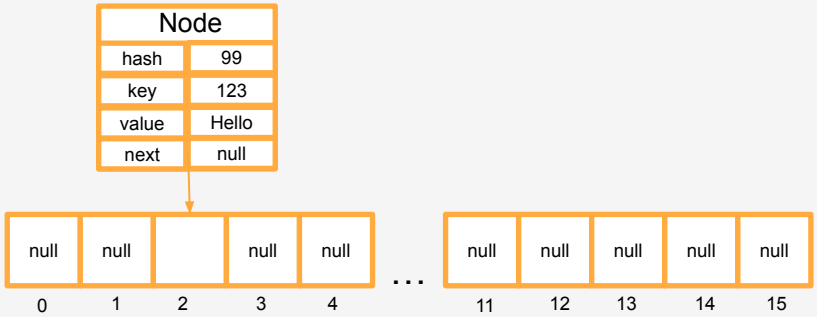
new HashMap()

new HashMap(100, 0.85)

size: 0 capacity: 16

Добавление элемента:

map.put(123, “Hello”)

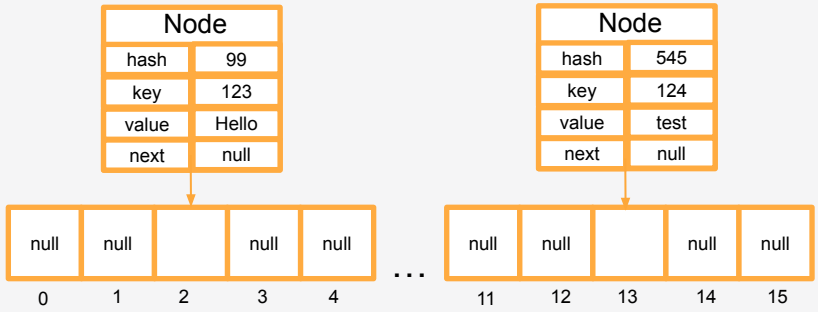


size: 1 capacity: 16

HashMap

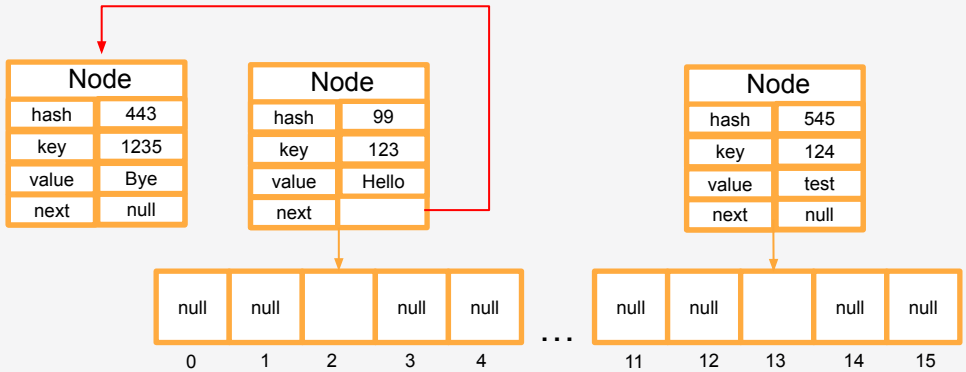
Добавление элемента:

put(124, “test”)



size: 2 capacity: 16

put(1235, “Bye”)



size: 3 capacity: 16

HashMap

Коллизия – ситуация, когда элементы с разными ключами попадают в одну и ту же корзину.

Возможные причины коллизий:

- Одинаковый hashCode ключей
- На основании разных hashCode ключей вычисляется один и тот же индекс в массиве

В HashMap коллизии разрешаются методом **external chaining** – создание цепочек элементов, ссылающихся друг на друга, то есть создается **связный список**

Таким образом добавление происходит следующим образом:

- Методом int hash(Object key) вычисляется hashcode ключа
- Вычисляем на основе hashcode индекс массива для хранения элемента
- Создаем объект Node
- Если в указанной ячейке массива уже есть элемент Node, то происходит проход по связному списку и сравнение ключей по equals
- Если найдено соответствие по equals – заменяем объект, если нет – добавляем в конец списка

HashMap

А если коллизий много?

При достижении размера списка в **TREEIFY_THRESHOLD** равному 8 элементам, происходит ребалансировка в красно-черное дерево

Метод **final void treeifyBin(Node<K,V>[] tab, int hash)**:

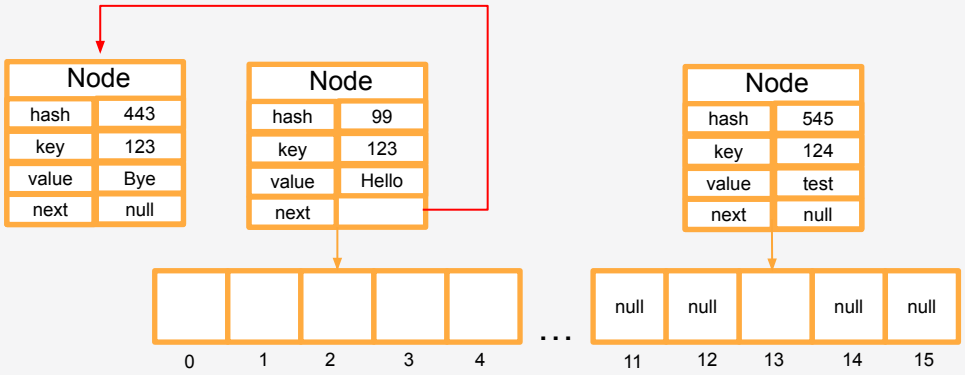
- Если кол-во элементов в хеш-таблице меньше **MIN_TREEIFY_CAPACITY** равному 64 элементам, то вызывается метод **resize()** для увеличения хеш-таблицы и перераспределения элементов
- Если кол-во элементов меньше порогового значения, то связный список для ячейки массива указанного значения hash ребалансируется в древовидную структуру **TreeNode**

Как происходит сравнение элементов для хранения в **TreeNode:**

- Сравниваются значения hash в Node
- Если хеши двух элементов равны, пробуем сравнить используя Comparable (если ключ реализовывает интерфейс)
- Если хеши двух элементов равны и не реализуют Comparable – используется метод **System.*identityHashCode*()**

HashMap

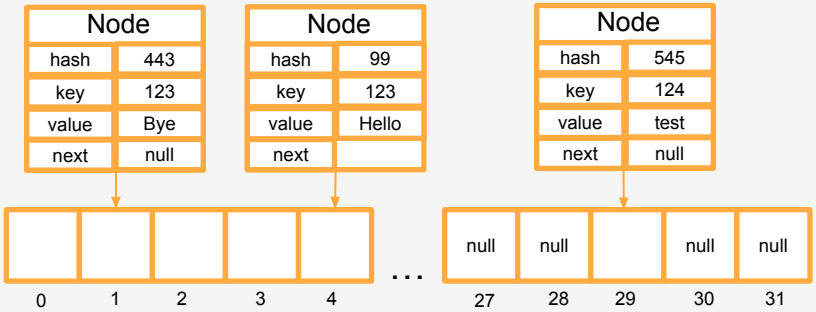
Расширение массива:



size: 12 capacity: 16

Когда размер хеш-таблицы превышает порог **threshold**, вызывается метод **resize()** увеличивающий размер вдвое и перераспределяет элементы:

Элементы перераспределяются при перестроении, таким образом устраняя коллизии (не гарантировано):

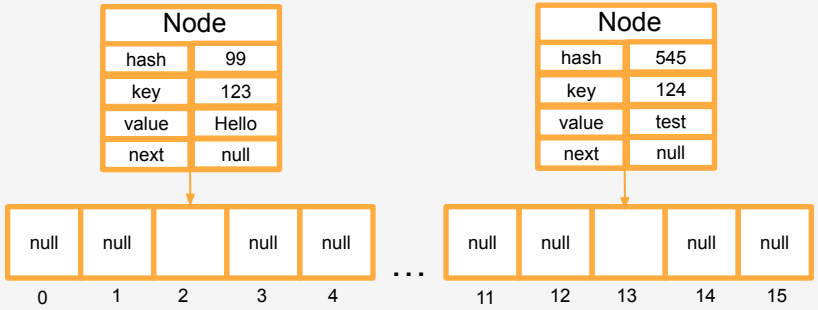


size: 12 capacity: 32

HashMap

Удаление элемента:

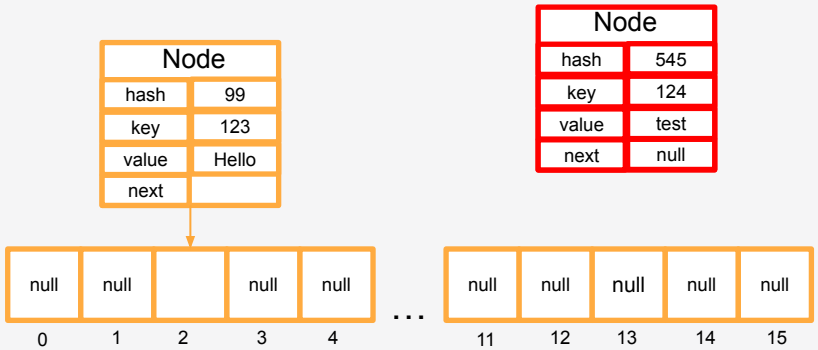
remove(124)



size: 2 capacity: 16

При удалении элемента из хеш-таблицы для поиска элемента происходит процесс аналогичный добавлению элемента.

remove(124)



size: 1 capacity: 16

Во избежание утечек памяти используется метод **trimToSize()**

HashMap

Оценка сложности основных операций:

Операция	Реализация	Сложность
Поиск по ключу	get(K key)	O(1) O(n) – коллизии в виде списка O(logN) – коллизии в дерева
Вставка элемента	put(K key, V value)	O(1) O(n) – коллизии в виде списка O(logN) – коллизии в дерева
Удаление	remove(int index)	O(1) O(n) – коллизии в виде списка O(logN) – коллизии в дерева

LinkedHashMap

LinkedHashMap – хеш-таблица, реализованная на основе динамического массива, являющаяся так же связным списком.

```
public class LinkedHashMap<K,V>
    extends HashMap<K,V>
    implements Map<K,V>
```

Поддерживает порядок вставки

Пример:

```
LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
map.put(123, "Hello");
map.put(1234, "Tinkoff");
map.put(59, "Test");

Set<Integer> keys = map.keySet();
System.out.println(keys); // 123, 1234, 59
```

LinkedHashMap

LinkedHashMap содержит те же поля что и HashMap, а так же пару дополнительных:

- LinkedHashMap.Entry<K,V> head – начало двусвязного списка
- LinkedHashMap.Entry<K,V> tail – конец двусвязного списка
- boolean accessOrder – порядок доступа к элементам

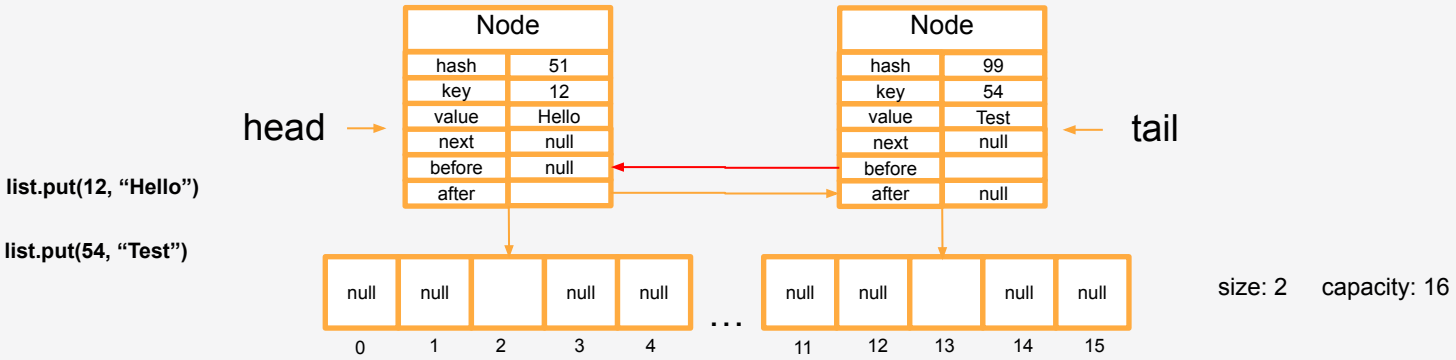
Для хранения элементов используется внутренний класс Entry, аналогичный классу Node в HashMap с дополнением ссылок на предыдущий и следующий элемент:

```
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) { super(hash, key, value, next); }
}
```

LinkedHashMap

Операции удаления, получения по ключу, удаления аналогичны операциям HashMap. Добавляется лишь логика связывания элементов в список

Элементы LinkedHashMap ссылаются друг на друга с использованием дополнительных полей **after** и **before**



LinkedHashMap

Поле **accessOrder** определяет порядок итерации по элементам:

- accessOrder=true – итерация в порядке обращения
- accessOrder=false – итерация в порядке вставки

accessOrder = true

```
Map<Integer, String> map =
        new LinkedHashMap<>( initialCapacity: 16, loadFactor: 0.75f, accessOrder: true);
map.put(1, "Hello");
map.put(2, "Tinkoff");
map.put(3, "Test");

System.out.println(map); // {1=Hello, 2=Tinkoff, 3=Test}
String secondValue = map.get(2);
System.out.println(map); // {1=Hello, 3=Test, 2=Tinkoff}
```

accessOrder = false

```
Map<Integer, String> map =
        new LinkedHashMap<>( initialCapacity: 16, loadFactor: 0.75f, accessOrder: false);
map.put(1, "Hello");
map.put(2, "Tinkoff");
map.put(3, "Test");

System.out.println(map); // {1=Hello, 2=Tinkoff, 3=Test}
String secondValue = map.get(2);
System.out.println(map); // {1=Hello, 2=Tinkoff, 3=Test}
```

TreeMap

TreeMap – реализация map, элементы которой отсортированы по ключу и хранятся в структуре RBT.

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
```

Порядок элементов согласуется с порядок сортировки

Natural order

```
TreeMap <Integer, String> map = new TreeMap<>();
map.put(45, "Hello");
map.put(1233, "Tinkoff");
map.put(3, "Test");

System.out.println(map); // {3=Test, 45=Hello, 1233=Tinkoff}
```

Comparator:

```
TreeMap<Integer, String> map = new TreeMap<>(Comparator.reverseOrder());
map.put(45, "Hello");
map.put(1233, "Tinkoff");
map.put(3, "Test");

System.out.println(map); // {1233=Tinkoff, 45=Hello, 3=Test}
```

TreeMap

TreeMap состоит из:

- final Comparator<? super K> comparator – компаратор для сравнения элементов
- Entry<K,V> root – корневой элемент
- int size – размер коллекции

Для хранения элементов используется внутренний класс Entry:

```
static final class Entry<K,V> implements Map.Entry<K,V> {
    K key;
    V value;
    Entry<K,V> left;
    Entry<K,V> right;
    Entry<K,V> parent;
    boolean color = BLACK;
```

TreeMap

О механизме сортировки TreeMap

- Сортировка элементов TreeMap происходит по ключу.
- По умолчанию сортировка происходит согласно natural order
- В конструкторе TreeMap можно указать кастомный **Comparator** для сортировки
- ...вит...

Object не реализует Comparable:

```
TreeMap<Object, String> map = new TreeMap<>();

map.put(new Object(), "Hello"); // ClassCastExcesption
map.put(new Object(), "Tinkoff");
map.put(new Object(), "Test");
```

String реализует Comparable:

```
TreeMap<String, String> map = new TreeMap<>();

map.put("First", "Hello");
map.put("Second", "Tinkoff");
map.put("Third", "Test");
```

Явно указан Comparator:

```
TreeMap<Object, String> map =
        new TreeMap<>(Comparator.comparing(Object::hashCode));

map.put(new Object(), "Hello");
map.put(new Object(), "Tinkoff");
map.put(new Object(), "Test");
```

TreeMap

```java
public class User implements Comparable<User> {

    private String name;
    private Integer age;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) && Objects.equals(age, user.age);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }

    @Override
    public int compareTo(User o) {
        return this.age.compareTo(o.getAge());

    }
```

Что будет выведено на экран?

```java
Map<User, String> usersToCompany = new TreeMap<>();

usersToCompany.put(new User( name: "Misha",  age: 29), "Tinkoff");
usersToCompany.put(new User( name: "Masha",  age: 24), "Google");
usersToCompany.put(new User( name: "Oleg",  age: 29), "Neflix");

System.out.println(usersToCompany);

// {User{name='Masha', age=24}=Google, User{name='Misha', age=29}=Neflix}
```

TreeMap

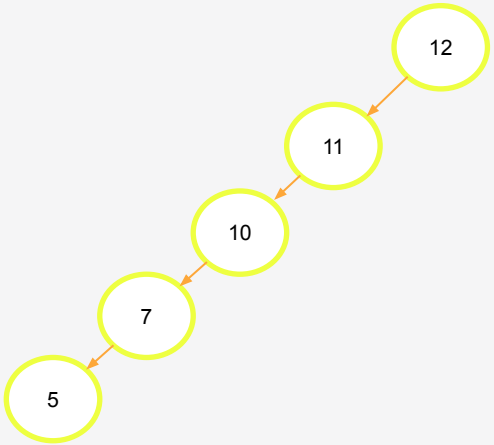
Особенности работы **TreeMap**:

- Для сравнения ключей TreeMap использует метод **compareTo** либо метод **compare**, если указан компаратор
- Методы **hashCode** и **equals** не используются в TreeMap
- Рекомендуется соблюдать контракт: ***(x.compareTo(y)==0) == (x.equals(y))***

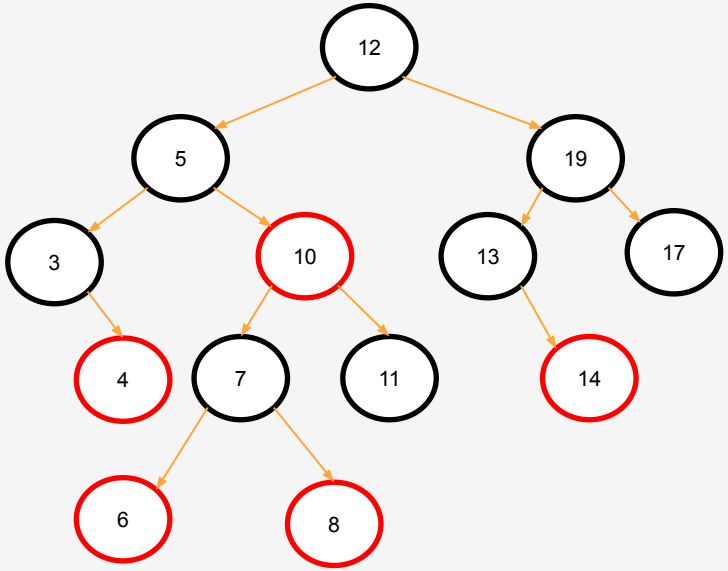
TreeMap

TreeMap реализует структуру Red-Black Tree для хранения элементов в отсортированном порядке.

Обычное бинарное дерево не сбалансировано, в худших случаях давая скорость поиска O(N)



RBT сбалансировано. Поиск за O(log(N))



TreeMap

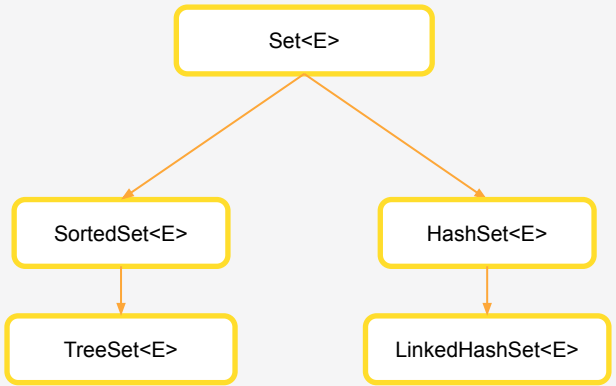
Оценка сложности основных операций:

Операция	Реализация	Сложность
Поиск по ключу	get(K key)	O(logN)
Вставка элемента	put(K key, V value)	O(logN)
Удаление	remove(int index)	O(logN)

Set

Set – представляет собой множество уникальных элементов

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



Особенности:

- Элементы уникальны в рамках коллекции
- Порядок элементов зависит от конкретной реализации

Представители:

- **TreeSet** – элементы отсортированы по возрастанию (RBT)
- **HashSet** – хеш-множество
- **LinkedHashSet** – хеш-множество, элементы хранятся в порядке вставки

Методы:

- **add(E e)** – добавить элемент
- **contains(E e)** – проверка наличия элемента
- **containsAll(Collection<E> c)** – проверка наличия элемнетов

HashSet

HashSet – множество элементов, использующее для хранения хеш-таблицу

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
```

Пример:

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

hashSet.add("Hello");
hashSet.add("Tinkoff");

System.out.println(hashSet.contains("Hello")); // true
```

HashSet

Получение элемента из Hashset:

Пример:

```
Set<String> set = new HashSet<>();
set.add("Hello");
set.add("Tinkoff");
// Compilation error!
System.out.println(set.get(0));
```

Set не предоставляет методов для получения объектов (кроме обхода через итератор)

HashSet

HashSet в своей реализации использует HashMap:

- В качестве ключа – сами объекты
- В качестве значения – «мусорный» объект

Пример:

HashSet состоит из:

```
private transient HashMap<E,Object> map;
```

Добавление элемента:

```
public boolean add(E e) { return map.put(e, PRESENT)==null; }

private static final Object PRESENT = new Object();
```

Конструктор HashSet

```
public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<>(initialCapacity, loadFactor);
}
```

Проверка наличия элемента:

```
public boolean contains(Object o) {
    return map.containsKey(o);
}
```

LinkedHashSet

LinkedHashSet – множество элементов, использующее для хранения хеш-таблицу в сочетании с двусвязным списком.

Основана на реализации **LinkedHashMap**

```
public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {
```

Пример:

```
LinkedHashSet<Integer> set = new LinkedHashSet<>();
set.add(1);
set.add(3);
set.add(3);

System.out.println(set); // 1, 3
```

Конструктор:

```
public LinkedHashSet() {
    super( initialCapacity: 16,  loadFactor: .75f,  dummy: true);
}
```

```
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```

TreeSet

TreeSet – множество элементов, отсортированных в natural order или согласно указанному Comparator.

Основана на реализации **TreeMap**

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
```

Пример:

```
TreeSet<Integer> set = new TreeSet<>();
set.add(10);
set.add(5);
set.add(3);
set.add(19);
set.add(10);

System.out.println(set); // 3, 5, 10, 19
```

Конструктор:

```
public TreeSet() {
    this(new TreeMap<>());
}
```

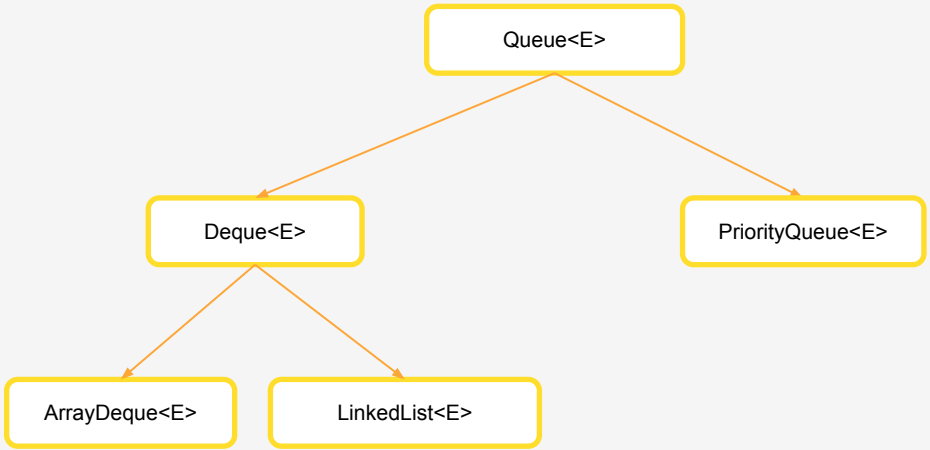
Добавление элемента:

```
public boolean add(E e) {
    return m.put(e, PRESENT)==null;
}
```

Queue

Queue – представляет структуру данных очередь, работающую по принципу FIFO (first in - first out)

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



Особенности:

- Элементы очереди упорядочены
- Порядок элементов очереди зависит от реализации

Методы:

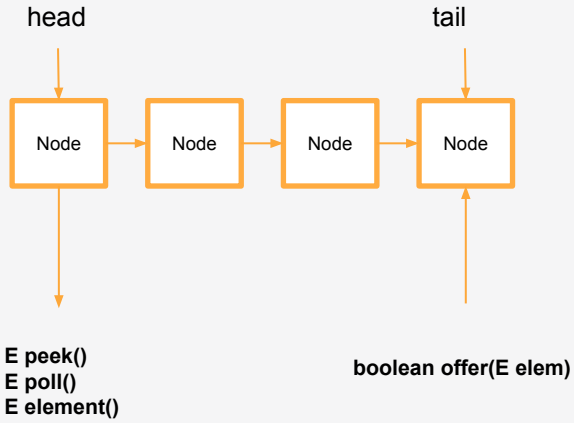
- boolean offer(E e)
- E remove()
- E poll()
- V put(K key, V value);
- E element()
- E peek()

LinkedList as Queue

LinkedList - пример реализации очереди с классически FIFO

Основные методы:

- **boolean offer(E elem)** – добавление элемента в конец очереди
- **E element()** – получить (не удалить) элемент из начала очереди
- **E peek()** - получить (не удалить) элемент из начала очереди (null if empty)
- **E poll()** – получить и удалить элемент из начала очереди (null if empty)



PriorityQueue

PriorityQueue - очередь с приоритетом. Элементы в очереди отсортированы в **natural order** или согласно указанному **Comparator**.

```
public class PriorityQueue<E> extends AbstractQueue<E>
    implements java.io.Serializable {
```

Основные моменты:

- Доступ к элементам очереди согласно приоритету (определяет порядок сортировки)
- Нарушает стандартный механизм FIFO

Пример:

```
PriorityQueue<Integer> queue = new PriorityQueue<>();
queue.offer( e: 1);
queue.offer( e: 10);
queue.offer( e: 3);
queue.offer( e: 4);

while (!queue.isEmpty()) {
    System.out.println(queue.poll()); // 1, 3, 4, 10
}
```

PriorityQueue

PriorityQueue состоит из:

- Object[] queue – массив с элементами очереди
- int size – размер очереди
- final Comparator<? super E> comparator – компаратор для сравнения элементов

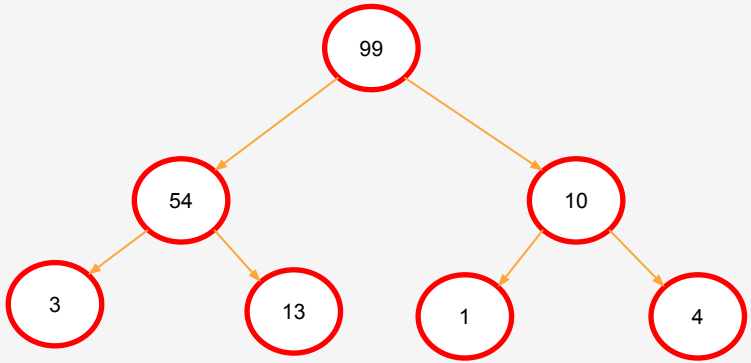
```
PriorityQueue<Integer> queue =
        new PriorityQueue<>(Comparator.reverseOrder());
queue.offer( e: 1);
queue.offer( e: 10);
queue.offer( e: 3);
queue.offer( e: 4);
queue.offer( e: 99);
queue.offer( e: 54);
queue.offer( e: 13);

while (!queue.isEmpty()) {
    System.out.println(queue.poll()); // 99, 54, 13, 10, 4, 3, 1
}
```

size: 7

PriorityQueue

В основе **PriorityQueue** лежит структура данных - бинарная куча:

- Значение в любой вершине не меньше, чем значения её потомков
- Глубина всех листьев отличается не более чем на 1 слой



Массивы удобны для представления двоичной кучи, элементы располагаются следующим образом:

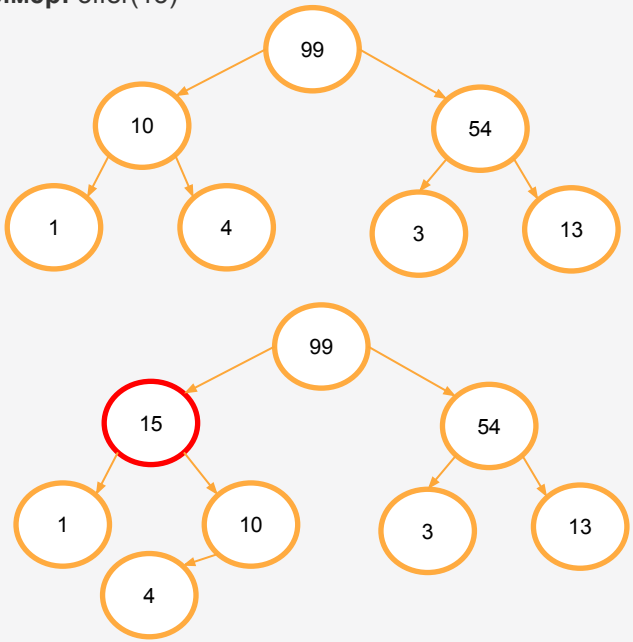
- Корень элемента - **A[0]**
- Листьями элемента **A[N]** являются **A[2N] и A[2N+1]**

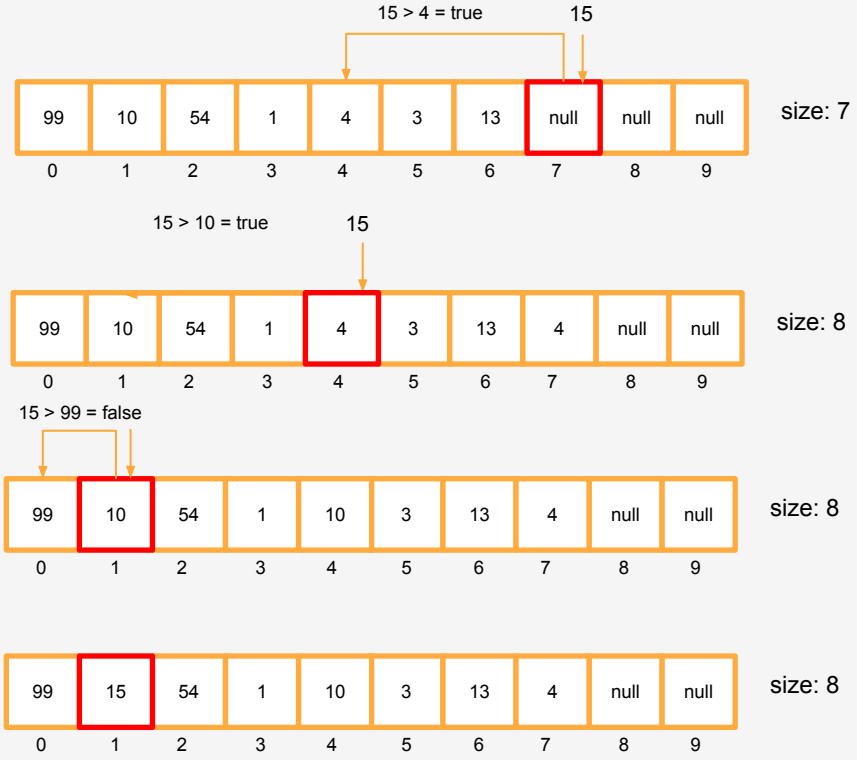
size: 7

PriorityQueue

Добавление элементов происходит методом **siftUp(int k, E x)** , где k – позиция вставки, E – элемент.

Пример: offer(15)

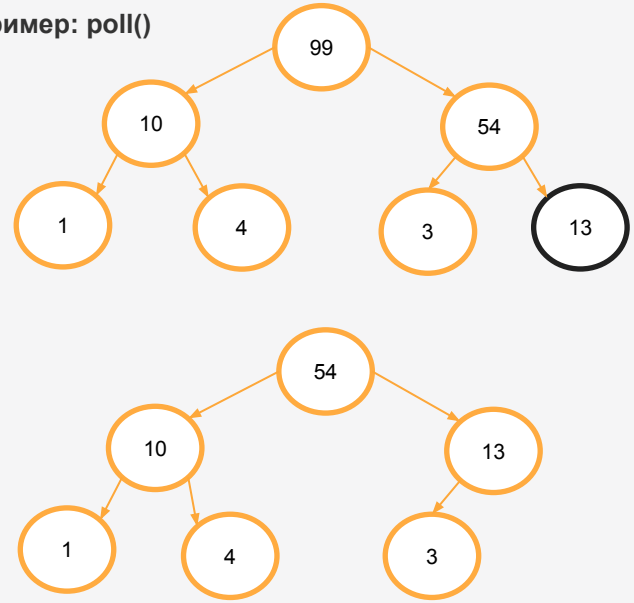


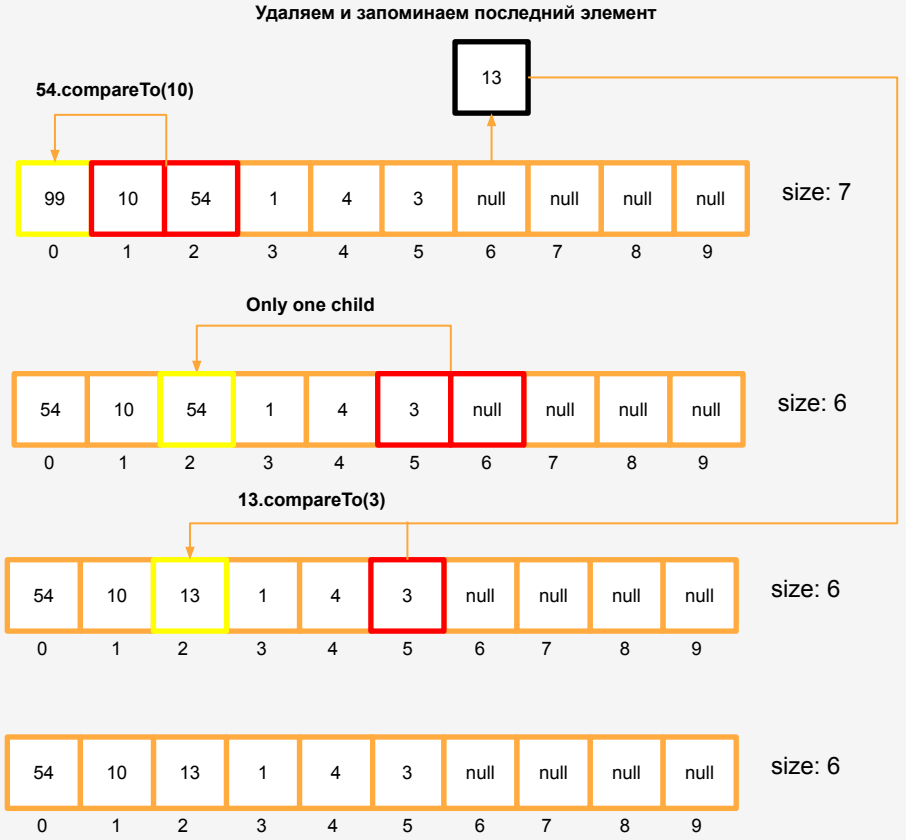


PriorityQueue

Получение следующего элемента очереди происходит методом **siftDown(int k, E x)** , где k – позиция вставки, E – элемент для вставки (последний элемент массив)

Пример: poll()





PriorityQueue

Оценка сложности основных операций:

Операция	Реализация	Сложность
Добавить в очередь	boolean offer(E elem)	O(logN)
Получить из очереди	E poll()	O(logN)

Deque

Deque – расширяет интерфейс Queue, добавляю функциональность двунаправленной очереди

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

Особенности:

- Элементы очереди упорядочены
- Возможность добавлять/выбирать элементы с обоих концов очереди

Методы:

- void addFirst(E e)
- void addLast(E e)
- boolean offerFirst(E e)
- boolean offerLast(E e)
- E pollFirst()
- E pollLast()
- E peekFirst()
- E peekLast()
- void push(E e)
- E pop()

ArrayDeque

ArrayDeque - реализация двухсторонней очереди на примере динамического массива

```
public class ArrayDeque<E> extends AbstractCollection<E>
                           implements Deque<E>, Cloneable, Serializable
```

Пример:

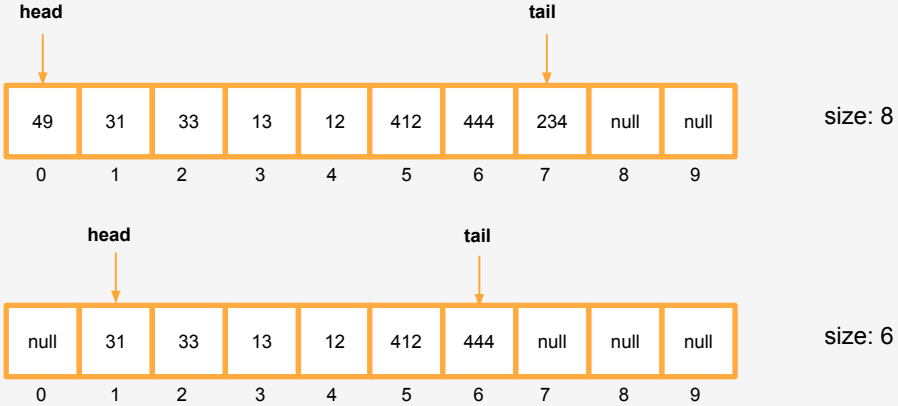
```
ArrayDeque<Integer> queue = new ArrayDeque<>();
queue.offer( e: 1);
queue.offer( e: 2);
queue.offerFirst( e: 3);
queue.offerFirst( e: 4);

while (!queue.isEmpty()) {
    System.out.println(queue.poll()); // 4, 3, 1, 2
}
```

ArrayDeque

ArrayDeque состоит из:

- Object[] elementData – массив с хранимыми объектами
- int size – размер массива
- int head
- int tail

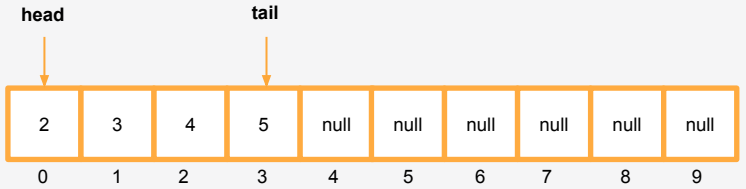
Работает аналогично ArrayList, дополнительно реализуя функциональность двухсторонней очереди при помощи указателей на начало и конец массива **head** и **tail**



pollFirst()
pollLast()

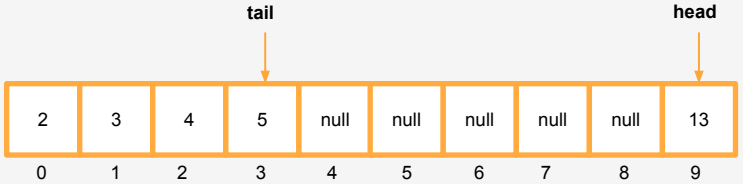
ArrayDeque

Что произойдет если вставить в начало очереди: **offerFirst(13)** ?



size: 4

ArrayDeque работает по принципу **circular buffer**, за счет указателей head и tail



size: 5

Thread safety?

Большая часть рассмотренных коллекций **НЕ** потокобезопасна (кроме hashtable, vector, stack).

Для достижения потокобезопасности применяется:

- Потокобезопасные реализации:
  - ConcurrentHashMap
  - CopyOnWriteArrayList
  - ArrayBlockingQueue
  - PriorityBlockingQueue

- Методы класса Collections:
  - *synchronizedList(List<T> list)*
  - *synchronizedCollection(Collection<T> coll)*
  - *synchronizedMap(Map<K,V> map)*
  - *synchronizedSet(Set<T> set)*

EnumSet

Типичный пример хранения Enum объектов:

```
public enum OrderStatus {
    CREATED,
    CANCELLED,
    PAID,
    DELETED
}
```

```
Set<OrderStatus> badStatuses =  new HashSet<>();
badStatuses.add(OrderStatus.DELETED);
badStatuses.add(OrderStatus.PAID);
```

EnumSet

Для хранения множества Enum объектов используем EnumSet:

```
public abstract class EnumSet<E extends Enum<E>> extends AbstractSet<E>
    implements Cloneable, java.io.Serializable
```

EnumSet использует **bit vector** для хранение enum, используя метод int ordinal()

Создание:

```
EnumSet.of(OrderStatus.DELETED, OrderStatus.PAID);
EnumSet.allOf(OrderStatus.class);
```

```
class RegularEnumSet<E extends Enum<E>> extends EnumSet<E>
```

Для хранения элементов использует:

```
private long elements = 0L;
```

```
class JumboEnumSet<E extends Enum<E>> extends EnumSet<E>
```

Для хранения элементов использует:

```
private long elements[];
```

EnumMap

При выборе Map можно использовать EnumMap, если в качестве ключа Enum объекты:

Не оптимальное использование HashMap:

```
public enum OrderStatus {
    CREATED,
    CANCELLED,
    PAID,
    DELETED
}
```

```
Map<OrderStatus, List<Order>> orderByStatus =
        new HashMap<>();
```

```
public class EnumMap<K extends Enum<K>, V> extends AbstractMap<K, V>
    implements java.io.Serializable, Cloneable
```

Создание:

```
EnumMap<OrderStatus, List<Order>> ordersByStatus =
        new EnumMap<>(OrderStatus.class);
```

Для хранения использует:

```
private transient Object[] vals;
```

Since Java 9

Добавились полезные фабричные утилитарные методы:

- Set.of()
- List.of()
- Map.of()

Stream API

Stream API – набор инструментов (since Java 8), предоставляющих возможность функционального подхода обработки данных

Зачем нужны?

```
static class User {
    private final String name;
    private final String sex;
    private final int age;

    // ... getters and setters
```

```
List<User> users = Arrays.asList(
        new User( name: "Misha",  sex: "MALE",  age: 19),
        new User( name: "Kolya",  sex: "MALE",  age: 34),
        new User( name: "Anya",  sex: "FEMALE",  age: 23),
        new User( name: "Sasha",  sex: "FEMALE",  age: 44),
        new User( name: "Vasya",  sex: "MALE",  age: 52),
        new User( name: "John",  sex: "MALE",  age: 12)
);
```

```
public static void printMaleYounger20(List<User> users){
    List<User> maleYoungerThan20 = new ArrayList<>();
    for (User user : users) {
        if (user.getAge() < 20) {
            if (user.getSex().equals("MALE")) {
                maleYoungerThan20.add(user);
            }
        }
    }
    maleYoungerThan20.sort(new Comparator<User>() {
        @Override
        public int compare(User u1, User u2) {
            return u1.getName().compareTo(u2.getName());
        }
    });
    for (User user : maleYoungerThan20) {
        System.out.println(user.getName());
    }
}
```

Stream API

Stream API предоставляет возможность работать со структурами данных в функциональном стиле, упрощает работу с обработкой потока данных, делая код более читабельным

```
static class User {
    private final String name;
    private final String sex;
    private final int age;

    // ... getters and setters
```

```
List<User> users = Arrays.asList(
        new User( name: "Misha",  sex: "MALE",  age: 19),
        new User( name: "Kolya",  sex: "MALE",  age: 34),
        new User( name: "Anya",  sex: "FEMALE",  age: 23),
        new User( name: "Sasha",  sex: "FEMALE",  age: 44),
        new User( name: "Vasya",  sex: "MALE",  age: 52),
        new User( name: "John",  sex: "MALE",  age: 12)
);
```

```
public static void printMaleYounger20(List<User> users) {
    users.stream()
            .filter(user -> user.getAge() < 20)
            .filter(user -> user.getSex().equals("MALE"))
            .sorted(Comparator.comparing(User::getName))
            .forEach(user -> System.out.println(user.getName()));
}
```

Stream

Stream – последовательность элементов для, потоковой обработки

```
public interface Stream<T> extends BaseStream<T, Stream<T>>
```

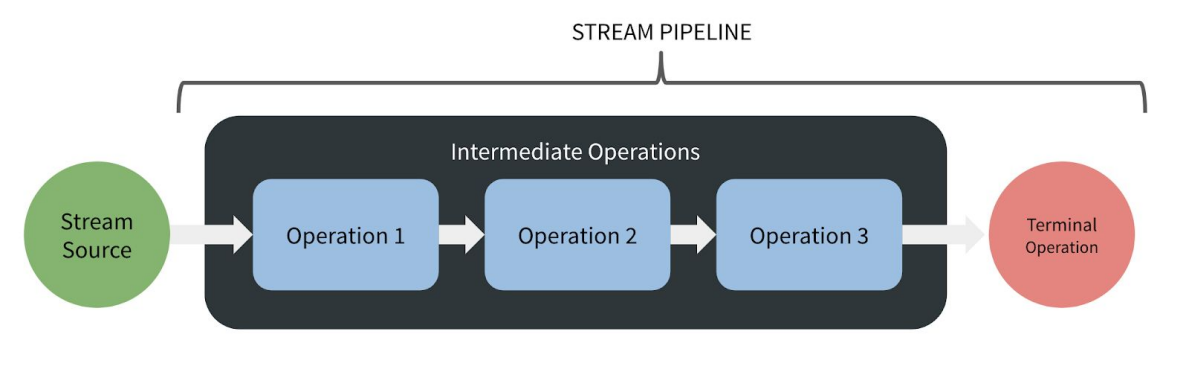
Основные моменты:

- Предоставляет интерфейс для последовательности элементов определенного типа
- Источником данных могут являться коллекции, массивы, файлы и т.д.
- Предоставляют промежуточные и терминальные операции для работы с последовательность элементов
- Stream не является хранилищем объектов

Stream API

Stream Pipeline состоит из:

- Источник данных (коллекция, массив, файл и т.д.)
- Промежуточные операции (filter, map, flatMap)
- Терминальные операции. (collect, forEach)



Stream Source

Как создается Stream?

```
// методы Collection
Stream<String> s1 = Arrays.asList("Hello", "Tinkoff").stream();
Stream<String> s2 = Arrays.asList("Hello", "Tinkoff").parallelStream();
// методы Stream
Stream<String> s3 = Stream.of("Hello", "Tinkoff");
Stream<String> s4 = Stream.empty();
Stream<String> s5 = Stream.concat(Stream.of("Tinkoff"), Stream.of("Tinkoff"));
Stream<UUID> s6 = Stream.generate(UUID::randomUUID);
// метод  Arrays
String[] arr = {"Hello", "Tinkoff"};
Stream<String> s7 = Arrays.stream(arr);
// метод Files
Stream<String> s8 = Files.lines(Paths.get( first: "/usr/test"));
// и другие
IntStream s9 = IntStream.range(1,1000);
```

Intermediate operations

Промежуточные операции:

- map(Function m)
- filter(Predicate p)
- flatMap(Function m)
- peek(Consumer c)
- skip(long n)
- limit(long n)
- sorted(Comparator r)
- boxed()

- parallel()
- sequential()

c Java 9

- takeWhile(Predicate p)
- dropWhile(Predicate p)

```
public static List<String> getNames(List<User> users) {
    return users.stream()  Stream<Examples.User>
            .map(user -> user.getName())  Stream<String>
            .filter(name -> name.startsWith("M"))
            .peek(System.out::println)
            .distinct()
            .sorted()
            .collect(Collectors.toList());
}
```

Terminal operations

Терминальные операции:

- collect(Collector c)
- reduce(BinaryOperator b)
- findFirst()
- findAny()
- forEach(Consumer c)
- allMatch(Predicate p)
- anyMatch(Predicate p)

anyMatch

```
boolean existVname = Stream.of("Misha", "Vasya", "Kolya")
        .anyMatch(name->name.startsWith("V")); // true
```

collect

```
List<String> names = Stream.of("Misha", "Vasya", "Kolya")
        .collect(Collectors.toList()); // Misha Vasya Kolya
```

count

```
long count = Stream.of("Misha", "Vasya", "Kolya")
        .count(); // 3
```

findFirst

```
Optional<String> firstName = Stream.of("Misha", "Vasya", "Kolya")
        .findFirst(); // Misha
```

Collectors

Основные коллекторы содержатся в утилитарном классе **Collectors**:

- toList()
- toSet()
- counting()
- toMap()
- groupingBy()
- joining()
- toCollection()
- averagingInt() averagingLong(), averagingLong()

toMap

```
Map<String, User> usersByAge = Stream.of(
        new User( name: "Misha",  sex: "MALE",  age: 13),
        new User( name: "Vasya",  sex: "MALE",  age: 13),
        new User( name: "Kolya",  sex: "MALE",  age: 20)
)
        .collect(Collectors.toMap(user -> user.getName(), user -> user));
```

groupingBy

```
Map<Integer, List<User>> usersByAge = Stream.of(
        new User( name: "Misha",  sex: "MALE",  age: 13),
        new User( name: "Vasya",  sex: "MALE",  age: 13),
        new User( name: "Kolya",  sex: "MALE",  age: 20)
)
        .collect(Collectors.groupingBy(user -> user.getAge()));
// 20 -> Kolya, 13 -> Misha, Vasya
```

Short circuit

Что будет выведено в stdout?

```
Optional<String> findVasya =
        Stream.of("Misha", "Vasya", "Kolya", "John")
                .peek(name -> System.out.println(name))
                .filter(name -> name.equals("Vasya"))
                .findFirst();
```

```
// Misha
// Vasya
```

Некоторые терминальные операции являются **короткозамкнутыми** (short circuit) что позволяет им не пробегаться по всему потоку, если результат уже известен.

Пример:

- anyMatch()
- findFirst()
- limit()

Stream API

Что будет выведено в stdout?

```
Stream<Integer> ints = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .filter(i -> i % 2 == 0)
        .peek(i -> System.out.println(i))
        .limit(2);

System.out.println("Before of after?");

List<Integer> intList = ints.collect(Collectors.toList());
```

```
Before of after?
2
4
```

Lazy evaluation

Стримы в Java “ленивы” т.е. оптимизированы таким образом что промежуточные операции выполняются только при вызове терминальной операции.

Обработка стрима не начнется пока не вызовется терминальная операция.

Parallel streams

Можно создавать параллельные потоки с помощью методов:

- parallelStream()
- parallel()

Перед использованием параллельных стримов подумайте а надо ли оно вам?

```
return Stream.of(
        new User( name: "Misha",  age: 29),
        new User( name: "Masha",  age: 44),
        new User( name: "Kolya",  age: 22)
)
        .parallel()
        .map(user -> findOrder(user))
        .collect(Collectors.toList());
```

Вопросы

Перерыв

Generics

Introduction

Что обсудим:

- Что такое дженерики и зачем они нужны
- Что такое wildcards, какие они бываю и как правильно использовать
- PECS
- Как устроены дженерики в Java
- Что такое Type erasure
- Type inference, bridge methods, super type token

Before Java 5

Как бы мы реализовывали свою коллекцию до Java 5?

```
public class CustomArray implements Iterable{

    private Object[] arr;

    public CustomArray(int size) {
        this.arr = new Object[size];
    }

    public void add(Object object){
        // add in array
    }

    public Object get(int index){
        // get from array
    }
```

```
CustomArray array = new CustomArray( size: 10);
array.add("Hello");
String value = (String) array.get(0);
```

Before Java 5

При необходимости реализации или использования классов-оберток/хранилищ (коллекций) мы сталкивались с проблемами:

- Необходимость явного приведения типа
- Возможность добавить объект некорректного типа

```
List ints = new ArrayList();
ints.add(123);
Integer value = (Integer) ints.get(0);
```

```
List ints = new ArrayList();
ints.add(123);
ints.add("Hello"); // no error
for (Object value : ints) {
    Integer integer = (Integer) value; //  ClassCastExeption
}
```

Before Java 5

Корректное использование Collection API до появления дженериков:

```
List ints = new ArrayList();
ints.add(123);
ints.add(1234);
for (Object value : ints) {
    if(value instanceof Integer) {
        Integer integer = (Integer) value;
        System.out.println(integer.intValue());
    }
}
```

Generic types introduction in Java 5

Согласно JSL: “**A *generic type* is a generic class or interface that is parameterized over types.**”

Создание дженерик типа:

```
class Store<T> {

    private T object;

    public Store(T object) {
        this.object = object;
    }

    public T get() {
        return this.object;
    }

    public void put(T object) {
        this.object = object;
    }
}
```

Использование:

```
Store<Integer> intStore = new Store<Integer>( object: 123);
Store<String> strStore = new Store<String>( object: "Hello");
Store<Object> objStore = new Store<Object>(new Object());


// since Java 7
Store<Integer> intStore = new Store<>( object: 123);
Store<String> strStore = new Store<>( object: "Hello");
Store<Object> objStore = new Store<>(new Object());
```

Generic types

Основные преимущества дженериков:

Типобезопасность во время компиляции:

```
ArrayList<Integer> integers = new ArrayList<Integer>();
integers.add(123);
integers.add("Hello"); // compile time error
```

Отсутствие необходимости приведения типа:

```
ArrayList<Integer> integers = new ArrayList<Integer>();
integers.add(123);
Integer value = integers.get(0);
```

Универсальность алгоритмов:

```
List<String> strings = new ArrayList<>();
List<Integer> ints = new ArrayList<>();
List<Number> numbers = new ArrayList<>();
```

Generic methods

Помимо типов, можно создавать дженерик методы:

```
public class CollectionUtils {

    public static <T> boolean contains(Collection<T> coll, T value) {
        return coll.stream()
                .anyMatch(v -> v.equals(value));
    }
}
```

```
List<Integer> ints = List.of(1, 2, 3, 4, 5);
List<String> strings = List.of("Foo", "Bar");

boolean containsSeven =
        CollectionUtils.<Integer>contains(ints, value: 7); // false
boolean containsFoo =
        CollectionUtils.<String>contains(strings, value: "Foo"); // true


// since Java 7
containsSeven = CollectionUtils.contains(ints, value: 7); // false
containsFoo = CollectionUtils.contains(strings, value: "Foo"); // true
```

Type inference

Java – статически типизированный язык, т.е. значение переменной должно быть известно на стадии компиляции

```
public static Integer sum(Integer first, Integer second) {
    return first + second;
}


Integer intSum = sum(123,124);

String strSum = sum(123,124); // Compile time error
```

Type inference

Type Inference – способность компилятора определять тип выражения из контекста

```
public static <T> T getAny(T first, T second){
    return first;
}

getAny(123,123); // ?

Integer value = getAny(123, 123);
```

```
getAny(123, 123.0); // ?

Number value = getAny(123, 123.0);
```

Type inference

Type Inference позволяет определить параметр типа и не указывать явно:

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

List<String> strList =
        Collections.<String>emptyList();
List<Integer> intList =
        Collections.<Integer>emptyList();
```

По ссылке компилятор определяет тип объекта:

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

List<String> strList = Collections.emptyList();
List<Integer> intList = Collections.emptyList();
```

Multiple parameter types

Можно указать более одного параметра для дженерик типа:

```
static class TwoValueStore<V,S> {

    private V first;
    private S second;

    public TwoValueStore(V first, S second) {
        this.first = first;
        this.second = second;
    }

    public V getFirst() { return this.first; }
    public S getSecond() { return this.second; }
}
```

```
TwoValueStore<String, Integer> store = new TwoValueStore<>("Hello", 123);
```

Naming conventions

Рекомендации по именованию параметров:

- E – элемент (при использовании Collection API)
- K – ключ
- V – значение
- N – номер
- T – тип
- S, U, V etc. – 2-й, 3-й, 4-й типы

Raw type

Raw types (или сырые типы) это дженерик тип, используемый без параметров типов

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

List list = ints;

list.add(123); // warning unchecked call
list.add("Hello"); // no error
```

Bounded type parameter

А что если есть необходимость ограничить параметр типа?

Выражение ***T extends Class*** задает ограничение для параметра типа дженерика:

```
class Store<T> {

    private T object;

    public Store(T object) {
        this.object = object;
    }

    public T get() {
        return this.object;
    }

    public void put(T object) {
        this.object = object;
    }
}
```

Параметр типа ограничен Number:

```
static class NumberStore<T extends Number> {

    private T number;

    public NumberStore(T number) {
        this.number = number;
    }

    public T get() { return this.number; }

    public void put(T number) { this.number = number; }

    public int intValue(){
        return this.number.intValue();
    }
}
```

```
NumberStore<Integer> iStore = new NumberStore<>( object: 123);
NumberStore<Double> dStore = new NumberStore<>( object: 123.0);
// Compile time error
NumberStore<String> sStore = new NumberStore<>( object: "Test");
```

Bounded type parameters

Аналогично ограничивать параметр типа можно и дженерик методе:

Параметр типа ограничен Number:

```
public static <T extends Number> int sum(List<T> list) {
    int sum = 0;
    for (T element : list) {
        sum += element.intValue();
    }
    return sum;
}
```

```
List<Integer> ints = List.of(1, 2, 3, 4, 5);
List<Number> numbs = List.of(1L, 2L, 3L, 4L, 5L);
List<String> strings = List.of("Foo", "Bar");

int intSum = sum(ints);
int numbSum = sum(numbs);
int strSum = sum(strings); // Compile time error
```

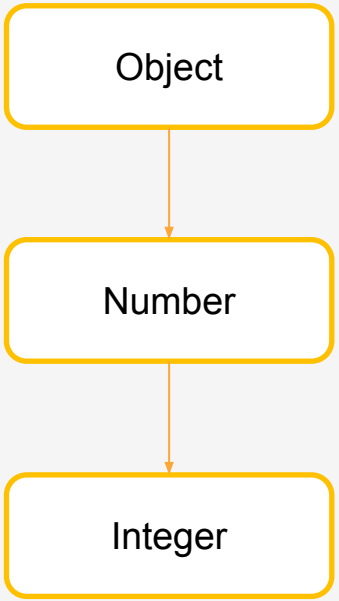
Multiple bounds

Можно указывать несколько ограничений для типа параметра.

```
class A{}
class B{}
interface C{}
interface D{}
```

```
class E<T extends A & C & D>{} // correct

class I<T extends A & B & C>{} // comptilation error

class K<T extends C & A & D>{} // comptilation error
```

Generics and inheritance




```
Integer integer = 123;
Number number = integer;
Object object = number;
```

Generics and inheritance

```
Integer intValue = 1234;
Number numberValue = intValue;
```

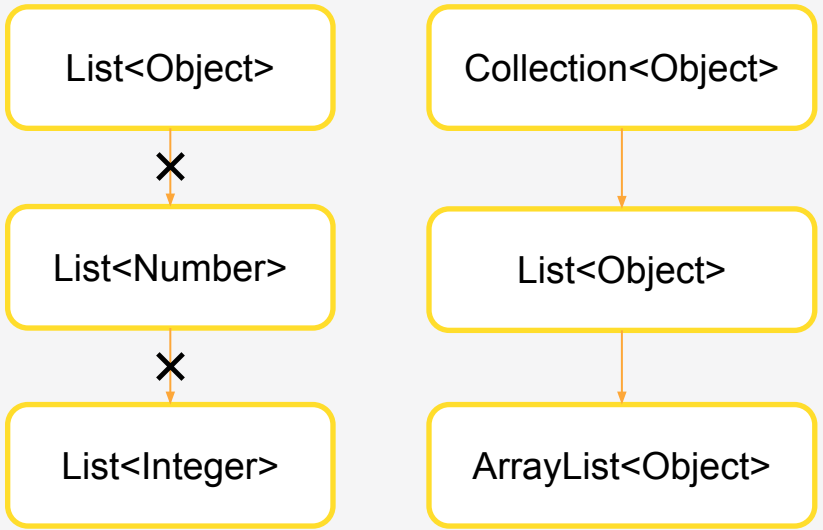
?

```
List<Integer> ints = List.of(1, 2, 3, 4, 5);
List<Number> numbs = ints; // ?
```

Generics and inheritance

Стандартный механизм наследования не применим к дженерикам из-за их **инвариантности**:

```
List<Integer> integers = new ArrayList<>();
List<Number> numbers = integers; // compile time error
List<Object> objects = integers; // compile time error
```



Covariance and Invariance

Ковариантность – это сохранение иерархии наследования исходных типов в производных в том же порядке.

Массивы ковариантны:

```
String[] strings = new String[10];
Object[] objects = strings; // no error
```

Что может пойти не так:

```
objects[0] = new Integer( value: 123); // Array store exception
// Runtime error!
```

Инвариантность – отсутствие наследования между производными типами

Дженерики инвариантны:

```
List<Integer> integers = new ArrayList<>();
List<Object> objects = integers; // compile time error
objects.add(123);
```

Wildcards

Wildcard в дженериках называется символ **?**, означающий неизвестный тип (unknown type).

Wildcards позволяют обходить ограничения инвариантности дженериков, добавляютс дополнительную типобезопасность, а так же способствуют написанию более гибкого кода.

Подразделяются на:

- Unbounded
- Upper bounded
- Lower bounded

Wildcards. Unbounded

Unbounded wildcard **?** означает любой или неограниченный тип

List<?> означает, что список может содержать любой объект:

```
List<?> objects;

objects = new ArrayList<Integer>();
objects = new ArrayList<String>();
objects = new ArrayList<Object>();
```

Используется когда нам не важен тип параметра:

```
public static void printAll(List<?> list) {
    for (Object value : list){
        System.out.println(value);
    }
}
printAll(ints);
printAll(strings);
```

Wildcards. Unbounded

```
public void someMethod(List<?> list){
    // what can I add to list ?

    // list.add(123);
    // list.add(new Object());
    // list.add("String");

}
```

Wildcards. Unbounded

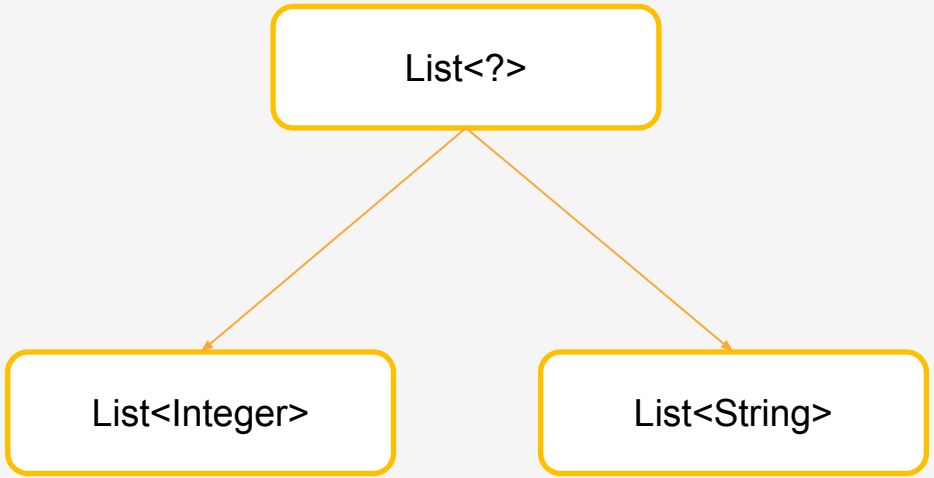
Нельзя поместить ни один объект в List<?> (кроме null):

```
List<?> objects = new ArrayList<>();

objects.add(new Object()); // compile time error
objects.add(123); // compile time error
objects.add("Hello"); // compile time error
objects.add(null); // ok
```

Возвращаемый тип - Object:

```
List<?> objects = Arrays.asList(123);
Object value = objects.get(0); // return type is Object
```

Wildcard добавляет ковариантность дженерикам



Upper Bounded Wildcards

Upper bounded wildcard добавляет границу «сверху» для типа параметра дженерика: список *List<? extends Number>* может содержать объекты Number или его наследников**:**

```
List<? extends Number> numbers;

numbers = new ArrayList<Number>();
numbers = new ArrayList<Integer>();
numbers = new ArrayList<String>(); // compile time error
```

```
public static int intSum(List<? extends Number> list) {
    int sum = 0;
    for (Number number : list) {
        sum += number.intValue();
    }
    return sum;
}
```

Upper Bounded Wildcards

```
public void someMethod(List<? extends Number> list) {
    // what can I add to list ?

    // list.add(123);
    // list.add(new Object());
    // list.add("String");

}
```

Upper Bounded Wildcards

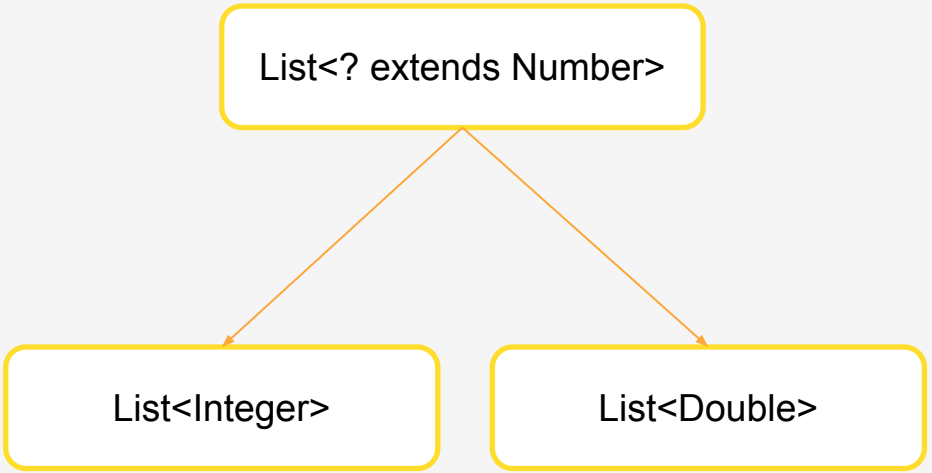
Нельзя поместить ни один объект в List<? extends Number> (кроме null):

```
List<? extends Number> numbers = new ArrayList<>();

numbers.add(new Object()); // compile time error
numbers.add(123); // compile time error
numbers.add(new Double( value: 2.1)); // compile time error
numbers.add(null); // ok
```

Возвращаемый тип - Number:

```
List<? extends Number> objects = Arrays.asList(123);
Number value = objects.get(0); // return something that extends Number
```

Wildcard с верхней границей так же ковариантны



Upper Bounded Wildcards

```
public class CustomArray<T> {

    private T[] objects;

    public void addAll(Collection<T> coll) {
        // add logic
    }
}
```

```
CustomArray<Number> numbers = new CustomArray<>();


List<Number> numbs = List.of(1, 2, 3);
numbers.addAll(numbs);


List<Integer> ints = List.of(4, 5, 6);
numbers.addAll(ints); // Compile time error
```

```
public class CustomArray<T> {

    private T[] objects;

    public void addAll(Collection<? extends T> coll) {
        // add logic
    }
}
```

Lower Bounded Wildcards

Lower bounded wildcard добавляет границу «снизу» для типа параметра дженерика: список *List<? super Number>* может содержать объекты Number или предков класса Number.

```
List<? super Number> objects = new ArrayList<>();

objects = new ArrayList<Number>();
objects = new ArrayList<Object>();
objects = new ArrayList<Integer>(); // compilation error
objects = new ArrayList<String>(); // compilation error
```

```
public static void addNumbers(List<? super Number> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

Lower Bounded Wildcards

```
public void someMethod(List<? super Number> list) {
    // what can I add to list ?

    // list.add(123);
    // list.add(new Object());
    // list.add("String");

}
```

Lower Bounded Wildcards

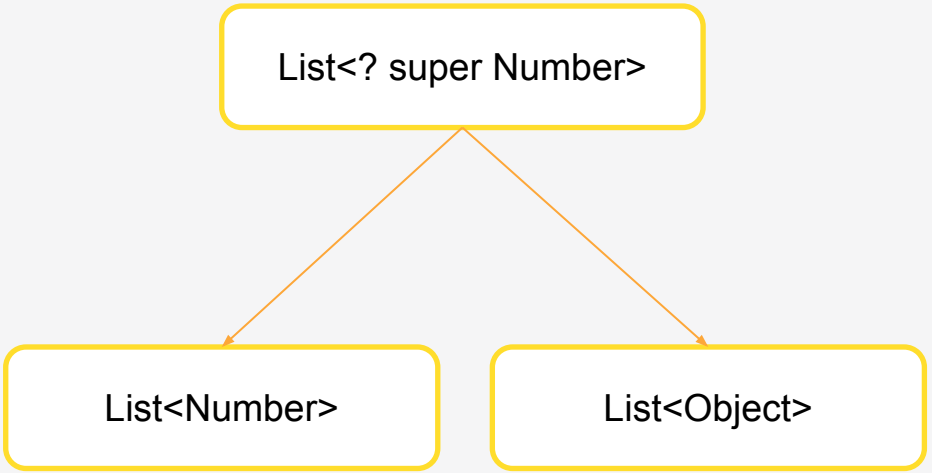
Положить можно все что extends Number (и null):

```
List<? super Number> objects = new ArrayList<>();

objects.add(new Integer( value: 123));
objects.add(new Double( value: 2.0));
objects.add(new Object());
objects.add("Hello");
objects.add(null);
```

Возвращаемый тип - Object:

```
List<? super Integer> integers = Arrays.asList(123);
Integer integer = integers.get(0); // compile time error
Object object = integers.get(0); // ok
```

Wildcard с нижней границей контравариантны



Upper Bounded Wildcards

```
public class CustomArray<T> {

    private T[] objects;

    public void copyTo(Collection<T> coll) {
        // copy logic
    }
}
```

```
CustomArray<Integer> ints = new CustomArray<>();

List<Integer> copy = new ArrayList<>();
ints.copyTo(copy);

List<Number> copyNumb = new ArrayList<>();
ints.copyTo(copyNumb); // Compile time error
```

```
public class CustomArray<T> {

    private T[] objects;

    public void copyTo(Collection<? super T> coll) {
        // copy logic
    }
}
```

PECS

Pecs (producer-extends, consumer-super) principle stands for:

- Если объявляем **wildcard extends**, то он является поставщиком данных и ничего не принимает
- Если объявляем **wildcard super**, то он является потребителем данных и сам ничего не поставляет

PECS

```
public static <T> void copy(List<? extends T> from, List<? super T> to) {
    for (T value : from) {
        to.add(value);
    }
}
```

Recursive type bounds

В редких случаях может быть полезно ограничивать тип параметра выражением, включающим его самого:

```
public static <T extends Comparable<T>> T getMax(List<T> list) {
    Iterator<T> i = list.iterator();
    T candidate = i.next();
    while (i.hasNext()) {
        T next = i.next();
        if (next.compareTo(candidate) > 0)
            candidate = next;
    }
    return candidate;
}
```

Generics under the hood

Во время компиляции стирается вся информация о типах параметров дженерика и становится недоступной в runtime.

Что видим коде:

```
public class Store<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Что делает компилятор:

```
public class Store {
    private Object value;

    public Object getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = value;
    }
}
```

Generics under the hood

Что видим коде:

```
public static <T> void print(List<T> list) {
    for (T value : list) {
        System.out.println(value);
    }
}
```

Что делает компилятор:

```
public static void print(List<Object> list) {
    for (Object value : list) {
        System.out.println(value);
    }
}
```

Что видим коде:

```
public static <T extends Number> int sum(List<T> list) {
    int sum = 0;
    for (T value : list) {
        sum += 0;
    }
    return sum;
}
```

Что делает компилятор:

```
public static int sum(List<Number> list) {
    int sum = 0;
    for (Number value : list) {
        sum += 0;
    }
    return sum;
}
```

Type erasure

Type erasure – стирание информации о типах-параметрах во время компиляции.
Реализуется через:

- Замену всех типов-параметров на Object или указанный ограничивающий тип
- Вставку явного преобразования типов (Class cast)
- Создание Bridge-методов

Type erasure. Type cast

Добавление явного преобразования типа

Что видим коде:

```
List<Integer> integers = new ArrayList<>();
integers.add(123);
integers.add(1234);
int sum = 0;
for (Integer value : integers) {
    sum += value;
}
```

Что делает компилятор:

```
List integers = new ArrayList();
integers.add(123);
integers.add(1234);
int sum = 0;
for (Object value : integers) {
    sum += (Integer) value; // Class cast
}
```

Type erasure. Bridge methods

Реализуем простой дженерик класс:

```
public class Store<T> {
    private T value;

    public void put(T value) {
        this.value = value;
    }
}

class IntStore extends Store<Integer> {

    @Override
    public void put(Integer value) {
        super.put(value);
    }
}
```

После стирания типов:

```
public class Store {
    private Object value;

    public void put(Object value) {
        this.value = value;
    }
}

class IntStore extends Store {

    @Override
    public void put(Integer value) {
        super.put(value);
    }
}
```

Type erasure. Bridge methods

Для сохранения полиморфизма компилятор создает синтетический Bridge метод:

Что видим коде:

```
public class Store<T> {
    private T value;

    public void put(T value) {
        this.value = value;
    }
}

class IntStore extends Store<Integer> {

    @Override
    public void put(Integer value) {
        super.put(value);
    }
}
```

Компилятор добавляем bridge method:

```
public class Store {
    private Object value;

    public void put(Object value) {
        this.value = value;
    }
}

class IntStore extends Store {

    public void put(Object value) {
        put((Integer) value);
    }

    public void put(Integer value) {
        super.put(value);
    }
}
```

Super type token

Действительно ли нельзя получить информацию о типе параметра дженерика в runtime?

Для примера рассмотрим **Jackson** – библиотека для работы с Json:

Jackson предоставляет методы для маппинга json в pojo:

```
ObjectMapper mapper = new ObjectMapper();
```

```
public <T> T readValue(String content, Class<T> valueType)
```

```
String jsonUser = "{\"name\" : \"Misha\", \"age\" : 32}";

User user = mapper.readValue(jsonUser, User.class);

String name = user.getName(); // Misha
Integer age = user.getAge(); // 13
```

Super type token

```
ObjectMapper mapper = new ObjectMapper();

String jsonUsers = "[" +
        "{\"name\" : \"Misha\", \"age\" : 29}," +
        "{\"name\" : \"Vasya\", \"age\" : 32}" +
        "]";

// Compile time error
User user = mapper.readValue(jsonUsers, List<User>.class);
```

Super type token

Super type token – механизм сохранения параметра типа при помощи анонимных классов и Reflection API.

```
public abstract class TypeReference<T> {

    private final Type type;

    public TypeReference() {
        Type superclass = getClass().getGenericSuperclass();
        type = ((ParameterizedType) superclass).getActualTypeArguments()[0];
    }

    public Type getType() {
        return type;
    }
}
```

Super type token

Тип параметра явно сохраняется в поле анонимного класса

```
ObjectMapper mapper = new ObjectMapper();

String jsonUsers = "[" +
        "{\"name\" : \"Misha\", \"age\" : 29}," +
        "{\"name\" : \"Vasya\", \"age\" : 32}" +
        "]";

List<User> users = mapper.readValue(jsonUsers, new TypeReference<List<User>>() {});

for (User user : users) {
    System.out.println(user);
}
```

Generic restrictions

Как нельзя использовать дженерики?

- Тип-параметр дженерика не может быть примитивом
- Нельзя создать объект типа-параметра (new V())
- Нельзя параметризировать статическое поле класса
- Нельзя использовать параметризированный тип с instance of
- Нельзя создавать массив параметризированных типов
- Нельзя использовать дженерики в исключениях
- Нельзя переопределять метод если после стирания типов он будет иметь такую же сигнатуру

Conclusion

Итог:

- Дженерики помогают писать обобщенный, универсальный код
- Дженерики приносят типобезопасность во время компиляции и удобство использования
- Wildcards снимают ограничения ковариантности, добавляю дополнительную гибкость
- При написании кода с использованием wildcards полагаемся на PECS
- Механизм Type Erasure стирает всю информацию о типах-параметрах во время компиляции
- Super type token позволяет получить информацию о типе в runtime

Вопросы

Homework

Спасибо за внимание!