Generics. Collections. Streams

Алексей Владыкин

```java
public class TreeNode {

    String value;

    TreeNode left;

    TreeNode right;

}
```

```java
public static BigDecimal minElement(
        BigDecimal[] values) {

    if (values.length == 0) {
        return null;
    }

    BigDecimal min = values[0];
    for (int i = 1; i < values.length; i++) {
        if (min.compareTo(values[i]) > 0) {
            min = values[i];
        }
    }
    return min;
}
```

```java
TreeNode rootNode = new TreeNode();

rootNode.value = "foobar";

// tree manipulation

String value = (String) rootNode.value;
```

```java
Object[] arrayOfBigDecimals = {...};

BigDecimal min = (BigDecimal)
        minElement(arrayOfBigDecimals);
```

```java
public static <T extends Comparable<T>> T
        minElement(T[] values) {

    if (values.length == 0) {
        return null;
    }

    T min = values[0];
    for (int i = 1; i < values.length; i++) {
        if (min.compareTo(values[i]) > 0) {
            min = values[i];
        }
    }
    return min;
}
```

```java
TreeNode<String> stringNode;

TreeNode<Integer> integerNode;

TreeNode<int[]> intArrayNode;

TreeNode<int> intNode;

TreeNode<10> tenNode;
```

```java
package java.util;

public final class Optional<T> {

    private final T value;

    private Optional(T value) {
        this.value = Objects.requireNonNull(value);
    }

    public static <T> Optional<T> of(T value) {
        return new Optional<>(value);
    }

    public T get() {
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
    }

    // ...
}
```

```java
String text = ???; // is null allowed?

@Nullable String nullableText = null;

@NonNull String nonNullText = "hello!";
```

```java
String text = "bar";

Optional<String> optionalText =
        Optional.of("baz");
```

```java
Optional<String> baz = Optional.of("baz");

baz.ifPresent(System.out::println);
```

```java
Optional<String> bar = Optional.empty();

String value = bar.orElse("bar");
```

```java
package java.util;

public final class Optional<T> {

    private final T value;

    private Optional(T value) {
        this.value = Objects.requireNonNull(value);
    }

    public static <T> Optional<T> of(T value) {
        return new Optional<>(value);
    }

    public T get() {
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
    }

    // ...
}
```

```java
Optional<String> foo =
        Optional.empty();

Optional<String> bar =
        Optional.of("bar");

Optional<String> baz =
        Optional.ofNullable("baz");

Optional<CharSequence> optionalCharSequence =
        Optional.<CharSequence>ofNullable("baz");

Optional<String> newOptional =
        new Optional<>("foobar");
```

```java
package java.util;

public final class Optional {

    private final Object value;

    private Optional(Object value) {
        this.value = Objects.requireNonNull(value);
    }

    public static Optional of(Object value) {
        return new Optional(value);
    }

    public Object get() {
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
    }

    // ...
}
```

```
Optional<String> optional = Optional.of("foo");

String value1 = optional.orElse("bar");

String value2 = optional.get();
```

```java
Optional optional = Optional.of("foo");

String value1 = (String) optional.orElse("bar");

String value2 = (String) optional.get();
```

```
T obj = new T();

T[] arr = new T[5];

if (obj instanceof T) {...}


T a = (T) b;
```

```java
import java.io.IOException;

public class Hack {

    public static void main(String[] args) {
        throwAsUnchecked(new IOException());
    }

    private static void throwAsUnchecked(Exception e) {
        Hack.<RuntimeException>genericThrow(e);
    }

    private static <T extends Throwable>
            void genericThrow(Exception e) throws T {
        throw (T) e;
    }
}
```

```
Number number = new Integer(1);
Number[] numberArray = new Integer[10];
```

```
Optional<Integer> optionalInt = Optional.of(1);
Optional<Number> optionalNumber = optionalInt;
optionalNumber.set(new BigDecimal("3.14"));
```

```java
package java.util;

public final class Optional<T> {

    private final T value;

    public void ifPresent(Consumer<T> consumer) {
        if (value != null)
            consumer.accept(value);
    }

    public T orElseGet(Supplier<T> other) {
        return value != null ? value : other.get();
    }

    // ...
}
```

```java
package java.util;

public final class Optional<T> {

    private final T value;

    public void ifPresent(Consumer<? super T> consumer) {
        if (value != null)
            consumer.accept(value);
    }

    public T orElseGet(Supplier<? extends T> other) {
        return value != null ? value : other.get();
    }

    // ...
}
```

```
Optional<?> optional = Optional.of(1);

Object value1 = optional.get();

Object value2 = optional.orElse(2);
```

```java
int[] oldArray = ...;
int oldLength = oldArray.length;


int newLength = oldLength + 10;
int[] newArray =
        Arrays.copyOf(oldArray, newLength);

newArray[oldLength] = newElement1;
newArray[oldLength + 1] = newElement2;
// ...
```

```java
final int[] array = new int[] {1, 2, 3};

array[0] = 10;
array[1] = 11;
array[2] = 12;
```

```java
package java.util;

public interface Collection<E>
        extends Iterable<E> {
    int size();

    boolean isEmpty();

    boolean contains(Object o);

    boolean add(E e);

    boolean remove(Object o);

    void clear();

    // ...
}
```

```java
Collection<ComplexNumber> c = ...;

c.add(new ComplexNumber(1, 2));

boolean contains =
        c.contains(new ComplexNumber(1, 2));
```

```java
Collection<Integer> collection = ...;


Iterator<Integer> it = collection.iterator();
while (it.hasNext()) {
    Integer element = it.next();
    System.out.println(element);
}


for (Integer element : collection) {
    System.out.println(element);
}
```

```java
Collection<Integer> collection = ...;

collection.forEach(System.out::println);
```

```
Collection<Integer> collection = ...;

for (Integer number : collection) {
    if (number > 5) {
        collection.remove(number);
    }
}


// java.util.ConcurrentModificationException
```

```java
package java.util;

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

    E get(int index);

    E set(int index, E element);

    void add(int index, E element);

    E remove(int index);

    int indexOf(Object o);

    int lastIndexOf(Object o);

    List<E> subList(int fromIndex, int toIndex);
}
```

```
List<String> words = ...;

words.subList(1, 3).clear();

int indexOfFoo =
    10 + words.subList(10, 15).indexOf("foo");
```

```java
List<String> list1 = new ArrayList<>();

List<Integer> list2 = new LinkedList<>();
```

```
package java.util;

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

    boolean add(E e);
    boolean offer(E e);

    E remove();
    E poll();

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

```java
package java.util;

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

    void addFirst(E e);
    void addLast(E e);

    boolean offerFirst(E e);
    boolean offerLast(E e);

    E removeFirst();
    E removeLast();

    // ...
}
```

```java
Deque<Object> deque1 = new ArrayDeque<>();


Deque<Integer> deque2 = new LinkedList<>();

deque2.offerLast(1);
deque2.offerLast(2);
deque2.offerLast(3);

Integer element;
while ((element = deque2.pollFirst()) != null) {
    System.out.println(element);
}
```

```java
package java.util;

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

    // ...

}
```

```
Set<ComplexNumber> numbers = new HashSet<>();

numbers.add(new ComplexNumber(3, 3));

numbers.remove(new ComplexNumber(3, 3));

// equals(), hashCode()
```

```java
Set<String> words1 = new HashSet<>();
words1.add(...);
```

```java
Set<String> words2 = new LinkedHashSet<>();
words2.add(...);
```

```java
package java.util;

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();
}
```

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

words.add("aaa");
words.add("bbb");
words.add("ccc");

words.headSet("bbb").clear();
```

```java
List<String> list = new ArrayList<>();
list.add("aaa");
list.add("aaa");
list.add("bbb");
list.add("aaa");

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

List<String> listWithoutDups =
        new ArrayList<>(set);
```

```java
package java.util;

public interface Map<K,V> {

    int size();
    boolean isEmpty();

    boolean containsKey(Object key);
    boolean containsValue(Object value);

    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();
}
```

```java
Map<A, B> map = new HashMap<>();

for (A key : map.keySet()) { ... }

for (B value : map.values()) { ... }

for (Map.Entry<A, B> entry : map.entrySet()) {
    System.out.printf("%s => %s\n",
        entry.getKey(), entry.getValue());
}

map.forEach((k, v) ->
        System.out.printf("%s => %s\n", k, v));
```

```java
Map<String, String> map1 = new HashMap<>();
map1.put("foo", "bar");
map1.put("bar", "baz");
map1.remove("bar");

SortedMap<String, String> map2 = new TreeMap<>();
map2.put("foo", "bar");
map2.put("bar", "baz");
map2.subMap("bar", "foo").clear();
```

Устаревшие классы

- `java.util.Vector`
- `java.util.Stack`
- `java.util.Dictionary`
- `java.util.Hashtable`

```
Collections.shuffle(list);

Collections.sort(list);
```

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

set.remove("abc");
// throws java.lang.UnsupportedOperationException
```

```
List<Integer> list = ...;

Object[] array1 = list.toArray();

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

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

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

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

```java
package java.util;

@FunctionalInterface
public interface Comparator<T> {

    int compare(T o1, T o2);

    // any number of default or static methods
}
```

```java
File directory = ...;

File[] javaSourceFiles = directory.listFiles(
    new FileFilter() {
        @Override
        public boolean accept(File file) {
            return file.getName().endsWith(".java");
        }
    });
```

```java
package java.io;

@FunctionalInterface
public interface FileFilter {

    boolean accept(File pathname);

}
```

```java
package java.util.function;

@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

}
```

```java
package java.util.function;

@FunctionalInterface
public interface Supplier<T> {

    T get();

}
```

```java
package java.util.function;

@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

}
```

```java
package java.util.function;

@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

}
```

```java
package java.util.function;

@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

}
```

```java
package java.util.function;

@FunctionalInterface
public interface UnaryOperator<T>
        extends Function<T, T> {

    // apply is inherited from Function
}
```

```java
class IntSquare implements IntUnaryOperator {

    @Override
    public int applyAsInt(int operand) {
        return operand * operand;
    }
}
```

```java
IntUnaryOperator square = x -> {
    return x * x;
};
```

```java
IntConsumer print = x -> System.out.print(x);
```

```java
IntUnaryOperator cube = x -> x * x * x;
```

```java
public class Demo {

    private int counter;

    public void foo() {
        IntUnaryOperator square =
                x -> x * x;

        IntSupplier sequence =
                () -> counter++;

        int bonus = 10;
        IntUnaryOperator bonusAdder =
                (x) -> x + bonus;
    }
}
```

```java
int[] counter = new int[] {0};

IntSupplier sequence = () -> counter[0]++;
```

```java
ToIntFunction<String> intParser =
        Integer::parseInt;

Consumer<Object> printer =
        System.out::println;

Function<Object, String> objectToString =
        Object::toString;


IntFunction<String[]> arrayAllocator =
        String[]::new;
```

```
IntPredicate isOdd = x -> x % 2 != 0;

IntPredicate isEven = isOdd.negate();
```

```
IntPredicate p1 = ..., p2 = ...;

IntPredicate p3 = p1.and(p2);
```

```java
Consumer<Object> printer =
        System.out::println;


List<Object> objects = new ArrayList<>();
Consumer<Object> collector = objects::add;


Consumer<Object> combinedConsumer =
        printer.andThen(collector);
```

```
DoubleUnaryOperator square = x -> x * x;
DoubleUnaryOperator sin = Math::sin;

DoubleUnaryOperator complexFunction1 =
        sin.andThen(square);

DoubleUnaryOperator complexFunction2 =
        sin.compose(square);
```

```
Comparator<Double> absoluteValueComparator =
        (a, b) -> Double.compare(
                Math.abs(a), Math.abs(b));
```

```
Comparator<Double> absoluteValueComparator2 =
        Comparator.comparing(
                Math::abs, Double::compare);
```

```java
package java.util.stream;

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


    // MANY methods


}
```

```java
int sum = IntStream.iterate(1, n -> n + 1)
    .filter(n -> n % 5 == 0 && n % 2 != 0)
    .limit(10)
    .map(n -> n * n)
    .sum();
```

```
Set<String> vocabulary = ...;
Stream<String> stream1 = vocabulary.stream();


BufferedReader reader = ...;
Stream<String> stream2 = reader.lines();


Path path = ...;
Stream<Path> stream3 = Files.list(path);
Stream<Path> stream4 = Files.walk(path);


IntStream chars = "hello".chars();
```

```
DoubleStream randomNumbers =
    DoubleStream.generate(Math::random);


IntStream integers =
    IntStream.iterate(0, n -> n + 1);


IntStream smallIntegers =
    IntStream.range(0, 100);


IntStream smallIntegers2 =
    IntStream.rangeClosed(0, 100);
```

```java
IntStream combinedStream =
    IntStream.concat(stream1, stream2);
```

```java
IntStream empty = IntStream.empty();
```

```java
double[] array = ...;
DoubleStream streamFromArray =
    Arrays.stream(array);
```

```java
IntStream streamOfElements =
    IntStream.of(2, 4, 6, 8, 10);
```

```
IntStream stream = ...;

stream.filter(n -> n > 100)

    .mapToObj(Integer::toString)

    .flatMapToInt(s -> s.chars())

    .distinct()

    .sorted()

    .skip(3)

    .limit(2);
```

```java
IntStream stream1 = ...;
stream1.forEach(System.out::println);


IntStream stream2 = ...;
OptionalInt result = stream2.findFirst();


Stream<String> stream3 = ...;
boolean allStringsAreAtLeast10Chars =
    stream3.allMatch(s -> s.length() > 10);
```

```
Stream<String> stream1 = ...;
Optional<String> minString = stream1.min(
    Comparator.comparing(
        String::length, Integer::compare));
```

```
IntStream stream2 = ...;
int count = stream2.count();
```

```
IntStream stream3 = ...;
int sum = stream3.sum();
```

```
Stream<String> stream1 = ...;
List<String> list =
    stream1.collect(Collectors.toList());
```

```
Stream<BigInteger> bigInts = ...;
BigInteger sum = bigInts.reduce(
    BigInteger.ZERO, BigInteger::add);
```

```java
public static BigInteger factorial(int n) {
    return IntStream.rangeClosed(1, n)
        .mapToObj(i -> BigInteger.valueOf(i))
        .reduce(BigInteger.ONE, BigInteger::multiply);
}
```

```java
public static boolean isPalindrome(String s) {

    StringBuilder leftToRight = new StringBuilder();

    s.chars().filter(Character::isLetterOrDigit)
        .map(Character::toLowerCase)
        .forEach(leftToRight::appendCodePoint);

    StringBuilder rightToLeft =
        new StringBuilder(leftToRight).reverse();

    return leftToRight.toString()
        .equals(rightToLeft.toString());

}
```