



Л-выражения. Java Stream API

Croc Java School



First-class citizen

К сущностям первого порядка в языке применимы базовые операции языка:

- передача в функцию в виде аргумента
- использование в качестве возвращаемого значения функции
- присваивание переменным
- изменение



Object - First-class citizen Java

Java - объектно-ориентированный язык.

При написании программ мы оперируем объектами: описываем их структуру в виде классов, передаем экземпляры в методы, получаем результат в виде объектов.

То есть, операции первого порядка мы применяем к данным (объектам). И в языке нет инструментов применить их к поведению (функциям).



Function - не First-class citizen Java (до Java 8*)

- Функцию не можем передать как аргумент в метод
- Функцию не можем вернуть как результат метода
- Функцию не можем присвоить переменной

** После Java 8 тоже с натяжкой*



Пример. Создание потока

Но уже с ранних версий в языке используются конструкции, которые подразумевают использования функций наравне с объектами.

```
int[] values = new int[] {1, 2, 3, 4, 5};  
Thread t = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        int sum = 0;  
        for (int value : values)  
            sum += value;  
    }  
});
```




В этом примере в конструктор класса Thread передается функция - желаемое поведение, которое должно выполняться в отдельном потоке.

```
int sum = 0;
for (int value : values)
    sum += value;
```



Еще пример. Сортировка

```
public class Product {  
  
    private final String name;  
    private final BigDecimal price;  
  
    Product(String name, BigDecimal price) {  
        this.name = name;  
        this.price = price;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public BigDecimal getPrice() {  
        return price;  
    }  
}
```



```
List<Product> products = Arrays.asList(  
    new Product("Tesla Model S", new BigDecimal("8561311.00")),  
    new Product("Сырок Б.Ю. Александров", new BigDecimal("29.90"))  
);  
  
products.sort(new Comparator<>(){  
    @Override  
    public int compare(Product p1, Product p2) {  
        return p1.getPrice().compareTo(p2.getPrice());  
    }  
});
```




В метод сортировки передается функция сравнения двух цен.

```
p1.getPrice().compareTo(p2.getPrice())
```

**Приходится писать много
лишнего кода**



Лямбда-выражения лаконичнее

До

```
products.sort(new Comparator<>(){  
    @Override  
    public int compare(Product p1, Product p2) {  
        return p1.getPrice().compareTo(p2.getPrice());  
    }  
});
```

После

```
products.sort((p1, p2) -> p1.getPrice().compareTo(p2.getPrice()));
```



Лямбда-выражение

Синтаксис для объявления функционального-объекта (функции) по месту использования, допускающий замыкание на лексический контекст.

Лямбда-оператор: `->` (переход).

Левая часть оператора определяет параметры, правая — само лямбда-выражение, определяющее анонимный метод.

Простейший лямбда-оператор:

`x -> x % 2`



Форма лямбда-выражений

В общем случае состоит из блока инструкций, заключенных в фигурные скобки и содержащий оператор возврата значения (`return`).

```
x -> { return x % 2; }
```

Фигурные скобки могут быть опущены в случае, если выражение в блоке единственное. Оператор `return` в таком случае тоже опускается.

```
x -> x % 2
```



Параметры лямбда-оператора

Лямбда-оператор допускает использование нескольких параметров. В таком случае они должны быть заключены в скобки.

```
(x, y) -> Integer.compare(x, y)
```

Если параметры лямбда-оператора совпадают с сигнатурой функции лямбда-выражения, он может быть заменен **ссылкой на метод** (method reference).

```
(x, y) -> Integer::compare
```



Method reference

Ссылка на метод экземпляра

```
word::contains // (s) -> word.contains(s)
```

Ссылка на метод экземпляра через название класса

```
String::contains // (word, s) -> word.contains(s)
```

Ссылка на статический метод класса

```
Math::max // (x, y) -> Math.max(x, y)
```

Ссылка на конструктор

```
Product::new // (name, price) -> new Product(name, price)
```



Замыкание (closure)

Функция, которая ссылается на переменные, определенные **вне тела** этой функции и не являющиеся ее параметрами.

Ссылки на “внешние” переменные действительны внутри замыкания, даже если переменная **вышла из области видимости**.

Замыкание связывает тело функции с ее **лексическим окружением** (лексическим контекстом) — местом, где функция определена.

```
int mod = 2;  
x -> x % mod
```

Effectively final



Effectively final

Если переменная (примитив или ссылка) объявлена с модификатором `final`, ее значение не может быть изменено после инициализации.

Если переменная объявлена без модификатора `final`, но ее значение не изменяется после инициализации, то такая переменная считается `effectively final`.




Примеры

```
// final  
final int year = 2021;
```

```
// effectively final, значение не меняется  
int month = 12;
```

```
// не effectively final, значение меняется  
int day = 1;  
day = 2;
```

```
// effectively final, значение не меняется  
List<String> friends = Arrays.asList("Даша", "Тима");  
friends.remove("Тима");
```



Final и effectively final переменные можно использовать в методах анонимных классов и лямбда выражениях через замыкания. В противном случае использовать переменную в лямбда выражении нельзя.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
int mod = 2;  
mod += 3;  
numbers.removeIf(x -> x % mod == 0);
```

Error: local variables referenced from a lambda expression must be final or effectively final




Корректно ли такое использование?

```
int[] values = new int[] {1, 2, 3, 4, 5};
```

```
int sum = 0; // effectively final
```

```
Thread t = new Thread(() -> {  
    for (int value : values)  
        sum += value;  
});
```



Нет, так как переменные в замыканиях effectively final изменять их значения в лямбда-выражениях нельзя.

```
int[] values = new int[] {1, 2, 3, 4, 5};
```

```
int sum = 0; // effectively final
```

```
Thread t = new Thread(() -> {  
    for (int value : values)  
        sum += value;  
});
```

Error: local variables referenced from a lambda expression must be final or effectively final

Лямбды упрощают многие
интерфейсы



Collection.removeIf(Predicate<E>)

```
Set<String> words = new HashSet<>();  
words.removeIf(word -> word.contains("a"));
```

Вместо

```
for (Iterator<String> it = words.iterator(); it.hasNext();) {  
    String word = it.next();  
    if (word.contains("a"))  
        it.remove();  
}
```




Iterable.forEach(Consumer<T>)

```
Set<String> words = new HashSet<>();  
words.forEach(System.out::println);
```

Вместо

```
for (String word : words) {  
    System.out.println(word);  
}
```



Map.computeIfAbsent(K, Function<K, V>)

```
Map<Integer, List<String>> wordsByLength = new HashMap<>();  
String word = "lambda";  
wordsByLength  
    .computeIfAbsent(word.length(), k -> new ArrayList<>())  
    .add(word);
```

Вместо

```
List<String> words = wordsByLength.get(word.length());  
if (words == null) {  
    words = new ArrayList<>();  
    wordsByLength.put(word.length(), words);  
}  
words.add(word);
```



А еще

`Map.forEach(BiConsumer<K, V>)`

`Map.compute(K, BiFunction<K, V>)`

`Map.merge(K, V, BiFunction<K, V>)`

**Java 8: Как вывести функции
на уровень first-class citizen,
но не сломать обратную
совместимость?**

—

@FunctionalInterface



Функциональный интерфейс

Функциональный интерфейс — это интерфейс, определяющий один единственный абстрактный метод.

Например, стандартный интерфейс `Runnable` является функциональным, так как определяет только метод `run()`.

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```



Экземпляры функциональных интерфейсов

Экземпляры функциональных интерфейсов могут быть созданы с помощью:

- лямбда-выражений
- ссылок на методы
- ссылок на конструкторы



Экземпляр Runnable как лямбда-выражение

```
new Thread(() -> {  
    // some task  
}).start();
```

Что на самом деле происходит:

```
Runnable r = new Runnable() {  
    @Override  
    public void run() {  
        // some task  
    }  
};  
new Thread(r).start();
```





Функциональные интерфейсы как функции первого порядка

```
Runnable r = () -> {  
    // some task  
};
```

С одной стороны `r` имеет тип `Runnable` и является объектом анонимного класса. Но с другой стороны этот объект интерпретируется как функция.

В итоге в Java нет функциональных типов.

Какое-то читерство



Да, формально, это не функции первого порядка, а все еще классы. Но в применении большой разницы нет, а обратная совместимость сохранена. Все довольны.

Стандартные функциональные интерфейсы



java.util.function.Function<T, R>

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Функция подсчета букв в строке

```
Function<String, Integer> numLetters = str -> {
    int n = 0;
    for (char c : str.toCharArray())
        if (Character.isLetter(c))
            n++;
    return n;
};
```



Композиция функций

```
Function<T, R> f;  
f.compose(Function<V, T> before) // R f(before(V x))
```

```
Function<T, R> f;  
f.andThen(Function<R, V> after) // V after(f(T x))
```



Что делает функция f?

```
Function<Integer, Integer> f  
    = ((Function<String, Integer>)String::length)  
        .compose(Object::toString);
```



Что делает функция f?

```
Function<Integer, Integer> f  
  = ((Function<String, Integer>)String::length)  
    .compose(Object::toString);
```

Возвращает количество символов в строковом представлении числа

```
f.apply(2021) => 4
```




java.util.function.Predicate<T>

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Предикат, проверяющий наличие в строке пробелов

```
Predicate<String> hasSpaces = str -> str.contains(" ");
```



java.util.function.Supplier<T>

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Генератор случайного месяца

```
Random rnd = new Random(System.currentTimeMillis());
Supplier<Integer> randomMonth = () -> rnd.nextInt(12) + 1;
```



java.util.function.Consumer<T>

```
@FunctionalInterface  
public interface Consumer<T> {  
    void accept(T t);  
}
```

Функция-консьюмер, печатающая числа на экран

```
Consumer<Integer> logger = System.out::println;
```

Пользовательские функциональные интерфейсы



Определение функционального интерфейса

```
@FunctionalInterface
public interface UnaryOperator<T, R> {

    R apply(T operand);
}
```

Абстрактный метод должен быть только один.

Аннотацию `@FunctionalInterface` можно не указывать.



```
UnaryOperator<Integer, Integer> minus = x -> -x;
```

Если лямбда-выражение совпадает по типам параметров и возвращаемого значения с функциональным методом, то это выражение может быть присвоено переменной типа функционального интерфейса.



FunctionalInterface + void

```
@FunctionalInterface
public interface VoidOperator<T> {

    void apply(T operand);
}
```

```
VoidOperator<Integer> abs = Math::abs;
```

У функционального интерфейса нет возвращаемого значения, но `Math.abs(int)` возвращает `int`. Несмотря на разницу в типах, такое использование допустимо.

Java Stream API



Stream API

Работа с коллекциями в функциональном стиле.

Основной класс — Stream.

Создание стрима из коллекции

```
public interface Collection<E> extends Iterable<E> {  
    default Stream<E> stream() {  
        return StreamSupport.stream(spliterator(), false);  
    }  
    default Stream<E> parallelStream() {  
        return StreamSupport.stream(spliterator(), true);  
    }  
}
```



Создание стримов

Стрим можно создать

- на основе данных: коллекции, массива, строк файла, символов строки;
- синтетический: определить условие его генерации.



Создание стримов (опций много)

```
Stream<Integer> stream = Stream.empty();
```

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
```

```
Integer[] array = new Integer[] {1, 2, 3, 4, 5};  
Stream<Integer> stream = Arrays.stream(array);
```

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);  
Stream<Integer> stream = list.stream();
```

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5);
```

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1).limit(5);
```



Конвейерные и терминальные методы

Все методы стримов можно разделить на две группы:

1. конвейерные
2. терминальные

Конвейерные методы возвращают модифицированный стрим, терминальные — завершают обработку и возвращают итоговый результат.

Терминальные методы



forEach

Применяет лямбда-выражение ко всем элементам стрима.

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1)  
    .limit(10);
```

```
stream.forEach(System.out::print);
```

12345678910



collect

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1)  
    .limit(10);
```

```
List<Integer> numbers = stream.collect(Collectors.toList());
```

numbers:

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



Collectors.joining

```
Stream<String> stream = Stream.of("a", "b", "c");  
String str = stream.collect(Collectors.joining(", "));
```

str:

a, b, c



min/max/count

Ищет максимальный элемент в соответствии с заданным компаратором.

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1)
    .limit(10);
Optional<Integer> max = stream.max(Integer::compareTo);
```

max:

Optional[10]



reduce

Применяет функцию-аккумулятор.

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1)
    .limit(10);
Optional<Integer> sum = stream.reduce((n1, n2) -> n1 + n2);
```

```
sum:
Optional[55]
```

Optional<T>



Optional<T>

Безопасная замена для null.

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5, 6, 1);
Optional<Integer> first = numbers.stream()
    .filter(n -> n > 100)
    .findFirst();
if (!first.isPresent()) {
    System.out.println("Not found");
} else {
    System.out.println(first.get());
}
```

Конвейерные методы



filter

Выбирает элементы, удовлетворяющие условию предиката.

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1)
    .limit(10);
```

```
stream.filter(n -> n % 2 == 0);
```

```
[2, 4, 6, 8, 10]
```



map

Преобразует элементы стрима, возможно, приводит к другому типу.

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1)
    .limit(10);
```

```
stream.map(n -> String.valueOf(n).length());
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```



flatMap

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1)
    .limit(5);
Stream<Integer> flatStream = stream
    .flatMap(n -> Stream.iterate(n, k -> k).limit(n));
```

stream:

```
[1, 2, 3, 4, 5]
```

flatStream:

```
[1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]
```




distinct

Исключает неуникальные элементы.

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1)
    .limit(10);
```

```
stream
    .map(n -> String.valueOf(n).length());
    .distinct();
```

[1, 2]

Схема использования



Типовая работа со стримами

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5, 6, 1);  
List<Character> filtered = numbers.stream()  
    .filter(n -> n % 2 == 0)  
    .map(n -> (char)('a' + n))  
    .collect(Collectors.toList());
```

1. создаем стрим
2. выполняем цепочку преобразований
3. выполняем терминальный оператор