MTC True Tech

Functional Interfaces, Optional, Stream API

Калайчьян Сурен
Backend developer

Functional Interfaces

Лямбда

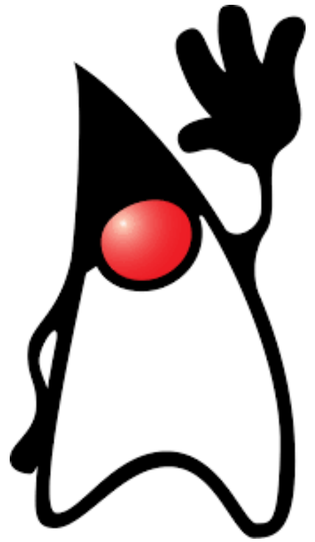
Лямбда-выражение в программировании — специальный синтаксис для определения функциональных объектов, заимствованный из λ-исчисления.

- → Применяется для объявления анонимных функций по месту их использования.
- → Обычно допускает замыкание на лексический контекст, в котором это выражение использовано.
- → Используя лямбда-выражения, можно объявлять функции в любом месте кода.

Лямбда

Лямбда-выражение в Java — это анонимная функция, позволяющая передавать блок кода как аргумент или сохранять его в переменной.

Она представляет собой упрощенную запись анонимного класса и используется для работы с **функциональными интерфейсами**, что делает код более лаконичным и читаемым, особенно при работе с коллекциями, потоками и событиями.



Лямбда

```
() -> {}
```

```
(a, b) -> {return a + b}
```

Лямбда

```
() -> {}
```

```
(a, b) ->                a + b
```

Лямбда

```
() -> {}
```

```
(a, b) -> Integer::sum
```

Лямбда vs Method Reference

Практически тоже самое но имеет одно основное отличие:

→ Method reference хранит состояние, которое было на момент его вызова

Functional interface

```java
@FunctionalInterface
public interface Runnable {
    /**
     * Runs this operation.
     */
    void run();
}
```

Functional interface

```
Runnable anonymousRunnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello Matan");
    }
};
```

Functional interface

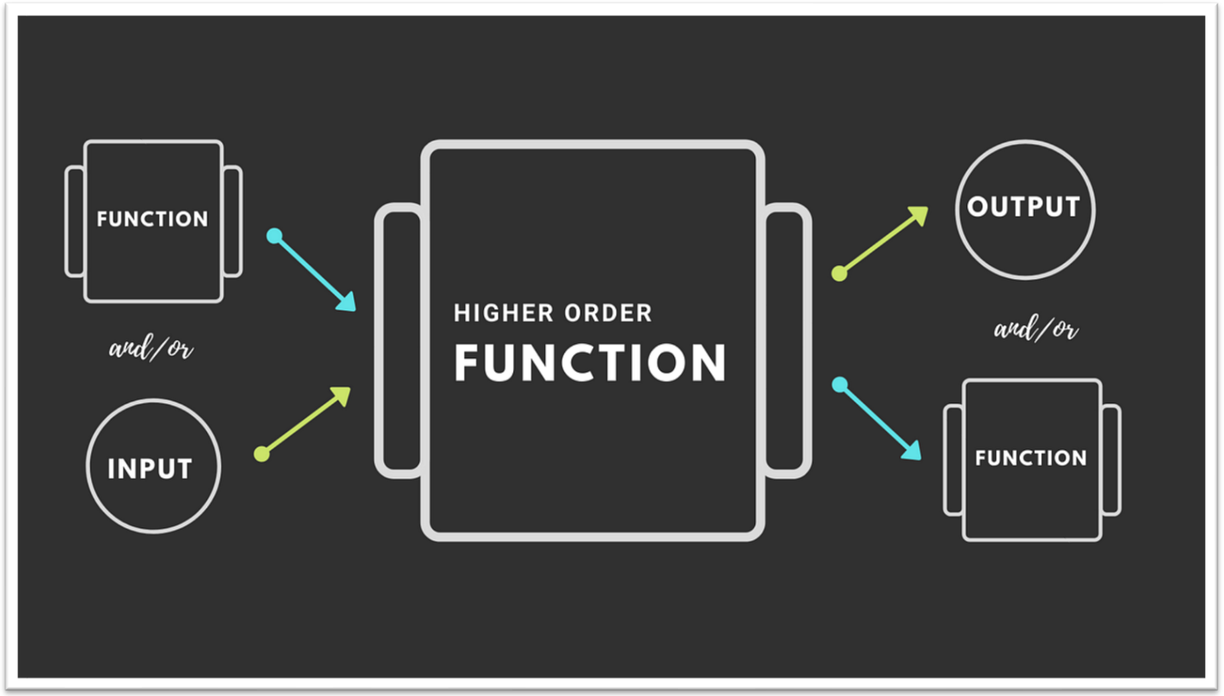
```
Runnable anonymousRunnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello Matan");
    }
};
```

```
Runnable lambdaRunnable = () -> System.out.println("Hello Matan");
```

Функция высшего порядка

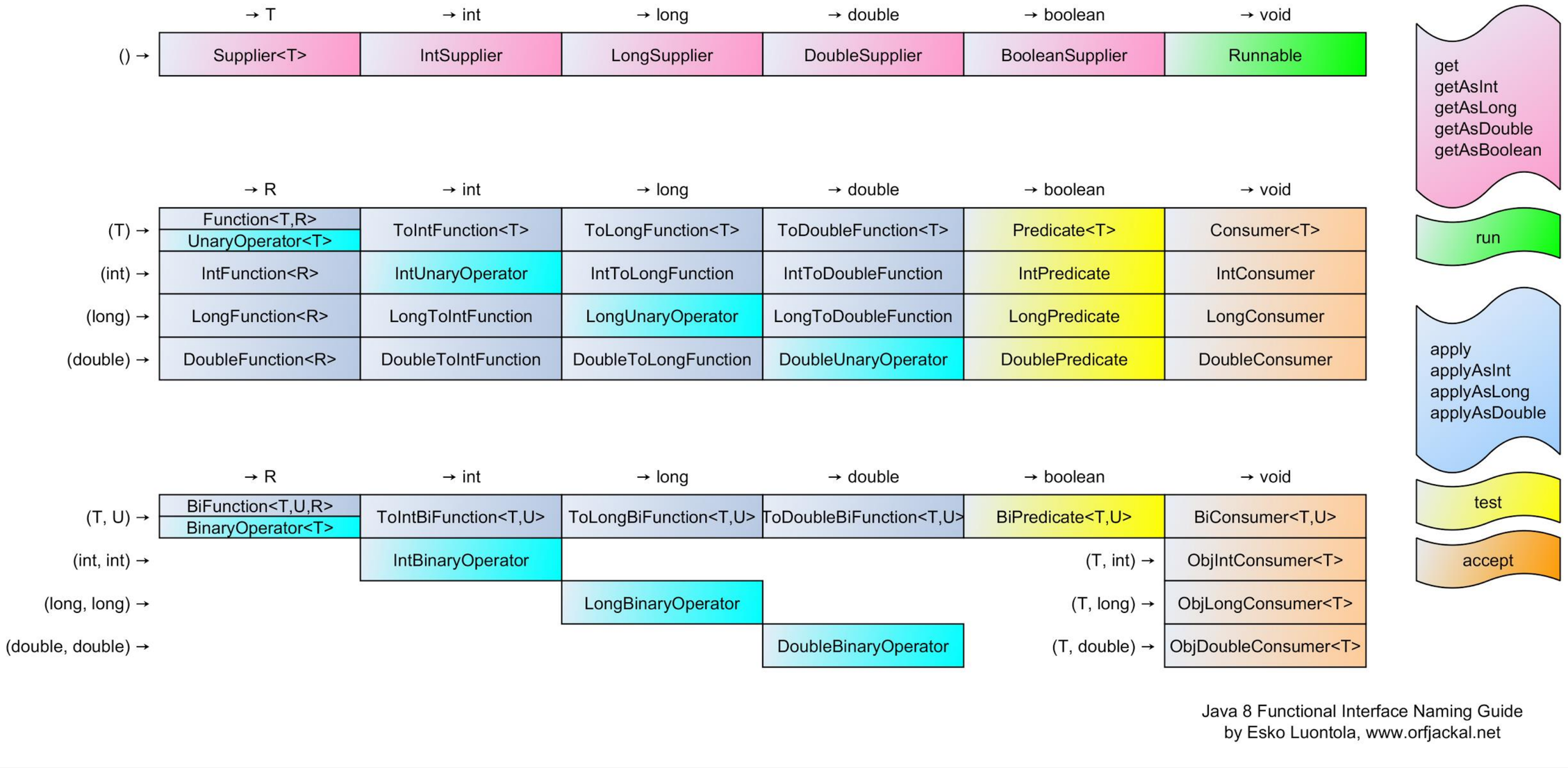
Функция высшего порядка (ФВП) — это функция, которая может принимать другие функции в качестве аргументов или возвращать функцию в качестве результата.

Это позволяет писать более абстрактный, модульный и переиспользуемый код, делегируя часть логики другим функциям, которые передаются в качестве параметров.



Functional interface

Function Type	Method Signature	Input parameters	Returns	When to use?
Predicate<T>	boolean test(T t)	one	boolean	Use in conditional statements
Function<T, R>	R apply(T t)	one	Any type	Use when to perform some operation & get some result
Consumer<T>	void accept(T t)	one	Nothing	Use when nothing is to be returned
Supplier<R>	R get()	None	Any type	Use when something is to be returned without passing any input
BiPredicate<T, U>	boolean test(T t, U u)	two	boolean	Use when Predicate needs two input parameters
BiFunction<T, U, R>	R apply(T t, U u)	two	Any type	Use when Function needs two input parameters
BiConsumer<T, U>	void accept(T t, U u)	two	Nothing	Use when Consumer needs two input parameters
UnaryOperator<T>	public T apply(T t)	one	Any Type	Use this when input type & return type are same instead of Function<T, R>
BinaryOperator<T>	public T apply(T t, T t)	two	Any Type	Use this when both input types & return type are same instead of BiFunction<T, U, R>



Java 8 Functional Interface Naming Guide
by Esko Luontola, www.orfjackal.net

Function interface

```
Predicate<String> universityChecker = (a) -> a.equals("MIPT");
universityChecker.test("HSE");
```

Functional interface

```
Predicate<String> universityChecker = (a) -> a.equals("MIPT");
universityChecker.test("HSE"); // false
```

Functional interface

```
Supplier<String> micronayshnikSupplier = () -> "10 po matany";
micronayshnikSupplier.get();
```

Functional interface

```
Supplier<String> micronayshnikSupplier = () -> "10 po matany";
micronayshnikSupplier.get(); // "10 po matany"
```

Functional interface

```
Consumer<String> micronayshnikConsumer = a -> System.out.println(a);
micronayshnikConsumer.accept("Spalili - poidu na peresdachy");
```

Functional interface

```
Consumer<String> micronayshnikConsumer = a -> System.out.println(a);
micronayshnikConsumer.accept("Spalili - poidu na peresdachy"); // void
```

Functional interface

```
Function<Integer, Integer> increment = a -> a + 1;
increment.apply(1);
```

Functional interface

```
Function<Integer, Integer> increment = a -> a + 1;
increment.apply(1); // 2
```

Functional interface

Function Type	Method Signature	Input parameters	Returns	When to use?
Predicate<T>	boolean test(T t)	one	boolean	Use in conditional statements
Function<T, R>	R apply(T t)	one	Any type	Use when to perform some operation & get some result
Consumer<T>	void accept(T t)	one	Nothing	Use when nothing is to be returned
Supplier<R>	R get()	None	Any type	Use when something is to be returned without passing any input
BiPredicate<T, U>	boolean test(T t, U u)	two	boolean	Use when Predicate needs two input parameters
BiFunction<T, U, R>	R apply(T t, U u)	two	Any type	Use when Function needs two input parameters
BiConsumer<T, U>	void accept(T t, U u)	two	Nothing	Use when Consumer needs two input parameters
UnaryOperator<T>	public T apply(T t)	one	Any Type	Use this when input type & return type are same instead of Function<T, R>
BinaryOperator<T>	public T apply(T t, T t)	two	Any Type	Use this when both input types & return type are same instead of BiFunction<T, U, R>

Effectively final

Effectively final— означает, что переменная явно не задана как final, но она в заданном блоке кода не изменяется.

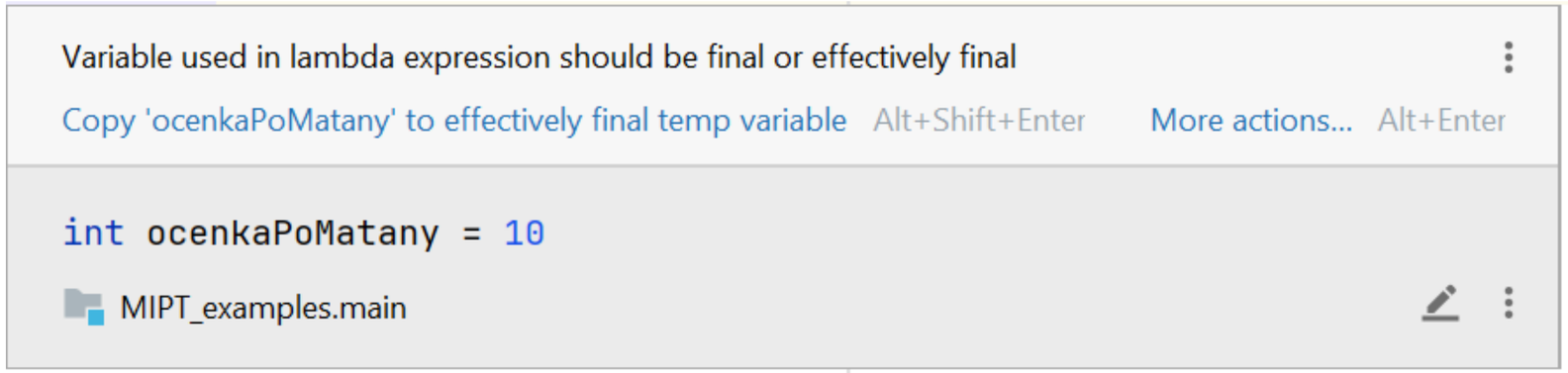
- → Если изменяемая переменная является локальной и не effectively final, то будет ошибка компиляции.
- → Если изменяемая переменная **НЕ** является локальной (достается из класса), то ошибки не будет

Effectively final

```
int ocenkaPoMatany = 10;

Supplier<String> telegramPublicSupplier =
        () -> "Send message to telegram: " + "Y menya po matany " + ocenkaPoMatany;
telegramPublicSupplier.get(); // "10 po matany"

Supplier<Integer> zloiPrepodSupplier = () -> 0;
ocenkaPoMatany = zloiPrepodSupplier.get();
```

Effectively final

```
int ocenkaPoMatany = 10;

Supplier<String> telegramPublicSupplier =
        () -> "Send message to telegram: " + "Y menya po matany " + ocenkaPoMatany;
telegramPublicSupplier.get(); // "10 po matany"

Supplier<Integer> zloiPrepodSupplier = () -> 0;
ocenkaPoMatany = zloiPrepodSupplier.get();
```



Effectively final

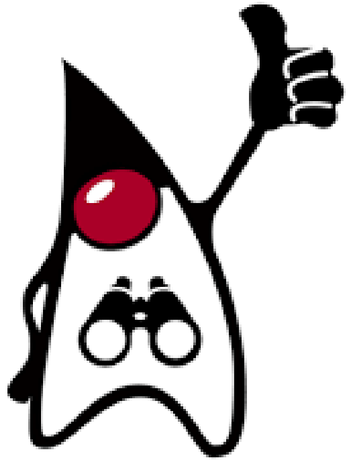
```
public int ocenkaPoMatany = 10;

public void doExam() {
    Supplier<String> telegramPublicSupplier =
            () -> "Send message to telegram: " + "Y menya po matany " + ocenkaPoMatany;
    telegramPublicSupplier.get(); // "10 po matany"

    Supplier<Integer> zloiPrepodSupplier = () -> 0;
    ocenkaPoMatany = zloiPrepodSupplier.get();
}
```

Optional

Optional

Optional – обертка для работы над объектами которые могут быть в потенциале как null

- → Не получили должной популярности как хотелось бы (если сравнивать с NullSafety в Kotlin)
- → Встречается на проектах где есть на это договорённости в команде
- → Можно встретить часто при обращении к бд (в слое repository) или в специфичных случаях



Значение null

```
Object object = new Object();
object.toString();
```

Значение null

```
Object object = new Object();
object.toString(); // java.lang.Object@8f73c56d
```

Значение null

```
Object object = new Object();
object.toString(); // java.lang.Object@8f73c56d
```

```
Object nullObject = null;
object.toString();
```

Значение null

```
Object object = new Object();
object.toString(); // java.lang.Object@8f73c56d
```

```
Object nullObject = null;
object.toString(); // java.lang.NullPointerException
```

Optional

```
String nullObject = null;

Optional<String> opt = Optional.ofNullable(nullObject);

opt.isPresent();

opt.isEmpty();

opt.get();
```

Optional

```
String nullObject = null;

Optional<String> opt = Optional.ofNullable(nullObject);

opt.isPresent();

opt.isEmpty();

opt.get();
```

```
opt.ifPresentOrElse(
        (s -> System.out.println(s)),
        () -> System.out.println("Null object allo")
);

Optional<String> optModifiedString = opt
        .filter((s) -> !s.isBlank())
        .map(s -> s + "!!!");
```

Optional

```
String nullObject = "null";

Optional<String> opt = Optional.ofNullable(nullObject);

opt.isPresent();

opt.isEmpty();

opt.get();
```

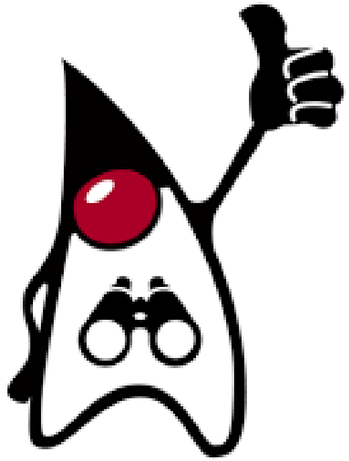
```
opt.ifPresentOrElse(
        (s -> System.out.println(s)),
        () -> System.out.println("Null object allo")
);

Optional<String> optModifiedString = opt
        .filter((s) -> !s.isBlank())
        .map(s -> s + "!!!");
```

Stream API

Stream API

Stream API – это API для конвейерной обработки данных в функциональном стиле

- → Имеет огромное количество методов для обработки
- → Может сэкономить довольно много времени и нервов в отличии от классических: if, for, while, break, etc..



До стримов

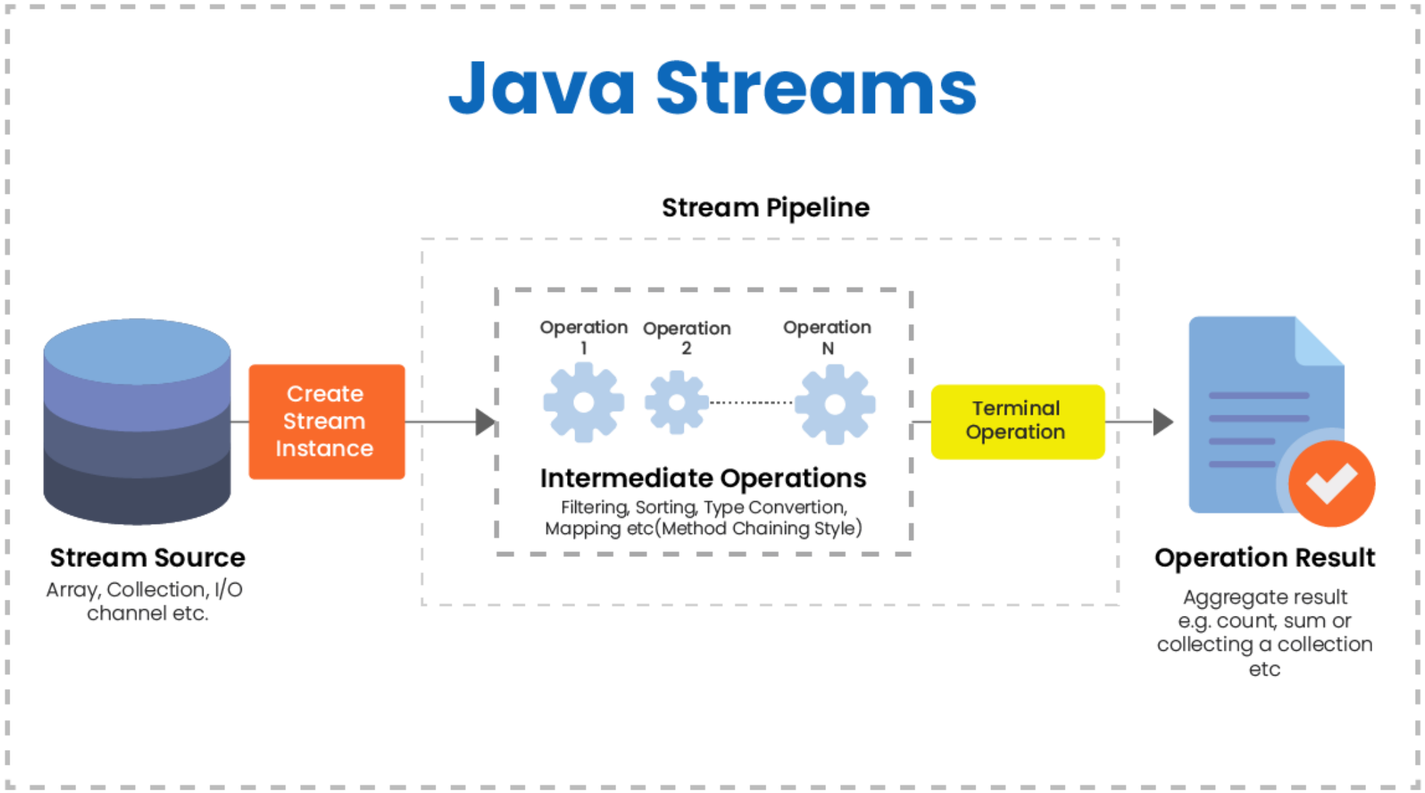
```
int[] arr = {50, 60, 70, 80, 90, 100, 110, 120}
int count = 0;
for (int x : arr) {
    if (x >= 90) {
        continue;
    }
    x += 10;
    count++;

    if (count > 3) {
        break;
    }
    System.out.print(x);
}
```

С помощью Stream API

```
IntStream.of(50, 60, 70, 80, 90, 100, 110, 120)
        .filter(x -> x < 90)
        .map(x -> x + 10)
        .limit(3)
        .forEach(System.out::print);
```

Stream API

- Источники (source) (builders, factory methods)
- Конвейерные (intermediate)
- Терминальные (terminal)



Stream API

```
List<Integer> ints = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Set<Integer> collect = ints.stream()
        .filter(i -> i > 7) // 8, 9, 10
        .map(i -> i + 1) // 9, 10, 11
        .collect(Collectors.toList());
```

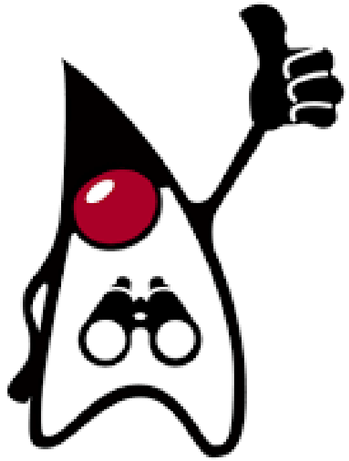
Stream API


КРАН ШАРОВЫЙ
ФИЛЬТР ГРУБОЙ ОЧИСТКИ
ОБРАТНЫЙ КЛАПАН
РЕГУЛЯТОР ДАВЛЕНИЯ С МАНОМЕТРОМ
СЧЕТЧИК
КОЛЛЕКТОР

Stream API

Особенности:

- → Стримы имеют ленивую обработку данных (отложенную)
- → Стрим начинает обрабатываться только тогда, когда вызывается терминальная операция
- → Все промежуточные операции выстраивают только будущий pipeline, сами они в моменте ничего не делают
- → Может обработать не все элементы**!!!**
- → Может быть как однопоточным так и параллельным



Sources

→Stream.empty()

→Stream.ofNullable(*элемент*)

→collection.stream()

→Stream.of(*значение1*,... *значениеN*)

→Arrays.stream(*массив*)

→Files.lines(*путь_к_файлу*)

→collection.parallelStream()

→Stream.iterate(*начальное_условие*, *выражение_генерации*)

→Stream.generate(*выражение_генерации*)

Intermediate

→filter

→map (mapToInt/mapToLong/mapToDouble)

→flatMap, (flatMapToInt, flatMapToDouble, flatMapToLong)

→distinct

→peek

→sorted

→limit

→skip

Full barrier

```
Stream.of("за экзамен я получу", "отл", "неуд")
        .limit(2)
        .sorted()
        .forEach(x -> System.out.println(x));
```

Full barrier

```
Stream.of("за экзамен я получу", "отл", "неуд")
        .limit(2)
        .sorted()
        .forEach(x -> System.out.println(x)); // за экзамен я получу \n отл
```

Full barrier

```
Stream.of("за экзамен я получу", "отл", "неуд")
        .sorted()
        .limit(2)
        .forEach(x -> System.out.println(x));
```

Full barrier

```
Stream.of("за экзамен я получу", "отл", "неуд")
        .sorted()
        .limit(2)
        .forEach(x -> System.out.println(x)); // за экзамен я получу \n неуд
```

Short circuit

```
Stream.iterate(1, x -> x + 1)
        .limit(3)
        .forEach(x -> System.out.println(x));
```

Short circuit

```
Stream.iterate(1, x -> x + 1)
        .limit(3)
        .forEach(x -> System.out.println(x));

// 1
// 2
// 3
```

Non Short circuit

```
Stream.iterate(1, x -> x + 1)
        .sorted()
        .limit(3)
        .forEach(x -> System.out.println(x));
```

Non Short circuit

```
Stream.iterate(1, x -> x + 1)
        .sorted()
        .limit(3)
        .forEach(x -> System.out.println(x));

// 2
// 3
// 4
// 5
// 6
// ...
// ∞
// OutOfMemoryError: Java heap space
```

Terminal

→count

→findFirst/findAny -> Optional

→allMatch/anyMatch/noneMatch

→forEach/forEachOrdered

→min/max -> Optional

→toArray

→collect

→reduce

→sum/average/summaryStatistics(примитивы)

Optimization - count

```
long count = Stream.of(1, 2, 3, 4, 5)
        .peek(System.out::println)
        .count();
```

Optimization - count

```
long count = Stream.of(1, 2, 3, 4, 5)
        .peek(System.out::println)
        .count();
// 5
```

Optimization - count

```
long count = Stream.of(1,2,3,4,5)
        .filter(x -> true)
        .peek(System.out::println)
        .count();
```

Optimization - count

```
long count = Stream.of(1,2,3,4,5)
        .filter(x -> true)
        .peek(System.out::println)
        .count();
// 1
// 2
// 3
// 4
// 5
```

Short circuit

```
List<List<String>> listOfLists = List.of(
        List.of("foo", "bar", "baz"),
        List.of("Java", "Kotlin", "Groovy"),
        List.of("Hello", "Good Bye")
);

listOfLists.stream()
        .flatMap(List::stream)
        .peek(System.out::println)
        .anyMatch("Java"::equals);
```

Short circuit

```
List<List<String>> listOfLists = List.of(
        List.of("foo", "bar", "baz"),
        List.of("Java", "Kotlin", "Groovy"),
        List.of("Hello", "Good Bye")
);

listOfLists.stream()
        .flatMap(List::stream)
        .peek(System.out::println)
        .anyMatch("Java"::equals);
```

```
// foo
// bar
// baz
// Java
```

Terminal -> Collectors

→toList()

→toSet()

→toCollection(supplier)

→toMap(key, value, **merger**, **mapSupplier**)

→groupingBy(classifier, **mapSupplier**, **downstream**)

→joining(**separator**, **prefix**, **suffix**)

→partitioningBy(predicate, **downstream**)

→reducing/counting/mapping/minBy/maxBy

→averagingInt/averagingLong/averagingDouble

→summingInt/summingLong/summingDouble

toMap(k, v)

```
String[] strings = {"Hello", "Java", "Course"};

Map<String, Integer> lengths = Arrays.stream(strings)
        .collect(Collectors.toMap(x -> x, String::length));
```

toMap(k, v)

```
String[] strings = {"Hello", "Java", "Course"};

Map<String, Integer> lengths = Arrays.stream(strings)
        .collect(Collectors.toMap(x -> x, String::length));

// {Java=4, Hello=5, Course=6}
```

toMap(k, v)

```
String[] strings = {"Hello", "Java", "Course"};

Map<String, Integer> lengths = Arrays.stream(strings)
        .collect(Collectors.toMap(Function.identity(), String::length));
```

toMap(k, v, merger)

```
String[] strings = {"Hello", "Java", "Course"};

Map<String, Integer> lengths = Arrays.stream(strings)
        .collect(Collectors.toMap(x -> x, String::length,
                (first, second) -> first));
```

toMap(k, v, merger, supplier)

```
String[] strings = {"Hello", "Java", "Course"};

Map<String, Integer> lengths = Arrays.stream(strings)
        .collect(Collectors.toMap(x -> x, String::length,
                (first, second) -> first, LinkedHashMap::new));
```

groupingBy

```
Map<Integer, List<String>> stringsByLength =
        Arrays.asList("a", "bb", "c", "dd", "eee").stream()
                .collect(Collectors.groupingBy(String::length));
```

groupingBy

```
Map<Integer, List<String>> stringsByLength =
        Arrays.asList("a", "bb", "c", "dd", "eee").stream()
                .collect(Collectors.groupingBy(String::length));

// {1=[a, c], 2=[bb,dd], 3[eee]}
```

groupingBy + joining

```
Map<Integer, String> collect = Arrays.asList("a", "bb", "c", "dd", "eee").stream()
        .collect(Collectors.groupingBy(String::length,
                Collectors.joining("+")));
```

groupingBy + joining

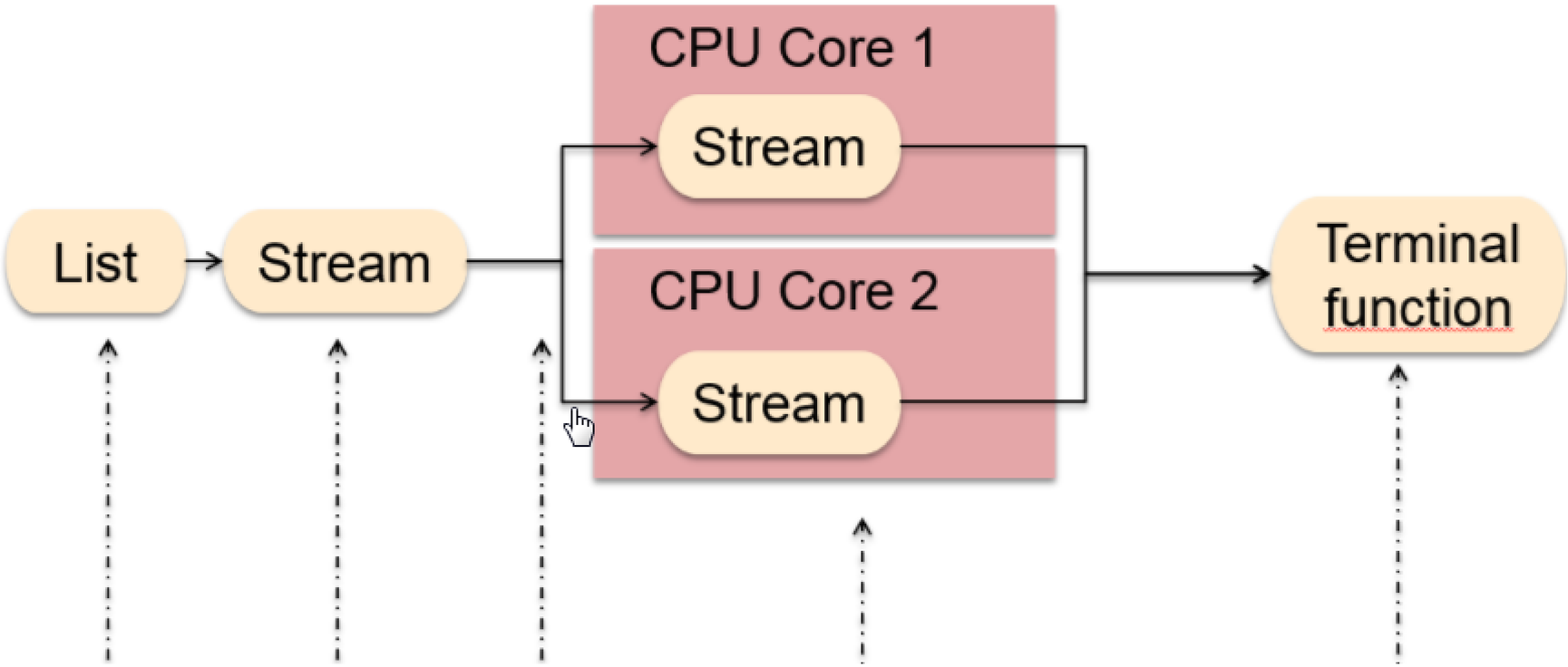
```
Map<Integer, String> collect = Arrays.asList("a", "bb", "c", "dd", "eee").stream()
        .collect(Collectors.groupingBy(String::length,
                Collectors.joining("+")));

// {1=a+c, 2=bb+dd, 3=eee}
```

Parallel stream

Parallel stream

- Используются редко, но метко (в основном для cpu-bound операций, high computing)
- Используют системный ThreadPool под капотом
- Надо замерять и знать конер кейсы, иначе получите преимущество на уровне однопоточного стрима

Parallel stream



Parallel stream

- →isParallel
- →parallel
- →sequential

Parallel stream

```
Stream.of(...)
        .peek(...) // операция последовательна
        .parallel()
        .map(...) // операция может выполняться параллельно
        .sequential()
        .reduce(...) // операция снова последовательна
```

Parallel stream

```
IntStream.range(0, 100_000_000)
        .skip(99_000_000)
        .sum();
```

```
    IntStream.range(0, 100_000_000)
            .parallel()
            .skip(99_000_000)
            .sum();
```

Parallel stream

```
IntStream.range(0, 100_000_000)
        .skip(99_000_000)
        .sum();
```

~100-200 +- мс

```
IntStream.range(0, 100_000_000)
        .parallel()
        .skip(99_000_000)
        .sum();
```

~1-10 +- мс

Parallel with custom thread pool

```
List<Long> aList = LongStream.range(1, 1_000_000)
        .boxed()
        .toList();

ForkJoinPool customThreadPool = new ForkJoinPool(4);

long actualTotal = customThreadPool.submit(
        () -> aList.parallelStream().reduce(0L, Long::sum)
).get();
```

Parallel stream

При использовании избегайте:

→forEachOrdered(...)

→findFirst()

MTC True Tech

Вопросы?

QR