Введение в программирование на Java

Лекция 16. Stream API.

Виталий Олегович Афанасьев

26 мая 2025

Optional

Проблема отсутствующего значения (1)

```
1 public class UserStorage {
2     public User getUserByName(String name) {
3         ...
4     }
5 }
```

Что делает метод getUserByName, если пользователя не существует?

```
1 public static void example(UserStorage storage) {
2     User user = storage.getUserByName("User that doesn't exist");
3     List<Message> inbox = user.getInbox();
4     ...
5 }
```

Проблема отсутствующего значения (2)

Вариант 1: вернуть `null`:

```
1 public class UserStorage {
2     public User getUserByName(String name) {
3         if (!userExists(name)) return null;
4         ...
5     }
6 }
```

Вариант 2: бросить исключение:

```
1 public class UserStorage {
2     public User getUserByName(String name) {
3         if (!userExists(name)) throw new UserDoesntExistException(name);
4         ...
5     }
6 }
```

Проблема отсутствующего значения (3)

Проблема этих двух подходов — нам приходится знать, **как** реализован данный метод.

Если забыть про нюансы реализации — код завершится с ошибкой **во время выполнения**.

Проблема отсутствующего значения (3)

Проблема этих двух подходов — нам приходится знать, **как** реализован данный метод.

Если забыть про нюансы реализации — код завершится с ошибкой **во время выполнения**.

Возможно ли закодировать эту функцию так, чтобы отсутствие проверки было **ошибкой компиляции**?

Optional (1)

Библиотечный тип Optional (из пакета java.util) представляет собой обёртку над значением, которое может отсутствовать.

```
1 public class UserStorage {
2     public Optional<User> getUserByName(String name) {
3         if (!userExists(name)) return Optional.empty();
4         User user = ...;
5         return Optional.of(user);
6     }
7 }
```

Optional может находиться в двух состояниях:

- empty — отсутствующее значение.
- present — наличие не-null значения.

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3
4
5
6
7
8
9
10
11
12
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  boolean userExists = optionalUser.isPresent();
4
5
6
7
8
9
10
11
12
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  boolean userExists = optionalUser.isPresent();
4  boolean userDoesntExist = optionalUser.isEmpty();
5
6
7
8
9
10
11
12
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  boolean userExists = optionalUser.isPresent();
4  boolean userDoesntExist = optionalUser.isEmpty();
5
6  if (user.isPresent()) {
7
8
9
10
11
12
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  boolean userExists = optionalUser.isPresent();
4  boolean userDoesntExist = optionalUser.isEmpty();
5
6  if (user.isPresent()) {
7      User user = user.get();
8
9
10
11
12
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  boolean userExists = optionalUser.isPresent();
4  boolean userDoesntExist = optionalUser.isEmpty();
5
6  if (user.isPresent()) {
7      User user = user.get();
8      List<Message> inbox = user.getInbox();
9
10
11
12
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  boolean userExists = optionalUser.isPresent();
4  boolean userDoesntExist = optionalUser.isEmpty();
5
6  if (user.isPresent()) {
7      User user = user.get();
8      List<Message> inbox = user.getInbox();
9      ...
10
11
12
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  boolean userExists = optionalUser.isPresent();
4  boolean userDoesntExist = optionalUser.isEmpty();
5
6  if (user.isPresent()) {
7      User user = user.get();
8      List<Message> inbox = user.getInbox();
9      ...
10 }
11
12
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  boolean userExists = optionalUser.isPresent();
4  boolean userDoesntExist = optionalUser.isEmpty();
5
6  if (user.isPresent()) {
7      User user = user.get();
8      List<Message> inbox = user.getInbox();
9      ...
10 }
11
12 User user = user.get(); // NoSuchElementException, если отсутствует
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8
9
10
11
12
13
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9
10
11
12
13
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9  User user = optionalUser.orElse(defaultUser);
10
11
12
13
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9  User user = optionalUser.orElse(defaultUser);
10
11 User user = optionalUser.orElseGet(() -> createDefaultUser());
12
13
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9  User user = optionalUser.orElse(defaultUser);
10
11 User user = optionalUser.orElseGet(() -> createDefaultUser());
12
13 User user =optionalUser.orElseThrow(()->new UserDoesntExistException());
14
15
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9  User user = optionalUser.orElse(defaultUser);
10
11 User user = optionalUser.orElseGet(() -> createDefaultUser());
12
13 User user =optionalUser.orElseThrow(()->new UserDoesntExistException());
14
15 Optional<List<Message>> optionalInbox = optionalUser
16
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9  User user = optionalUser.orElse(defaultUser);
10
11 User user = optionalUser.orElseGet(() -> createDefaultUser());
12
13 User user =optionalUser.orElseThrow(()->new UserDoesntExistException());
14
15 Optional<List<Message>> optionalInbox = optionalUser
16     .map(user -> user.getInbox());
17
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9  User user = optionalUser.orElse(defaultUser);
10
11 User user = optionalUser.orElseGet(() -> createDefaultUser());
12
13 User user =optionalUser.orElseThrow(()->new UserDoesntExistException());
14
15 Optional<List<Message>> optionalInbox = optionalUser
16     .map(user -> user.getInbox());
17 Optional<Integer> optionalInboxSize = optionalUser
18
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9  User user = optionalUser.orElse(defaultUser);
10
11 User user = optionalUser.orElseGet(() -> createDefaultUser());
12
13 User user =optionalUser.orElseThrow(()->new UserDoesntExistException());
14
15 Optional<List<Message>> optionalInbox = optionalUser
16     .map(user -> user.getInbox());
17 Optional<Integer> optionalInboxSize = optionalUser
18     .map(user -> user.getInbox())
19
```

Optional (2)

```
1  Optional<User> optionalUser = storage.getUser(...);
2
3  optionalUser.ifPresent(user -> {
4      List<Message> inbox = user.getInbox();
5      ...
6  });
7
8  User defaultUser = ...;
9  User user = optionalUser.orElse(defaultUser);
10
11 User user = optionalUser.orElseGet(() -> createDefaultUser());
12
13 User user =optionalUser.orElseThrow(()->new UserDoesntExistException());
14
15 Optional<List<Message>> optionalInbox = optionalUser
16     .map(user -> user.getInbox());
17 Optional<Integer> optionalInboxSize = optionalUser
18     .map(user -> user.getInbox())
19     .map(inbox -> inbox.size());
```

Optional (3)

```
1 int messagesForUser = storage.getUser(...)
2     .map(User::getInbox)
3     .map(List::size)
4     .orElse(0);
```

Optional (4)

`Optional` — это не "серебряная пуля".

Чтобы он был безопаснее `null`, с ним нужно обращаться с умом: не вызывать методы `get`, `orElseThrow` и подобные, если не уверены, что значение точно присутствует.

Stream API

Stream API

```
1 List<String> words = List.of("a", "bc", "def", "ghij", ...);
2
3 int shortWordsCount = 0;
4 for (String word : words) {
5     if (word.length() < 5) {
6         ++shortWordsCount;
7     }
8 }
9 System.out.println(shortWordsCount);
```

Stream API

```
1 List<String> words = List.of("a", "bc", "def", "ghij", ...);
2
3 int shortWordsCount = 0;
4 for (String word : words) {
5     if (word.length() < 5) {
6         ++shortWordsCount;
7     }
8 }
9 System.out.println(shortWordsCount);
```

Stream API позволяет использовать декларативный подход для обработки потоков данных:

```
1 long shortWordsCount = words.stream()
2     .filter(word -> word.length() < 5)
3     .count();
```

Создание потока данных

- Поток можно получить из любой коллекции:
  - `Stream<E> stream();`
  - `list.stream()`
- Поток можно получить из массива методами класса `Arrays`:
  - `static <T> Stream<T> stream(T[] array);`
  - `static <T> Stream<T> stream(T[] array,int from,int to);`
  - `Arrays.stream(values)`
- Поток можно создать перечислением элементов:
  - `static <T> Stream<T> of(T... values);`
  - `Stream.of(1, 2, 3, 4, 5)`
- Поток можно сгенерировать генератором:
  - `static <T> Stream<T> generate(Supplier<T> s);`
  - `Stream.generate(() -> (int) (Math.random() * 10))`
- Поток можно сгенерировать итератором:
  - `static <T> Stream<T> iterate(T seed,UnaryOperator<T> f);`
  - `Stream.iterate(1, x -> x + 1)`

Операции над потоками данных

Операции в Stream API разделяются на **промежуточные** и **терминальные**:

- **Терминальные операции** потребляют поток данных и возвращают результат.
  - Над потоком данных можно выполнить только одну терминальную операцию.
- **Промежуточные операции** выполняются не сразу, а только при необходимости (т.н. отложенное/ленивое поведение, lazy fashion).
  - Промежуточных операций может быть сколько угодно.

Операции над потоками данных **не меняют оригинальных коллекций!**

Операция filter

Промежуточная операция `filter` возвращает поток, элементы которого удовлетворяют предикату.

```
Stream<T> filter(Predicate<? super T> predicate);
```

```
1 List<Integer> list = Stream.of(1, 2, 3, 4, 5)
2     .filter(x -> x % 2 == 0)
3     .toList();
4 System.out.println(list); // 2, 4
```

Операция map

Промежуточная операция map применяет функцию отображения к каждому элементу потока.

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

```
1 List<Integer> list = Stream.of("Hello", "123", "A", "", "...")
2     .map(x -> x.length())
3     .toList();
4 System.out.println(list); // 5, 3, 1, 0, 3
```

Операция findFirst

Терминальная операция `findFirst` возвращает первый элемент из потока.

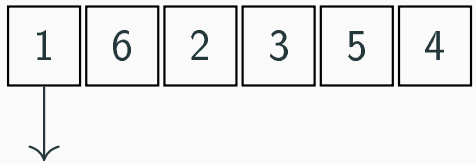
```
Optional<T> findFirst();
```

```
1 Optional<String> first = Stream.of("Hello", "123", "a", "Hi", "")
2     .filter(x -> x.length() < 5)
3     .findFirst();
4 System.out.println(first); // Optional("123")
```

Пример ленивого вычисления

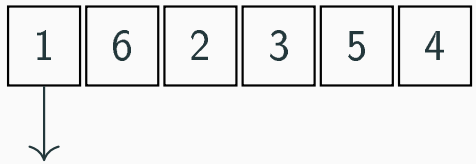

```
.filter(x -> x % 2 == 0)

    .map(x -> x * 2)

  .filter(x -> x < 10)

      .findFirst()
```

Пример ленивого вычисления




```
.filter(x -> x % 2 == 0)

    .map(x -> x * 2)

  .filter(x -> x < 10)

      .findFirst()
```

Пример ленивого вычисления

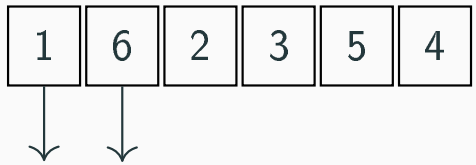



```
.filter(x -> x % 2 == 0)
```

X

```
    .map(x -> x * 2)

  .filter(x -> x < 10)

     .findFirst()
```

Пример ленивого вычисления

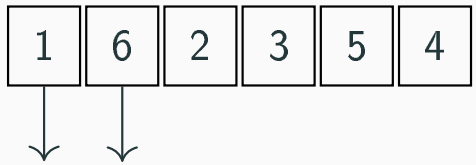



```
.filter(x -> x % 2 == 0)
```

X

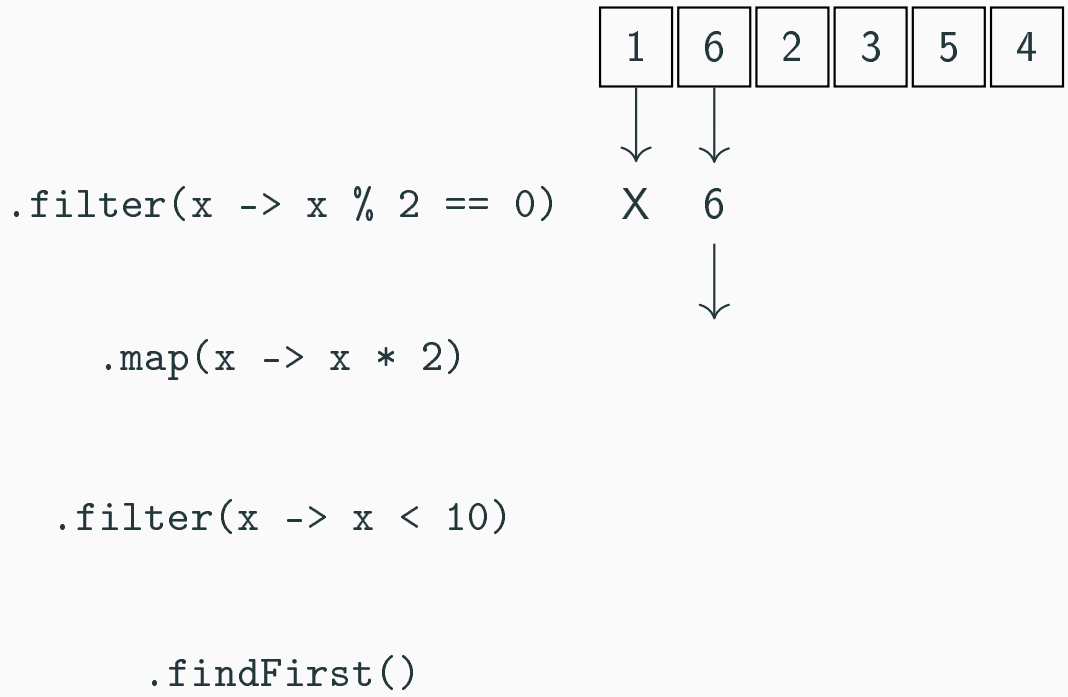
```
    .map(x -> x * 2)

  .filter(x -> x < 10)

     .findFirst()
```

Пример ленивого вычисления

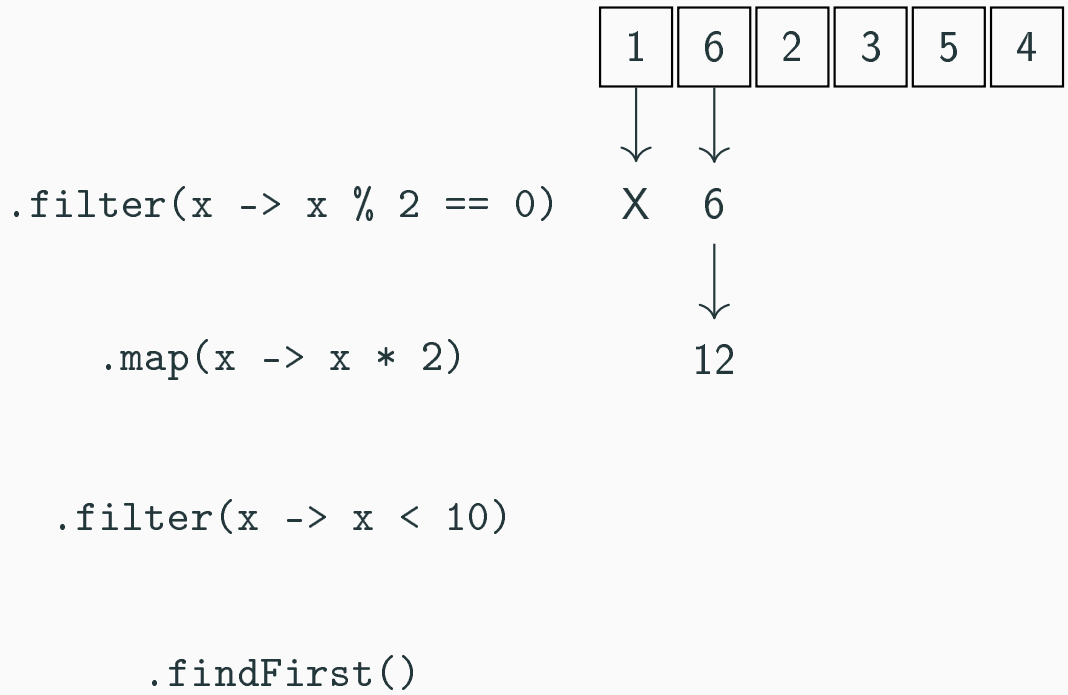



```
.filter(x -> x % 2 == 0)   X  6

    .map(x -> x * 2)

  .filter(x -> x < 10)

     .findFirst()
```

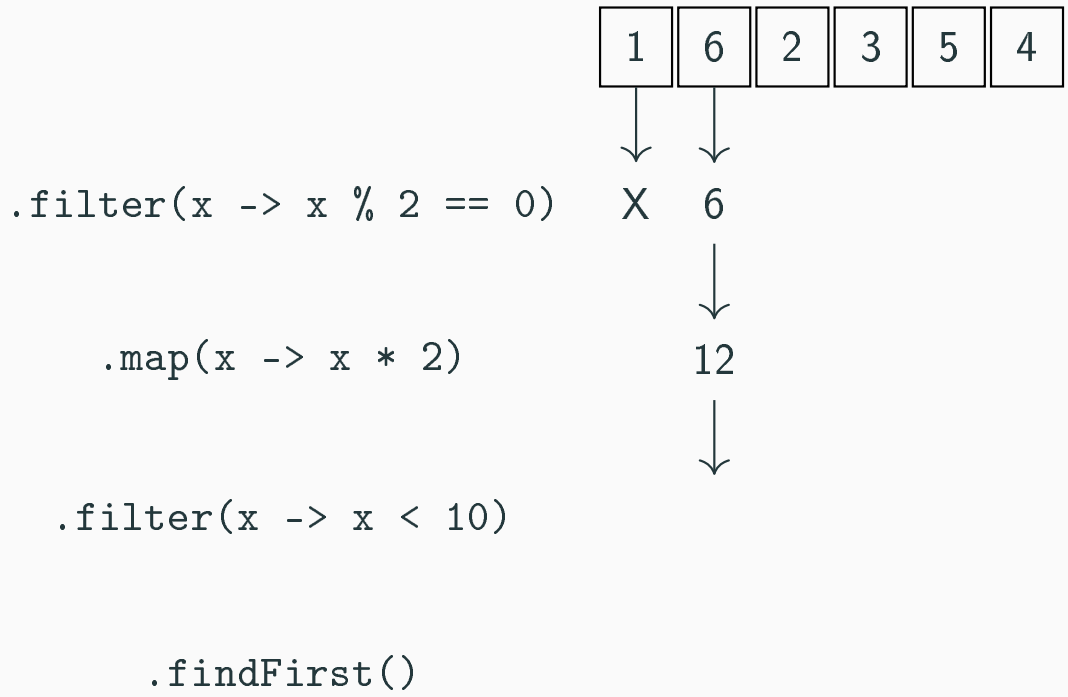
Пример ленивого вычисления



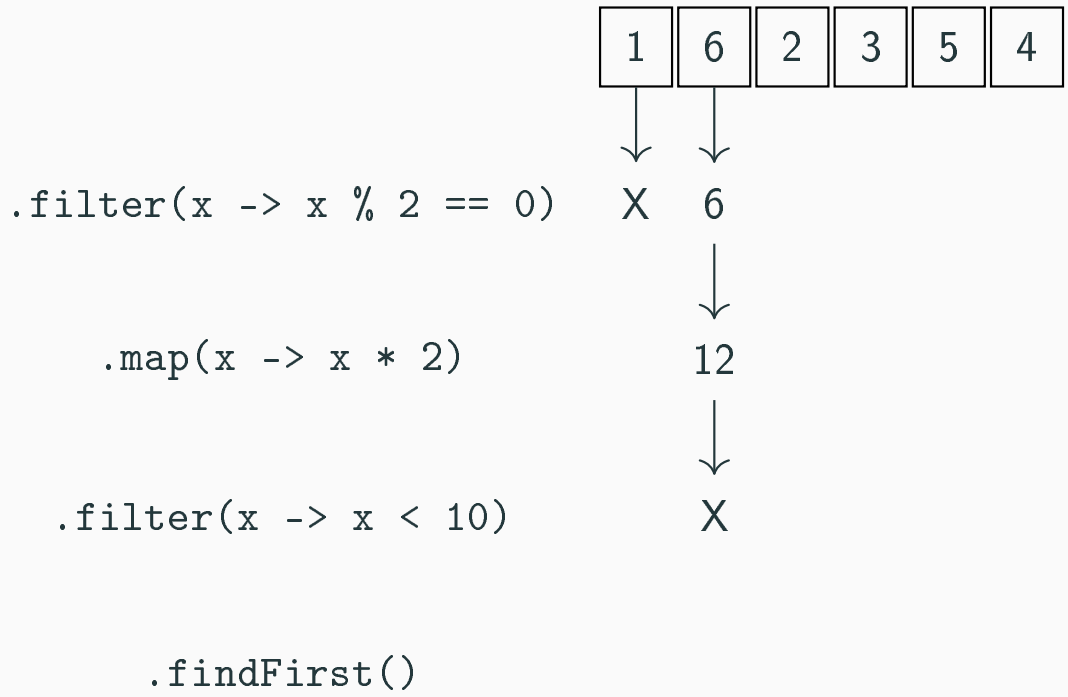
Пример ленивого вычисления



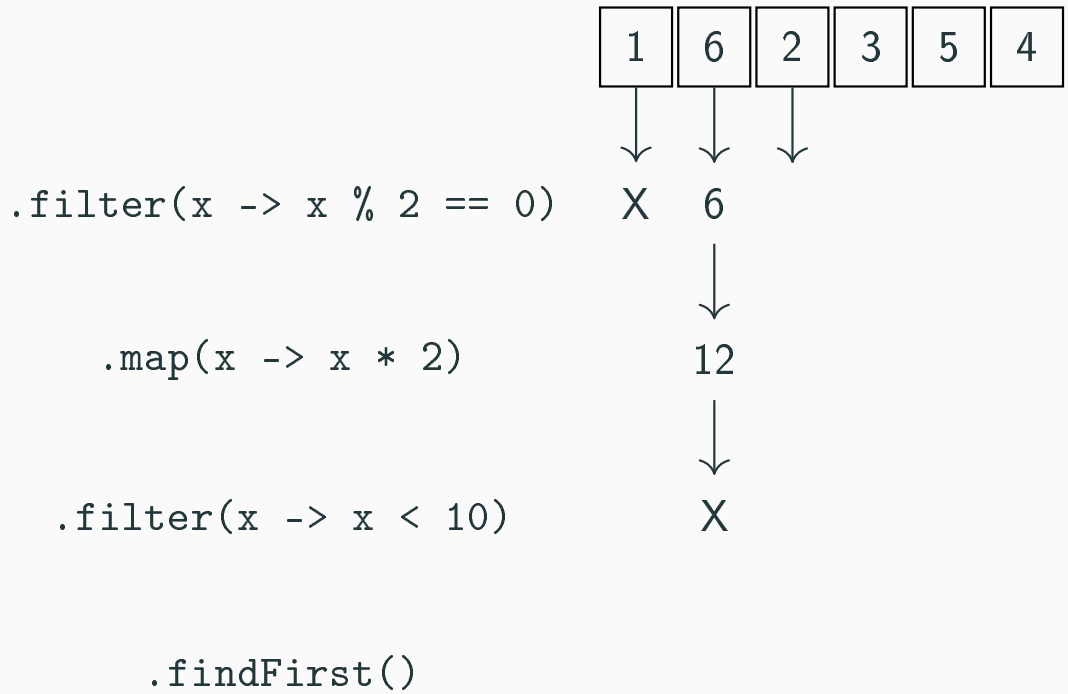
Пример ленивого вычисления



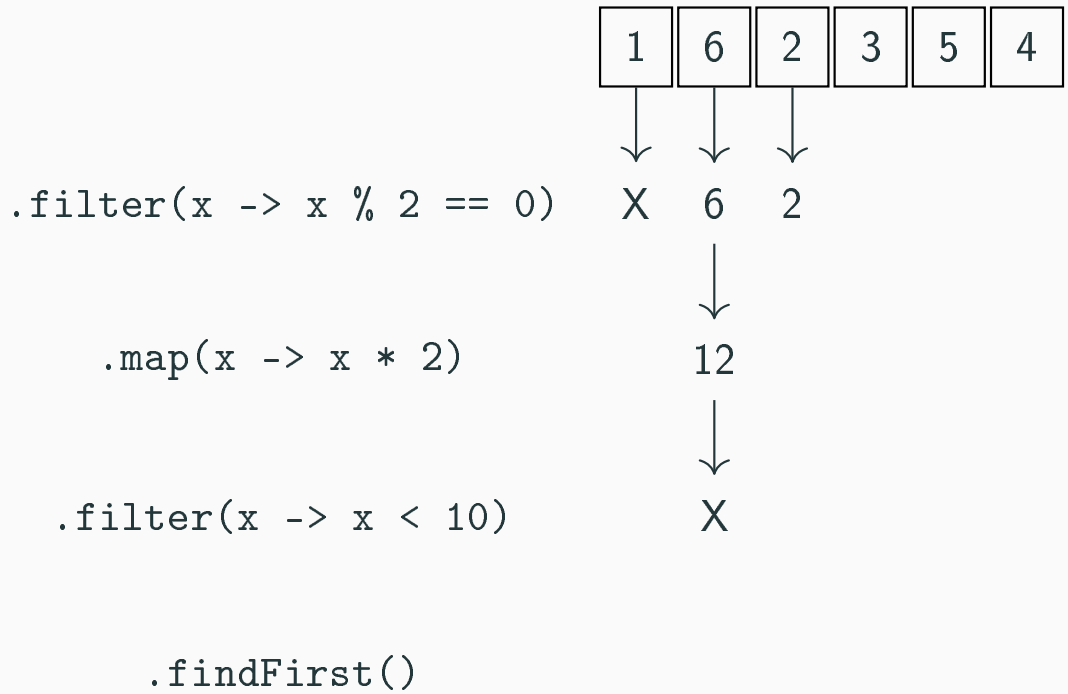
Пример ленивого вычисления



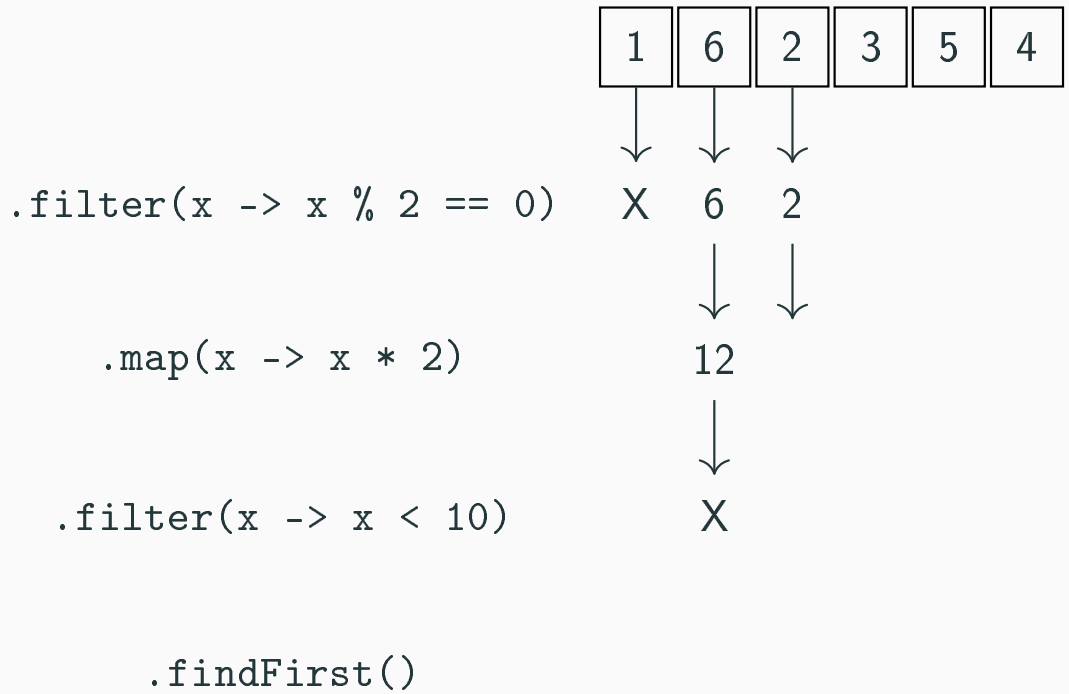
Пример ленивого вычисления



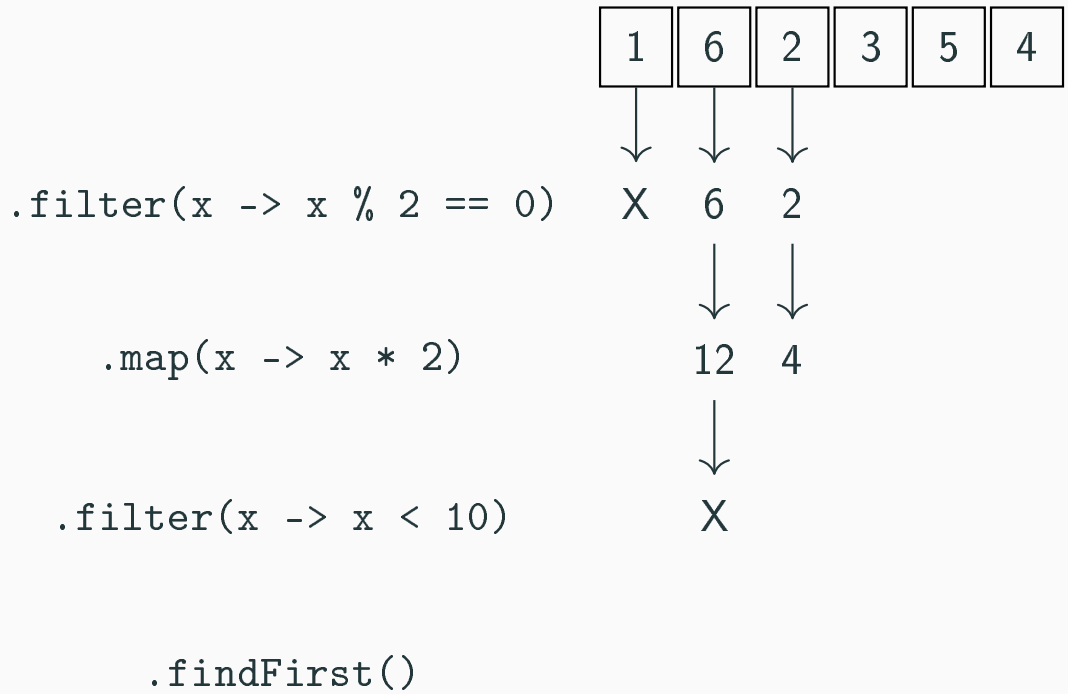
Пример ленивого вычисления



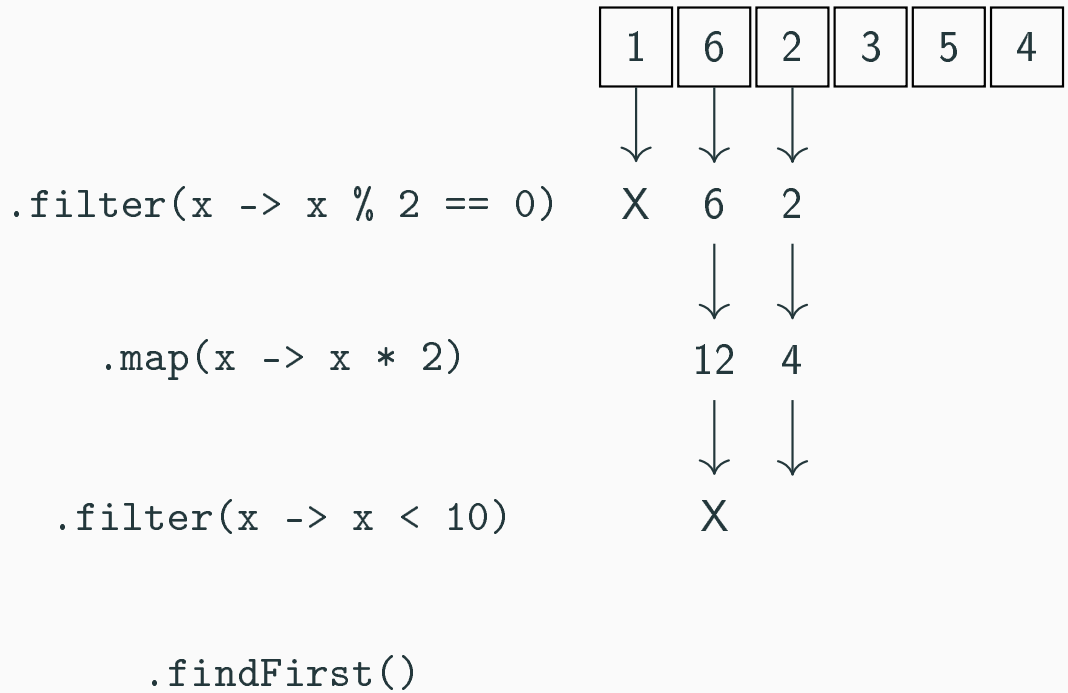
Пример ленивого вычисления



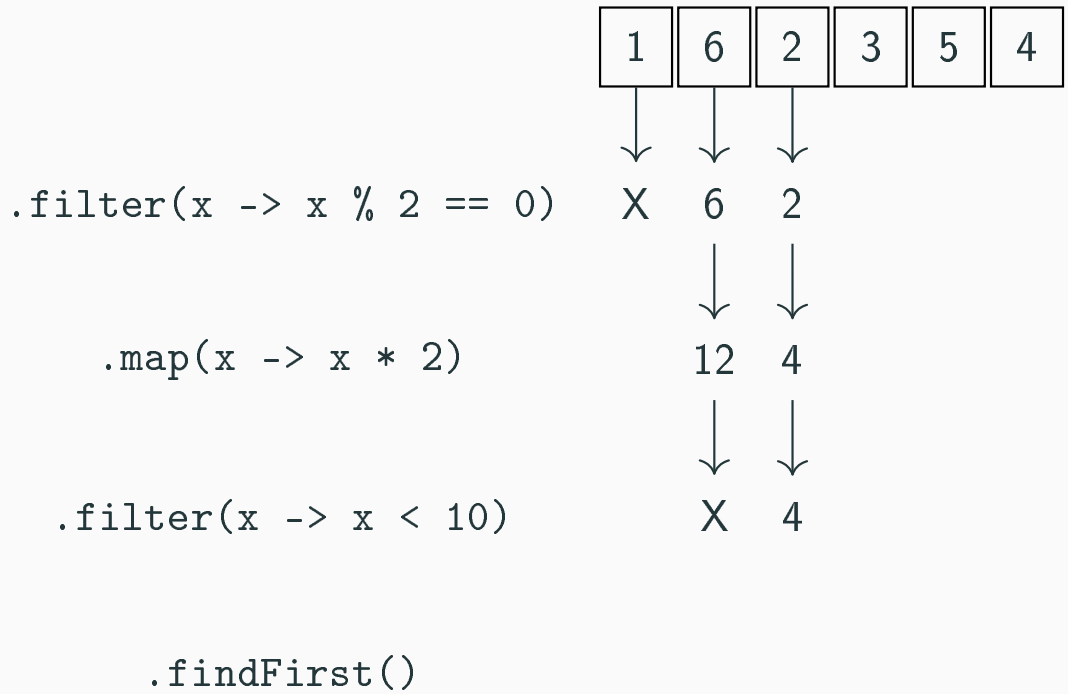
Пример ленивого вычисления



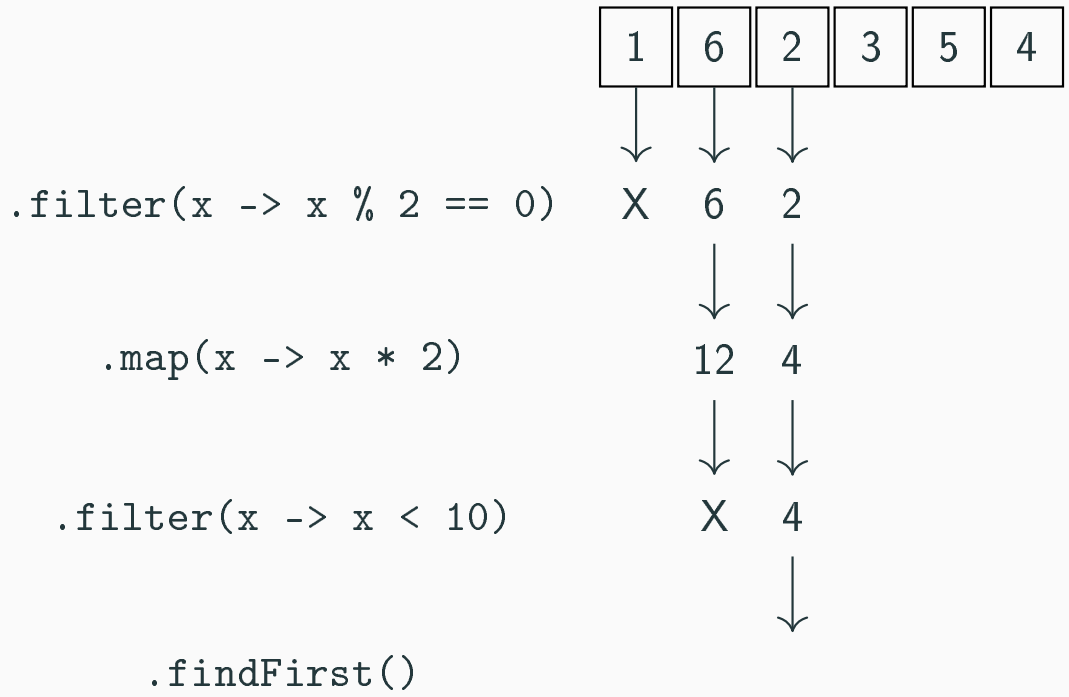
Пример ленивого вычисления



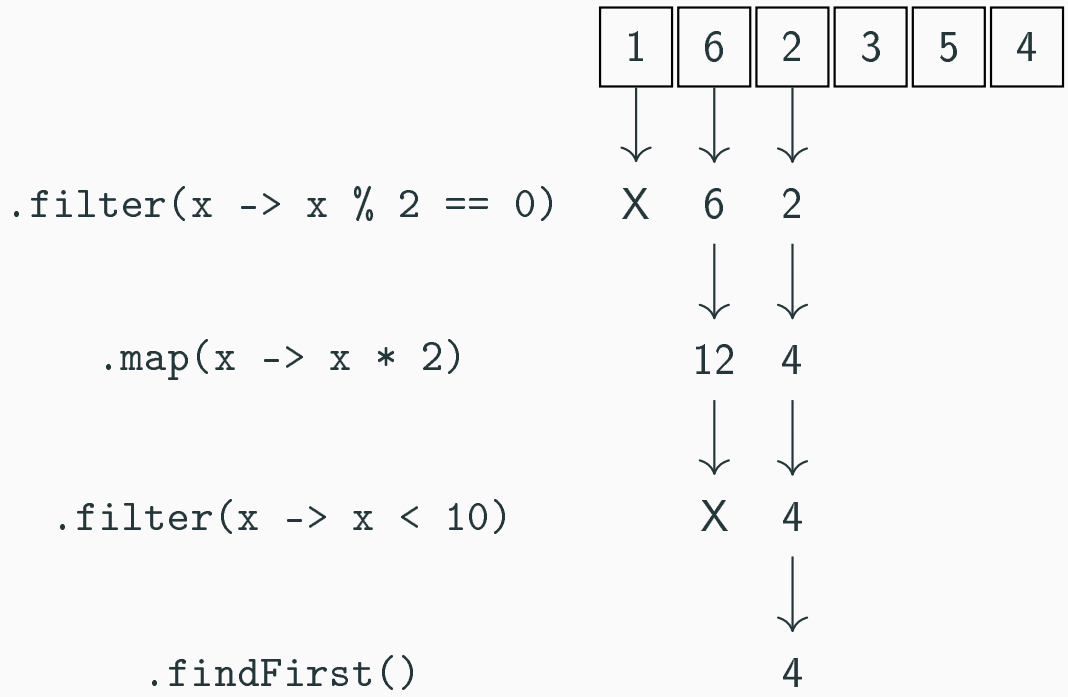
Пример ленивого вычисления



Пример ленивого вычисления



Пример ленивого вычисления



Переиспользование потоков данных

Поток данных **нельзя переиспользовать** после выполнения терминальной операции:

```
1 Stream<String> shortWords = words.stream()
2     .filter(word -> word.length() < 5);
3
4 Optional<String> word = shortWords.findFirst();
5 Optional<String> otherWord = shortWords
6     .map(word -> "!" + word + "!") // IllegalStateException
7     .findFirst();
```

Операция flatMap

Промежуточная операция `flatMap` применяет функцию отображения к каждому элементу потока. В отличии от функции `map`, функция отображения может возвращать произвольное кол-во элементов.

Другими словами, `flatMap` — это отображение "один-ко-многим".

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>>
      mapper);
```

```
1 List<String> list = Stream.of("Hello World", "a b c")
2     .flatMap(x -> Stream.of(x.split(" ")))
3     .toList();
4 System.out.println(list); // Hello, World, a, b, c
```

Операция flatMap

Промежуточная операция `flatMap` применяет функцию отображения к каждому элементу потока. В отличии от функции `map`, функция отображения может возвращать произвольное кол-во элементов.

Другими словами, `flatMap` — это отображение "один-ко-многим".

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>>
      mapper);
```

```
1 List<String> list = Stream.of("Hello World", "a b c")
2     .flatMap(x -> Stream.of(x.split(" ")))
3     .toList();
4 System.out.println(list); // Hello, World, a, b, c
```

Если же применить `map`, то мы получим коллекцию из коллекций:

```
1 List<String[]> list = Stream.of("Hello World", "a b c")
2     .map(x -> x.split(" "))
3     .toList();
4 System.out.println(list); // [Hello, World], [a, b, c]
```

Операция limit

Промежуточная операция `limit` возвращает поток, состоящий из первых `N` элементов (если исходных элементов меньше — возвращает все).

```
Stream<T> limit(long maxSize);
```

```
1 List<Integer> list = Stream.of("Hello", "123", "A", "", "...")
2     .map(x -> x.length())
3     .limit(2)
4     .toList();
5 System.out.println(list); // 5, 3
```

Операция skip

Промежуточная операция `skip` удаляет из потока первые `N` элементов (если исходных элементов меньше — возвращает пустой поток).

```
Stream<T> skip(long n);
```

```
1 List<Integer> list = Stream.of("Hello", "123", "A", "", "...")
2     .map(x -> x.length())
3     .skip(2)
4     .toList();
5 System.out.println(list); // 1, 0, 3
```

Операция concat

Вспомогательная операция concat объединяет два потока.

```
static<T> Stream<T> concat(Stream<? extends T> a,Stream<? extends T> b);
```

```
1 List<Integer> list = Stream.concat(
2         Stream.of("Hello", "123"),
3         Stream.of("A", "", "...")
4     ).map(x -> x.length())
5     .toList();
6 System.out.println(list); // 5, 3, 1, 0, 3
```

Операция distinct

Промежуточная операция `distinct` возвращает поток, состоящий из уникальных элементов.

```
Stream<T> distinct();
```

```
1 List<String> list = Stream.of("Hello", "123", "Hello")
2     .distinct()
3     .toList();
4 System.out.println(list); // Hello, 123
```

Операция sorted

Промежуточная операция `sorted` возвращает упорядоченный поток.

```
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);
```

```
1 List<String> list = Stream.of("Hello", "123", "a", "Hi")
2     .sorted()
3     .toList();
4 System.out.println(list); // 123, Hello, Hi, a
```

Важно: сортировка относится к операциям, которые нельзя выполнить частично — чтобы отсортировать поток, нужно знать **все** его элементы.

Операция peek

Промежуточная операция peek возвращает исходный поток без изменений, но выполняет действие на каждом элементе по мере потребления.

Данная операция очень полезна при отладке.

```
Stream<T> peek(Consumer<? super T> action);
```

```
1 List<String> list = Stream.of("Hello", "123", "a", "Hi", "")
2     .filter(x -> x.length() > 1)
3     .peek(x -> System.out.println(x)) // Hello, 123, Hi
4     .map(x -> x + x)
5     .toList();
6 System.out.println(list); // HelloHello, 123123, HiHi
```

Операция count

Терминальная операция count возвращает количество элементов в потоке.

```
long count();
```

```
1 long count = Stream.of("Hello", "123", "a", "Hi", "")
2     .filter(x -> x.length() > 1)
3     .count();
4 System.out.println(count); // 3
```

Операции min и max

Терминальные операции `min` и `max` возвращают минимум/максимум из элементов в потоке.

```
Optional<T> min(Comparator<? super T> comparator);
Optional<T> max(Comparator<? super T> comparator);
```

```
1 Optional<Integer> max = Stream.of("Hello", "123", "a", "Hi", "")
2     .filter(x -> x.length() > 1)
3     .map(x -> x.length())
4     .max(Integer::compare);
5 System.out.println(max); // Optional(5)
```

Операции anyMatch, allMatch, noneMatch

Терминальные операции anyMatch/allMatch/noneMatch проверяют, что предикату удовлетворяют хотя бы один/все/ни один элемент потока.

```
boolean anyMatch(Predicate<? super T> predicate);
boolean allMatch(Predicate<? super T> predicate);
boolean noneMatch(Predicate<? super T> predicate);
```

```
1 boolean any = Stream.of("Hello", "123", "a", "Hi", "")
2     .map(x -> x + x)
3     .anyMatch(x -> x.length() < 3);
4 System.out.println(any); // true
```

Операция forEach

Терминальная операция `forEach` выполняет операцию на всех элементах потока.

```
void forEach(Consumer<? super T> action);
```

```
1 Stream.iterate(1, x -> x + 2)
2     .filter(x -> x < 10)
3     .map(x -> x * 2)
4     .limit(3)
5     .forEach(System.out::println); // 2, 6, 10
```

Операция collect

Терминальная операция `collect` собирает элементы потока произвольным образом.

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

```
1 Set<String> set = Stream.of("A", "B", "A")
2     .collect(Collectors.toSet());
3 System.out.println(set); // A, B
```

Collectors API: joining

Коллектор `joining` позволяет собрать несколько строк в одну.

```
1 String joining1 = Stream.of("A", "B", "C")
2     .collect(Collectors.joining()); // "ABC"
3 String joining2 = Stream.of("A", "B", "C")
4     .collect(Collectors.joining(":")); // "A:B:C"
5 String joining3 = Stream.of("A", "B", "C")
6     .collect(Collectors.joining(", ")); // "A, B, C"
```

Collectors API: toMap

Коллектор `toMap` позволяет собрать элементы в `Map` с заданными ключами и значениями.

```
1 List<Person> people = List.of(new Person(...), ...);
2 Map<Integer, String> idToName = people.stream()
3     .collect(Collectors.toMap(Person::getId, Person::getName));
```

Collectors API: groupingBy

Коллектор groupingBy позволяет собрать элементы в Map, сгруппировав их по заданному ключу.

```
1 List<Person> people = List.of(new Person(...), ...);
2 Map<String, List<Person>> cityToPeople = people.stream()
3     .collect(Collectors.groupingBy(Person::getCity));
```

Collectors API: partitioningBy

Коллектор `partitioningBy` позволяет разбить элементы по предикату.

```
1 List<Person> people = List.of(new Person(...), ...);
2 Map<Boolean, List<Person>> adultsAndNonAdults = people.stream()
3     .collect(Collectors.partitioningBy(p -> p.age() >= 18));
```

Операция reduce (1)

Терминальная операция `reduce` производит операцию свёртки/редукции/сведения над потоком данных.

```
Optional<T> reduce(BinaryOperator<T> accumulator);
T reduce (T identity, BinaryOperator<T> accumulator);
```

Данная операция позволяет обобщить операцию сбора потока данных в одно значение.

Операция reduce (2)

Представим список следующим образом:

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

Здесь `[]` — пустой список, а `:` — операция добавления элемента в конец списка.

Представим, что мы выполняем `reduce(42, (x, y) -> x + y)`. Тогда `[]` заменится на `42`, а `:` — на сложение:

```
42 + 1 + 2 + 3 + 4 + 5
```

Таким образом, `reduce` последовательно применяет заданную операцию слева направо, постепенно сворачивая поток в одно значение.

Операция reduce (3)

Сумма:

```
Stream.of(1, 3, 2).reduce(
    0,
    (sum, cur) -> sum + cur
); // 6
```

Произведение:

```
Stream.of(2, 3, 2).reduce(
    1,
    (prod, cur) -> prod * cur
); // 12
```

Максимум:

```
Stream.of(1, 3, 2).reduce(
    Integer.MIN_VALUE,
    (prevMax, cur) -> Math.max(prevMax, cur)
); // 3
```

Операция reduce (4)

Количество:

```
Stream.of(1, 3, 2).reduce(
    0,
    (total, cur) -> total + 1
); // 3
```

Количество положительных:

```
Stream.of(1, 3, -2).reduce(
    0,
    (total, cur) -> cur > 0 ? total + 1 : total
); // 2
```

Операция reduce (5)

У операции `reduce` есть несколько требований к оператору накопления:

- **Stateless** — оператор накопления не должен хранить никакое состояние, и результат не должен от него зависеть.
- **Non-interfering** — оператор накопления не должна менять промежуточные значения.
- **Ассоциативность** — $(1 + 2) + 3 == 1 + (2 + 3)$.

Свёртка потока списков в один список:

```
Stream.of(List.of(1, 3), List.of(2, 4)).reduce(
    new ArrayList<>(),
    (result, cur) -> {
        // Копируем result! Нельзя вызывать result.addAll(cur)!
        List<Integer> newResult = new ArrayList<>(result);
        newResult.addAll(cur);
        return newResult;
    }
); // [1, 3, 2, 4]
```