# Функциональное программирование

Различие состоит в том, что в объектно-ориентированном программировании абстрагируются главным образом данные, а в функциональном – поведение.

## Общее описание

### История

Слово «лямбда» ведет свое начало от разработанной в научной среде системы под названием лямбда-исчисление (lambda calculus), которая использовалась для описания вычислений.

Лямбда выражения появились в Java 8.

**До Java 8**: Java полностью объектно-ориентированный язык. За исключением примитивных типов, все в Java – это объекты (в т.ч. массивы, экземпляры классов и т.д.). Невозможно определить вне класса какую-нибудь функцию и невозможно передать метод как аргумент или вернуть тело метода как результат другого метода. Надо было писать анонимные классы, когда нужно было передать некую функциональность в какой-нибудь метод.

Есть пределы уровню абстрагирования, доступному авторам библиотек на Java. Хороший пример – отсутствие эффективных параллельных операций с большими коллекциями данных. Чтобы такие библиотеки распараллеливания массовых операций над данными были возможны, в Java пришлось внести дополнение на уровне языка: лямбда-выражения.

# Lambda

### Определение

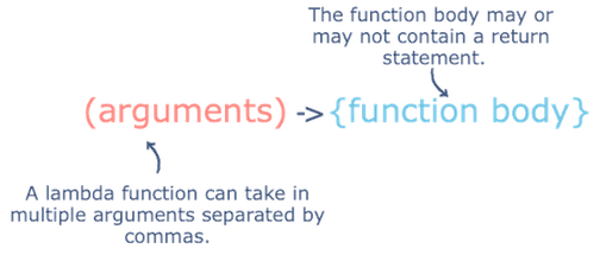
A lambda function is a function declared without a name.

*Лямбда-выражение* (lambda expression, lambda) можно рассматривать как краткую форму анонимной функции, которую разрешено куда-либо передавать (безымянный метод, который служит для передачи поведения из одного места программы в другое так, будто это данные.) У нее отсутствует название, но имеется список параметров, тело, тип возвращаемого значения, а также, возможно, список потенциально генерируемых исключений:

* *Анонимное* — анонимным лямбда-выражение является потому, что у него отсутствует явное название, как у метода.
* *Функция* — лямбда-выражение является функцией потому, что не связано с конкретным классом, подобно методу. Но, как и у метода, у лямбда-выражения есть  
  список параметров, тело, возвращаемый тип, а также, возможно, список потенциально генерируемых исключений.
* *Возможность передачи* — лямбда-выражение можно передать в качестве аргумента в метод или сохранить в переменной.
* *Лаконичное* — нет необходимости писать длинный стереотипный ( «шаблонный») код, как в случае анонимных классов.

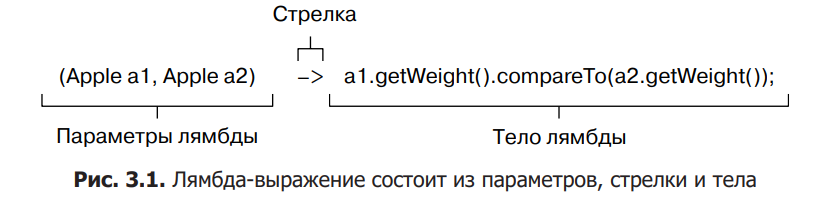
Lambda functions have been incorporated in Java 8 to provide preliminary support for functional programming. They are primarily used to implement the abstract function of a functional interface.

### Синтаксис



*Пример*:

Comparator<Apple> byWeight =  
 (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());



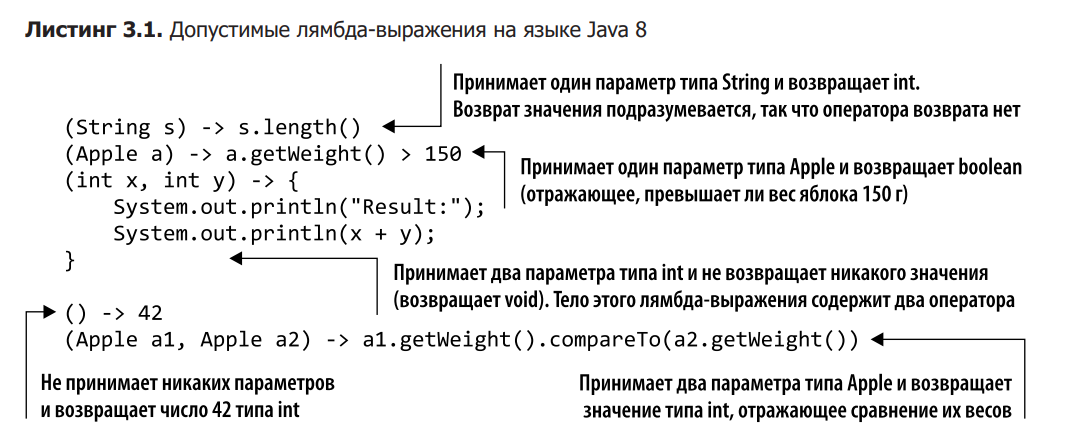
Список параметров в данном случае отражает параметры метода compare интерфейса Comparator — два объекта Apple.

Стрелка (arrow) -> отделяет список параметров от тела лямбда-выражения.

Тело лямбды — сравнение двух объектов Apple по весу. Это выражение считается возвращаемым значением лямбды.

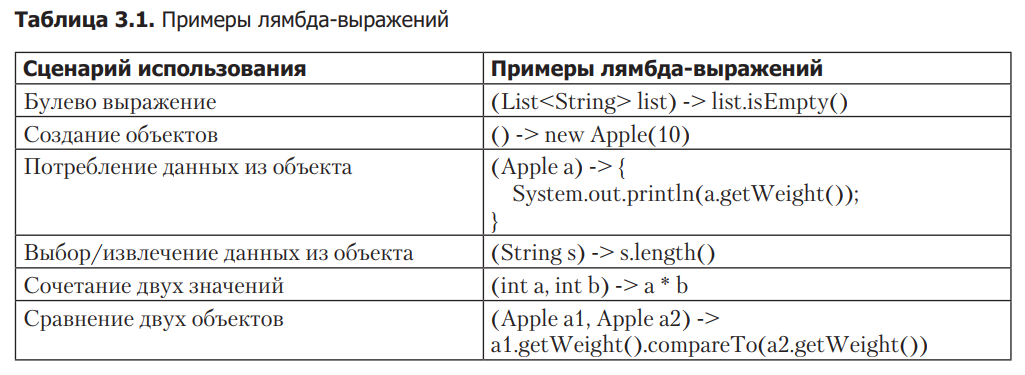
Базовый синтаксис лямбда-выражения имеет вид (обычно называется лямбдой *в форме выражения* (expression-style lambda)):  
 (параметры) -> выражение  
или вид (обратите внимание на фигурные скобки вокруг операторов — такие лямбды часто называют *лямбдами в форме блока* (block-style lambdas)):  
 (параметры) -> { операторы; }

Целевым типом лямбда-выражения называется тип контекста, в котором это выражение встречается, – например, тип локальной переменной, которой оно присваивается, или тип параметра метода, вместо которого оно передается.



BinaryOperator<Long> add = (x, y) -> x + y; Как читать такое лямбда-выражение: В этой строке мы не складываем два числа, а создаем функцию, складывающую два числа. Переменная с именем add, имеющая тип BinaryOperator<Long>, – это не результат сложения чисел, а код, который их складывает.

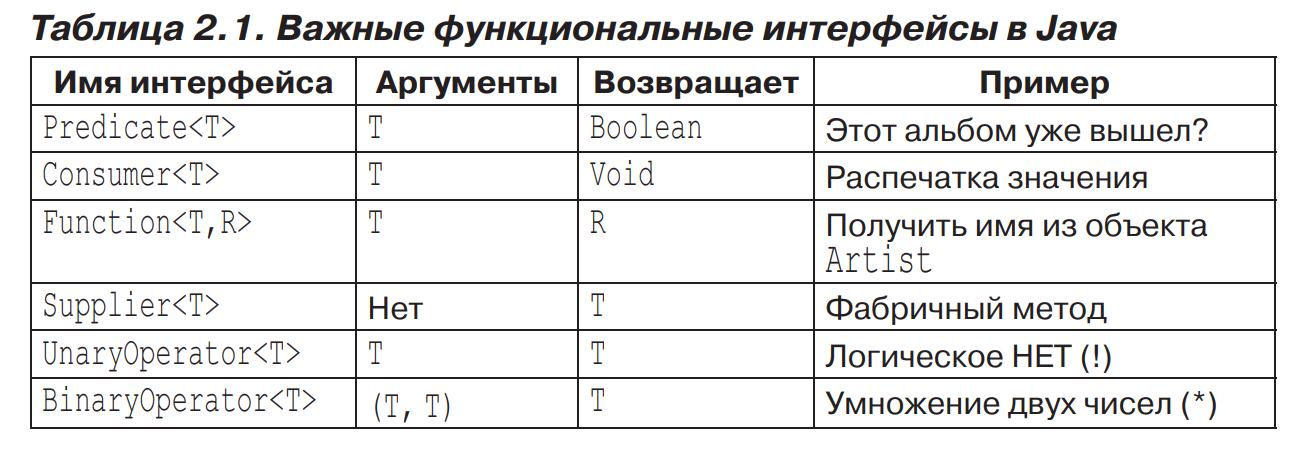
BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y; Можно задать тип параметров явно. Нужно заключить аргументы лямбда-выражения в круглые скобки.



### Функциональные интерфейсы

**Definition**: Функциональным интерфейсом называется интерфейс с единственным абстрактным методом; он используется в качестве типа лямбда-выражения.

**Аннотация @FunctionalInterface** - Этой аннотацией следует снабжать любой интерфейс, который предполагается использовать как функциональный. Причина - в Java есть интерфейсы с единственным методом, которые не предназначены для реализации лямбда-выражениями. Наличие этой аннотации заставляет javac проверить, отвечает ли интерфейс критерию «функциональности».



Consumer — это функциональный интерфейс, который принимает один параметр на вход и не возвращает никаких выходных данных. Используется в случае, если необходимо передать объект на вход и произвести над ним некоторые операции, не возвращая результат. Самый частый случай использования этого интерфейса - это вывод на консоль.

Supplier — это интерфейс, указывающий, что данная реализация является поставщиком какого то результата. Интерфейс Supplier используется тогда, когда на вход не передаются значения, но необходимо вернуть результат.

Интерфейс Predicate принимает на вход значение, проверяет состояние и возвращает boolean значение в качестве результата. Он в основном используется для фильтрации данных из потока (stream) Java.

Интерфейс Function — это интерфейс, который принимает значение в качестве аргумента одного типа и возвращает другое значение. Часто используется для преобразования одного значения в другое. Одним из распространенных вариантов использования этого интерфейса является метод Stream.map.

### Default Methods

*Default method* or *Defender method* is one of the newly added features in Java 8.

**Definition**: They will be used to allow an interface method to provide an implementation used as default in the event that a concrete class doesn't provide an implementation for that method.

**Features**:

* Если в классе с имплементацией метод не реализован, наследуется Default method. Если есть выбор между Default method и методом, переопределенным в классе, предпочтение всегда отдается последнему. Короче говоря: Классу всегда отдается предпочтение перед интерфейсом..
* Нужно ключевое слово default в начале определения метода. Оно говорит javac, что мы хотим добавить метод в интерфейс.
* Методы по умолчанию могут существовать в любом интерфейсе, а не только функциональном.
* в отличие от классов, у интерфейсов не может быть полей, поэтому методы по умолчанию могут модифицировать состояние дочерних классов, только вызывая их методы.

**Example**:

public interface Hello {

default void sayHello() {

System.out.println("Hello");

}

}

public class HelloImpl implements Hello {

}

Hello hello = new HelloImpl();

hello.sayHello(); // This will invoke the default method in interface

# **Optional**

[Guide To Java 8 Optional | Baeldung](https://www.baeldung.com/java-optional)

Класс Optional<T> (java.util.Optional) представляет собой класс-контейнер, отражающий наличие или отсутствие значения. Вместо того чтобы возвращать null — стратегия, часто приводящая к возникновению ошибок, — создатели библиотеки языка Java 8 ввели в нее класс Optional<T>. Optionals allow you to handle null values without writing if statements.

**Creation**

* To create an empty Optional object, we simply need to use its empty() static method:

Optional<String> empty = Optional.empty();

* We can also create an Optional object with the static method of(). Argument passed to the of() method can't be null.

Optional<String> opt = Optional.of(name);

* in case we expect some null values, we can use the ofNullable() method. If we pass in a null reference, it doesn't throw an exception but rather returns an empty Optional object

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

**Checking Value Presence**

* isPresent() возвращает true, если объект Optional содержит значение, и false в противном случае.

assertFalse(opt.isPresent());  
we can do the opposite with the isEmpty method

assertTrue(opt.isEmpty());

**Conditional Action**

ifPresent(Consumer<T> block) выполняет заданный блок кода, если значение есть.

Optional.ofNullable(port).ifPresent(settings::setPort);

Is similar to:

if (port != null) {

settings.setPort(port);

}

**Retrieve Value**

* orElse() method is used to retrieve the value wrapped inside an Optional instance. It takes one parameter, which acts as a default value. The orElse() method returns the wrapped value if it's present, and its argument otherwise:

String name = Optional.ofNullable(nullName).orElse("john");

* The orElseGet() method is similar to orElse(). However, instead of taking a value to return if the Optional value is not present, it takes a supplier functional interface, which is invoked and returns the value of the invocation:

String name = Optional.ofNullable(nullName).orElseGet(() -> "john");

* orElseThrow() method - Instead of returning a default value when the wrapped value is not present, it throws an exception:

String name = Optional.ofNullable(nullName).orElseThrow(IllegalArgumentException::new);

* get() can only return a value if the wrapped object is not null; otherwise, it throws a no such element exception

Optional<String> opt = Optional.of("baeldung");

String name = opt.get();

# Stream

## Описание

Stream API Java 8 позволяет писать код:

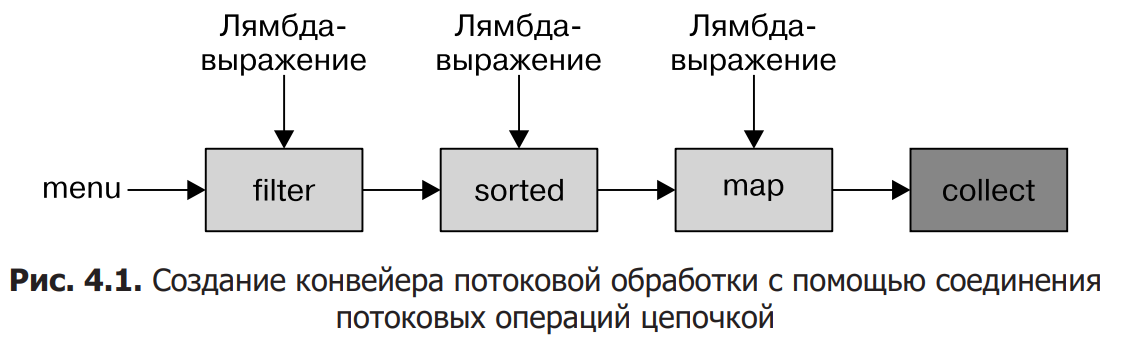
* *декларативный* — более лаконичный и удобочитаемый;
* *удобный для компоновки* — более гибкий;
* *параллелизуемый* — более производительный.

Потоки данных (streams) — новая возможность Java API, позволяющая декларативно описывать операции над коллекциями данных (писать запрос, а не код импровизированной реализации).

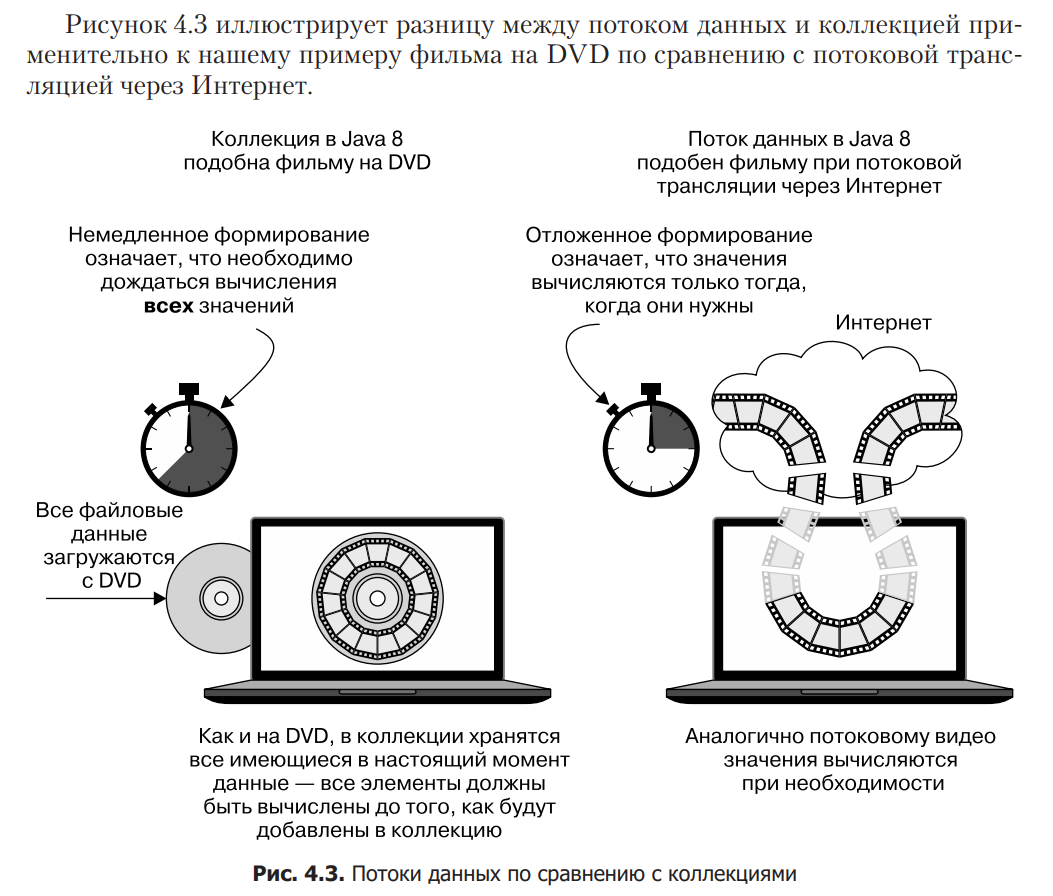
Код написан в декларативном стиле: вы указываете, что хотели бы получить (например, отфильтровать), а не как реализовать операцию (с помощью операторов управления потоком выполнения, например циклов и условных операторов if). Как вы видели в предыдущей главе, такой подход вместе с параметризацией поведения позволяет адаптироваться к изменению требований. Другая сторона этого преимущества: отделение модели многопоточного выполнения от самого запроса. Ведь благодаря тому, что описывается только «рецепт» запроса, его можно выполнять последовательно или параллельно.

Stream – аналог Iterator, только с внутренним итерированием.

Для выражения сложного конвейера обработки данных несколько стандартных операций соединяются цепочкой.



### Сравнение с коллекциями



## Определения

*Поток данных (stream)*: «полученная из источника последовательность элементов, поддерживающая операции обработки данных».

*Последовательность элементов* — подобно коллекции, поток данных обеспечивает интерфейс для упорядоченного набора значений определенного типа. Коллекции, как структуры данных, в основном решают задачи хранения и доступа к элементам с конкретными нюансами временного или пространственного характера (например, сравните ArrayList и LinkedList). Потоки данных же выражают такие вычислительные операции, как вышеупомянутые filter, sorted и map. При работе с коллекциями упор делается на данных; а при работе с потоками данных — на их обработке.

*Источник* — потоки потребляют данные из таких источников, как коллекции, массивы или ресурсы ввода/вывода.

*Операции обработки данных* — потоки данных поддерживают операции в стиле баз данных и распространенные операции над данными из функциональных языков программирования, такие как filter, map, reduce, find, match, sort и т. д. Выполнять потоковые операции можно как последовательно, так и параллельно. Кроме того, у потоковых операций есть две важные характеристики:

*Возможность конвейерной организации* — многие из потоковых операций сами возвращают поток данных, благодаря чему их можно соединять цепочкой в конвейер.

*Внутренняя итерация* — в отличие от коллекций, циклы по которым описываются явным образом, с помощью итераторов, при потоковых операциях итерация производится незаметно для вас.

## Операции - общее

max и min - операции нахождения минимума или максимума. Пример: .min(Comparator.comparing(track -> track.getLength()))

filter — принимает в качестве параметра лямбда-выражение, служащее для исключения из потока определенных элементов. Пример: .fillter(value -> isDigit(value.charAt(0))) .filter(artist -> artist.getName().startsWith("The")) .filter(track -> track.getLength() > 60)

map — принимает в качестве параметра лямбда-выражение для преобразования элемента в другой или извлечения информации. Пример: .map(string -> string.toUpperCase()) .map(artist -> artist.getNationality())

limit — укорачивает список так, чтобы он содержал не более заданного количества элементов

distinct() возвращает новый поток отдельных элементов.

flatMap – позволяет заменить значение объектом Stream и конкатенировать все потоки. Например, имеется объект Stream, содержащий несколько списков чисел, и мы хотим получить все числа из всех списков. Пример: .flatMap(numbers -> numbers.stream())

sorted – сортировка. Пример: .sorted(Comparator.reverseOrder())

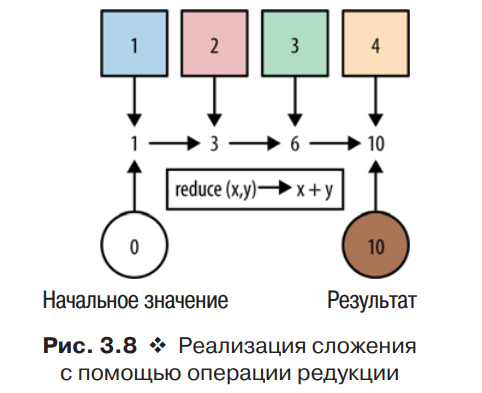
Завершающие:

collect — преобразует поток данных в другую форму. Пример: .collect(toList()); .collect(Collectors.toList()); .collect(toSet());

count() - returns the count of elements in the stream

forEach - used to iterate over a collection in Java and to perform an Consumer action on each element of the Stream. Аналог enhanced for-loop. Пример: .forEach(System.out::println); .forEach(name -> trackNames.add(name));

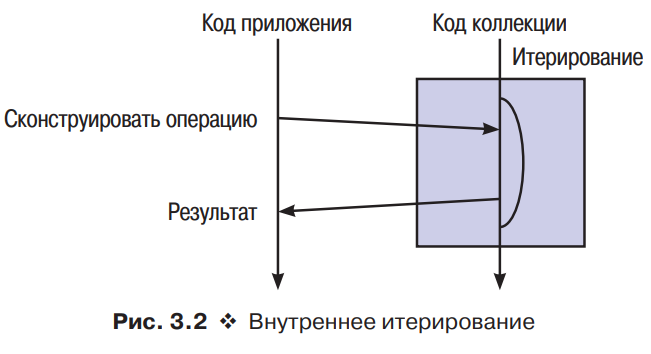
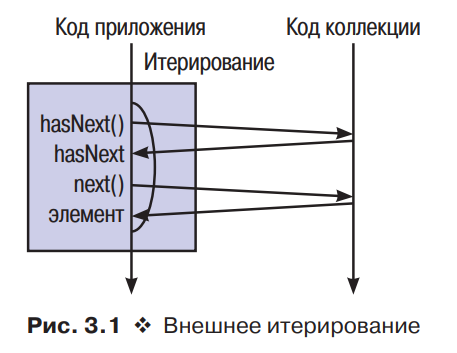
reduce - стоит использовать, когда имеется коллекция значений, а нужно получить единственное значение в качестве результата. Лямбда-выражение, которое называется редуктором, принимает два аргумента и возвращает их сумму. есть две формы reduce: одна принимает начальное значение, а другая не принимает. Если начальное значение опущено, то при первом обращении к редуктору используются первые два элемента потока. Пример: .reduce(0, (acc, element) -> acc + element);



### Внутренняя и внешняя итерация

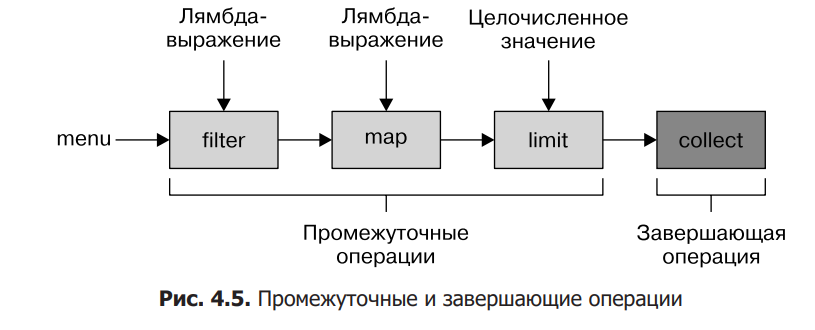
В потоках данных используется внутренняя итерация: организация цикла отделяется от выполнения за счет таких операций, как filter, map и sorted.

При использовании интерфейса Collection организовать цикл должен пользователь (например, с помощью for-each); это называется *внешней итерацией* (external iteration). В библиотеке потоков данных, наоборот, применяется *внутренняя итерация* (internal iteration) — библиотека сама выполняет проход по данным и сохраняет где-то полученные потоковые значения; вам же достаточно указать функцию, которая определяет, что нужно сделать. Подобный способ работы с данными удобен тем, что управление обработкой данных берет на себя Stream API и может неявно оптимизировать выполнение кода. Кроме того, при внутренней итерации Stream API может самостоятельно принять решение о параллельном выполнении кода. При внешней итерации это невозможно, ведь вы должны выполнять цикл однопоточно, пошагово и последовательно.

### **Intermediate и terminal operations**

Потоковые операции, допускающие соединение в цепочку, называются *промежуточными* (**intermediate**), а закрывающие поток данных операции — *завершающими*(**terminal**).



**Промежуточные операции**, такие как filter и map, возвращают поток данных (объект Stream), их можно соединять цепочкой. Они применяются для организации конвейеров операций, но не генерируют сами по себе никаких результатов. Промежуточные операции не выполняют никаких действий до тех пор, пока не будет вызвана завершающая операция, поскольку они носят отложенный характер, т.е. промежуточные - Lazy. Причина в том, что обычно промежуточные операции сливаются воедино и обрабатываются завершающей операцией за один проход. Генерация результата потокового конвейера происходит в результате вызова завершающей операции.

**Завершающие операции**, такие как forEach и count, возвращают непотоковое значение и инициируют выполнение обработки потокового конвейера для возвращения результата. Завершающие операции – Eager.

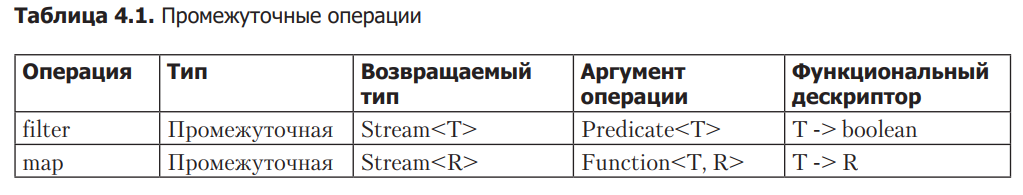
Вычисление элементов потока данных производится по запросу (отложенным образом).

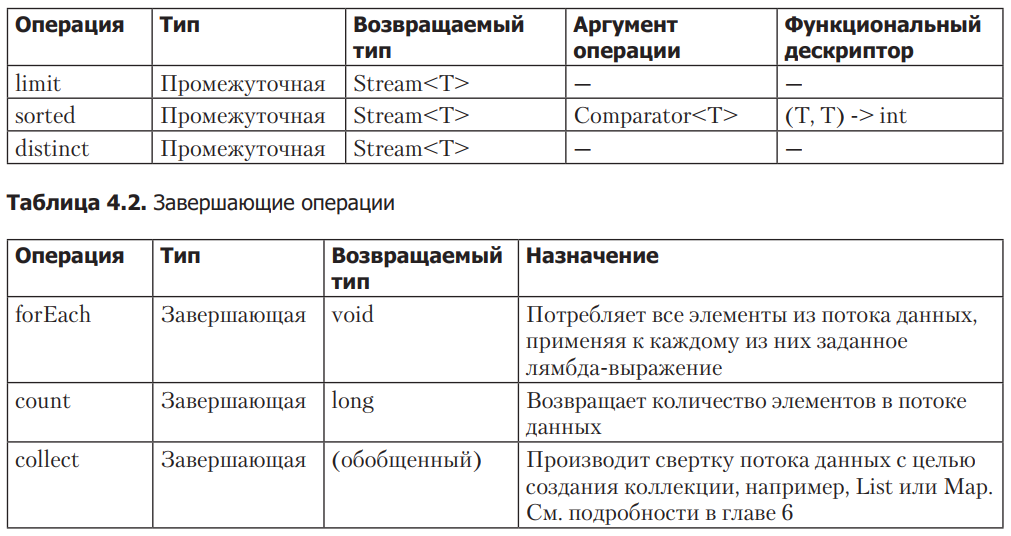
Для работы с потоками данных в целом необходимы три элемента:

1. *источник данных* (например, коллекция), к которому производится запрос;
2. цепочка *промежуточных операций*, составляющая потоковый конвейер;
3. *завершающая операция*, которая выполняет потоковый конвейер и выдает результат.

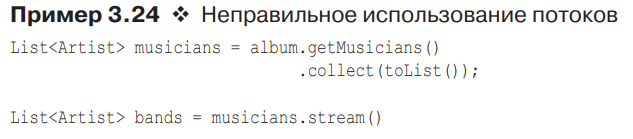
Концепция потокового конвейера напоминает паттерн «Строитель» (builder). Паттерн проектирования «Строитель» описывает цепочку вызовов, предназначенную для задания настроек (в случае потоков — цепочку промежуточных операций), за которой следует вызов метода build (в случае потоков — завершающая операция).

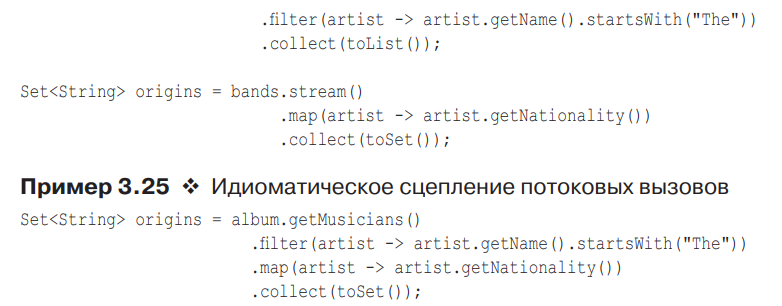
#### Виды операций





### Неправильное использование





Версия из примера 3.24 хуже идиоматической по нескольким причинам.

* Труднее понять, что в ней происходит, потому что доля стереотипного кода, не имеющего отношения к собственно бизнес-логике, выше.
* Она менее эффективна, так как на каждом промежуточном шаге энергично создается новая коллекция объектов.
* Код загроможден лишними переменными, нужными только для хранения промежуточных результатов.
* Операции сложнее автоматически распараллелить

### Функции высшего порядка

Это функция, которая либо принимает другую функцию в качестве аргумента, либо возвращает функцию в качестве значения. Если тип аргумента или возвращаемого значения – функциональный интерфейс, значит, мы имеем функцию высшего порядка. Например, map – функция высшего порядка, потому что ее аргумент mapper – функция.

### Method reference ( :: )

Если нужно создать лямбда-выражение, которое вызывает метод от имени своего параметра, можно использовать сокращенный синтаксис, который позволяет повторно использовать существующий метод. Это просто эквивалент лямбда-выражения, вызов которого приведет к вызову метода. Ссылки на методы можно использовать всюду, где допустимы лямбда-выражения. С помощью такого же сокращенного синтаксиса можно:

* вызывать конструкторы,
* вызывать методы,
* создавать массивы.

Пример:

|  |  |
| --- | --- |
| **Обычный код** | **Ссылка на метод** |
| artist -> artist.getName() | Artist::getName |
| Consumer<String> consumer = str -> System.out.**println**(str); | Consumer<String> consumer = System.out**::**println; |
| (name, nationality) -> new Artist(name, nationality) | Artist::new |
|  | String[]::new |

Ссылки на методы бывают четырех видов:

|  |  |
| --- | --- |
| **Тип** | **Пример** |
| Ссылка на статический метод | ***ContainingClass::staticMethodName*** |
| Ссылка на нестатический метод конкретного объекта | ***containingObject::instanceMethodName*** |
| Ссылка на нестатический метод любого объекта конкретного типа | ***ContainingType::methodName*** |
| Ссылка на конструктор | ***ClassName::new*** |

### Как упорядочены элементы в потоках

В потоке имеется определенный порядок, потому что операции производятся над каждым элементом по очереди. Такой порядок называется порядком поступления (encounter order). Если поток создается из коллекции, в которой определен порядок,

то и порядок поступления в потоке тоже определен. Если изначально порядок не был определен, то и в потоке, созданном на основе такого источника, нет определенного порядка. Операции могут создать порядок поступления там, где его изначально не

было (например, .sorted).

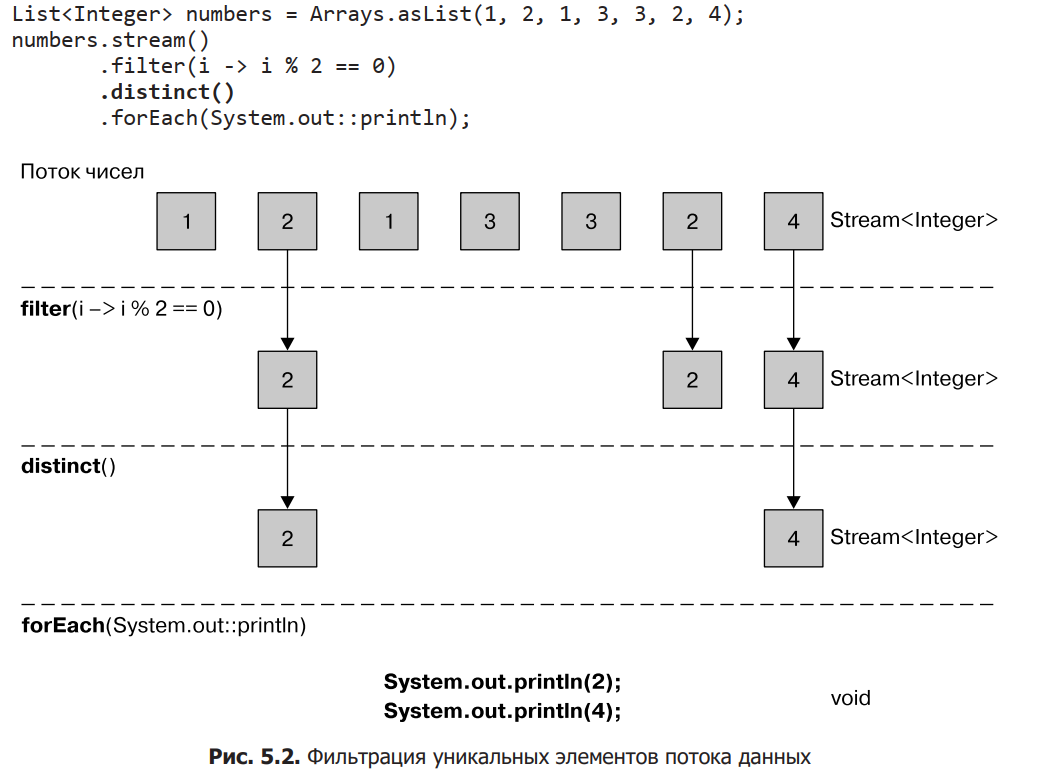
Можно отказаться от упорядочения. Для этого достаточно вызвать метод потока unordered.

## Фильтрация

Фильтрация с помощью предиката

Интерфейс Stream поддерживает метод filter (с которым вы уже хорошо знакомы).  
Он принимает в качестве аргумента *предикат*, или *условие* (функцию, возвращающую boolean), и возвращает поток данных, включающий все соответствующие  
этому предикату элементы.

Потоки также поддерживают метод distinct, который возвращает поток данных,  
содержащий только уникальные элементы (в соответствии с реализацией методов  
hashcode и equals, генерируемых потоком объектов).



## Срез, усечение, пропуск элементов потока данных

Существуют операции для эффективного выбора или отбрасывания элементов с помощью предиката, пропуска нескольких первых элементов потока или усечения потока до заданного размера.

Срез с помощью предиката

Метод takeWhile - позволяет выполнить срез любого потока данных с помощью предиката, и прекращает работу сразу же по обнаружении неподходящего элемента.

Метод dropWhile - отбрасывает  
первые элементы, для которых предикат ложен. Как только результат вычисления  
предиката становится истинным, прекращает работу и возвращает все оставшиеся элементы

Усечение потока данных

метод limit(n). Он возвращает еще один поток, не  
превышающий заданную длину, переданную в limit в виде аргумента. Если поток  
данных упорядоченный, возвращаются первые элементы, но не более чем n.

Пропуск элементов

метод skip(n), возвращающий поток данных с отброшенными первыми n элементами источника.

## Применение функции к каждому из элементов потока данных

Потоки данных поддерживают метод map, принимающий в качестве аргумента  
функцию. Эта функция затем применяется к каждому из элементов потока данных,  
отображая его в новый элемент (это называется термином *«отображение»* (mapping),  
потому что он синонимичен термину *«преобразование»* (transforming), но с дополнительным оттенком смысла «создание новой версии», а не просто «модификация»).

## Схлопывание потоков данных

Метод flatMap При использовании метода flatMap каждый из массивов отображается не в отдельный поток данных, а в *содержимое единого потока*. Все потоки, генерируемые при использовании map(Arrays::stream), сливаются (схлопываются) в единый поток.

## Поиск и сопоставление с шаблоном

**Поиск элементов из набора данных, которые удовлетворяют заданному свойству.**

Stream API предоставляет такие возможности с помощью методов allMatch, anyMatch, noneMatch, findFirst  
и findAny.

Метод anyMatch служит для ответа на вопрос: «Удовлетворяет ли заданному предикату хотя бы один элемент из потока данных?»

Метод allMatch работает аналогично anyMatch, но проверяет, удовлетворяют ли заданному предикату все элементы потока данных.

Противоположностью allMatch является метод noneMatch. Он проверяет, точно ли ни один элемент списка не соответствует заданному предикату.

Метод findAny возвращает произвольный элемент текущего потока данных.

метод findFirst работаюет аналогично методу findAny. Применяется в потоках, где есть (encounter order).

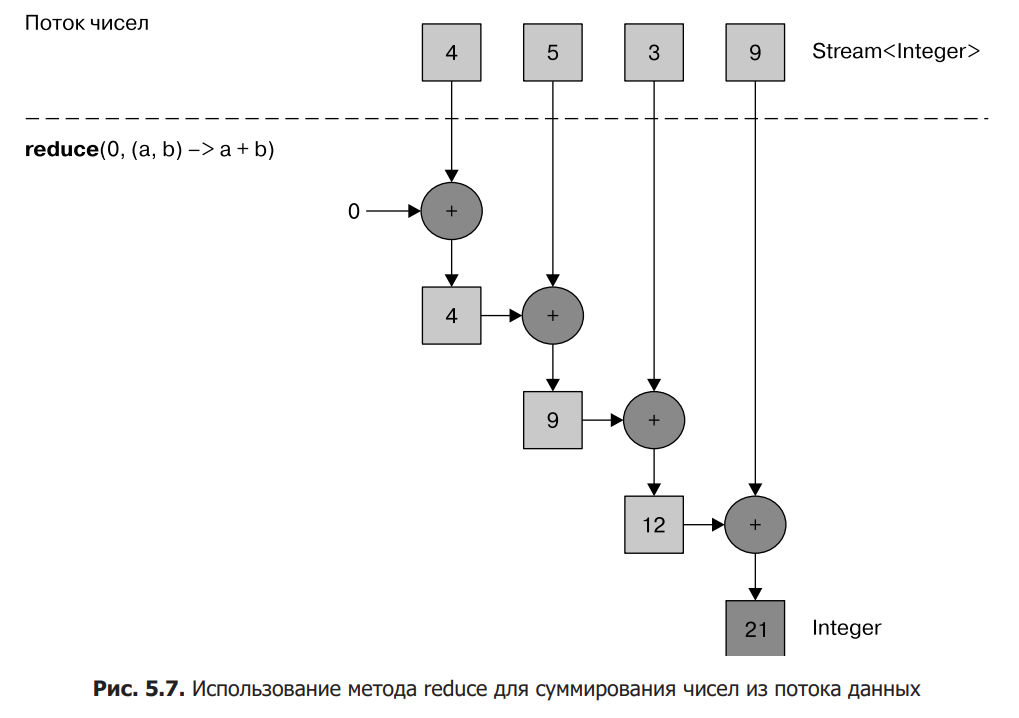
## Reduction (Свертка)

reduction operations, поток данных сворачивается в одно значение

Суммирование элементов

операция reduce

Просуммировать все элементы потока  
данных можно следующим образом:  
int sum = numbers.stream().reduce(0, (a, b) -> a + b);



Максимум и минимум

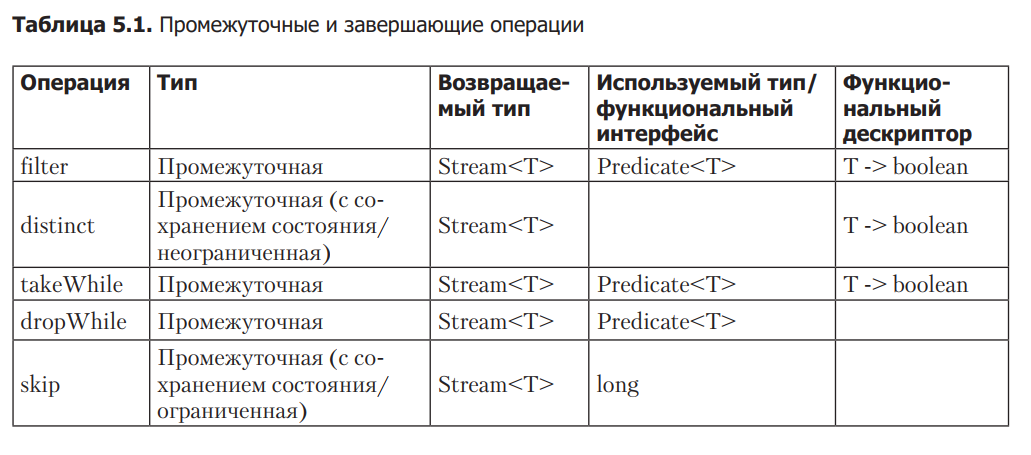
Optional<Integer> max = numbers.stream().reduce(Integer::max);

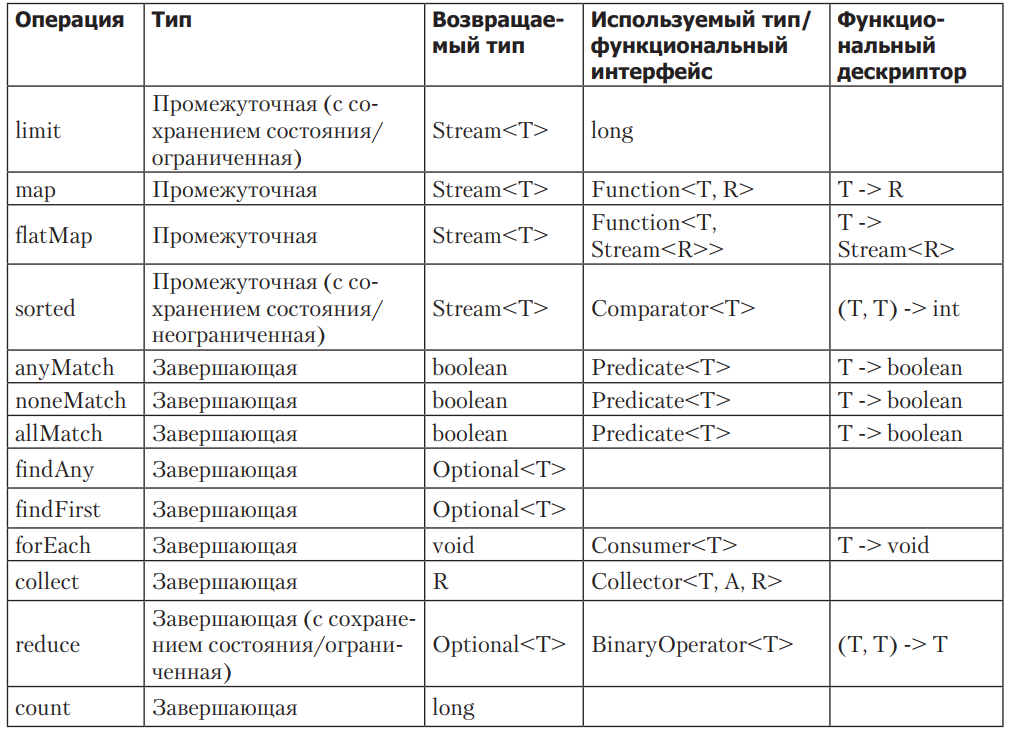
Optional<Integer> min = numbers.stream().reduce(Integer::min);

## **Потоковые операции: с сохранением состояния и без сохранения состояния**

Такие операции, как map и filter, обрабатывают *по одному* элементу из входящего по  
тока данных, выводя в выходной поток *не более одного* результирующего значения. Это  
операции, вообще говоря, *без сохранения состояния*: у них нет внутреннего состояния  
(при условии, что у передаваемого пользователем лямбда-выражения или ссылки на  
метод нет внутреннего изменяемого состояния).

В отличие от них такие операции, как sorted и distinct, на первый взгляд, ведут  
себя подобно map и filter — принимают на входе поток и генерируют другой поток  
(промежуточная операция), — но между ними существует ключевое различие. Как  
для сортировки, так и для удаления дубликатов из потока нужно знать предыдущие  
состояния. Такие операции называются *операциями с сохранением состояния*(stateful operations)





## Версии потоков для простых типов данных

IntStream, DoubleStream и LongStream. Они предназначены для работы соответственно с элементами типов int, double и long,

Отображение в числовой поток данных Самые распространенные методы для преобразования потока данных в специализированные версии — это mapToInt, mapToDouble и mapToLong.

## Преобразование обратно в поток объектов

Для преобразования специализированного потока данных в общий (в котором каждый int упакован в Integer) можно воспользоваться методом boxed

## Числовые диапазоны

В Java 8 появилось два статических метода в интерфейсах IntStream и LongStream, предназначенных для генерации подобных диапазонов значений: range и rangeClosed. Оба они принимают начальное значение диапазона в качестве первого параметра, а конечное значение — в качестве второго параметра. Различие лишь в том, что rangeClosed включает границы диапазона, а range — нет.

## Создание потоков данных

Потоки данных можно создавать на основе явно указываемых значений с помощью статического метода Stream.of

Stream<String> stream = Stream.of("Modern ", "Java ", "In ", "Action");

stream.map(String::toUpperCase).forEach(System.out::println);

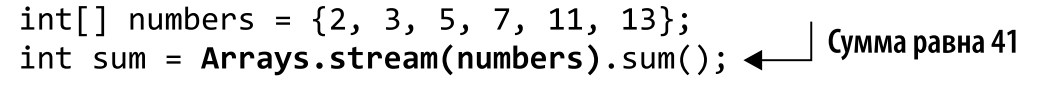
Создание потока данных из объекта, допускающего неопределенное значение В Java 9 появился новый метод для создания потока данных объекта, допускающего неопределенное значение (null).

Stream<String> homeValueStream

= Stream.ofNullable(System.getProperty("home"));

Создание потоков данных из массивов

Поток данных можно создать на основе массива, с помощью статического метода Arrays.stream, принимающего в качестве параметра массив.



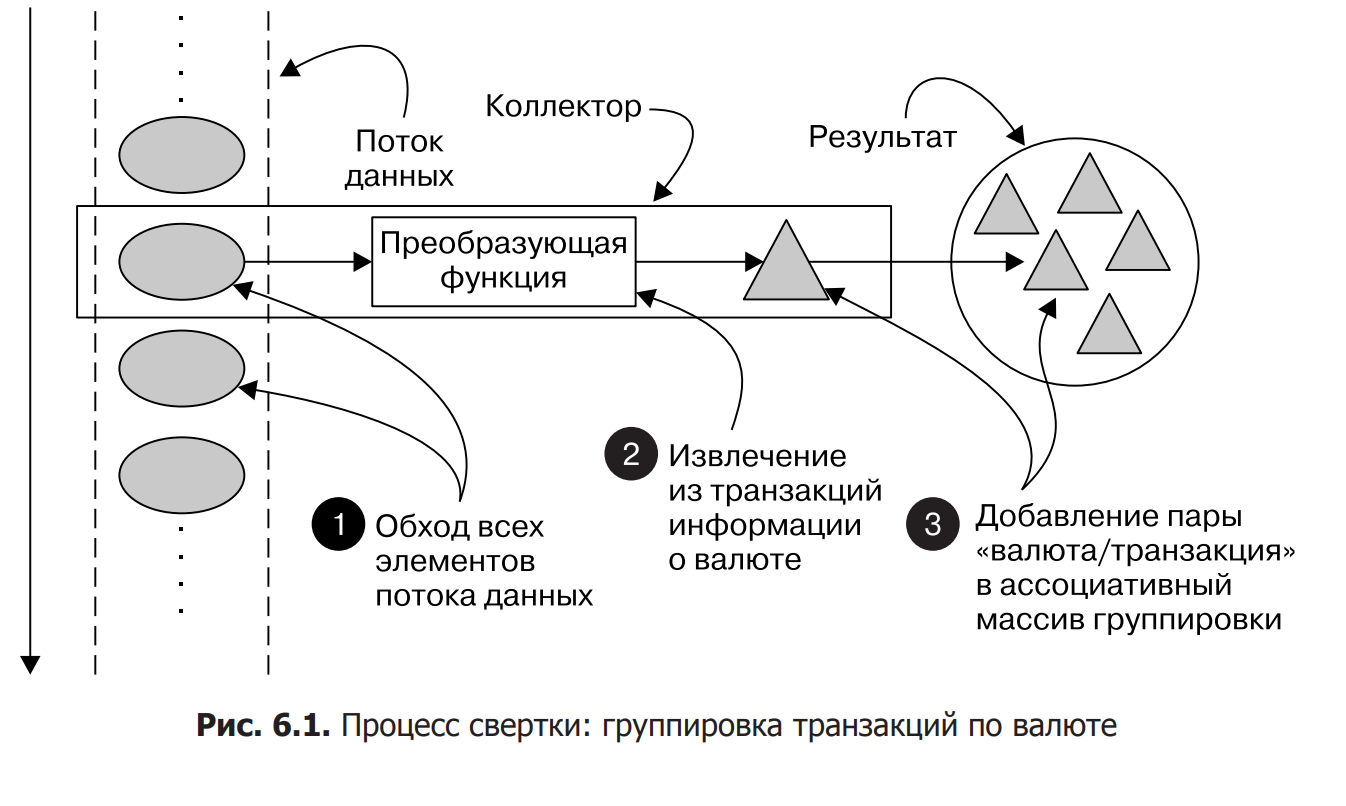
Потоки на основе функций: создание бесконечных потоков

В Stream API есть два статических метода для генерации потока данных на основе функции: Stream.iterate и Stream.generate. С помощью этих двух операций можно создать то, что называется бесконечным потоком данных (infinite stream): поток данных без фиксированного размера, в отличие от потока, созданного из конкретной коллекции.

Обычно имеет смысл использовать для подобных потоков limit(n) во избежание вывода в консоль бесконечного количества значений.

Метод generate Подобно методу iterate, метод generate дает возможность генерировать бесконечный поток значений, вычисляемых по требованию. Но метод generate не применяет функцию последовательно к каждому сгенерированному значению. Для генерации новых значений он получает в качестве параметра лямбда-выражение типа Supplier. Рассмотрим пример его использования: Stream.generate(Math::random) .limit(5) .forEach(System.out::println);

## Collector (Сбор данных с помощью потоков)



**Definition**: Collector – конструкцию общего характера для порождения составных значений из потоков.

Коллекторы позволяют вычислять конечные результаты обработки потоков и являются изменяемым аналогом метода reduce.

Collector может преобразовывать данные, а может и осуществлять тождественное преобразование, не влияющее на данные (например, как в случае с преобразованием toList).

Самый простой и чаще всего используемый коллектор — статический метод toList, собирающий все элементы потока данных в объект типа List.

Все статические фабричные методы класса Collectors уже импортированы с помощью оператора: import static java.util.stream.Collectors.\*

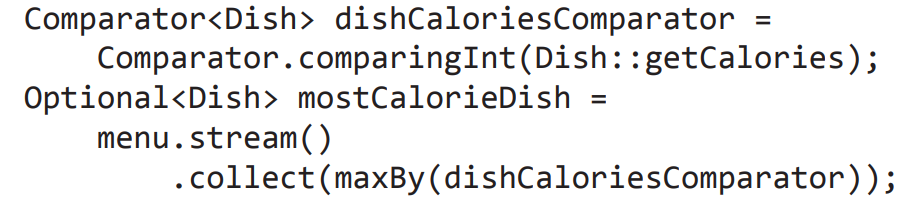
Если мы использовали второй коллектор для обработки части конечного результата, такие коллекторы называются **подчиненными (downstream).** Если коллектор – это рецепт построения конечного значения, то подчиненный коллектор – рецепт построения части этого значения, которая затем используется главным коллектором.

### Виды

**toCollection -** создание конкретной коллекции с помощью метода. Пример: stream.collect(*toCollection*(TreeSet::new));

**Collectors.maxBy и Collectors.minBy** - вычисление максимального или минимального значения в потоке данных.

Они принимают в качестве аргумента объект Comparator для сравнения элементов потока данных.



**Collectors.summingInt –** суммирование. Он принимает на входе функцию, отображающую объект в суммируемое значение типа int, и возвращает коллектор, который при передаче обычному методу collect вычисляет нужные сводные показатели. Например, можно найти общее число калорий в меню с помощью следующего кода: int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));

**Collectors.averagingInt** вместе с его аналогами averagingLong и averagingDouble позволяет вычислить среднее значение набора числовых значений

**joining -** Объединение строк. Пример:

String shortMenu = menu.stream().map(Dish::getName).collect(joining());

String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));

**Collectors.reducing -** Обобщение вычисления сводных показателей с помощью свертки

int totalCalories = menu.stream().collect(reducing( 0, Dish::getCalories, (i, j) -> i + j));

**Методы collect и reduce.** Метод reduce предназначен для группировки двух значений в одно новое, то есть неизменяемой свертки. Напротив, метод collect накапливает требуемый результат путем изменения контейнера.

**groupingBy -** группировка

Map> dishesByType = menu.stream().collect(**groupingBy**(Dish::getType));

**Многоуровневая группировка -** Двумя аргументами фабричного метода Collectors.groupingBy можно воспользоваться для группировки второго уровня. Для этого можно передать внешнему методу groupingBy второй внутренний groupingBy, задав в нем критерий второго уровня для классификации элементов потока

**filtering -** фильтрация группированных элементов

Map> caloricDishesByType = menu.stream() .collect(groupingBy(Dish::getType, filtering(dish -> dish.getCalories() > 500, toList())));

**Mapping -** преобразование

public Map <Artist,List <String >> nameOfAlbums (Stream albums) {

return albums.collect(groupingBy(Album::getMainMusician, mapping(Album::getName, toList())));

}

**partitioningBy -** Разбиение потока на две коллекции значений.

public Map<Boolean, List<Artist>> bandsAndSolo (Stream artists) {

return artists.collect(partitioningBy(artist -> artist.isSolo()));

}

**counting -** Композиция коллекторов. Коллектор **counting** разбивает множество элементов на кластеры. Каждый кластер ассоциируется с ключом, который предоставляет функция классификации: getMainMusician. Затем операция groupingBy использует подчиненный коллектор для сбора данных из каждого кластера и создает отображение Map, содержащее результаты.

public Map<Artist, Long> numberOfAlbums(Stream<Album> albums) {

return albums.collect(groupingBy(album -> album.getMainMusician(),

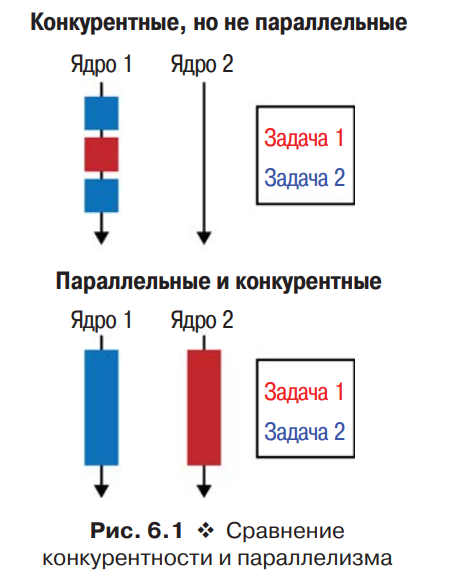
counting()));

}

### Пользовательский коллектор

## Параллелизм и конкурентность

Конкурентность означает, что выполнение двух задач перекрывается во времени. В случае параллелизма выполнение двух задач реально происходит в одно и то же время, как, например, на многоядерном процессоре. Если программа запускает две задачи, которые попеременно получают небольшие кванты времени на одноядерном процессоре, то имеет место конкурентность, но не параллелизм.



Цель параллелизма – уменьшить время работы конкретной задачи, разбив ее на меньшие части, которые работают параллельно.

Параллелизму по данным часто противопоставляют параллелизм на уровне задач, когда каждый поток выполнения решает свою задачу. Пожалуй, самым известным примером параллелизма на уровне задач является контейнер приложений в Java EE. Различные потоки могут не только работать над задачами разных пользователей, но и выполнять совершенно разные задачи, например один поток занимается аутентификацией пользователя, а другой добавляет товар в корзину.

Чтобы распараллелить операцию, реализованную с помощью потоковой библиотеки, достаточно изменить вызов одного метода. Если уже имеется объект Stream, то для превращения его в параллельный нужно вызвать его метод parallel. Если объект Stream создается из Collection, то для получения параллельного потока нужно создавать его с по мощью метода parallelStream.

Параллельная потоковая библиотека особенно хороша для задач, в которых над большим количеством данных производятся простые операции.

Под капотом параллельные потоки реализованы с помощью разветвления и соединения. На этапе разветвления задача рекурсивно разбивается на части. Затем каждая часть обрабатывается параллельно. На стадии соединения частичные результаты объединяются.

Говоря об операциях в потоковом конвейере, применяемых к отдельным порциям данных, мы можем выделить два типа таковых: с состоянием и без состояния. У операций без состояния нет никаких данных, которые нужно было бы сохранять на протяжении всей операции; у операций с состоянием такие данные есть. Если удастся обойтись только операциями без состояния, то производительность параллельных вычислений будет выше. Примеры операций без состояния: map, filter и flatMap; у операций sorted, distinct и limit состояние есть.

Производительность определяется пятью основными факторами: объем данных, структура исходных данных, являются ли данные упакованными или примитивными, число доступных ядер и время, затрачиваемое на обработку одного элемента.

## Тестирование

Поскольку лямбда-выражение не имеет имени, его невозможно вызвать из тестового кода напрямую.

Есть два заслуживающих внимания решения этой проблемы. Первое – рассматривать лямбда-выражение как блок кода внутри объемлющего его метода. Если пойти по этому пути, то мы должны будем тестировать поведение объемлющего метода, а не самого лямбда-выражения. Второе - используйте ссылки на методы. Любой метод, который можно было бы написать в виде лямбда-выражения, можно также записать в виде обычного метода, а затем обращаться к нему в других местах программы с помощью ссылки на метод.

При написании автономных тестов нередко применяются тестовые двойники , которые описывают ожидаемое поведение других компонентов системы. Есть два типа двойников: подставки и заглушки . Часто применяются каркасы типа Mockito , которые автоматически генерируют тестовые двойники.

Метод **peek** - позволяет по очереди просматривать каждое значение и при этом продолжать операции с потоком. Чтобы отлаживать поток поэлементно, по аналогии с пошаговым прохождением цикла мы можем поставить точку остановки в теле метода peek.