**Чистая функция** - это функция, результаты которой зависят только от входных параметров и работа которой не вызывает никаких побочных эффектов, то есть не оказывает никакого внешнего воздействия, кроме возвращаемого значения. Есть только аргументы функции и результат который она возвращает. По сути она не меняет состояние объекта. Изменяет только значение

**Фу́нкция вы́сшего поря́дка** — в программировании функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

Пример 1:

var filteredUsers = CollectionUtils.filter(users, (u) -> u.getFriends().isEmpty());

class CollectionUtils {

private static ArrayList<User> filter(ArrayList<User> users, Predicate<User> fn) {

var filteredUsers = new ArrayList<User>();

for (var user : users) {

if (fn.test(user)) { // запуск лямбда-функции

filteredUsers.add(user);

}

}

return filteredUsers;

}

}

Здесь функцией высшего порядка filter которая принимает u.getFriends().isEmpty().

Что самое интересное при вызове fn.test будет срабатывать функция u.getFriends().isEmpty().

Пример 2:

Это очень простой пример функции высшего порядка. Он принимает функцию и Integer и вычисляет данную функцию с помощью Integer.

[?](https://developerkiselev.blogspot.com/2017/03/java-1.html)

|  |  |
| --- | --- |
| 1  2  3 | public static Integer compute(Function<Integer, Integer> function, Integer value) {      return function.apply(value);  } |

А теперь мы хотим использовать эту функцию, чтобы инвертировать число.

[?](https://developerkiselev.blogspot.com/2017/03/java-1.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class AwesomeClass {      private static Integer invert(Integer value) {          return -value;      }      public static Integer invertTheNumber(){          Integer toInvert = 5;          Function<Integer, Integer> invertFunction = AwesomeClass::invert;          return compute(invertFunction, toInvert);      }    } |

Здесь два интересных момента, первый из них:

[?](https://developerkiselev.blogspot.com/2017/03/java-1.html)

|  |  |
| --- | --- |
| 1 | return function.apply(value); |

Вызов метода объекта-функции, просто используются аргументы и возвращается результат метода. Для нашего примера необходимо написать:

[?](https://developerkiselev.blogspot.com/2017/03/java-1.html)

|  |  |
| --- | --- |
| 1 | return invert(value); |

Второй интересный момент:

[?](https://developerkiselev.blogspot.com/2017/03/java-1.html)

|  |  |
| --- | --- |
| 1 | Function<Integer, Integer> invertFunction = AwesomeClass::invert; |

То, что мы здесь используем называется ссылкой на метод. Мы делаем из метода invert() объект-функцию, используя оператор ::. Это один из двух способов хранить функцию, как объект. Но этот код ничего не упощает. Его можно изменить:

[?](https://developerkiselev.blogspot.com/2017/03/java-1.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class AwesomeClass {      private static Integer invert(Integer value) {          return -value;      }      public static Integer invertTheNumber(){          Integer toInvert = 5;          return invert(toInvert);      }  } |

Такое решение не нуждается в функции compute, да и в самом ФП тоже. Чтобы сделать ФП подходящим в данном примере, нужно представить второй способ хранения функции как объект. Он опирается на анонимные функции или так называемые лямбды

<https://www.youtube.com/watch?v=-d4_gtRH5uY>

Лямда- упрощенная запись для создания экземпляра анонимного класса

Длинная запись

Button button=new Button("Press me!");

button.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

System.out.println("Pressed");

}

});

Короткая запись

Button button=new Button("Press me!");

button.addActionListener(e -> System.out.println("Pressed"));

**Замыкание** – когда лямда завхатывает переменную из внешнего контекста

//захват переменной из внешнего контекста. Это называется замыкание

int i=10;

new Thread(()-> System.out.println("A thread Variable capture... i="+i))

.start();

Ограничение что для анонимного класса что для лямбд- если локальная переменная захватывается, то она должна быть final и не изменяемая

**Java 8 Stream API: шпаргалка для программиста**

Обработка данных — стандартная задача при разработке. Раньше для этого приходилось использовать циклы или рекурсивные функции. С появлением в Java 8

Stream API

 процесс обработки данных значительно ускорился. Этот инструмент языка позволяет описать, как нужно обработать данные, кратко и емко.

**Содержание статьи:**  
[1. Что такое Java Stream API?](https://highload.today/java-stream-api/#1)  
[2. Пример Java Stream API](https://highload.today/java-stream-api/#2)  
[3. Преимущества Java Stream API](https://highload.today/java-stream-api/#3)  
[4. Как создавать стримы](https://highload.today/java-stream-api/#4)  
[5. Методы стримов](https://highload.today/java-stream-api/#5)  
[5.1 Конвейерные](https://highload.today/java-stream-api/#6)  
[5.2 Терминальные](https://highload.today/java-stream-api/#7)  
[5.3 Методы числовых стримов](https://highload.today/java-stream-api/#8)  
[5.4 Еще несколько методов](https://highload.today/java-stream-api/#9)  
[6. Решение задач с помощью Stream API](https://highload.today/java-stream-api/#10)  
[7. Заключение](https://highload.today/java-stream-api/#11)

**Что такое Java Stream API**

Это новый инструмент языка Java, который позволяет использовать функциональный стиль при работе с разными структурами данных.

**Для того что овладеть знаниями инструмента Java Stream API в короткие сроки, можно записаться на и начать использовать в работе в короткие сроки**

Для начала стриму нужен источник, из которого он будет получать объекты. Чаще всего это коллекции, но не всегда. Например, можно взять в качестве источника генератор, у которого заданы правила создания объектов.

Данные в стриме обрабатываются на промежуточных операциях. Например: мы можем отфильтровать данные, пропустить несколько элементов, ограничить выборку, выполнить сортировку. Затем выполняется терминальная операция. Она поглощает данные и выдает результат.

**Stream на примере простой задачи**

Для наглядности посмотрим на примере использование стримов в сравнении со старым решением аналогичной задачи.

Задача — найти сумму нечетных чисел в коллекции.

**Решение с методами стрима:**

**Integer** odd = collection.stream().filter(p -> p % 2 != 0).reduce((c1, c2) -> c1 + c2).orElse(0);

Здесь мы видим функциональный стиль. Без стримов эту же задачу приходится решать через использование цикла:

**Integer** oldOdd = 0;

**for**(**Integer** i: collection) {

**if**(i % 2 != 0) {

oldOdd += i;

}

}

Да, на первый взгляд цикл выглядит более понятным. Но это вопрос опыта взаимодействия со стримами. Очень быстро привыкаешь к тому, что можно обрабатывать данные без использования циклов.

**Преимущества Stream**

Благодаря стримам больше не нужно писать стереотипный код каждый раз, когда приходится что-то делать с данными: сортировать, фильтровать, преобразовывать. Разработчики меньше думают о стандартной реализации и больше времени уделяют более сложным вещам.

**Еще несколько преимуществ стримов:**

* Поддержка слабой связанности. Чем меньше классы знают друг про друга, тем лучше.
* Распараллеливать проведений операций с коллекциями стало проще. Там, где раньше пришлось бы проходить циклом, стримы значительно сокращают количество кода.
* Методы

Stream API

 не изменяют исходные коллекции, уменьшая количество побочных эффектов.

Даже сложные операции по обработке данных благодаря

Stream API

 выглядят лаконично и понятно. В общем, писать становится удобнее, а читать — проще.

**Как создавать стримы**

**В таблице ниже — основные способы создания стримов.**

|  |  |  |
| --- | --- | --- |
| **Источник** | **Способ** | **Пример** |
| Коллекция | collection.stream() | Collection<String> collection = Arrays.asList("f5", "b6", "z7");  Stream<String> collectionS = collection.stream(); |
| Значения | Stream.of(v1,… vN) | Stream<String> valuesS = Stream.of("f5", "b6", "z7"); |
| Примитивы | IntStream.of(1, … N) | IntStream intS = IntStream.of(9, 8, 7); |
| DoubleStream.of(1.1, … N) | DoubleStream doubleS = DoubleStream.of(2.4, 8.9); |
| Массив | Arrays.stream(arr) | String[] arr = {"f5","b6","z7"};  Stream<String> arrS = Arrays.stream(arr); |
| Файл — каждая новая строка становится элементом | Files.lines(file\_path) | Stream<String> fromFileS = Files.lines(Paths.get("doc.txt")) |
| Stream.builder | Stream.builder().add(...)....build() | Stream.builder().add("f5").add("b6").build() |

Есть и другие способы. Например, с перечисляемыми типами можно создавать стримы не только из существующих последовательностей, но еще и задавать

range

, причем сразу двух типов:

IntStream rangeS = IntStream.range(9, 91); // 9 … 90

IntStream rangeS = IntStream.rangeClosed(9, 91); // 9 … 91

Стримы можно создавать не только из файлов, но и из списка объектов какой-либо директории или файлов, находящихся в какой-либо части дерева файловой системы.

Если требуется параллельный стрим, то просто напишите

collection.parallelStream()

.

Почти все перечисленные способы создания потоков не выглядят необычно для тех, кто привык постоянно работать с коллекциями. Но есть еще два интересных варианта:

Stream.iterate

 и

Stream.generate

. Их предназначение — бесконечные стримы.

В

Stream.iterate

 мы задаем начальное значение, а также указываем, как будем получать следующее, используя предыдущий результат:

Stream<Integer> iterStream = Stream.iterate(1, m -> m + 1)

Stream.generate

 позволяет бесконечно генерировать постоянные и случайные значения, которые соответствуют указанному выражению.

Stream<String> generateStream = Stream.generate(() -> "f5")

Если хотите узнать больше об этих и других способах, читайте [документацию Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html).

**Методы стримов**

В

Java 8 Stream API

 доступны методы двух видов — конвейерные и терминальные. Кроме них можно выделить ряд спецметодов для работы с числовыми стримами и несколько методов для проверки параллельности/последовательности. Но это формальное разделение.

Конвейерных методов в стриме может быть много. Терминальный метод — только один. После его выполнения стрим завершается.

Пока вы не вызвали терминальный метод, ничего не происходит. Все потому, что конвейерные методы ленятся. Это значит, что они обрабатывают данные и ждут команды, чтобы передать их терминальному методу. Мы рекомендуем не лениться как конвейерные методы, а чтобы иметь полноценные знания для работы с Java Stream API.

**Курс QA Manual (Тестування ПЗ мануальне).**

Навчіться знаходити помилки та контролювати якість сайтів та додатків.

[**Записатися на курс**](https://powercode.academy/uk/online-courses/course-qa/?utm_campaign=highload)

**Конвейерные**

|  |  |  |
| --- | --- | --- |
| **Метод** | **Что сделает** | **Использование** |
| filter | отработает как фильтр, вернет значения, которые подходят под заданное условие | collection.stream().filter(«e22»::equals).count() |
| sorted | отсортирует элементы в естественном порядке; можно использовать  Comparator | collection.stream().sorted().collect(Collectors.toList()) |
| limit | лимитирует вывод по тому, количеству, которое вы укажете | collection.stream().limit(10).collect(Collectors.toList()) |
| skip | пропустит указанное вами количество элементов | collection.stream().skip(3).findFirst().orElse("4") |
| distinct | найдет и уберет элементы, которые повторяются; вернет элементы без повторов | collection.stream().distinct().collect(Collectors.toList()) |
| peek | выполнить действие над каждым элементом элементов, вернет стрим с исходными элементами | collection.stream().map(String::toLowerCase).peek((e) -> System.out.print("," + e)). collect(Collectors.toList()) |
| map | выполнит действия над каждым элементом; вернет элементы с результатами функций | Stream.of("3", "4", "5") .map(Integer::parseInt) .map(x -> x + 10) .forEach(System.out::println); |
| mapToInt  ,  mapToDouble  ,  mapToLong | Сработает как  map  , только вернет числовой  stream | collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray() |
| flatMap  ,  flatMapToInt  ,  flatMapToDouble  ,  flatMapToLong | сработает как  map  , но преобразует один элемент в ноль, один или множество других | collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]:: |

**Терминальные**

|  |  |  |
| --- | --- | --- |
| **Метод** | **Что сделает** | **Использование** |
| findFirst | вернет элемент, соответствующий условию, который стоит первым | collection.stream().findFirst().orElse("10") |
| findAny | вернет любой элемент, соответствующий условию | collection.stream().findAny().orElse("10") |
| collect | соберет результаты обработки в коллекции и не только | collection.stream().filter((s) -> s.contains("10")).collect(Collectors.toList()) |
| count | посчитает и выведет, сколько элементов, соответствующих условию | collection.stream().filter("f5"::equals).count() |
| anyMatch | **True**  , когда хоть один элемент соответствует условиям | collection.stream().anyMatch("f5"::equals) |
| noneMatch | **True**  , когда ни один элемент не соответствует условиям | collection.stream().noneMatch("b6"::equals) |
| allMatch | **True**  , когда все элементы соответствуют условиям | collection.stream().allMatch((s) -> s.contains("8")) |
| min | найдет самый маленький элемент, используя переданный сравнитель | collection.stream().min(String::compareTo).get() |
| max | найдет самый большой элемент, используя переданный сравнитель | collection.stream().max(String::compareTo).get() |
| forEach | применит функцию ко всем элементам, но порядок выполнения гарантировать не может | set.stream().forEach((p) -> p.append("\_2")); |
| forEachOrdered | применит функцию ко всем элементам по очереди, порядок выполнения гарантировать может | list.stream().forEachOrdered((p) -> p.append("\_nv")); |
| toArray | приведет значения стрима к массиву | collection.stream().map(String::toLowerCase).toArray(String[]::**new**); |
| reduce | преобразует все элементы в один объект | collection.stream().reduce((c1, c2) -> c1 + c2).orElse(0) |

Совет: подробнее изучите метод

collect

. Он позволяет гибко управлять преобразованием значений в разные типы: коллекции, массивы,

map

. Делается это благодаря статистическим методам

Collectors

.

**Вот несколько интересных примеров:**

* toList

 — стрим приводится к списку;

* toCollection

 — получаем коллекцию;

* toSet

 — получаем множество;

* toConcurrentMap

,

toMap

 — если нужен

map

;

* summingInt

,

summingDouble

,

summingLong

 — если требуется получить сумму чисел;

* averagingInt

,

averagingDouble

,

averagingLong

 — если хотите вернуть среднее значение;

* groupingBy

 — если необходимо разбить коллекцию на части.

Это не все статистические методы

Collectors

. Другие возможности с подробным описанием [смотрите в документации](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html). Помимо тех

Collectors

, которые определены в документации, можно использовать собственноручно созданные, кастомные варианты.

**Методы числовых стримов**

Это специальные методы, которые работают только со стримами с числовыми примитивами.

|  |  |  |
| --- | --- | --- |
| **Метод** | **Что сделает** | **Использование** |
| sum | вернет сумму чисел, представленных в коллекции | collection.stream().mapToInt((s) -> Integer.parseInt(s)).sum() |
| average | вернет среднее арифметическое | collection.stream().mapToInt((s) -> Integer.parseInt(s)).average() |
| mapToObj | преобразует числовой стрим в объектный | intStream.mapToObj((id) -> **new** Key(id)).toArray() |

**Еще несколько методов**

Напоследок посмотрим еще несколько полезных методов, которые помогают управлять последовательными и параллельными стримами — как минимум быстро их определять.

|  |  |  |
| --- | --- | --- |
| **Метод** | **Что сделает** | **Использование** |
| isParallel | скажет, параллельный стрим или нет | someStream.isParallel() |
| parallel | сделает стрим параллельным или вернет сам себя | someStream = stream.parallel() |
| sequential | сделает стрим  последовательным или вернет сам себя | someStream = stream.sequential() |

Стримы могут быть последовательными и параллельными. Первые выполняются в текущем потоке, вторые используют общий пул

ForkJoinPool.commonPool()

. В параллельном стриме элементы разделяются на группы. Их обработка проходит в каждом потоке по отдельности. Затем они снова объединяются, чтобы вывести результат. С помощью методов

parallel

 и

sequential

 можно явно указать, что нужно сделать параллельным, а что — последовательным.

**Не рекомендуется применять параллельность для выполнения долгих операций (например, извлечения данных из базы), потому что все стримы работают с общим пулом.** Долгие операции могут остановить работу всех параллельных стримов в

Java Virtual Machine

 из-за того, что в пуле не останется доступных потоков.

Чтобы избежать такой проблемы, используйте параллельные стримы только для коротких операций, выполнение которых занимает миллисекунды, а не секунды и тем более минуты.

В

Stream API

 по умолчанию скрыта работа с потоконебезопасными коллекциями, разделение на части и объединение элементов. Это отличное решение. Разработчику остается только выбирать нужные методы и следить за тем, чтобы не было зависимостей от внешних факторов.

**Решение задач с помощью Stream API**

Давайте изучим на практике, как работать с разными методами

Stream API

, на примере несложных задач.

Допустим, у нас есть коллекция состоящая из строк.

Arrays.asList(«Highload», «High», «Load», «Highload»)

. Применим к ней разные методы.

Посчитаем, сколько раз объект «

High

» встречается в коллекции:

collection.stream().filter(«High»::equals).count() // 1

А теперь посмотрим, какой элемент в коллекции находится на первом месте. Если мы получили пустую коллекцию, то пусть возвращается

0

:

collection.stream().findFirst().orElse(«0») // Highload

Благодаря методам

filter

 и

findFirst

 можно находить элементы, равные заданным в условии:

collection.stream().filter(«Load»::equals).findFirst().get() // Load

Допустим, нам нужно вернуть последний элемент. Получили пустую коллекцию — пусть возвращается

0

. Используем метод

skip

, чтобы пропустить заданное количество элементов. А

findFirst

, чтобы вывести первое встреченное совпадение:

collection.stream().skip(collection.size() — 1).findFirst().orElse(«0») // Highload

С помощью метода

skip

 можно искать элементы по порядку. Например, пропустить первый и вывести второй:

collection.stream().skip(1).findFirst().get() // High

Можно также использовать методы

skip

 и

limit

, чтобы явно задавать, сколько элементов нужно пропустить, а сколько — вернуть. Полученные значения соберем в массив:

collection.stream().skip(1).limit(2).toArray()// [High, Load]

Аналогичным образом можно поиграться с методами

min

 и

max

. Пусть у нас будет коллекция строк вида

Arrays.asList("f10", "f15", "f2", "f4")

. Найти самый маленький элемент не составит труда:

collection.stream().min(String::compareTo).get() // f2

С максимальным значением тоже все очень просто:

collection.stream().max(String::compareTo).get() // f15

Посмотрим несколько примеров работы сортирующих методов. Используем ту же коллекцию строк, что и выше —

Arrays.asList("f10", "f15", "f2", "f4", "f4")

. Единственное отличие — теперь в нем появился дубликат.

Первая задача — отсортировать строки в алфавитном порядке и добавить их в массив:

collection.stream().sorted().collect(Collectors.toList()) // [f2, f4, f4, f10, f15]

А вот чуть более интересное задание — нужно выполнить сортировку в обратном алфавитному порядке и удалить дубликаты. В массиве должны оказаться только уникальные значения:

collection.stream().sorted((o1, o2) -> -o1.compareTo(o2)).distinct().collect(Collectors.toList())

Здесь мы используем не только

sorted

 для сортировки, но и метод

distinct

 для удаления неуникальных значений при обработке коллекции.

**Задачи про группу студентов**

Теперь давайте посмотрим чуть более комплексные, взрослые задачи. Например, у нас есть коллекция, которая имеет следующий вид:

Arrays.asList( new Student("Дмитрий", 17, Gender.MAN), new Student("Максим", 20, Gender.MAN), new Student("Екатерина", 20, Gender.WOMAN), new Student("Михаил", 28, Gender.MAN)

Мы можем обрабатывать эти данные используя методы

Stream API

. Например, давайте найдем средний возраст студентов мужского пола. Естественно, это может быть не целочисленное значение.

**Сначала создадим коллекцию студентов и опишем их:**

Collection<Student> students = Arrays.asList(

new Student("Дмитрий", 17, Gender.MAN),

new Student("Максим", 20, Gender.MAN),

new Student("Екатерина", 20, Gender.WOMAN),

new Student("Михаил", 28, Gender.MAN)

);

**private** **enum** Gender {

MAN,

WOMAN

}

**private** **static** **class** Student {

**private** **final** **String** name;

**private** **final** **Integer** age;

**private** **final** Sex gender;

**public** Student(**String** name, **Integer** age, Gender gender) {

**this**.name = name;

**this**.age = age;

**this**.gender = gender;

}

**public** **String** getName() {

**return** name;

}

**public** **Integer** getAge() {

**return** age;

}

**public** Gender getGender() {

**return** gender;

}

@Override

**public** **String** toString() {

**return** "{" +

"name='" + name + '\'' +

", age=" + age +

", gender=" + gender +

'}';

}

@Override

**public** **boolean** equals(Object o) {

**if** (**this** == o) **return** **true**;

**if** (!(o instanceof Student)) **return** **false**;

Student student = (Student) o;

**return** Objects.equals(name, student.name) &&

Objects.equals(age, student.age) &&

Objects.equals(gender, student.gender);

}

@Override

**public** **int** hashCode() {

**return** Objects.hash(name, age, gender);

}

}

Теперь мы можем использовать методы стримов для обработки этой коллекции. Посчитаем средний возраст, используя метод

average

:

students.stream().filter((s) -> s.getGender() == Gender.MAN). mapToInt(Student::getAge).average().getAsDouble() // 21,7

Получилась немного странная группа студентов мужского пола, но средний возраст вполне себе студенческий. Что мы здесь сделали:

1. Отфильтровали студентов по половому признаку.
2. Выбрали для обработки их возраст.
3. Вернули среднее арифметическое с помощью метода average.

Теперь давайте посмотрим, кому из наших студентов грозит получение повестки в этом году при условии, что призывной возраст установлен в диапазоне от 18 до 27 лет.

students.stream().filter((s)-> s.getAge() >= 18 && s.getAge() < 27 && s.getGender() == Gender.MAN).collect(Collectors.toList()) // [{name='Максим', age=20, gender=MAN}]

Не повезло Максиму. Он мужского пола, ему 20 лет. Другие студенты мужского пола не подходят под условие

s.getAge() >= 18 && s.getAge() < 27

. Один младше 18 лет, другой — старше 27 лет. Единственная студентка в нашей выборке не подходит под условие

s.getGender() == Gender.MAN

. При этом по возрасту она вполне проходит по первым двум условиям. Но так как используется оператор

&&

, в итоге мы получаем

**False**

.

**Задачи на поиск в строке**

Представим, что у нас есть большое количество логинов сотрудников. Нам нужно создать программу, которая будет выводить логины, начинающиеся на определенную букву. И использовать для решения этой задачи

Stream API

.

**Вот как будет выглядеть код этой программы:**

**import** *java.util.stream.*\*;

**import** *java.util.*\*;

**import** *java.util.function.*\*;

**public** **class** DemoStreamAPI {

**public** **static** **void** TrainStream() {

Scanner scanner = new Scanner(System.in);

**String** s;

ArrayList<**String**> ALL = new ArrayList<**String**>();

System.out.println("Введите имя: ");

**while** (**true**) {

System.out.print("имя = ");

s = scanner.nextLine();

**if** (s.equals("")==**true**)

**break**;

ALL.add(s);

}

System.out.println();

System.out.println("ALL = " + ALL); // Выводим массив введенных имен

Predicate<**String**> fn;

fn = (str) -> {

**if** (str.charAt(0)=='A')

**return** **true**;

**return** **false**;

}; // Определяем, что нам нужны имена, начинающиеся на 'A'

Stream<**String**> stream = ALL.stream(); // Конвертация массива в поток строк

Stream<**String**> resStream = stream.filter(fn); // Получаем список, отфильтрованный по предикату

System.out.println("count = " + resStream.count()); // Выводим количество имен

}

**public** **static** **void** main(**String**[] args) {

TrainStream();

}

}

Программа предлагает ввести имена сотрудников. Все они сохраняются в массив

ALL

 без предварительной обработки. Чтобы остановить ввод имен, нужно ввести пустую строку.

Сначала на экране выведется массив со всеми введенными именами. Чтобы отфильтровать их, нужно добавить условие. В нашем случае это будет первая буква — например, ‘

a

‘.

Для фильтрации используется метод

filter

. Затем данные записываются в результирующий стрим. Чтобы вывести количество имен подходящих под заданное ранее условие, мы используем метод

count

. Вот такое простое и элегантное решение.

**Заключение**

Stream

 в Java дает разработчикам удобные инструменты для обработки данных в коллекциях. Методы позволяют проще обрабатывать объекты и писать меньше кода. Чтобы научиться работать еще более эффективно с Java Stream API рекомендуем вам пройти у профессионалов.

Но стрим — не серебряная пуля. Опытные разработчики собрали несколько советов по их использованию:

1. Стримы можно не использовать, если задача решается красиво и эффективно без них.
2. Не обязательно сохранять

stream

 в переменную. Достаточно использовать цепочку вызовов методов.

1. Старайтесь ограничить или почистить стрим от лишних элементов, прежде чем выполнять преобразования.
2. Используйте параллельные потоки разумно. В отдельных случаях разбиение, обработка в разных потоках и последующее объединение данных могут быть дороже, чем работа в одном потоке.

Чтобы закрепить свои знания, посмотрите это наглядное пособие по

Java Stream API

. В нем рассматривается функциональный подход к работе с коллекциями. В видео есть примеры создания стримов из объектов файловой системы, примитивов, объектов, а также примеры использования конвейерных и терминальных методов: