Stream — последовательность элементов (потенциально бесконечных) с возможностью применять к ней простые или сложные многоэтапные преобразования. Позволяет это делать без циклов и условных операторов, то есть программа будет иметь простую линейную структуру.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.util.stream;  public **interface** Stream**<T>**  extends BaseStream<T, Stream<T>> {  // MANY methods  } |

IntS, LongS и DoubleS.

В отличие от итератора он представляет собой не просто средство обхода элементов, а средство описания алгоритма, обработки и преобразования последовательности элементов.

6.4.2 Пример и общая схема

|  |  |
| --- | --- |
| Коллекция | Стрим |
| конечна | потенциальна бесконечна |
| доступ к элементам | нет |
| менять, добавлять, удалять | трансформация не влияет на источник |

Этапы работы со стримом: порождение, промежуточные операции, терминальные операции. Если стрим выделял какие‑то системные ресурсы, то его надо закрыть: метод .close() или блок try с ресурсами.

(1) Порождение Stream'а

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Set<String> vocabulary = ...;  Stream<String> stream1 = vocabulary.stream();  BufferedReader reader = ...;  Stream<String> stream2 = reader.lines();  Path path = ...;  Stream<Path> stream3 = Files.list(path);  Stream<Path> stream4 = Files.walk(path);  IntStream chars = "hello".chars(); |

Стрим можно получить из:  
1) любой коллекции (stream());  
2) BufferedReader (lines()) (вернет поток строчек строк из данного потока символов);  
3) директории на диске (list() (содержимое директории на один уровень) и walk() (рекурсивно обойдет поддиректории)).

2-3 нужно закрывать (try с ресурсами)

4) строчки (chars()) (будет содержать символы строки — IntStream, т.к. нет CharStream).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | DoubleStrem randomNumbers =  DoubleSream.generate(Math::random);  IntStream integers =  IntStream.iterate(0, n -> n + 1);  IntStream smallIntegers =  IntStream.range(0, 100);  IntStream smallIntegers2 =  IntStream.rangeClosed(0, 100); |

Стримы можно порождать динамически:  
1) генерировать (generate()) при помощи Supplier (Math::random). (Supplier — интерфейс с единственным методом get(); метод должен последовательно один за другим возвращать элементы последовательности);  
2) итерирование (iterate()) функции (5: n -> n + 1) принимает число (первый элемент последовательности) и аргумент функции (вычисляет следующий элемент последовательности по предыдущему).

3) (речь о целых числах) получить диапазон чисел в виде стрима (range() и rangeClosed()). (первый возвращает стрим, состоящий из элементов от начала до конца диапазона не включительно, а второй — до конца диапазона включительно)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | IntStream combainedStream =  IntStream.concat(stream1, strem2);  IntStream empty = IntStream.empty();  double[] array = ...;  DoubleStream streamFromArray =  Arrays.stream(array);  IntStream streamOfElements =  IntStream.of(2, 4, 5, 6, 8, 10); |

Стрим можно получить:  
1) конкатенацией (concat()) из других стримов;  
2) взять пустой стрим (empty());  
3) из массива (Arrays.stream());  
4) просто взять и пересилить все элементы стрима явно.

(2) Промежуточные операции

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | IntStream stream = ...;  stream.filter(n -> n > 100)  .mapToObj(Integer::toString)  .flatMapToInt(s -> s.chars())  .distinct()  .sorted()  .skip(3)  .limit(2); |

1) filter() (фильтрация) принимает Predicate. (В данном случае мы проверяем, что число больше 100. Соответственно дальше в этом стриме будут только числа, удовлетворяющие предикату)

2) map() принимает Function, которая из каждого элемента стрима делает какой‑то новый элемент потенциально другого типа (в данном случае мы каждое число конвертируем в строку при помощи метода Integer toString);

3) flatMap() принимает Function, возвращающую Stream (в данном случае мы передаем туда функцию, которая для каждой строки возвращает стрим ее символов)

И дальше применяет эту Function к каждому элементу стрима (получается соответственно набор стримов и все их конкотентинирует в один стрим, на выходе у нас опять обычный плоский IntStream);

5) distinct() убирает из стрима дубликаты элементов;

6) sorted() преобразует стрим, что элементы в нем начинают идти по порядку, по возрастанию (в случае, если у нас стирм не примитивных типов, а объектов, то туда можно передать компаратор);

7) skip() (13) позволяет пропустить некоторое количество первых элементов. В данном случае нас не интересуют первые 3 (13);

8) limit() (15) ограничивает оставшиеся элементы заданным количеством (в данном случае мы получим стрим из 2‑х элементов);

9) peek() принимающая Concumer. (позволяет подсмотреть какие именно элементы летают на 8, 14 этапе обработки стрима. Для отладки в качестве Concumer удобно передавать System.out::println.)

(3) Терминальные операции

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | IntStream stream1 = ...;  stream1.forEach(System.out::println);  IntStream stream2 = ...;  OptionalInt result = stream2.findFirst();  Stream<String stream3 = ...;  boolean allStringsAreAtLeast10Chars =  stream3.allMatch(  s -> s.length() > 10); |

Терминальные операции производят какой‑то полезный результат и запускают стрим на исполнение.

Примеры терминальных операций:   
1) forEach() (2) принимает Consumer, которому будут отданы все элементы, которые в стриме остались (будут выведены в консоль);  
2) findFirst() (5) возвращает первый в порядке следования элемент из стрима. Возвращается OptionalInt (5), потому что стрим может быть пустой и тогда в нем никакого элемента нет;  
3) findAny() не гарантирует, что вернется первый элемент, а тот, который стриму удобнее;  
4) allMatch() позволяет проверить, что все элементы стрима удовлетворяют переданному условию (10), предикату. Возвращается булевское значение (8: boolean). Если действительно этот предикат выполняется для всех элементов;

5) anyMatch для проверки того, что хотя бы один элемент удовлетворяет предикату;

6) noneMatch проверяет, что не один элемент не удовлетворяет предикату;

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Stream<String> stream1 = ...;  Optional<String> minString = stream1.min(  Comparator.comparing(  String::length, Integer::compare));  IntStream stream2 = ...;  int count = stream2.count();  IntStream stream3 = ...;  int sum = stream3.sum(); |

7) min() (2) возвращает минимальный элемент из стрима в аналогичном виде Optional (2), потому что стрим может быть пустой;

8) max() возвращает максимум;  
\* (min() и max() в случае стрима объектов принимают Comparator (3). Правило, по которому определяются минимум и максимум)

9) count() (7) возвращает просто количество элементов оставшихся в стриме после применения всех трансформаций и фильтраций;

10) (примитивы) sum() (10) вернет обычную арифметическую сумму элементов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | Stream<String> stream1 = ...;  List<String> list =  stream1.collect(Collectors.toList());  Stream<BigInteger> bigInts = ...;  BigInteger sum = bigInts.reduce(  BigInteger.ZERO, BigInteger::add); |

11) collect() (3) позволяет собрать элементы стримов в какое‑то новое хранилище (в список (2: List). Тип этого нового хранилище определяется параметром, который передается в метод collect() (3: Collectors.toList()). Это некий Collector. И он может собирать элементы стрима во что угодно, не только в коллекции. Многие стандартные Collector есть в классе Collectors.);

12) reduce() (6) позволяет вычислить свертку элементов стрима (то есть результат применения некоторого бинарного оператора (7: BigInteger::add) к каждой паре элементов стрима пока от стрима не останется один единственный элемент. Это и есть результат свертки, и он возвращается в качестве результата (6: BigInteger sum) из метода reduce(). Если стрим был пуст, то возвращается некоторое нулевое значение (7: BigInteger.ZERO).).

Важно, что вызвать на стриме терминальную операцию можно только один раз. После вызова терминальной операции стрим считается более непригодным к использованию и этот объект остается только выбросить. Если к тем же элементам нужно применить еще какое‑то преобразование, посчитать по ним какое‑то другое значение, то надо заново сконструировать стрим, снова настроить его и запустить какой‑то терминальной операцией.

6.4.6 Два примера

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | public static BigInteger factorial(int n) {  return IntStream.rangeClosed(1, n)  .mapToObj(i -> BigInteger.valueOf(i))  .reduce(BigInteger.ONE,  BigInteger::multiply); |

На закуску приведу пару примеров того, как написать при помощи стримов известные нам уже алгоритмы. Например, алгоритм вычисления факториала (1). Здесь мы можем породить интервал целых чисел от 1 до n в виде стрима (2: rangeClosed(1, n)). Затем каждое из этих чисел превратить в BigInteger (5: BigInteger.valueOf(i)), а затем вычислить свертку при помощи операции умножения (8: BigInteger::multiply). Результат будет тем же самым факториалом, который мы с вами вычисляли в одном из домашних заданий.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | public static boolean isPalindrome(  String s) {  StringBuilder leftToRight =  new StringBuilder();  s.chars().filter(  Character::isLetterOrDigit)  .map(Character::toLowerCase)  .forEach(  leftToRight::appendCodePoint);  StringBuilder rightToLeft =  new StringBuilder(  leftToRight).reverse();  return leftToRight.toString()  .equals(rightToLeft.toString());  } |

И второй пример определение того, что строка является палиндромом (1), то есть одинаково читается слева направо и справа налево. Основная сложность здесь это отфильтровать символы строки (7), оставить только буквы и цифры и проверять палиндромность уже результирующей строки. Эту задачу мы можем решить следующим образом: получить стрим символов из строки (7: s.chars), отфильтровать только те, которые является только буквами или цифрами (8: Character::isLetterOrDigit), привести каждый из них в нижний регистр (9: Character::toLowerCase) и сохранить результат (11) в StringBuilder (4). Ну а дальше остается реверсировать (13) и сравнить (17) 2 строки.