Streams

Modern Java in Action: Lambda, streams, functional and reactive programming
https://docs.oracle.com/javase/tutorial/collections/streams/
https://www.baeldung.com/java-streams
http://tutorials.jenkov.com/java-collections/streams.html

스트림이란 무엇인가?

- 데이터를 다룰 때 컬렉션이나 배열에 데이터를 담고 원하는 결과를 얻기 위해 for 문과 iterator 를 이용해서 코드를 작성함
  - 코드가 길고 이해하기 어려움
  - 재사용성이 떨어짐
  - 데이터 소스(배열, 컬렉션, 파일 등)마다 다른 방식으로 처리해야 함
  - Collection, Iterator 와 같은 인터페이스를 이용해 표준화 했지만 중복 정의됨
    - Collections.sort(), Arrays.sort()

- 스트림을 이용하면 선언형 (SQL 같이) 으로 컬렉션 데이터를 처리할 수 있음 (you say what needs to be done)
  - "xx 조건을 만족하는 데이터만 가져와라!" – 행위를 파라미터화 할 수 있음(람다)
  - 함수형 언어 스타일 – 고수준의 함수들을 조합해 데이터를 처리함 (map, filter, reduce 등)
  - 병렬 처리가 가능함 - parallelStream() 호출하면 자동으로 병렬 처리함
  - 데이터 소스를 추상화함 (배열, 컬렉션, 파일 등)

Streams (Java 8)

- **A sequence of elements** from **a source** that supports **data-processing operations**.

  - A sequence of elements
    - a stream provides an interface to a sequenced set of values of a specific element type, which is not about manupulating values such as Collection but expressing computations

  - Source
    - Streams consume from a data-providing source such as collections, arrays, or I/O resources.
    - Note that generating a stream from an ordered collection preserves the ordering.

  - Data-processing operations
    - Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on.
    - Stream operations can be executed either sequentially or in parallel.

Exmaple – menu

- We'll use the following domain for our examples: a menu that's nothing more than a list of dishes

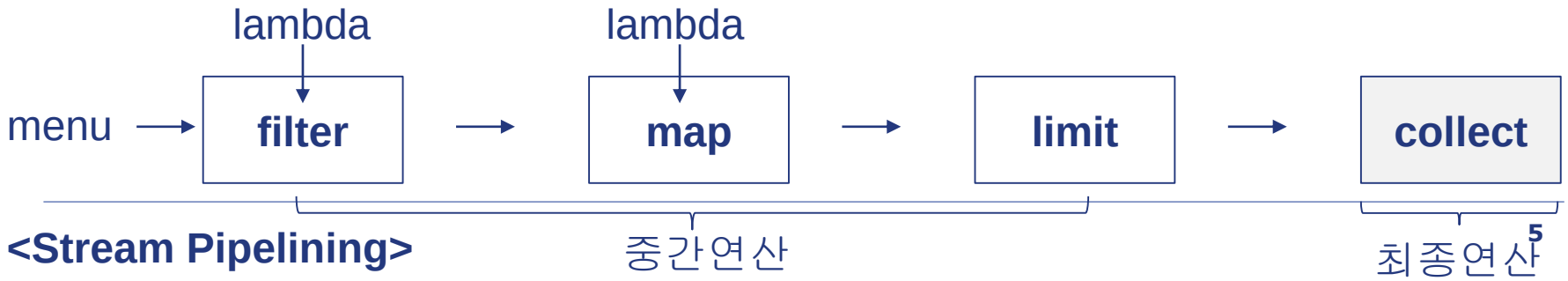
```
List<Dish> menu = Arrays.asList(
 new Dish("pork", false, 800, Dish.Type.MEAT),
 new Dish("beef", false, 700, Dish.Type.MEAT),
 new Dish("chicken", false, 400, Dish.Type.MEAT),
 new Dish("french fries", true, 530, Dish.Type.OTHER),
 new Dish("rice", true, 350, Dish.Type.OTHER),
 new Dish("season fruit", true, 120, Dish.Type.OTHER),
 new Dish("pizza", true, 550, Dish.Type.OTHER),
 new Dish("prawns", false, 300, Dish.Type.FISH),
 new Dish("salmon", false, 450, Dish.Type.FISH) );
```

```
public class Dish {
   public enum Type {MEAT, FISH, OTHER}
   private final String name;
   private final boolean vegetarian;
   private final int calories;
   private final Type type;

   // constructor
   // only getters
   // toString ()

}
```

Exmaple - Dish class

- You first get a stream from the list of dishes by calling the stream method on menu.
  - The **data source** is the list of dishes (the menu) and it provides **a sequence of elements** to the stream.
  - You apply a series of **data-processing operations** on the stream: filter, map, limit, and collect.

```
import static java.util.stream.Collectors.toList;      //remember!
// "Find names of three high-calorie dishes." (declarative way)
List<String> threeHighCaloricDishNames =
                                menu.stream() // 스트림 생성
                                .filter ( dish -> dish.getCalories() > 300 ) //조건을 만족하는 데이터
                                .map ( Dish::getName )    // Dish 객체를 String(이름) 객체로 변환
                                .limit ( 3 )              // 데이터 개수를 3개로 제한
                                .collect ( toList() );    // 스트림을 List 로 변환함
```



Exmaple - Dish class

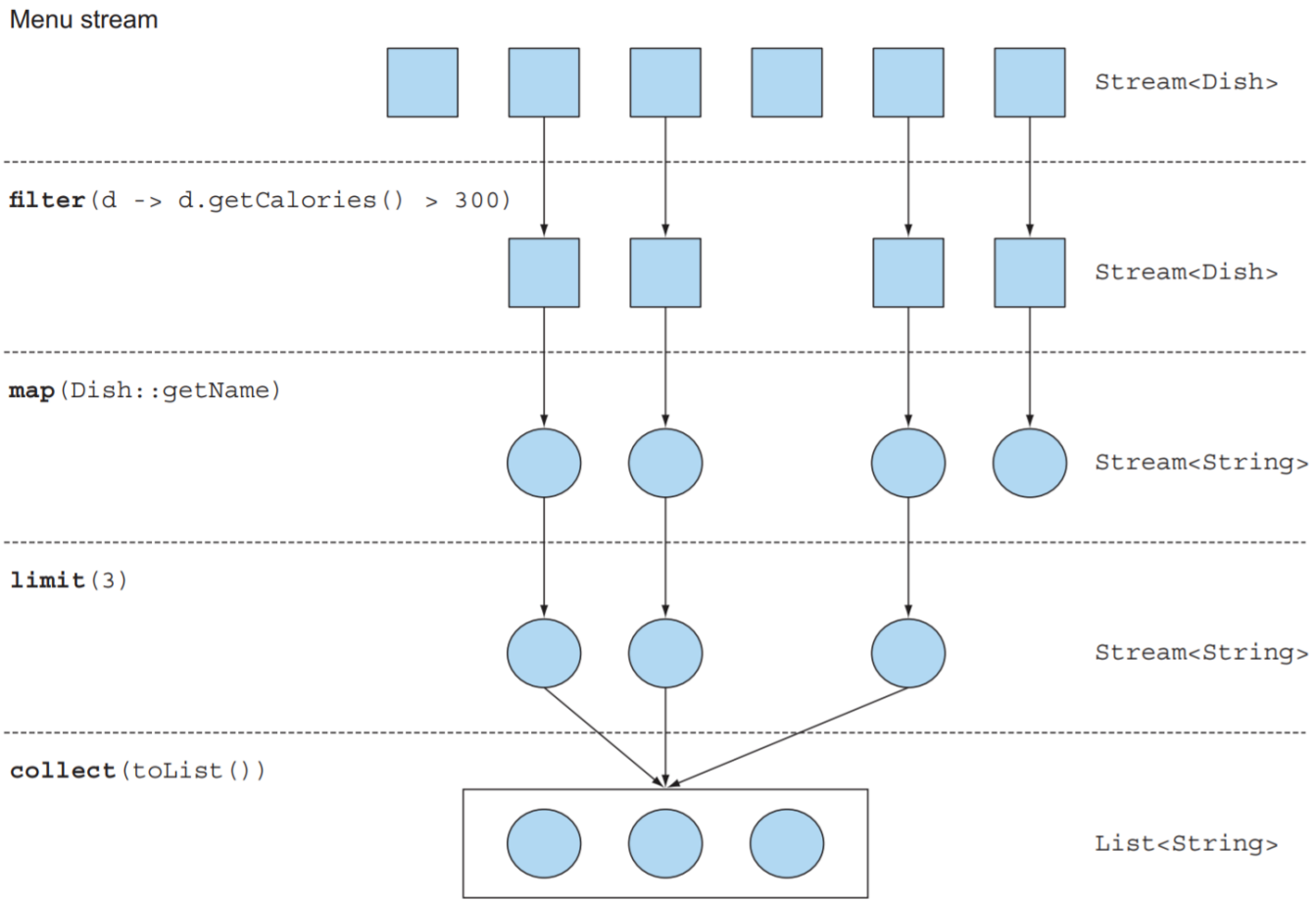


Figure 4.2 Filtering a menu using a stream to find out three high-calorie dish names

Stream vs Collection

- The difference between collections and streams has to do with when things are computed.
  - A collection is an in-memory data structure that holds all the values the data structure currently has - every element in the collection has to be computed before it can be added to the collection. (동영상을 전부 다운로드 해서 재생)
  - A stream is a conceptually fixed data structure (you can't add or remove elements from it) whose elements are computed on demand. (스트리밍)
- Traversable only once
  - A collection can be traversed multiple times, but a stream can not (only once)
- External vs. internal iteration
  - Using the Collection interface requires iteration to be done by the user (for example, using for-each); this is called external iteration.
  - The Streams library uses internal iteration (it does the iteration for you and takes care of storing the resulting stream value somewhere). You merely provide a function **saying what's to be done**.
- Pipelining - Many stream operations return a stream themselves, allowing operations to be chained to form a larger pipeline. This enables certain optimizations such as laziness and short-circuiting.

Stream operations (1/2)

- Stream operators can be classified into two categories
  - filter, map, and limit can be connected together to form a pipeline.
  - collect causes the pipeline to be executed and closes it

```
List<String> threeHighCaloricDishNames =
                              menu.stream() // 스트림 생성
                              .filter ( dish -> dish.getCalories() > 300 )  //조건을 만족하는 데이터
                              .map ( Dish::getName )     // Dish 객체를 String(이름) 객체로 변환
                              .limit ( 3 )                    // 데이터 개수를 3개로 제한
                              .collect ( toList() );          // 스트림을 List 로 변환함
```

- Intermediate operations
  - Intermediate operations such as filter or sorted **return another stream** as the return type. This allows the operations to be connected to form a query.
  - What's important is that intermediate operations **don't perform any processing until a terminal operation is invoked** on the stream pipeline - they're lazy
    - Only the first three dishes are selected because of the **limit** operation and a technique called **short-circuiting**
    - Despite the fact that **filter** and **map** are two separate operations, they were **merged into the same pass** (**loop fusion**)

Stream operations (2/2)

- Terminal operations
  - Terminal operations produce a result from a stream pipeline.
  - A result is any nonstream value such as a List, an Integer, or even void.
  - for example, menu.stream().forEach(System.out::println);

Table 4.1 Intermediate operations

Operation	Type	Return type	Argument of the operation	Function descriptor
`filter`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`map`	Intermediate	`Stream<R>`	`Function<T, R>`	`T -> R`
`limit`	Intermediate	`Stream<T>`		
`sorted`	Intermediate	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`
`distinct`	Intermediate	`Stream<T>`		

Table 4.2 Terminal operations

Operation	Type	Return type	Purpose
`forEach`	Terminal	`void`	Consumes each element from a stream and applies a lambda to each of them.
`count`	Terminal	`long`	Returns the number of elements in a stream.
`collect`	Terminal	(generic)	Reduces the stream to create a collection such as a `List`, a `Map`, or even an `Integer`. See chapter 6 for more detail.

Building Streams

- You can create streams in many ways
  - From a Collection
  - From an array
  - From a generative functions: range, iterate, generate
  - From a Files

https://www.geeksforgeeks.org/10-ways-to-create-a-stream-in-java/

Streams from Collection and Array

- You can create a stream from a Collection using the static method **Collection.stream()**

```
List<String> list1 = Arrays.asList("Modern ", "Java ", "in ", "Action ");
list1.stream().forEach(System.out::print);  // Modern Java In Action
list1.stream().forEach(System.out::print);   // OK!
```

- You can create a stream from an array using the static method **Arrays.stream** or **Stream.of**, which takes an array as parameter

```
String[] strings = {"Modern ", "Java ", "In ", "Action"};
Stream<String> stream1 = Arrays.stream(strings);
stream1.forEach(System.out::print);  // Modern Java In Action

Stream<String> stream2 = Stream.of("Modern ", "Java ", "In ", "Action");
stream2.forEach(System.out::print);  // Modern Java In Action
stream2.forEach(System.out::print);  // IllegalStateException

int[] numbers = {1, 2, 3, 4, 5};
IntStream number1 = Arrays.stream(numbers);
number1.forEach(System.out::print);  //12345
```

Streams from range, iterate, generate

- Streams produced by **range** create values from start to end by 1
- **Infinite streams** produced by **iterate** and **generate** create values on demand given a function

```
IntStream.range(1, 5)
   .forEach(System.out::print);     // 1234

IntStream.rangeClosed(1, 5)
   .forEach(System.out::print);     // 12345

Stream.iterate(0, n->n+2).limit(5)
    .forEach(System.out::print);    // 02468

Stream.generate(() -> 1).limit(5)
    .forEach(System.out::print);  // 11111
```

```
// IntStream class
static IntStream range (
        int startInclusive, int endExclusive)

static IntStream rangeClosed (
        int startInclusive, int endExclusive)
// interface UnaryOperator<T> extends Function<T, T>
static <T> Stream<T> iterate (
T seed, UnaryOperator<T> f) // f(f(seed))
// T get()
static <T> Stream<T> generate (
    Supplier<T> s)
```

Streams from Random and Math.random

- Generating random numbers is using nextInt method of Random or random method of Math class

```
Random random = new Random();
int rn1 = random.nextInt();
int rn2 = random.nextInt(100 - 1) + 1;

List<Integer> rns = new ArrayList<>();
for (int i =0; i < 10; i++)
    rns.add(random.nextInt(100 - 1) + 1);
System.out.println(rns);
```

```
IntStream intStream =
            new Random.ints(10, 1, 100);
intStream.forEach(System.out::print);

// infinite stream
IntStream intStream2 =
                new Random.ints();
intStream2.forEach(System.out::print);

DoubleStream randomStream =
        Stream.generate (Math::random);
```

Streams from Files

- Many static methods in java.nio.file.Files return a stream.
- For example, a useful method is Files.lines, which returns a stream of lines as strings from a given file.
  - Files.list(Path dir), which returns a stream of file names as Path objects from a given directory

```
// Stream<String> Files.lines(Path path)

try ( Stream<String> lines =
        Files.lines(Paths.get("data.txt"), Charset.defaultCharset()) ) {
      lines.forEach(System.out::println);
} catch ( IOException e ) {
   System.out.println(e);
}
```

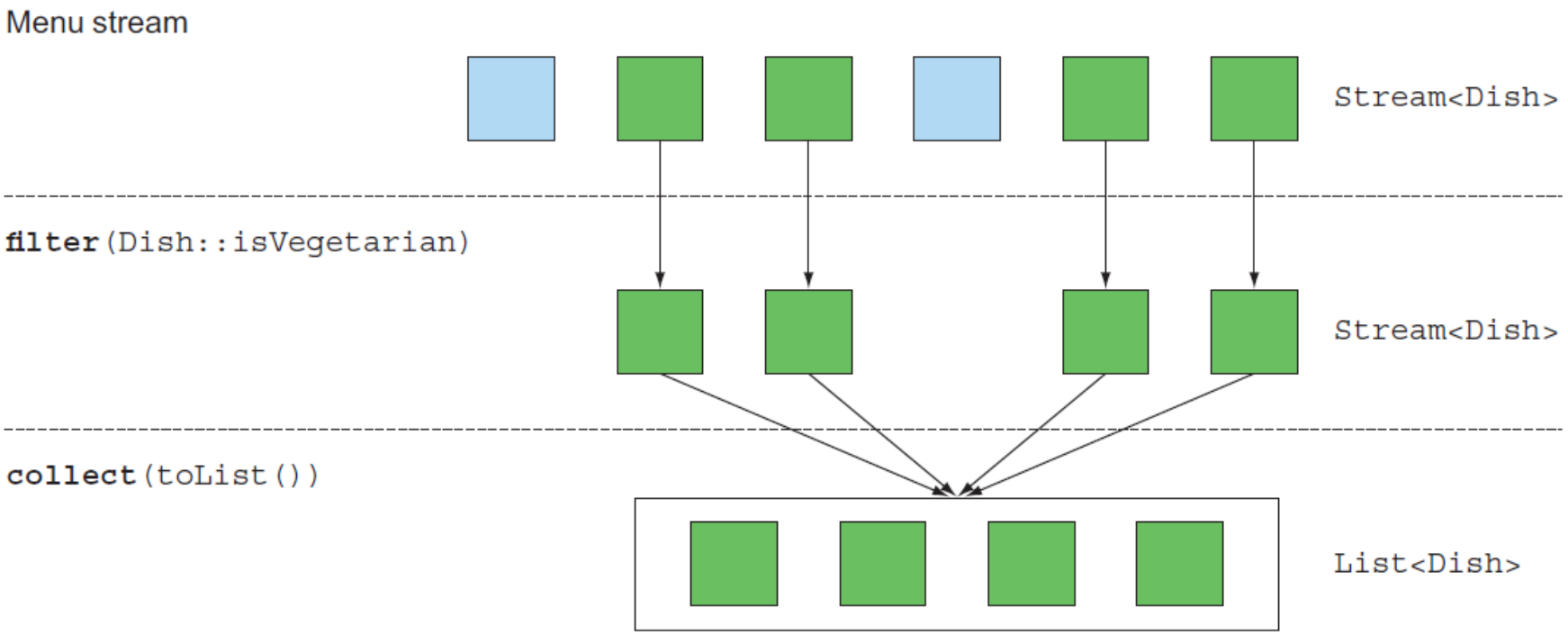
Working with Streams

- You can use the Streams API (internal iteration)
  - Filtering, slicing, and mapping
  - Finding, matching, and reducing
  - Using numeric streams

Filtering with Predicate

- **filter** operation takes as argument a predicate and returns a stream of all elements matching the predicate

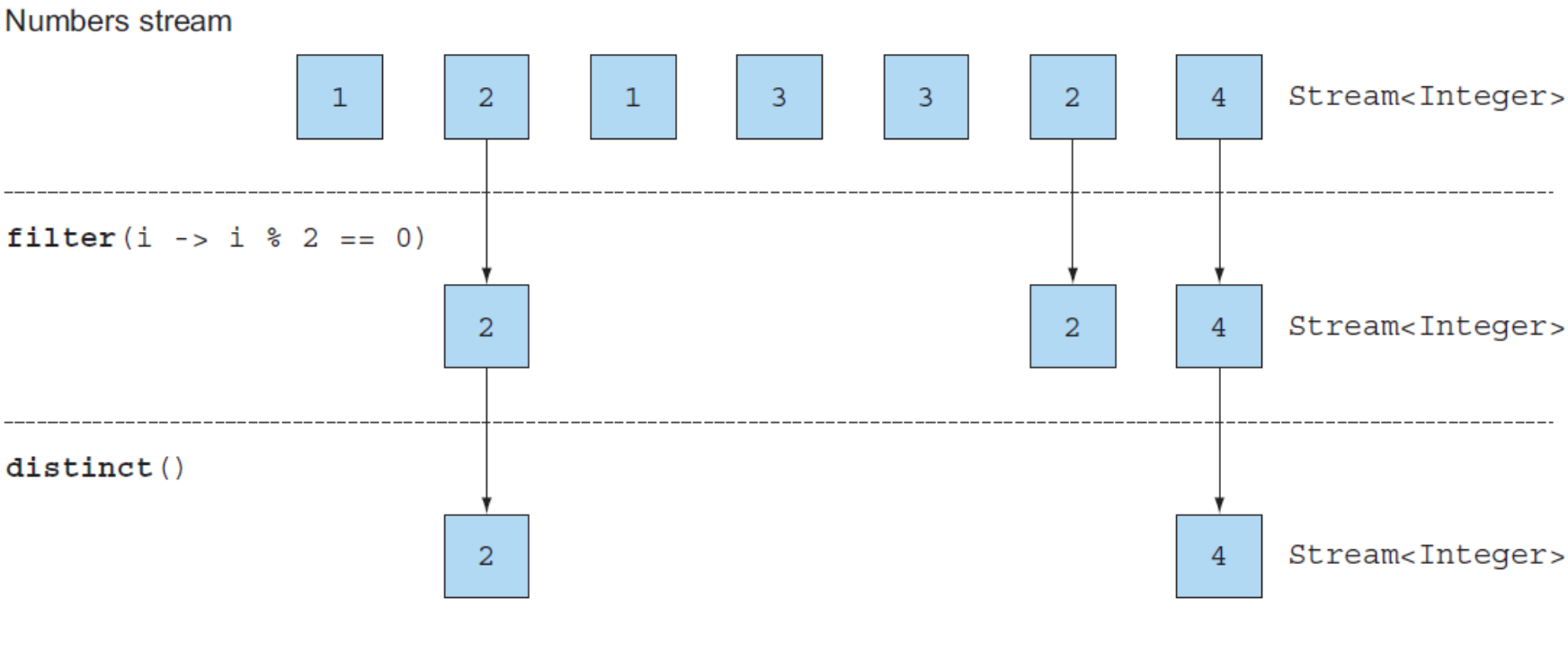
```
List<Dish> vegetarianDishes = menu.stream()
  .filter(Dish::isVegetarian).collect(Collectors.toList());
```



Filtering Unique Elements

- **distinct** returns a stream with unique elements (according to the implementation of the hashcode and equals methods)

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream().filter(i -> i % 2 == 0).distinct().forEach(System.out::println);
```



Slicing using a predicate

- The initial list (specialMenu) was already sorted on the number of calories!

```
List<Dish> specialMenu = Arrays.asList(
  new Dish("seasonal fruit", true, 120, Dish.Type.OTHER),
  new Dish("shrimp", false, 300, Dish.Type.FISH),
  new Dish("rice", true, 350, Dish.Type.OTHER),
  new Dish("chicken", false, 400, Dish.Type.MEAT),
  new Dish("french fries", true, 530, Dish.Type.OTHER) );
```

```
List<Dish> filteredMenu = specialMenu.stream()
  .filter( dish -> dish.getCalories() < 320 )
  .collect(Collectors.toList());     // seasonal fruit, shrimp
```

Slicing Using takeWhile and dropWhile (Java 9)

- **takeWhile** - Returns, if this stream is ordered, a stream consisting of the longest prefix of elements taken from this stream that match the given predicate.

```
List<Dish> slicedMenu1 = specialMenu.stream()
                         .takeWhile (dish -> dish.getCalories() < 320)
                         .collect(toList());     // seasonal fruit, shrimp
```

- **dropWhile** - Returns, if this stream is ordered, a stream consisting of the remaining elements of this stream after dropping the longest prefix of elements that match the given predicate.

```
List<Dish> slicedMenu2 = specialMenu.stream()
                         .dropWhile (dish -> dish.getCalories() < 320)
                         .collect(toList());    // rice, chicken, french fries
```

Truncating a stream

- Streams support the limit(n) method, which returns another stream that's no longer than a given size.

```
List<Dish> dishes = specialMenu
 .stream()
 .filter(dish -> dish.getCalories() > 300)
 .limit(3)
 .collect(toList());
```

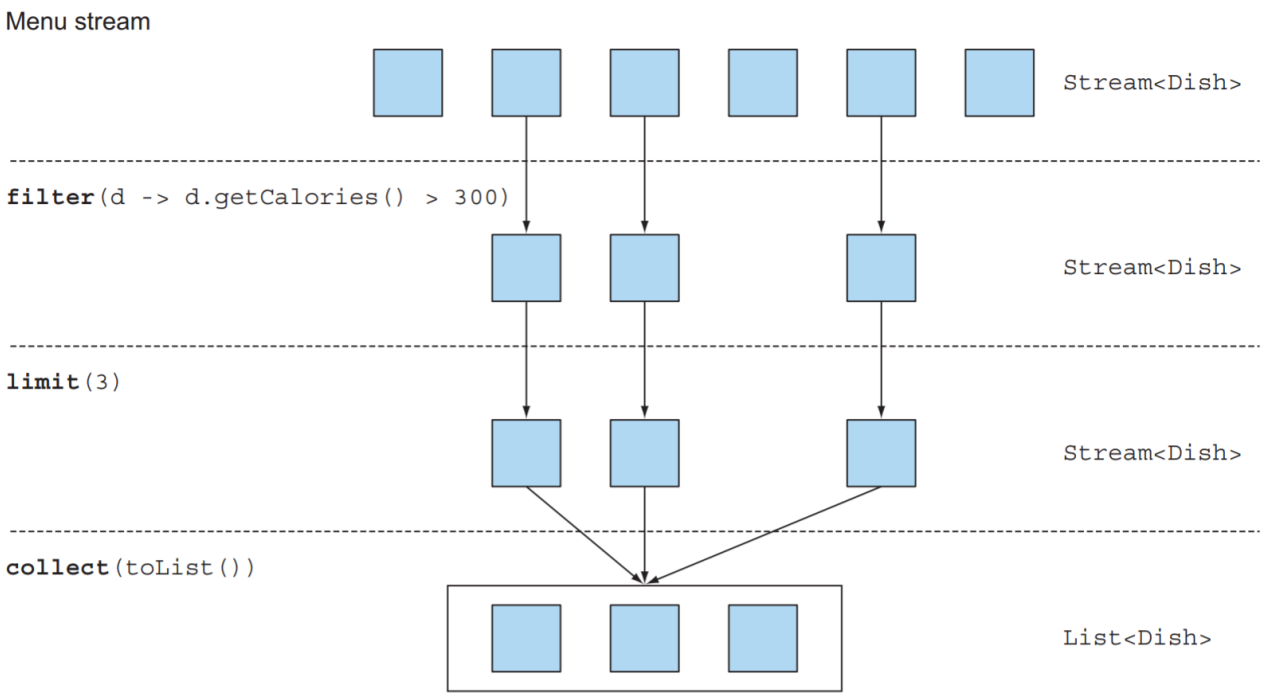


Figure 5.3 Truncating a stream

Skipping elements

- Streams support the skip(n) method to return a stream that discards the first n elements

```
List<Dish> dishes = menu.stream()
 .filter(d -> d.getCalories() > 300)
 .skip(2)
 .collect(toList());
```

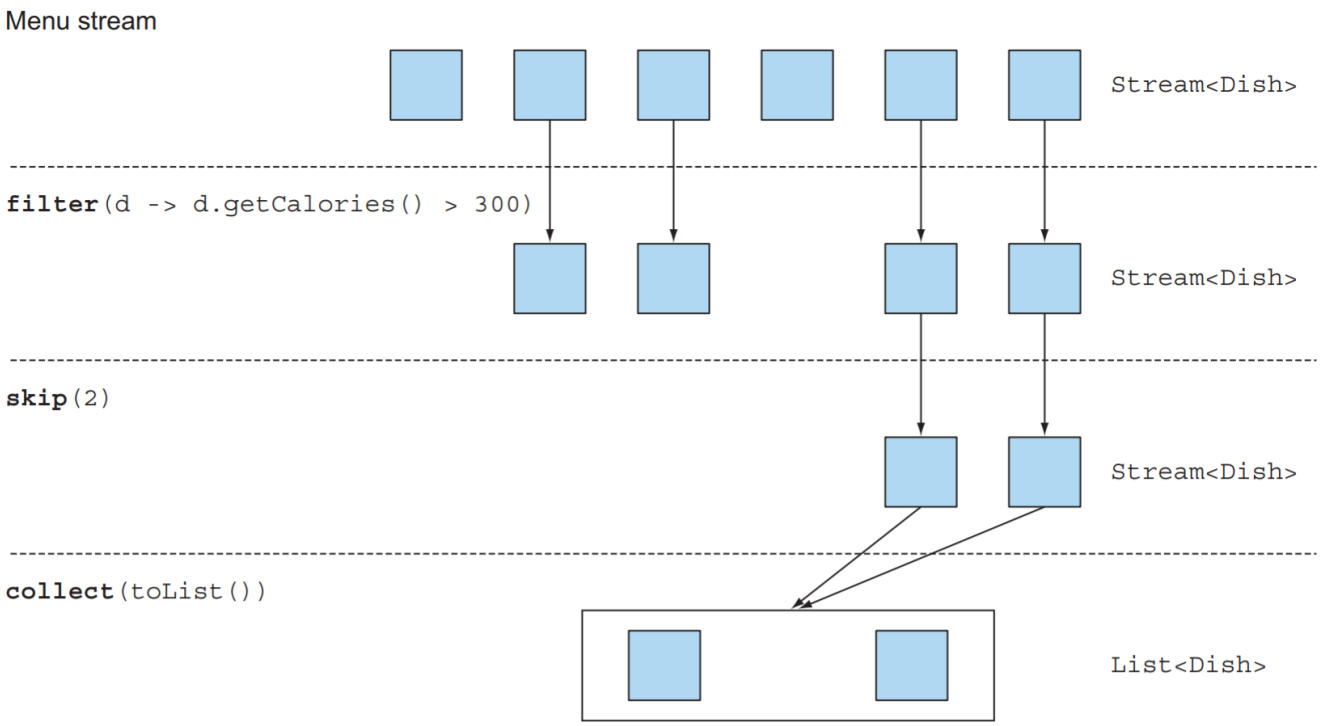


Figure 5.4 Skipping elements in a stream

Mapping

- **map** - Applying a function to each element of a stream
- Streams support the map method, which takes a function as argument.
- The function is applied to each element, mapping it into a new element

```
List<String> dishNames = menu.stream()
   .map(Dish::getName)
   .collect(toList());              // (menu on 4 page)
System.out.println(dishNames);
// [pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon]

List<Integer> dishNameLengths = menu.stream()
   .map(Dish::getName)
   .map(String::length)
   .collect(toList());
System.out.println(dishNameLengths); // [4, 4, 7, 12, 4, 12, 5, 6, 6]
```

Flattening streams (1/2)

- How could you return a list of all the unique characters for a list of words?
  - For example, given the list of words ["Hello," "World"] you'd like to return the list **["H", "e", "l", "o", "W", "r", "d"].**

```
List<String> words =
  Arrays.asList("Hello", "World");

// String[ ] split(String regex)
List<String[ ]> chs =
  words.stream()
 .map(word -> word.split(""))
 .distinct()
 .collect(toList());

System.out.println(chs);
//[[String@xxxx], [String@yyyy]]
```

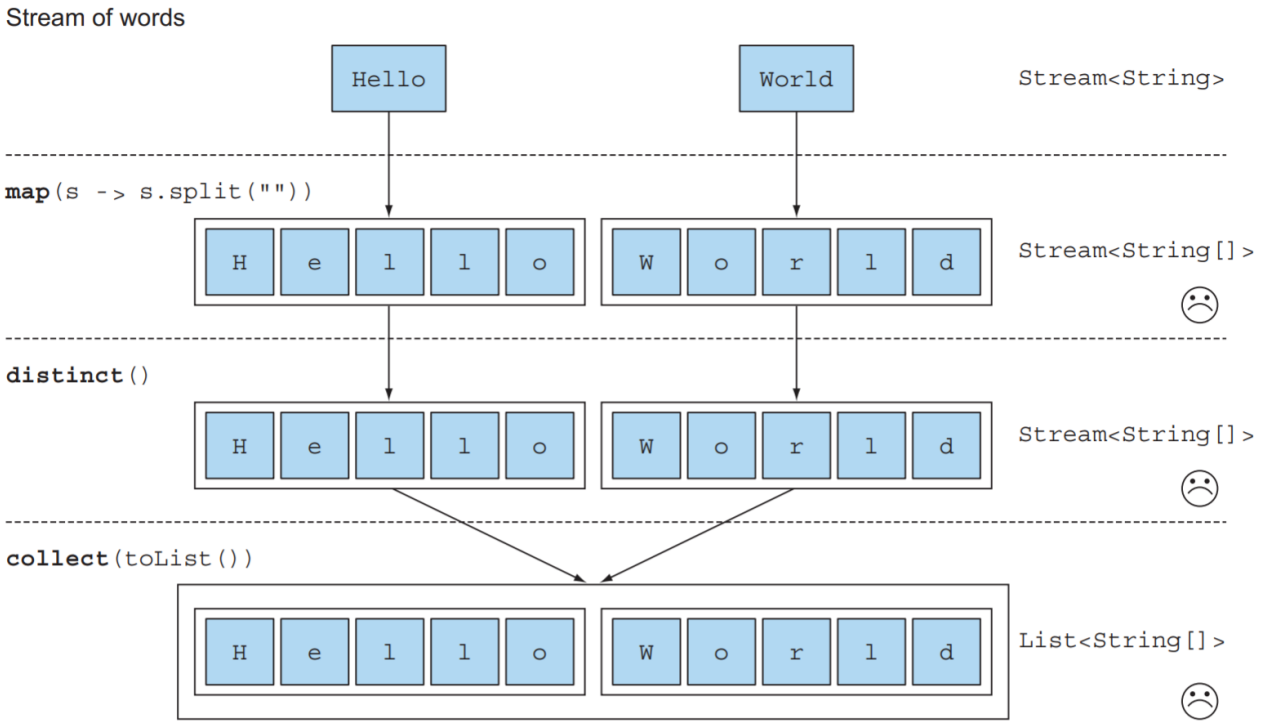


Figure 5.5 Incorrect use of `map` to find unique characters from a list of words

Flattening streams (2/2)

- Using the **flatMap** method has the effect of mapping each array not with a stream but with the contents of that stream.

```
List<String> uniqueCharacters =
  words.stream()
 .map(word -> word.split(""))
 .flatMap(Arrays::stream)
 .distinct()
 .collect(toList());

System.out.println(
                 uniqueCharacters);
//[H, e, l, o, W, r, d]
```

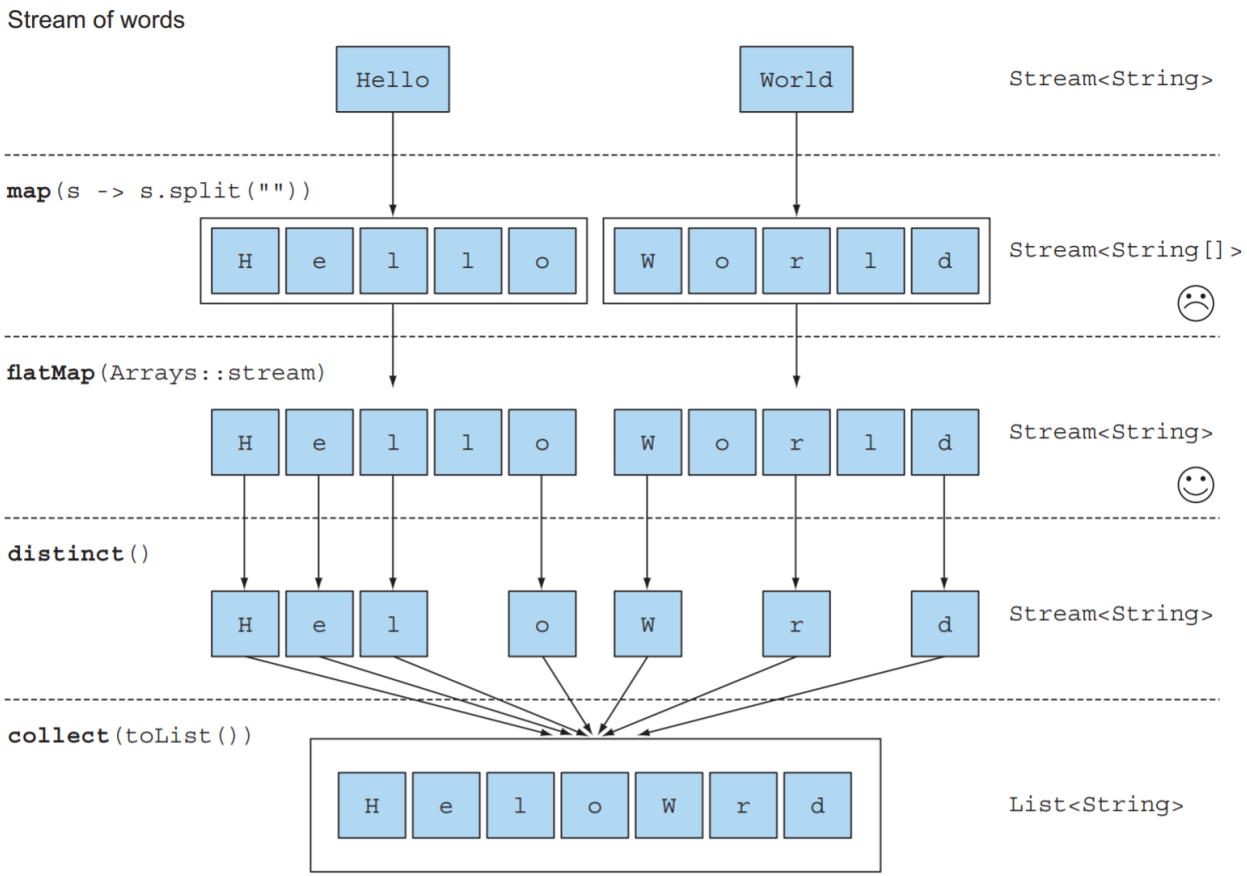


Figure 5.6 Using `flatMap` to find the unique characters from a list of words

Finding and matching (1/2)

- Another common data processing idiom is finding whether some elements in a set of data match a given property.
- The Streams API provides such facilities through the allMatch, anyMatch, noneMatch, findFirst, and findAny methods of a stream.

```
//  Checking to see if a predicate matches at least one element
if(menu.stream().anyMatch(Dish::isVegetarian)) {
      System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
// Checking to see if a predicate matches all elements or none of all
boolean isHealthy1 = menu.stream()
          .allMatch (dish -> dish.getCalories() < 1000);
boolean isHealthy2 = menu.stream()
          .noneMatch (d -> d.getCalories() >= 1000);
```

Finding and matching (2/2)

- The **findAny** returns an arbitrary element of the current stream.
- The **findFirst** returns the first element of the current stream.

```
// Finding an element
// parallelism
Optional<Dish> dish =
 menu.stream()
 .filter(Dish::isVegetarian)
 .findAny();
```

```
// Finding the first element
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree =
 someNumbers.stream()
 .map(n -> n * n)
 .filter(n -> n % 3 == 0)
 .findFirst(); // 9
```

- **Optional<T>** class is a container class to represent the existence or absence of a value.
  - **isPresent**() returns true if Optional contains a value, false otherwise.
  - **ifPresent**(Consumer<T> block) executes the given block if a value is present.
  - T **get**() returns the value if present; otherwise it throws a NoSuchElementException.
  - T **orElse**(T other) returns the value if present; otherwise it returns a default value

Reducing (1/2)

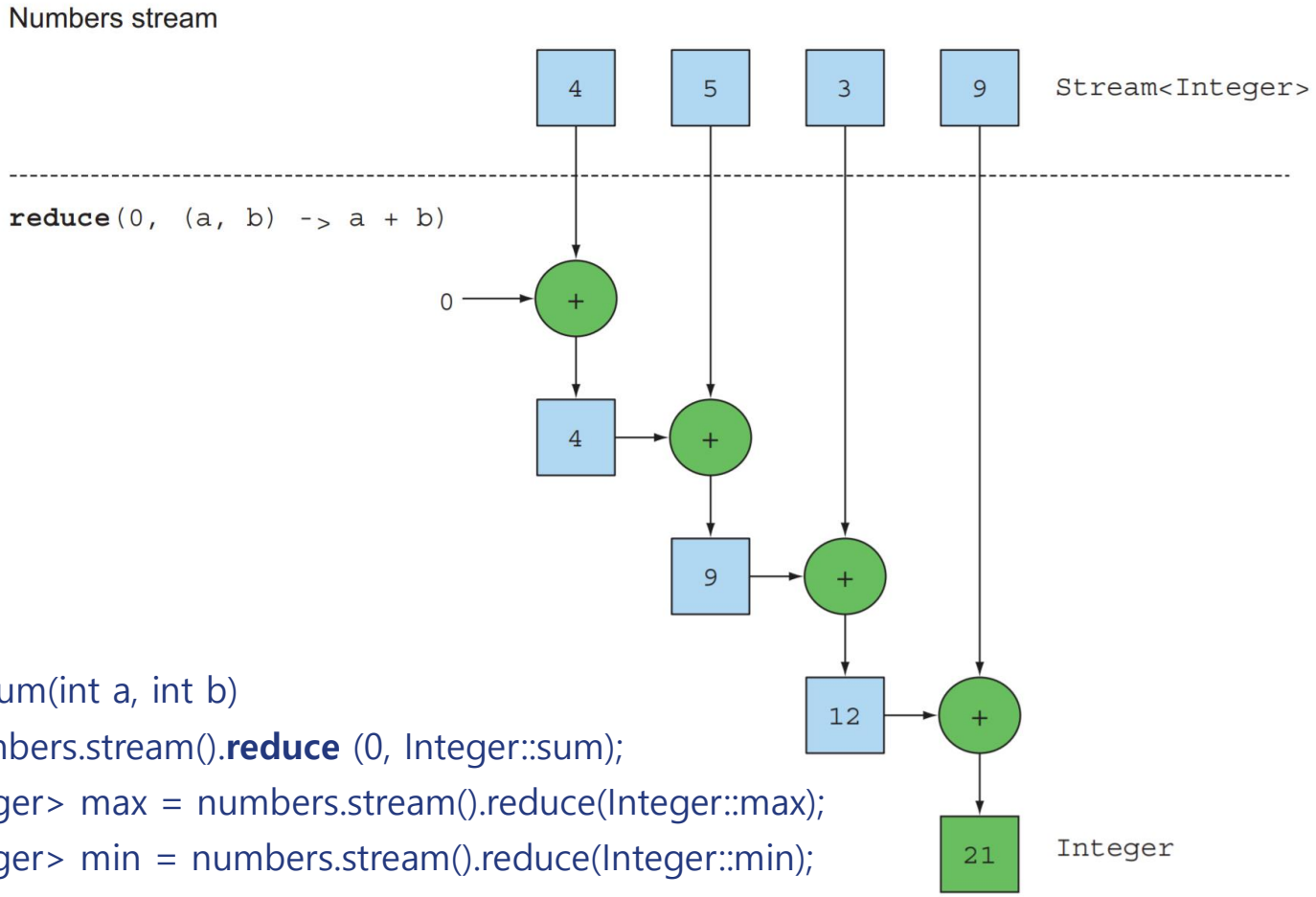
- Such queries combine all the elements in the stream repeatedly to produce a single value such as an Integer.
  - "Calculate the sum of all calories in the menu"
  - "What is the highest calorie dish in the menu?"
- In functional programming-language jargon, this is referred to as a **fold** because you can view this operation as repeatedly folding a long piece of paper until it forms a small square.
- **reduce** takes two arguments:
  - An initial value, here **0**.
  - A **BinaryOperator<T>** to combine two elements and produce a new value
    - implements BiFunction<T,T,T>

```
int sum = 0;
for (int x : numbers)
 sum += x;
```

```
Stream<Integer> numbers =
        Arrays.asList(4, 5, 3, 9).stream();
int sum = numbers.stream()
      .reduce ( 0, (a, b) -> a + b );
```

Reducing (2/2)

- The lambda combines each element repeatedly until the stream containing the integers 4, 5, 3, 9, are reduced to a single value.




```
// static int   sum(int a, int b)
int sum = numbers.stream().reduce (0, Integer::sum);
Optional<Integer> max = numbers.stream().reduce(Integer::max);
Optional<Integer> min = numbers.stream().reduce(Integer::min);
```

Figure 5.7 Using `reduce` to sum the numbers in a stream

Numeric Streams

- The problem with the previous code is that there's an insidious **boxing cost**. Behind the scenes each Integer needs to be unboxed to a primitive before performing the summation.

```
int sum = numbers.stream().reduce (0, Integer::sum);
```

- Java 8 introduces three primitive specialized stream interfaces to tackle this issue:
  - IntStream, DoubleStream, and LongStream
  - Thereby avoid hidden boxing costs
  - common numeric reductions: sum, max, etc.

```
int calories = menu.stream().mapToInt(Dish::getCalories).sum();
```

- Converting back to a stream objects

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> stream = intStream.boxed();
```

Summary

Table 5.1 Intermediate and terminal operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
`filter`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`distinct`	Intermediate (stateful-unbounded)	`Stream<T>`		
`takeWhile`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`dropWhile`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`skip`	Intermediate (stateful-bounded)	`Stream<T>`	`long`	
`limit`	Intermediate (stateful-bounded)	`Stream<T>`	`long`	
`map`	Intermediate	`Stream<R>`	`Function<T, R>`	`T -> R`
`flatMap`	Intermediate	`Stream<R>`	`Function<T, Stream<R>>`	`T -> Stream<R>`
`sorted`	Intermediate (stateful-unbounded)	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`
`anyMatch`	Terminal	`boolean`	`Predicate<T>`	`T -> boolean`
`noneMatch`	Terminal	`boolean`	`Predicate<T>`	`T -> boolean`

Performance Summary

- On a relatively small array old-fashion loop shows the best results
- For arrays of large size, parallel streams show better results.
- For a performance perspective, complex filter is better than multiple filters.
  - In JMH, the default benchmark mode is (Throughput) 1 and in this case **higher value is better.**

Array Elements	Version	Stream complex filter	Stream multiple filters	Parallel stream complex filter	Parallel stream multiple filters	Old fashion java iteration
10	Java 8	5,947,577.65	3,785,766.91	24,515.74	23,896.81	45,874,144.76
	Java 12	10,338,525.55	5,460,308.05	21,289.44	20,403.99	41,024,334.06
100	Java 8	3,131,081.56	1,806,210.04	25,584.77	25,314.61	4,902,625.83
	Java 12	4,381,301.19	2,227,583.84	20,105.24	19,426.22	6,011,852.03
1,000	Java 8	489,666.69	211,435.45	24,313.07	23,113.39	662,102.44
	Java 12	607,572.43	287,157.19	19,418.83	17,692.43	553,243.59
10,000	Java 8	17,297.42	12,614.67	11,909.09	12,676.06	29,390.91
	Java 12	30,643.29	16,268.02	13,874.59	12,108.48	29,188.75
100,000	Java 8	1,398.70	1,228.13	3,260.86	3,373.37	1,999.03
	Java 12	1,450.34	1,531.52	5,334.95	3,782.76	2,061.74
1,000,000	Java 8	81.31	99.15	406.30	477.87	200.56
	Java 12	139.00	123.88	781.05	589.97	196.11

https://github.com/volkodavs/javafilters-benchmarks

Q&A