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

Streams: Since Java 8

- Streams are an update to the Java API that let you manipulate collections of data in a declarative way

- The following example prints the name of all members contained in the collection roster with a for-each loop

```
for (Person p : roster) {            // external iteration
  System.out.println(p.getName());
}
```

- The following example prints all members contained in the collection roster but with the aggregate operation forEach:

```
roaster.stream()       // build Stream<Person> from List<Person>
  .forEach(e -> System.out.println(e.getName()); // internal iteration
```

Streams

- 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
- Source
  - Streams consume from a data-providing source such as values, collections, arrays, or I/O resources
- Data-processing operations
  - Streams support common operations to manipulate data, such as **filter**, **map**, **reduce**, **find**, **match**, **sort**, and so on
  - Stream operations can be executed either sequentially or in parallel.

Stream vs Collection

- The following example create a list of the name of all dishes contained in the collection *menu* with a **for-each loop**

```
List<String> names = new ArrayList<>();
for (Dish dish: menu) {
   names.add(dish.getName());
}
```

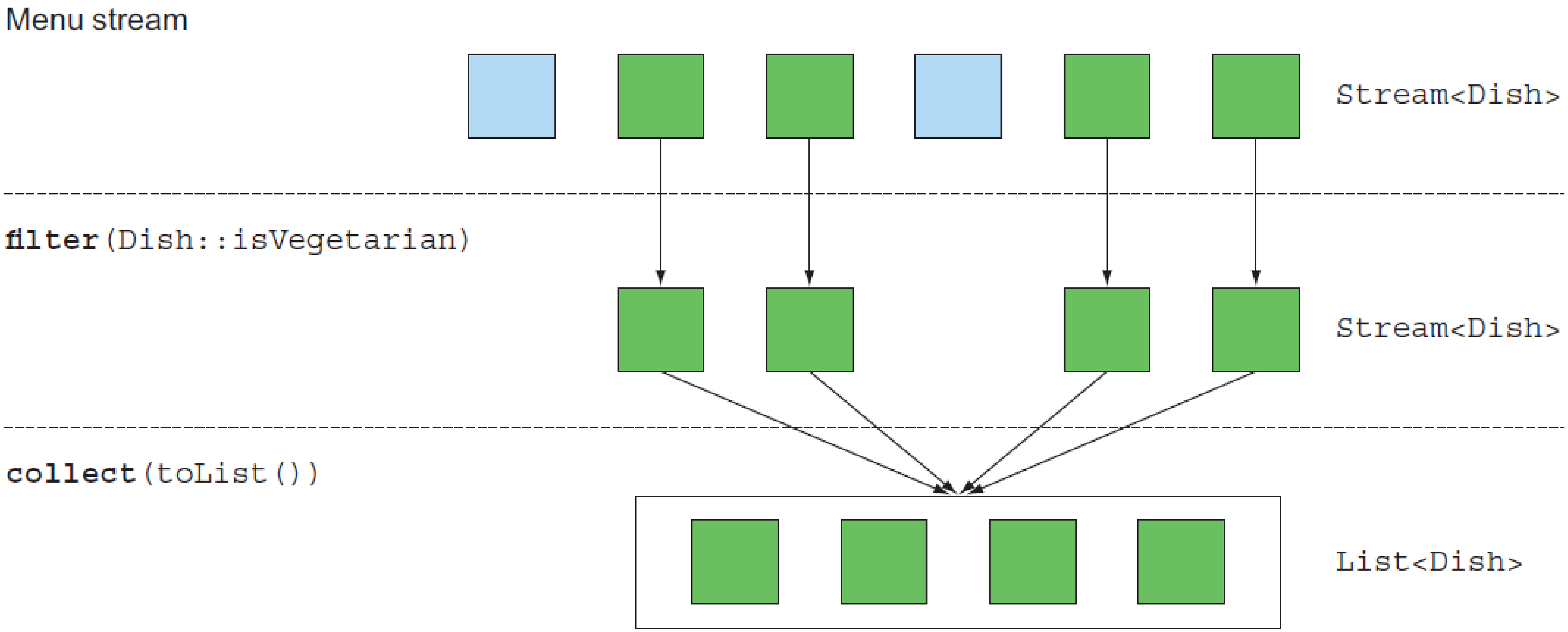
- The following example create a list of the name of all dishes contained in the collection *menu* with a **stream**

```
List<String> names = menu.stream()
   .map( (Dish dish) -> dish.getName() ) // or .map(Dish::getName)
   .collect(Collectors.toList());
```

Filtering with Predicate

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

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



Slicing using Filter

```
List<Dish> specialMenu = Arrays.asList(
  new Dish("seasonal fruit", true, 120, Dish.Type.OTHER),
  new Dish("prawns", 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, prawns
```

But, you notice that the initial **list was already sorted** on the number of calories!

Slicing using Filter

```
List<String> versions = new ArrayList<>();
versions.add("Lollipop");
versions.add("KitKat");
versions.add("Jelly Bean");
versions.add("Ice Cream Sandwidth");
versions.add("Honeycomb");
versions.add("Gingerbread");

// print all versions whose length is greater than 10 character
System.out.println("All versions whose length greater than 10");
versions.stream()
   .filter(s -> s.length() > 10)
   .forEach(System.out::println);

System.out.println("first element which has letter 'e' ");
String first = versions.stream()
   .filter(s -> s.contains("e"))
   .findFirst().get();
System.out.println(first);
```

Counting using Filter

```
// Count the empty strings
List<String> strList = Arrays.asList("abc", "", "bcd", "", "defg", "jk");
long count = strList.stream()
  .filter(s -> s.isEmpty()) // String::isEmpty
  .count();
System.out.printf("List %s has %d empty strings %n", strList, count);

// Count String with length more than 3
long num = strList.stream()


System.out.printf("List %s has %d strings of length > 3 %n", strList, num);

// Count number of String which startswith "a"
count = strList.stream()


System.out.printf("List %s has %d strings starting with 'a' %n", strList, count);
```

Creating List using Filter

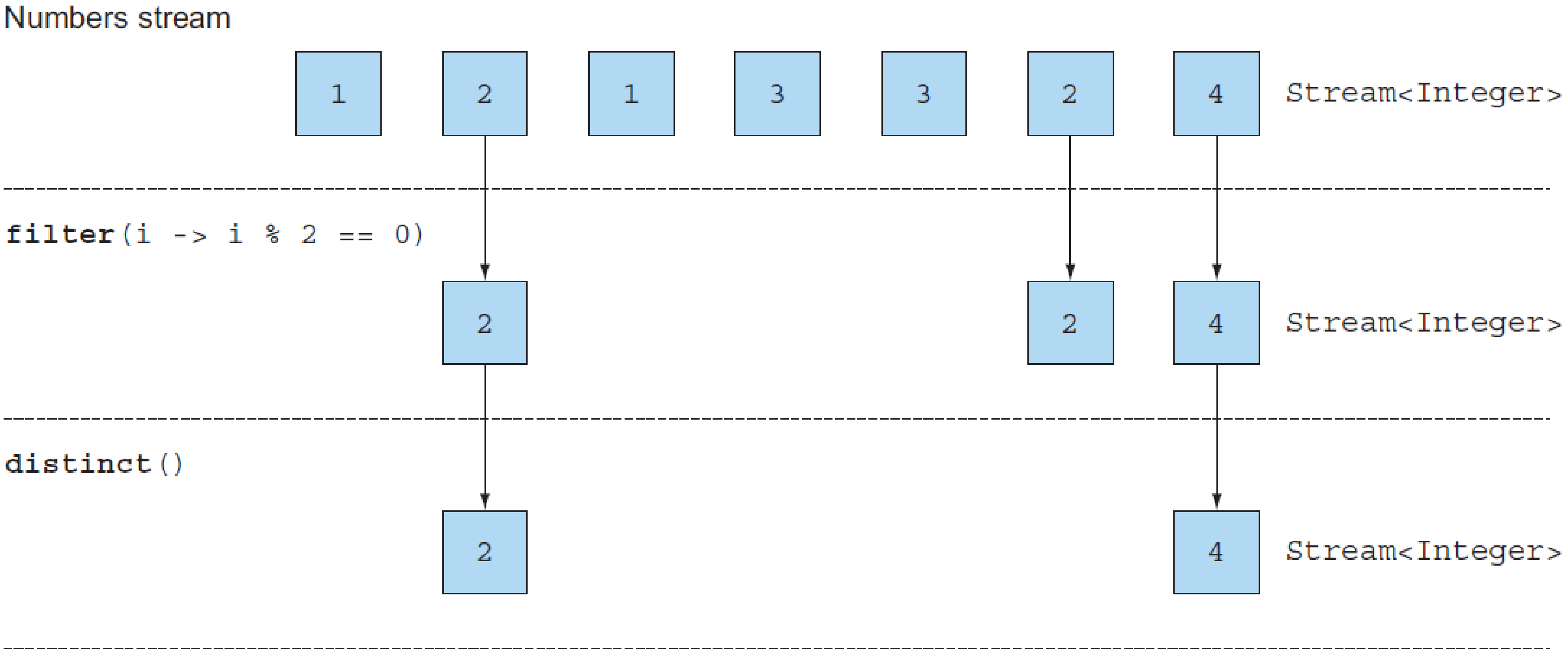
```
// Remove all empty Strings from List
List<String> filtered = strList.stream()
   .filter(s -> !s.isEmpty())
   .collect(Collectors.toList());
System.out.printf("Original List : %s, List without Empty Strings : %s %n",
strList, filtered);

// Create a List with String more than 2 characters
filtered = strList.stream()
   .filter(s -> s.length()> 2)
   .collect(Collectors.toList());
System.out.printf("Original List : %s, filtered list : %s %n", strList, filtered);
```

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 takeWhile and dropWhile

- **takeWhile** stops once it has found an element that fails to match

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

- **dropWhile** throws away the elements at the start where the predicate is false. Once the predicate evaluates to true it stops and returns all the remaining elements,

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

```
List<Dish> specialMenu = Arrays.asList(
  new Dish("seasonal fruit", true, 120, Dish.Type.OTHER),
  new Dish("prawns", 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) );
```

Mapping

- **map** is applied to each element, mapping it into a new element

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

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

Converting using Map

```
// Create List of square of all distinct numbers
List<Integer> numbers = Arrays.asList(9, 10, 3, 4, 7, 3, 4);
List<Integer> distinct = numbers.stream()
  .map( i -> i*i ).distinct()
  .collect(Collectors.toList());
System.out.printf("Original List : %s,  Square Without duplicates : %s %n",
  numbers, distinct);
// Original List : [9, 10, 3, 4, 7, 3, 4],  Square Without duplicates : [81, 100, 9,
16, 49]

// Convert String to Uppercase and join them using coma
List<String> G7 = Arrays.asList("USA", "Japan", "France", "Germany",
  "Italy", "U.K.","Canada");
String G7Countries = G7.stream()
  .map(x -> x.toUpperCase())
  .collect(Collectors.joining(", ", "<", ">")); // delimiter, prefix, suffix
System.out.println(G7Countries);
// <USA, JAPAN, FRANCE, GERMANY, ITALY, U.K., CANADA>
```

Matching

```
if ( Dish.menu.stream().anyMatch(Dish::isVegetarian) ) {
  System.out.println("The menu is (somewhat) vegetarian friendly!!");
}

// all dishes are below 1000 calories
boolean isHealthy = Dish.menu.stream()
  .allMatch(dish -> dish.getCalories() < 1000);

// no dishes are abover 1000 calories
boolean isHealthy = Dish.menu.stream()
  .noneMatch(d -> d.getCalories() >= 1000);
```

Finding

```
Dish.menu.stream()
   .filter(Dish::isVegetarian)
   .findAny() // Returns an Optional describing some element of the stream
   .ifPresent(dish -> System.out.println(dish.getName()));

List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree = someNumbers.stream()
   .map(n -> n * n)                   // 1, 4, 9, 16, 25
   .filter(n -> n % 2 == 0)           // 4, 16
   .findFirst();                      // 4
```

Optional<T>: A container object which may or may not contain a non-null value.
If a value is present, isPresent() will return true and get() will return the value.

Pipelines

- A *pipeline* is a sequence of data-processing operations.

```
List<Person> roaster = Arrays.asList(
            new Person("Kim", Gender.MALE),
            new Person("Lee", Gender.FEMALE),
            new Person("Park", Gender.MALE)
        );
```

```
for ( Person p : roaster ) {
  if ( p.getGender() == Gender.MALE ) {
     String name = p.getName();
     String upperCaseName = name.toUpperCase();
     System.out.println(upperCaseName);
  }
}
```

```
roaster.stream()
  .filter( p -> p.getGender() == Gender.MALE)
  .map( p -> p.getName())
  .map( s -> s.toUpperCase())
  .forEach( s -> System.out.println(s));
```

Pipelines with Method Reference

- Pipeline can be written more simply with method reference

```
roaster.stream()
   .filter( p -> p.getGender() == Gender.MALE)
   .map( p -> p.getName())
   .map( s -> s.toUpperCase())
   .forEach( s -> System.out.println(s));
```

```
roaster.stream()
   .filter( p -> p.getGender() == Gender.MALE)
   .map(Person::getName)
   .map(String::toUpperCase)
   .forEach(System.out::println);
```

Pipelines Example

```
List<String> versions = new ArrayList<>();
versions.add("Lollipop");
versions.add("KitKat");
versions.add("Jelly Bean");
versions.add("Ice Cream Sandwidth");
versions.add("Honeycomb");
versions.add("Gingerbread");

System.out.println("Element whose length is > 5 and contains I or i");
versions.stream()
   .filter(s -> s.length() > 5)
   .map(String::toUpperCase)
   .filter(s -> s.startsWith("I"))
   .forEach(System.out::println);
```

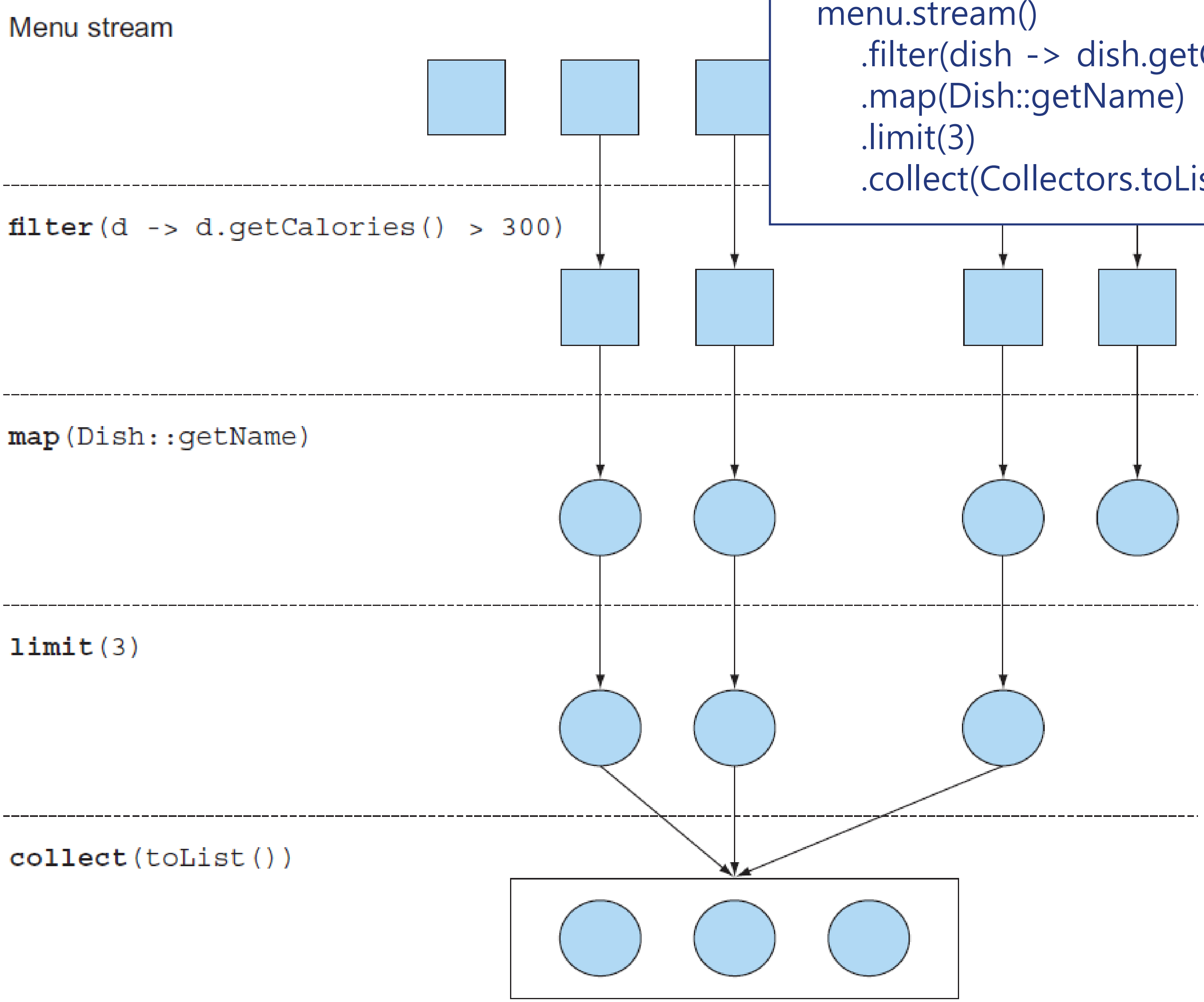
```
ICE CREAM SANDWIDTH
GINGERBREAD
```

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 12, 18);
Integer lcm = listOfNumbers.stream()
   .filter(i -> i % 2 == 0)
   .filter(i -> i % 3 == 0)
   .findFirst().get();
System.out.println("first number divisible by 2 and 3 in the list is : " + lcm); // 6
```

```
List<Dish> lowCalorieDishes = new ArrayList<>();
for (Dish dish: Dish.menu) {      // filter
  if (dish.getCalories() < 400) lowCalorieDishes.add(dish);
}
Collections.sort(lowCalorieDishes, new Comparator<Dish>() { // sort
  public int compare(Dish dish1, Dish dish2) {
     return Integer.compare(dish1.getCalories(), dish2.getCalories());
  }
});
List<String> lowCalorieDishesName1 = new ArrayList<>(); // map: a list of name
for (Dish dish: lowCalorieDishes1) {
  lowCalorieDishesName1.add(dish.getName());
}
System.out.println(lowCalorieDishesName1); // [season fruit, rice]
```

```
List<String> lowCalorieDishesName2 = menu.stream()
  .filter(dish -> dish.getCalories() < 400)
  .sorted(Comparator.comparing(Dish::getCalories))
  .map(Dish::getName)
  .collect(Collectors.toList());
System.out.println(lowCalorieDishesName1); // [season fruit, rice]
```

```
List<String> threeHighCalorieDishNames =
  menu.stream()
      .filter(dish -> dish.getCalories() > 300)
      .map(Dish::getName)
      .limit(3)
      .collect(Collectors.toList());
```



Building Streams

- You can create streams in many ways

- From a sequence of values
- From an array
- From a Collection
- From a Files
- From a generative function to create infinite streams

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

Streams from a Sequence of Values

- You can create a stream with explicit values by using the static method **Stream.of**, which can take any number of parameters

```
Stream<String> stream1 = Stream.of("Modern ", "Java ", "In ", "Action");
stream1.map(String::toUpperCase).forEach(System.out::println);
```

```
MODERN
JAVA
IN
ACTION
```

```
Stream<String> stream2 = Stream.of("Modern ", "Java ", "In ", "Action");
stream2.map(String::toLowerCase).forEach( s -> System.out.print('[' + s + ']') );
```

```
[modern ][java ][in ][action]
```

Streams from Arrays

- 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.map(String::toLowerCase).forEach(System.out::print);

Stream<String> stream2 = Stream.of(strings);
long longWordCount = stream2.filter(s -> s.length() > 4).count();
System.out.println(longWordCount);        // 3

int[] numbers = {1, 2, 3, 4, 5};
long count = Arrays.stream(numbers).count(); // 5

int sum1 = Arrays.stream(numbers).filter( n -> n % 2 == 0).sum(); // 6(=2+4)

// 41(=4*4+5*5)
int sum2 = Arrays.stream(numbers).filter( n -> n >= 4).map(n -> n*n).sum();
```

```
modern java in action
```

Streams from Collection

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

```
List<String> list1 = Arrays.asList("Modern ", "Java ", "in ", "Action ");
long wordCountWithDistinctiveLength = list1.stream()
  .map(s -> s.length())
  .distinct()
  .count();
System.out.println(wordCountWithDistinctiveLength); // 3

List<String> list2 = new ArrayList<>();
String[] helloJava = {"Hello ", "Modern ", "Java "};
Collections.addAll(list2, helloJava);

list2.stream()
  .filter(s -> s.contains("o"))
  .map(String::toUpperCase)
  .forEach(System.out::print); // HELLO MODERN
```

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.

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

Streams from Functions: iterate

- Streams produced by **iterate** and **generate** create values on demand given a function

```
Stream.iterate(0, n -> n + 2)
  .limit(5)
  .forEach(System.out::println);  // 0 2 4 6 8

IntStream.iterate(0, n -> n < 20, n -> n + 4)
  .forEach(System.out::println);  // 0 4 8 12 16

IntStream.iterate(0, n -> n + 4)
  .takeWhile(n -> n < 20)
  .forEach(System.out::println);  // 0 4 8 12 16
```

Streams from Functions: generate

- generate doesn't apply successively a function on each new produced value.
- It takes a lambda to provide new values

```
Stream.generate(Math::random)
        .limit(5)
        .forEach(System.out::println); // five random numbers

IntStream ones = IntStream.generate(() -> 1).limit(5);
ones.forEach(System.out::println); // 1 1 1 1 1

IntStream twos = IntStream.generate(new java.util.function.IntSupplier() {
  public int getAsInt() {
     return 2;
  }
}).limit(3);
twos.forEach(System.out::println); // 2 2 2
```

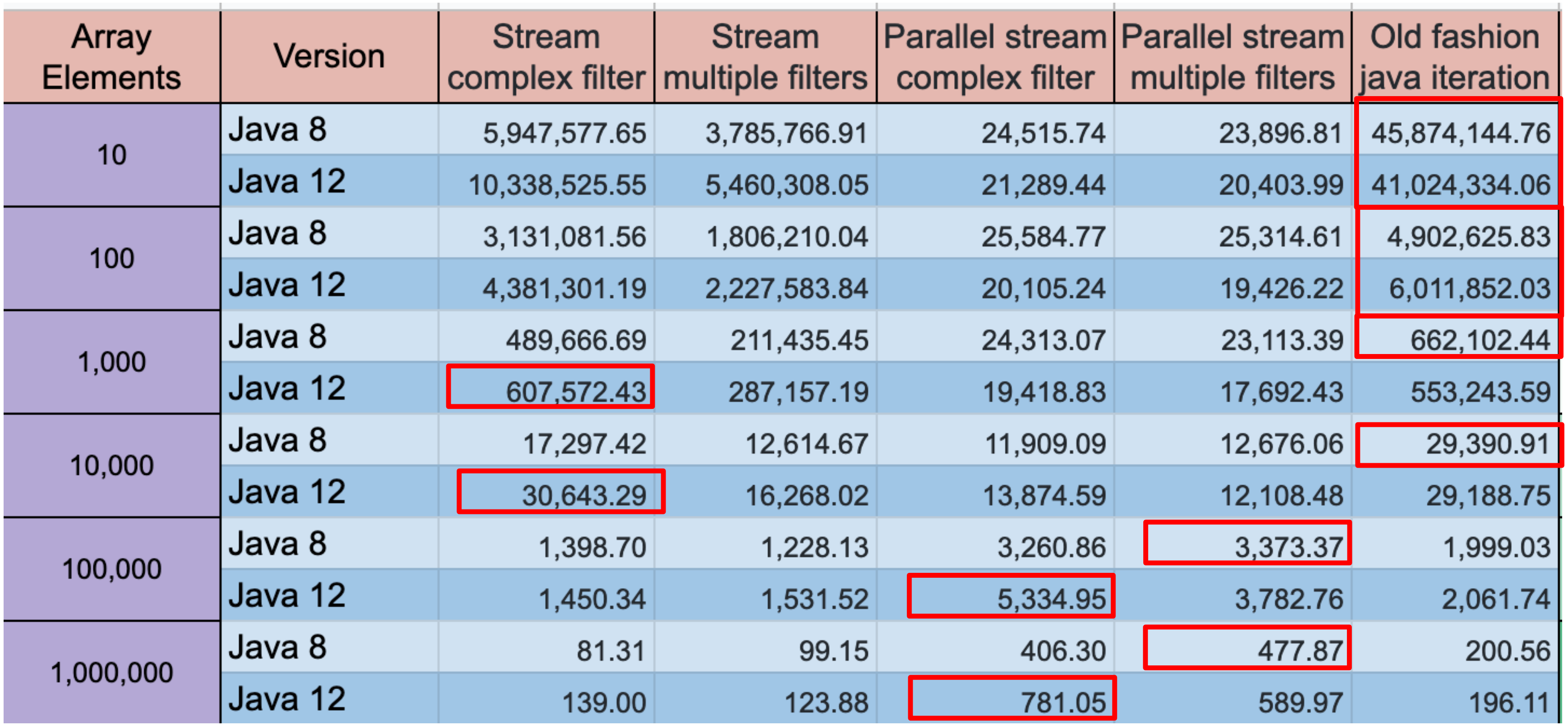
Streams from Functions: generate

```
import java.util.function.IntSupplier;

IntSupplier fib = new IntSupplier() {
        private int previous = 0;
        private int current = 1;
        public int getAsInt() {
            int oldPrevious = this.previous;
            int nextValue = this.previous + this.current;
            this.previous = this.current;
            this.current = nextValue;
            return oldPrevious;
        }
    };
IntStream.generate(fib).limit(5).forEach(System.out::println); // 0 1 1 2 3
```

Performance Summary

- On a relatively small array old fashion loop shows the best results
- For arrays of large size, parallel streams show better results.
- Complex filter is better than multiple filters



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