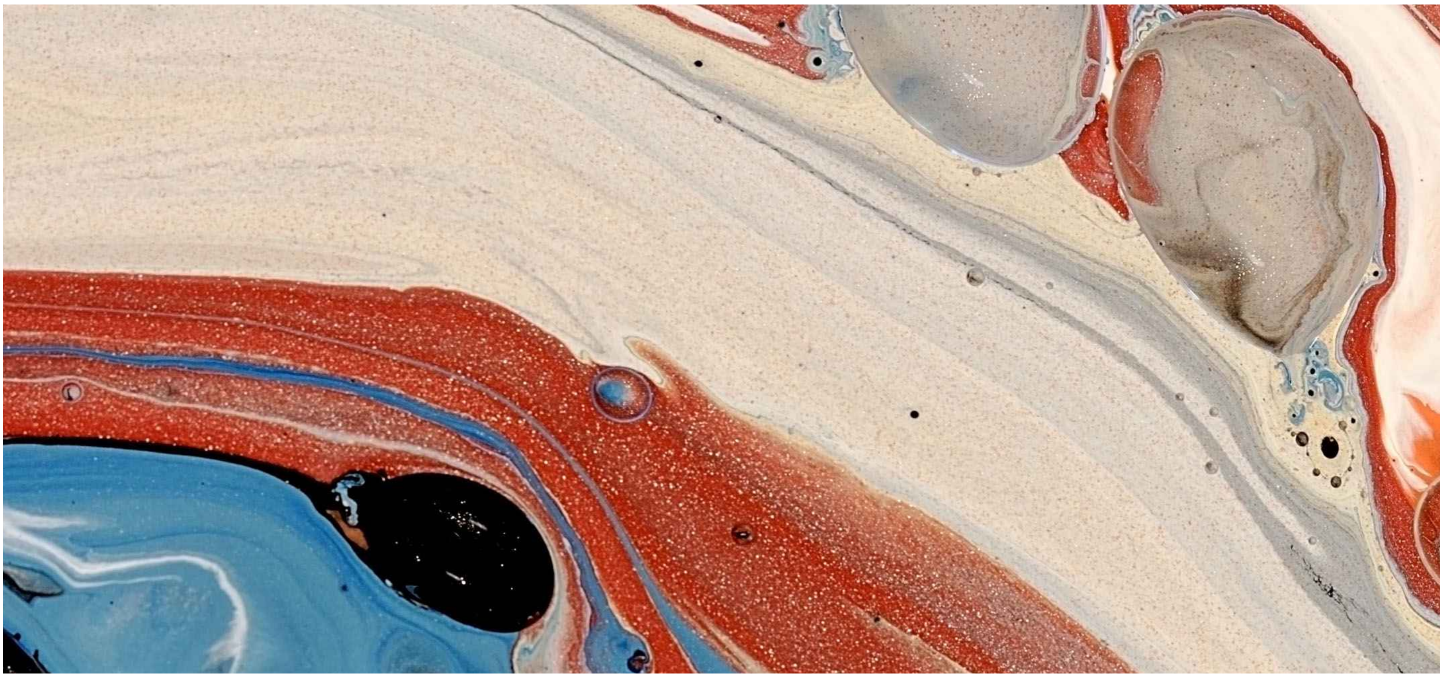

31-Java 8: Stream

Author: Vincent Lau

Note: This material is intended for educational purposes only. All rights reserved. Any unauthorized sharing or copying of this material, in any form, to any individual or party, for any use without prior permission, is strictly prohibited.

Learning Objectives

> Describe what a stream is.
>
> Create various types of stream.
>
> Understand the differences between intermediate and terminal operations.
>
> List the common intermediate and terminal operations.
>
> Perform multiple operations on a stream to return a value or a collection.
>
> Use the *Collectors* class to construct various types of collection.

Overview

What is a *Stream*?

- Introduced in **Java 8**, the *java.util.stream* package contains classes for processing sequences of elements.
- A *stream* is a sequence of data. In Java, it is represented in the type of `Stream<T>`.
- A *stream pipeline* is the operations that run on a stream to produce a result. ***Streams* can be chained together to perform complex operations**.
- As an **alternative to loops**, *streams* are commonly used to operate on the contents of a collection or array.
- Many ***stream*** methods use ***lambda expressions*** to perform operations on objects in a stream.

Creating a Stream

- Streams can be **finite** or *infinite.*
- Streams can be created from different element sources, e.g. a collection or array, with the help of:
  - *empty()*
  - *stream()*
  - *of()*
  - *generate()*
  - *iterate()*

```
// Stream.empty() creates an empty stream.
Stream<String> empty = Stream.empty();
// Stream.of() can create a stream with a single element
Stream<Integer> singleElement = Stream.of(1);
// Stream.of() also accepts varargs
Stream<Integer> multipleElements = Stream.of(1, 2, 3);

// Convert a list into a stream
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> fromList = list.stream();

// Create an infinite stream of random numbers
Stream<Double> randoms = Stream.generate(Math::random);
// Create an infinite stream of odd numbers starting from 1
Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);

// Use limit() to limit the number of elements to produce in a stream
```

```java
Stream<Double> randoms = Stream.generate(Math::random).limit(10);
Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2).limit(10);

// Streams are not executed until the terminal operation is called on them
oddNumbers.forEach(System.out::println);
```

Intermediate Operations vs Terminal Operations

- There are three parts to a **stream pipeline**:
  - ***Source***: Where the stream comes from.
  - ***Intermediate operations:*** **Transforms the stream into another one**. There can be as few or as many intermediate operations as you'd like. Since streams use lazy evaluation, the **intermediate operations do not run until the terminal operation runs**.
  - ***Terminal operation:*** Actually **produces a result**. Since streams can be used only once, the stream is no longer valid after a terminal operation completes.

Quick Example

```java
List<String> names = Arrays.asList("Peter", "Paul", "Mary", "Peter");
long count = names.stream()
                  .distinct()  // intermediate operation, return Stream<String>
                  .count();  // terminal operation, return non-Stream
System.out.println(count);  // prints 3
```

- The *distinct()* method represents an intermediate operation, which creates a new stream of unique elements of the previous stream. And the count() method is a terminal operation, which returns the stream's size.

More Examples

```java
List<String> names = Arrays.asList("Peter", "Carl", "Benny", "Alex");

// filtering
List<String> filteredNames = names.stream()
        .filter(name -> name.contains("A")) // intermediate operation
        .collect(Collectors.toList()); // terminal operation
System.out.println(filteredNames);  // [Alex]

```

```
// mapping
List<String> mappedNames = names.stream()
        .map(name -> name.toUpperCase()) // intermediate operation
        .collect(Collectors.toList()); // terminal operation
System.out.println(mappedNames);  // [PETER, CARL, BENNY, ALEX]

// sorting
List<String> sortedNames = names.stream()
        .sorted()      // intermediate operation, natural order
        .collect(Collectors.toList()); // terminal operation
System.out.println(sortedNames);  // [Alex, Benny, Carl, Peter]

// What is the sorting approach if it is not a String ArrayList? Try it out.
// Do you remember Comparable & Comparator?

// matching
boolean hasAlex = names.stream()
        .anyMatch(name -> name.contains("Alex")); // terminal operation
System.out.println(hasAlex);  // true

// generating a sequence of numbers from 1 to 10
List<Integer> numbers = Stream.iterate(1, n -> n + 1)  // intermediate operation
        .limit(10) // intermediate operation
        .collect(Collectors.toList()); // terminal operation
System.out.println(numbers);  // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// reducing - adding numbers in a list with initial value of 0
Integer reduced = numbers.stream()
        .reduce(0, (a, b) -> a + b); // terminal operation
System.out.println(reduced);  // 55

// finding the maximum element which returns an optional
List<Integer> numbers = Arrays.asList(9992, 2121, 2184, 5539, 3120);
Optional<Integer> opt = numbers.stream()
        .max((o1, o2) -> o1.compareTo(o2)); // terminal operation
opt.ifPresent(System.out::println);  // 9992
```

Working with *Maps*

- It is common to use *Streams* to work with collections such as *Maps*.
- The *Collectors* class provides various *static helper methods* to construct *Map*s such as:
    - *toMap()*

  - *groupingBy() [Nice to have]*
  - *partitioningBy() [Nice to have]*

Examples

```java
Stream<String> animals = Stream.of("lions", "tigers", "bears");
Map<String, Integer> map = animals.collect(
                                Collectors.toMap(s -> s, String::length));
System.out.println(map); // {lions=5, bears=5, tigers=6}

List<Employee> employees = ......
// Group employees by department
// Employees: [department, name]
// 1, John
// 1, Mary
// 2, Jason
// 3, Eric
// 3, Oscar
// Result: Map<Department, List<Employee>>
// Entry 1: 1, [{1, John}, {1, Mary}]
// Entry 2: 2, [{2, Jason}]
// Entry 3: 3, [{3, Eric}, {3, Oscar}]
Map<Department, List<Employee>> byDept = employees.stream()
                                .collect(Collectors.groupingBy(e -> e.getDepartment()));

// Compute sum of salaries by department
// Employees: [department, name, salary]
// 1, John, 10000
// 1, Mary, 20000
// 2, Jason, 15000
// 3, Eric, 23000
// 3, Oscar, 30000
// Result: Map<Department, Integer>
// Entry 1: 1, 30000
// Entry 2: 2, 15000
// Entry 3: 3, 53000
Map<Department, Integer> totalByDept = employees.stream()
                                .collect(Collectors.groupingBy(Employee::getDepartment,
                                Collectors.summingInt(Employee::getSalary)));

// Partition students into passing and failing
// students: [id, name, grade]
// 1, John, 40
// 2, Mary, 30
// 3, Oscar, 80
// PASS_THRESHOLD = 50
// Result: Map<Boolean, List<Student>>
// Entry 1: true, [{3, "Oscar", 80}]
// Entry 2: false, [{1, "John", 40}, {2, "Mary", 30}]
Map<Boolean, List<Student>> passingFailing = students.stream()
                                .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));

// .collect(Collectors.partitioningBy(...)) return Map<Boolean, List<Student>>
```

Reading Examples

Since `map` is a lazy operation, the following code will print nothing. This `Stream` is missing a terminal operation which would execute it, which would invoke the intermediate operations.

```
Stream.of(1, 2, 3).map(i -> {
    System.out.println(i);
    return i;
});
// Print nothing, due to no terminal operation
// intermediate operation is lazy operation

list.stream().filter(a -> a > 20 && a < 7);
// Return a Stream
// No element from the list has been filtered as no terminal operation here.
```

- To determine the result of count(), the map() is irrelevant. Thus, this code will still print nothing. But since `count()` is a terminal operation, the stream is processed and `count` gets the value `3` assigned.

```
long streamCount = Stream.of(1, 2, 3) //
.map(i -> {
    System.out.println(i);
    return i;
}).count();
// streamCount = 3
```

Questions

- List the common intermediate and terminal operations.
- Perform various operations on streams to return a value or a collection.
- Look up Java documentation to find the right helper methods to construct maps based on the requirements.