



COMP8710 Advanced Java for  
Programmers

## Lecture 13 Stream (1)

---

Yang He

# Topics

- Introduction to Java Streams
- Stream operations
- Collections vs. streams

# What are streams? (1)

- Collections is the most heavily used API in Java
- Stream
  - Is an update to the Java API that lets you manipulate collections of data in a *declarative* way
  - Can be processed in *parallel transparently*, without you having to write any multithreaded code
  - Represents a sequence of elements from a source that supports data processing operations
  - Takes Collections, Arrays, or I/O resources as input source

# What are streams? (2)

- E.g. Print all even numbers in an ascending order

```
var myList = Arrays.asList(3, 2, 5, 1, 6, 4);
```

- Before Java 8:

```
Collections.sort(myList);  
for (var i: myList) {  
    if (i % 2 == 0 ) {  
        System.out.println(i);  
    }  
};
```

# What are streams? (3)

- Using stream:

```
myList.stream().filter( i -> i % 2 == 0 )  
          .sorted()  
          .forEach(System.out::println);
```

- Why is this better?

- Declarative — more concise and readable
- Composable — greater flexibility
- Parallelisable — better performance

# Stream operations (1)

- Streams support database-like operations and common operations from functional programming languages to manipulate data, e.g. `filter`, `find`
- Stream operations can be executed either sequentially or in parallel
- Two important characteristics of stream operations:
  - Pipelining
    - Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline
  - Internal iteration
    - In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration internally behind the scenes for you

# Stream operations (2)

- There are two kinds of operations on streams:
  - Intermediate operations:
    - Operations that can be pipelined because they return another stream
    - E.g. `filter`, `sorted`
  - Terminal operations
    - Produce a result from a stream pipeline (i.e. any non-stream value, e.g. `List`)
    - E.g. `forEach` is a terminal operation that returns `void` and applies a lambda to each element of the stream
    - Other examples: `collect`, `count`, ...

# Working with streams

- Three steps to work with streams:
  - 1) a data source to perform a query on
  - 2) a chain of intermediate operations
  - 3) a terminal operation to collect the results



# Demo: Stream operations

---

`StreamDishes.java`

# An example (1)

- Define a method that takes a List of Dishes and returns the names of the dishes that are low in calories (<400 calories), sorted by number of calories.

```
public static List<String> getLowCaloricDishesNames(List<Dish> dishes) {  
    List<Dish> lowCaloricDishes = new ArrayList<>();  
  
    dishes.forEach(d -> {  
        if (d.calories() < 400) {  
            lowCaloricDishes.add(d);  
        }  
    });  
  
    lowCaloricDishes.sort(Comparator.comparing(Dish::calories));  
  
    List<String> names = new ArrayList<>();  
    lowCaloricDishes.forEach(d -> names.add(d.name()));  
    return names;  
}
```

# An example (2)

- Using Stream:

```
public static List<String> getLowCaloricDishesNames(List<Dish> dishes) {  
    return dishes.stream()  
        .filter(d -> d.calories() < 400)  
        .sorted(Comparator.comparing(Dish::calories))  
        .map(Dish::name)  
        .toList();  
}
```



# Exercise

- What does this print out?

```
menu.stream()
    .filter(d -> {
        System.out.println(" Filtering: " + d.name() + " " + d.calories());
        return d.calories() > 500;
    })
    .map(d -> {
        System.out.println(" Mapping: " + d.name());
        return d.name();
    })
    .limit(3)
    .forEach(d -> System.out.println("Dish: " + d));
```



# Answer

## ■ Output:

```
Filtering: beef 700
Mapping: beef
Dish: beef
Filtering: chicken 400
Filtering: french fries 530
Mapping: french fries
Dish: french fries
Filtering: rice 350
Filtering: season fruit 120
Filtering: pizza 550
Mapping: pizza
Dish: pizza
```

*Intermediate operations don't perform any processing until a terminal operation is invoked on the stream pipeline: they are lazy*

*Note: there are 4 dishes with calories > 500, only the first 3 are selected*

# Peeking into Streams (1)

- Signature: `Stream<T> peek(Consumer<T> action)`
- It consumes a stream without ending the pipeline of operations, i.e. intermediate operation
- It is mainly used for debugging, where you want to see the elements at a certain point in a pipeline

# Peeking into Streams (2)

- E.g.

```
menu.stream()  
    .peek(d -> System.out.println("Starting: " + d))  
    .filter(d -> d.calories() > 500)  
    .peek(d -> System.out.println("  Filtering: " + d.name() + " " + d.calories()))  
    .map(Dish::name)  
    .peek(d -> System.out.println("  Mapping: " + d))  
    .limit(3)  
    .peek(d -> System.out.println("  Limited: " + d))  
    .forEach(d -> System.out.println("Dish: " + d));
```

# Some stream operations

- Intermediate operations:
  - `Stream<T> filter(Predicate<T> p)`
  - `Stream<R> map(Function<T,R> f)`
  - `Stream<T> sorted(Comparator<T> c)`
  - `Stream<T> distinct()`
  - `Stream<T> limit(long n)`
- Terminal operations:
  - `forEach` consumes each element from a stream and applies a lambda to each of them (returns nothing, i.e. void)
  - `count` returns the number of elements in a stream (returns a long)
  - `collect` reduces the stream to create a collection such as a List, a Set, a Map, etc.



# Collections vs. streams (1)

- Both provide interfaces to data structures representing a *structured set of values* of a given type
- A stream is like a **lazily constructed** collection
  - Values are computed when they are solicited by a consumer
- Collections are an in-memory data structure that holds all the values

# Collections vs. streams (2)

- Like iterators, streams can be **traversed only once**, e.g.

```
List<String> letters = Arrays.asList("A", "B", "C");  
Stream<String> s = letters.stream();  
s.forEach(x -> System.out.println(x + "1"));  
s.forEach(x -> System.out.println(x + "2"));    // Not allowed!
```

How to fix this?



- Collection interface requires the iteration to be done by the user, for-each or with an iterator
- But you can't add/remove elements to/from a stream!

# Creating a stream

- E.g.

```
Stream<String> letters = Stream.of("a", "b", "c");
```

```
Stream<Apple> apples = inventory.stream();
```

```
IntStream num = IntStream.rangeClose(1, 100);
```

# More on stream operations

- Filtering, slicing, and matching:
  - `filter`
  - `distinct`
  - `limit`, `skip`
  - `map`
  - `flatMap`
- Finding, matching, and reducing:
  - `anyMatch`, `noneMatch`, `allMatch`
  - `findAny`, `findFirst`
  - `reduce`