Lecture 9

Functional-style Data Processing

Introduction to Streams

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

René Witte

Concordia University

From Collections to Streams

Given a list of `Dish` objects, return the names of dishes low in calories, sorted by calories

```
// Step 1: Filter
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: menu){
  if(d.getCalories() < 400){
    lowCaloricDishes.add(d);
  }
}

// Step 2: Sort
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
  public int compare(Dish d1, Dish d2){
    return Integer.compare(d1.getCalories(), d2.getCalories());
  }
});

// Step 3: Process
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d: lowCaloricDishes){
  lowCaloricDishesName.add(d.getName());
}
```

René Witte

With Java 8 Streams

```
List<String> lowCaloricDishesName =
    menu.stream()
        .filter(d -> d.getCalories() < 400)
        .sorted(comparing(Dish::getCalories))
        .map(Dish::getName)
        .collect(toList());
```

René Witte

Multicore architecture: Run in parallel

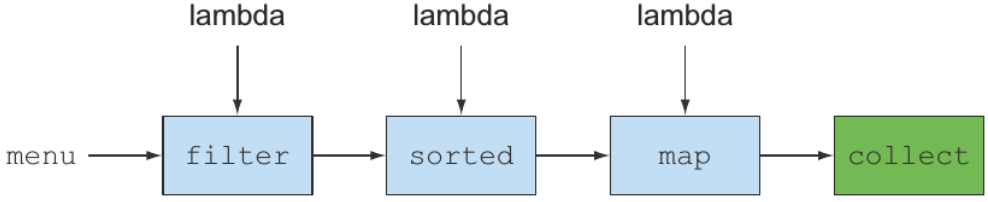
```
List<String> lowCaloricDishesName =
    menu.parallelStream()
        .filter(d -> d.getCalories() < 400)
        .sorted(comparing(Dish::getCalories))
        .map(Dish::getName)
        .collect(toList());
```

René Witte

Stream Pipelines

Chaining stream operations forming a stream pipeline

René Witte



Stream Processing Example

René Witte

Group dishes by type into a `Map`

```
Map<Dish.Type, List<Dish>> dishesByType =
    menu.stream().collect(groupingBy(Dish::getType));
```

Output Example

```
{FISH=[prawns, salmon],
 OTHER=[french fries, rice, season fruit, pizza],
 MEAT=[pork, beef, chicken]}
```

René Witte
Concordia University

New Streams API (`java.util.stream.Stream`)

With Java 8 you can write code that is

Declarative: More concise and readable

Composable: Greater flexibility

Parallelizable: Better performance

Outline

Motivation
Streams
Example
Definition
Streams vs. collections
Stream Traversal
Stream Iteration
Stream Operations
Intermediate operations
Terminal operations
Working with streams
Summary
Notes and Further Reading

Example Class: Dish

```
public class Dish {
  private final String name;
  private final boolean vegetarian;
  private final int calories;
  private final Type type;

  public Dish(String name, boolean vegetarian, int calories, Type type) {
    this.name = name;
    this.vegetarian = vegetarian;
    this.calories = calories;
    this.type = type;
  }

  public String getName() { return name; }
  public boolean isVegetarian() { return vegetarian; }
  public int getCalories() { return calories; }
  public Type getType() { return type; }

  @Override
  public String toString() {
    return name;
  }

  public enum Type { MEAT, FISH, OTHER }
}
```

René Witte

Example Data: Menu items

René Witte


```
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) );
```

Stream Definition

René Witte

Stream

A stream is a sequence of elements from a source that supports data processing operations.

Stream Characteristics

René Witte

Details

Sequence of elements: A stream provides an interface to a sequenced set of values of a specific element type. Unlike collections, they focus on expressing computations (e.g., `filter`, `sorted`, `map`).
Collections are about data; streams are about computations.

Source: Streams consume from a data-providing source such as collections, arrays, or I/O resources. 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.

Pipelining: Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline. This enables certain optimizations, such as laziness and short-circuiting.

Internal iteration: In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration behind the scenes for you.

Stream Example

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

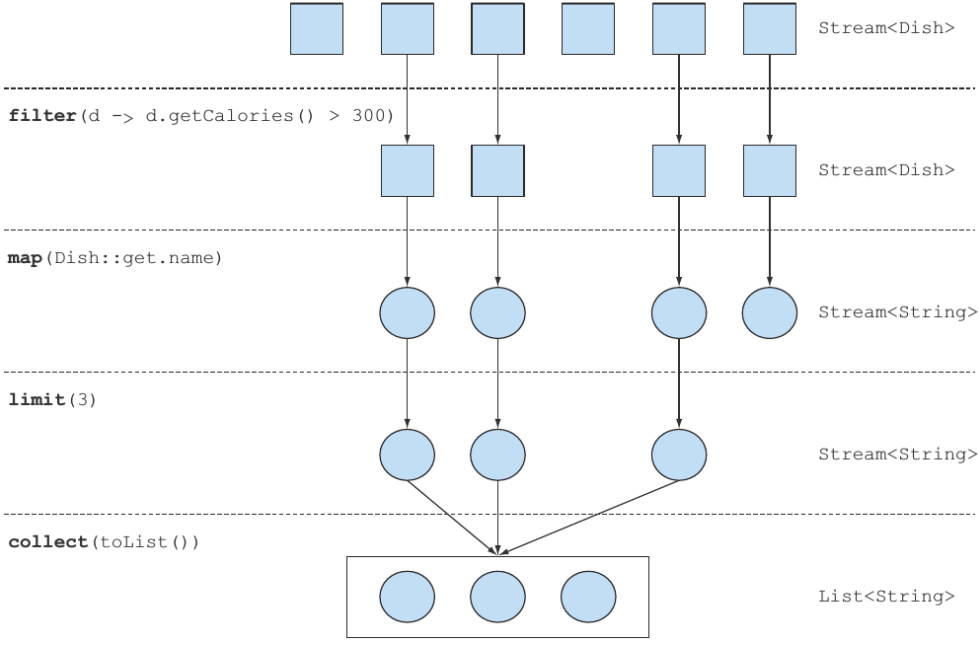
```
import static java.util.stream.Collectors.toList;

List<String> threeHighCaloricDishNames =
  menu.stream()
      .filter(d -> d.getCalories() > 300)
      .map(Dish::getName)
      .limit(3)
      .collect(toList());

System.out.println(threeHighCaloricDishNames);
```

René Witte

Menu stream



René Witte

René Witte
Concordia University

Stream operations in the example

`filter` takes a lambda to exclude certain elements from the stream

```
d -> d.getCalories() > 300
```

`map` takes a lambda to transform an element into another one or to extract information

```
d -> d.getName()
```

`limit` truncates a stream to a given number of elements

```
limit(3)
```

`collect` converts a stream into another form (e.g., a `List`)

```
toList()
```

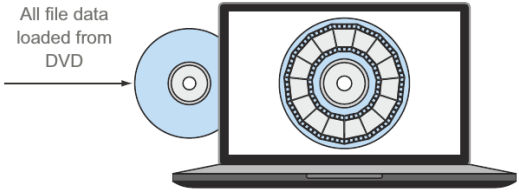
Streams vs. collections

René Witte

A collection in Java 8 is like a movie stored on DVD

Eager construction means waiting for computation of ALL values

All file data loaded from DVD

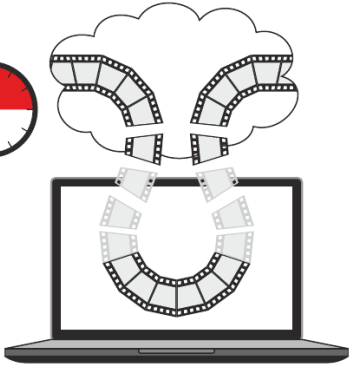


Like a DVD, a collection holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.

A stream in Java 8 is like a movie streamed over the internet.

Lazy construction means values are computed only as needed.

Internet



Like a streaming video, values are computed as they are needed.

Streams vs. collections

René Witte
Concordia University

Some differences

Collections:

- Can add or remove elements
- Contents eagerly constructed before processing
- E.g., *"make a list of all even numbers"* before processing will not work
- A set of values spread out in space (computer memory) – all exist at the same time

Streams:

- Conceptually fixed data structure (you can't add or remove elements)
- Elements are lazily constructed – only *as* and *when* required (computed on demand)
- Can process a *stream of even numbers* – next number will only be computed when needed
- Compare with, e.g., Internet search (showing next 10 search results) or "infinite scrolling" on web pages
- A set of values spread out in time – not all of them have to exist in the same space (computer memory)

Stream Traversal

René Witte

A stream can be traversed only once

```
List<String> title = Arrays.asList("Java8", "In", "Action");
Stream<String> s = title.stream();
s.forEach(System.out::println);
```

OK!

```
s.forEach(System.out::println);
```

```
java.lang.IllegalStateException:
stream has already been operated upon or closed.
```

Collection Iterators: External Iteration

René Witte

Iterating with `for-each` loop

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

Translated by compiler into `Iterator` object

```
List<String> names = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
  Dish d = iterator.next();
  names.add(d.getName());
}
```

Syntactic Sugar

The `for-each` construct is an example of so-called syntactic sugar

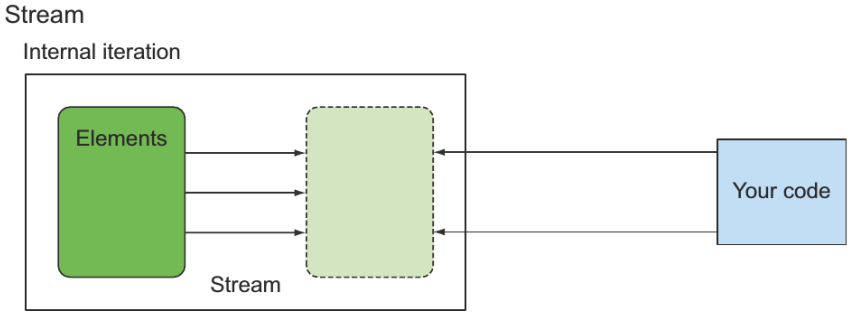
Stream Iterators

René Witte

Streams use *internal* iteration

```
List<String> names = menu.stream()
                          .map(Dish::getName)
                          .collect(toList());
```

Internal vs. external iteration



René Witte
Concordia University

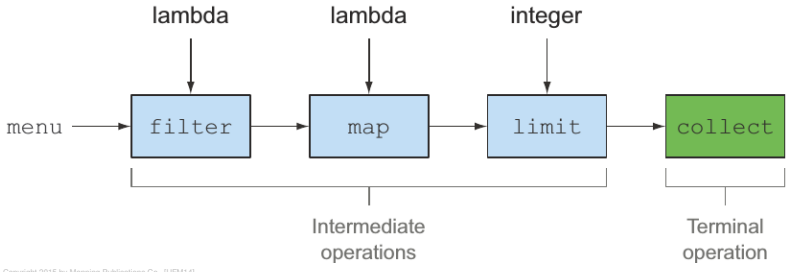
Outline

René Witte
Concordia University

Stream Operation Categories

Stream Pipeline Example

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



Two groups of operations: Intermediate & Terminal

- `filter`, `map`, `limit` etc. can be connected to form a pipeline
- `collect` causes the pipeline to be executed and closes it

René Witte
Concordia University

Intermediate operations

Understanding lazy execution

```
List<String> names =
  menu.stream()
      .filter(d -> {
        System.out.println("filtering" + d.getName());
        return d.getCalories() > 300; })
      .map(d -> {
        System.out.println("mapping" + d.getName());
        return d.getName(); })
      .limit(3)
      .collect(toList());

System.out.println(names);
```

Output

```
filtering pork
mapping pork
filtering beef
mapping beef
filtering chicken
mapping chicken
[pork, beef, chicken]
```

René Witte

Terminal operations

René Witte

Definition

Terminal operations produce a result from a stream pipeline. A result is any nonstream value, e.g., `List`, `Integer`, `void`.

Example

```
menu.stream().forEach(System.out::println);
```

Identify the operations

René Witte
Concordia University

What are the *intermediate* and *terminal* operations in this pipeline?

```
long count = menu.stream()
                 .filter(d -> d.getCalories() > 300)
                 .distinct()
                 .limit(3)
                 .count();
```

Working with streams

René Witte
Concordia University

In general, you need three items

- A data source (such as a collection) to perform a query on
- A chain of intermediate operations that form a stream pipeline
- A terminal operation that executes the stream pipeline and produces a result

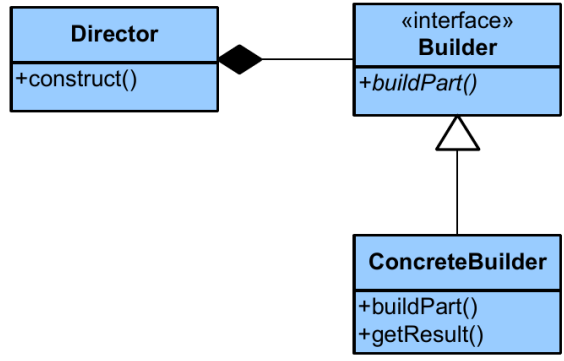
Similar idea to the Builder design pattern.

René Witte
Concordia University

Builder

Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Intermediate and Terminal Stream Operations (excerpt)

René Witte

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>`		

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

Summary

René Witte
Concordia University

Streams

- A stream is a sequence of elements from a source that supports data processing operations.
- Streams make use of internal iteration: the iteration is abstracted away through operations such as `filter`, `map`, and `sorted`.
- There are two types of stream operations: intermediate and terminal operations.
- Intermediate operations (e.g., `filter`, `map`) return a stream and can be chained together. They're used to set up a pipeline of operations but don't produce any result.
- Terminal operations such as `forEach` and `count` return a nonstream value and process a stream pipeline to return a result.
- The elements of a stream are computed on demand.

Outline

Reading Material

René Witte

Required

- [UFM14, Chapter 4] (Introducing Streams)

Supplemental

- [GHJV95] (Builder Design Pattern)

References

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.