Multicore Programming
Java Streams

Louis-Claude Canon
louis-claude.canon@univ-fcomte.fr

Bureau 414C

Master 1 computer science – Semester 8
Motivations

- Express simple iterations in a declarative way without writing the loop logic.
- Simple way to parallelize without writing the thread logic.
- Java Streams introduced in Java 8 (2014, LTS, support until 2025).
- Called *iterators* in Rust.
Outline

Concepts

Stream Definition

Streams vs. Collections
Intermediate and Terminal Operations

Stream Operations

Summary and References
Example without Streams

List<Dish> highCal = new ArrayList<>();
for (Dish dish: menu)
    if (dish.getCalories() > 300)
        highCal.add(dish);
List<String> highCalName = new ArrayList<>();
for (Dish dish: highCal)
    lowCalName.add(dish.getName());
lowCalName = highCalName.subList(0, 3);
Example with Streams

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

\[
\text{filter}(d \rightarrow d \text{.getCalories()} > 300)
\]

\[
\text{map}(	ext{Dish::getName})
\]

\[
\text{limit}(3)
\]

\[
\text{toList()}
\]

\[
\text{Stream<Dish>}
\]

\[
\text{Stream<String>}
\]

\[
\text{List<String>}
\]
Benefits

- declarative (no flow control): readable and concise
- composable/flexible/extensible: easy to modify
- parallelizable
A stream is a sequence of elements coming from a source and that supports data-processing operations.

sequence of elements   interface to access a sequenced set of values (not necessarily sorted)
source    data are provided by a collection, array or I/O resource
data-processing operations support for database-like or functional operations
Characteristics

**Pipelining** stream operation may return a stream

**Internal iteration** iteration (loop logic) is implicit
Outline

Concepts
- Stream Definition
- Streams vs. Collections
- Intermediate and Terminal Operations

Stream Operations

Summary and References
Similarity between Streams and Collections

- Iteration on a sequenced set of values.
Differences

- Lazy vs. Eager Evaluation: values are only computed as needed.
- Traversable only once: a stream cannot be consumed multiple times.
- Internal vs. external iterations: iterations can be transparently done in parallel or in a different more optimized order.
Outline

Concepts

Stream Definition
Streams vs. Collections
Intermediate and Terminal Operations

Stream Operations

Summary and References
Intermediate Operations

Intermediate operations:

- return another stream
- only define processing operations that will be performed (lazy evaluation)
Example

```java
List<String> names = menu.stream()
    .filter(dish -> {
        System.out.println("filtering:" + dish.getName());
        return dish.getCalories() > 300;
    })
    .peek(dish -> System.out.println("mapping:" + dish.getName()))
    .map(Dish::getName)
    .limit(3)
    .toList();
```
Example Output

Example with 10 dishes in menu:

<table>
<thead>
<tr>
<th>filtering: salad</th>
</tr>
</thead>
<tbody>
<tr>
<td>filtering: pork</td>
</tr>
<tr>
<td>mapping: pork</td>
</tr>
<tr>
<td>filtering: beef</td>
</tr>
<tr>
<td>mapping: beef</td>
</tr>
<tr>
<td>filtering: chicken</td>
</tr>
<tr>
<td>mapping: chicken</td>
</tr>
</tbody>
</table>

Terminal Operations

Terminal operations produce a result from a stream pipeline (*lazy evaluation*).
Summary

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

Stream Operations

- Filtering
- Slicing
- Mapping
- Finding and Matching
- Reducing

Summary and References
Java: Lambda Function and Method Reference

Equivalences:

- \((a, b) \rightarrow a + b\)
- \((\text{int } a, \text{int } b) \rightarrow a + b\)
- \((\text{int } a, \text{int } b) \rightarrow \{ \text{return } a + b; \}\)
- \(\text{Integer::sum}\)

```java
class Integer {
    public static int sum(int a, int b) {
        return a + b;
    }
}
```
Java: Special Function Names

From package java.util.function:

- Supplier void -> T
- Consumer T -> void
- Function T -> R (for map)
- UnaryOperator T -> T
- Predicate T -> boolean (for filter)

Equivalences with different arity:

- BiConsumer (T, U) -> void
- BiFunction (T, U) -> R
- BinaryOperator (T, T) -> T (for reduce)
- BiPredicate (T, U) -> boolean
Outline

Concepts

Stream Operations
  Filtering
  Slicing
  Mapping
  Finding and Matching
  Reducing

Summary and References
Predicate

- A predicate is a function returning a boolean.
- Used to filter values.
**filter**

Menu stream

```plaintext
filter(Dish::isVegetarian)
```

toList()
ReactiveX

filter { }
distinct

Numbers stream

Stream<Integer>

2

filter(i -> i % 2 == 0)

Stream<Integer>

2

2

4

distinct()

Stream<Integer>

2

4
distinct
Outline

Concepts

Stream Operations

- Filtering
- Slicing
- Mapping
- Finding and Matching
- Reducing

Summary and References
Definition

- Slicing: skip elements by ignoring either first or last elements.
- With a fixed number of elements: limit, skip.
- With a predicate (from Java 9): takeWhile, dropWhile.
**limit**

Menu stream

\[
\text{filter}(d \rightarrow d.get\text{Calories}() > 300)
\]

\[
\text{limit}(2)
\]

\[
\text{collect}(\text{toList()})
\]

List\langle Dish\rangle
limit(2)
skip

Menu stream

\[
\text{filter}(d \rightarrow d.getCalories() > 300)
\]

\[
\text{skip}(1)
\]

\[
\text{collect}(\text{toList()})
\]
skip(1)
takeWhile

List<String> slicedMenu = specialMenu.stream()
  .takeWhile(dish -> dish.getCalories() < 320)
  .toList();

- The methods `takeWhile` and `dropWhile` are more efficient than a filter when you know that the source is sorted.
takeWhile(≠)
dropWhile( ≠ )
Outline

Concepts

Stream Operations
  Filtering
  Slicing
  Mapping
  Finding and Matching
  Reducing

Summary and References
Stream Operations  Mapping

**map**

Applying a function:

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

Types:

1. **menu**: List<Dish>
2. **menu.stream()**: Stream<Dish>
3. **menu.stream().map(...)**: Stream<String>
4. **menu.stream().map(...).toList()**: List<String>
flatMap

Flattening streams:

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

- Apply a function that returns a stream (as map).
- Merge the resulting streams into a single one.
Stream of words

map(s -> s.split(""))

flatMap(Arrays::stream)

distinct()

collect(toList())
Difference between map and flatMap

Types:
1. words: List<String>
2. words.stream(): Stream<String>
3. words.stream().map(s -> s.split("")): Stream<String[]>
4. Arrays::stream: String[] -> Stream<String>
5. words.stream().map(s -> s.split("")).map(Arrays::stream): Stream<Stream<String>>
flatMap { ... }
Stream Operations Finding and Matching

Outline

Concepts

Stream Operations

Filtering
Slicing
Mapping
Finding and Matching
Reducing

Summary and References
Matching: anyMatch, allMatch, noneMatch

Rely on a predicate:

```java
boolean isHealthy = menu.stream()
    .allMatch(dish -> dish.getCalories() < 1000);

boolean isHealthy = menu.stream()
    .noneMatch(d -> d.getCalories() >= 1000);
```
allMatch(◯)
Stream Operations  Finding and Matching

Short-Circuiting

- Interrupt the processing iteration on the stream as soon as a condition is met.
- Similar to evaluation mechanism with || and &&.
- Equivalent to a break in a loop.
- Apply to limit and takeWhile as well.
- Allow infinite stream.
Finding: `findAny`, `findFirst`

```java
Optional<Dish> dish = menu.stream()
    .filter(Dish::isVegetarian)
    .findAny();
```

- `findAny` is better for parallelization than `findFirst`
Java: Optional<T>

An object that is either defined or null (explicit management of null references).

```java
boolean isPresent()
ifPresent(Consumer<T> block)
T get()
T orElse(T other)
```
Outline

Concepts

Stream Operations
  Filtering
  Slicing
  Mapping
  Finding and Matching
  Reducing

Summary and References
Reducing a list of elements into a single value (also called fold):

```java
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```
Numbers stream

\text{reduce}(0, (a, b) \rightarrow a + b)

Stream\langle\text{Integer}\rangle

\text{Integer} = 21
reduce \{ (\Diamond, \square) \rightarrow \Diamond \}
No Initial Value

```java
Optional<Integer> product = numbers.stream()
    .reduce((a, b) -> (a * b));
```
Maximum and Minimum

```java
Optional<Integer> max = numbers.stream()
    .reduce(Integer::max);
Optional<Integer> min = numbers.stream()
    .reduce(Integer::min);
```
Stateless vs. Stateful

- reduce is stateful (with a bounded state), i.e. it keeps an intermediate result (called an accumulator).
- sorted and distinct are even more stateful (unbounded state): they need to buffer all the elements of the stream to proceed.
Can use the natural order or a custom comparison criterion.

```java
menu.stream()
  .sorted(comparing(Dish::getCalories))
  .toList();
```
Java: Comparator<T>

- A comparator is used to compare two objects.
- A static helper function builds a comparator. It requires a function that extracts the key on which the objects must be compared to.

```java
int compare(T o1, T o2)
static Comparator<T> comparing(Function<T, Comparable> keyExtractor)
```
Summary

In addition to the distinction between intermediate and terminal operations, operations are characterized by whether they:

- allow short-circuiting
- return an optional (for terminal operations)
- are stateful (bounded or not)
<table>
<thead>
<tr>
<th>Operation</th>
<th>terminal</th>
<th>stateful</th>
<th>short-circuiting</th>
<th>interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td></td>
<td></td>
<td></td>
<td>Predicate</td>
</tr>
<tr>
<td>map</td>
<td></td>
<td></td>
<td></td>
<td>Function</td>
</tr>
<tr>
<td>flatMap</td>
<td></td>
<td></td>
<td></td>
<td>Function*</td>
</tr>
<tr>
<td>sorted</td>
<td></td>
<td>✓(unbounded)</td>
<td></td>
<td>Comparator</td>
</tr>
<tr>
<td>distinct</td>
<td></td>
<td>✓(unbounded)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>limit</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>long</td>
</tr>
<tr>
<td>skip</td>
<td>✓</td>
<td></td>
<td></td>
<td>long</td>
</tr>
<tr>
<td>takeWhile</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>Predicate</td>
</tr>
<tr>
<td>dropWhile</td>
<td>✓</td>
<td></td>
<td></td>
<td>Predicate</td>
</tr>
<tr>
<td>allMatch</td>
<td>✓(boolean)</td>
<td></td>
<td>✓</td>
<td>Predicate</td>
</tr>
<tr>
<td>noneMatch</td>
<td>✓(boolean)</td>
<td></td>
<td></td>
<td>Predicate</td>
</tr>
<tr>
<td>anyMatch</td>
<td>✓(boolean)</td>
<td></td>
<td></td>
<td>Predicate</td>
</tr>
<tr>
<td>findAny</td>
<td>✓(Optional)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>findFirst</td>
<td>✓(Optional)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reduce</td>
<td>✓(Optional)</td>
<td>(✓)</td>
<td></td>
<td>BinaryOperator</td>
</tr>
<tr>
<td>toList</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Outline

Concepts

Stream Operations

Summary and References
Official Documentation

- Documentation of package stream
- Documentation of interface Stream
Demonstration

Compute the list of all square numbers between 1 and 5.

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream()
        .filter(i -> {
            var x = Math.floor(Math.sqrt(i));
            return x * x == i;
        })
        .toList();
```
Summary

- A stream is a sequence of elements from a source that supports data-processing operations.
- The iteration is abstracted away and computed on demand ("lazily").
- Intermediate operations return a stream.
- Terminal operations return a result.
- Some operations use short-circuiting and some are stateless.