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 Operations

Summary and References
Outline

Concepts
  Stream Definition
    Streams vs. Collections
    Intermediate and Terminal Operations

Stream Operations

Summary and References
Example without Streams

```java
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

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

\[ \text{map}(\text{Dish}::\text{name}) \]

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

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

```java
filtering: salad
filtering: pork
mapping: pork
filtering: beef
mapping: beef
filtering: chicken
mapping: chicken
```
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.
Stream Operations

Outline

Concepts

Stream Operations
  Filtering
  Slicing
  Mapping
  Matching and Finding
  Reducing

Summary and References
Java: Lambda Function and Method Reference

Equivalences:

▶ \((a, b) \rightarrow a + b\)
▶ \((\text{int } a, \text{int } b) \rightarrow a + b\)
▶ \((\text{int } a, \text{int } b) \rightarrow \{ \text{return } a + b; \}\)
▶ `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
  Matching and Finding
  Reducing

Summary and References
Predicate

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

Menu stream

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

toList()
distinct

Numbers stream

1 2 1 3 3 2 4

\text{filter}(i \rightarrow i \% 2 == 0)

2 2 4

\text{distinct}()

2 4

Stream<Integer>
distinct
Outline

Concepts

Stream Operations

Filtering
Slicing
Mapping
Matching and Finding
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

\[
\text{Stream\langle Dish\rangle} \quad \text{filter}(d \rightarrow d.getCalories() > 300) \quad \text{Stream\langle Dish\rangle} \quad \text{limit}(2) \quad \text{Stream\langle Dish\rangle} \quad \text{collect}(toList()) \quad \text{List\langle Dish\rangle}
\]
skip

Menu stream

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

skip(1)

collect(toList())
skip(1)
**takeWhile**

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

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

Concepts

Stream Operations
   Filtering
   Slicing
   Mapping
   Matching and Finding
   Reducing

Summary and References
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>
map \{ \circ \rightarrow \diamond \}\}
**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

\[
\text{map}(s \to s.split(""))
\]

\[
\text{flatMap}(	ext{Arrays::stream})
\]

\[
\text{distinct}()
\]

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

List<String>
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
Outline

Concepts

Stream Operations
  Filtering
  Slicing
  Mapping
  Matching and Finding
  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(☐)
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: **findFirst**, **findAny**

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

- **findFirst** is better for parallelization than **findAny**
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

Stream Operations

Filtering
Slicing
Mapping
Matching and Finding
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<\text{Integer}>

\[ 0 \rightarrow \text{+} \rightarrow 4 \rightarrow \text{+} \rightarrow 9 \rightarrow \text{+} \rightarrow 12 \rightarrow \text{+} \rightarrow 21 \]
reduce \{ (\cdot, \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

- Intermediate operations are either *stateless* or *stateful* depending on whether they keep some information from previously seen elements.
- *sorted* and *distinct* have even an unbounded state: they keep track of part or all previously seen elements instead of a single value.
- *sorted* needs additionally to buffer all the elements of the stream to proceed, blocking the following data-processing operations.
- Terminal operations (such as *reduce*) are implicitly stateful because they keep an intermediate result (called an *accumulator*).
Can use the natural order or a custom comparison criterion.

```java
menu.stream()
    .sorted(comparing(Dish::getCalories))
    .toList();
```
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
<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>✓ (unbounded/blocking)</td>
<td>✓ (unbounded)</td>
<td></td>
<td>Comparator</td>
</tr>
<tr>
<td>distinct</td>
<td></td>
<td></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>✓ (List)</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
- Java tutorial
Demonstration

Compute the list of all numbers between 1 and 5 which square is below 20:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream()
    .filter(i -> i < 20)
    .peek(System.out::println)
    .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.