Multicore Programming
Java Streams (Advanced)

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

Bureau 414C

Master 1 computer science – Semester 8
Outline

Special Streams

Collectors

Summary and References
Primitive Stream Specializations

- IntStream, DoubleStream and LongStream.
- Avoid unboxing and boxing (more efficient data representation).
- Additional methods for efficiency and convenience: min, max, sum, average, etc.
Java: Boxing and Unboxing

- Collection of primitive type are forbidden in Java (e.g. List<int>).
- Boxing consists in encapsulating a primitive type into an object (e.g. Integer).
- Boxing and unboxing are performed automatically when needed.
- Working on boxed values is costly.
To and From a Primitive Stream

mapToInt:

```java
int calories = menu.stream()
    .mapToInt(Dish::getCalories)
    .sum();
```

boxed:

```java
Stream<Integer> stream = intStream.boxed();
```
Specialized Optionals

- OptionalInt
- OptionalDouble
- OptionalLong
Outline

Special Streams
- Numeric Streams
- Building Streams
- Infinite Streams

Collectors

Summary and References
From Values

Stream<String> stream =
    Stream.of("Modern ", "Java ", "In ", "Action");
From Arrays

```java
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```
Arrays.stream
From a Collection

```java
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
int sum = numbers.stream()
    .mapToInt(Integer::intValue)
    .sum();
```
From Numeric Ranges

```java
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
    .filter(n -> n % 2 == 0);
```

- range is exclusive
- rangeClosed is inclusive
long uniqueWords = 0;
try (Stream<String> lines =
    Files.lines(Paths.get("data.txt"),
    Charset.defaultCharset())) {
    uniqueWords = lines
        .flatMap(line -> Arrays.stream(line.split(" ")))
        .distinct()
        .count();
} catch (IOException e) {
From Nullable

From Java 9:

```java
String homeValue = System.getProperty("home");
Stream<String> homeValueStream =
    homeValue == null ? Stream.empty() : Stream.of(value);
```

```java
Stream<String> homeValueStream =
    Stream.ofNullable(System.getProperty("home"));
```

A stream with a null value differs from an empty stream.
Outline

Special Streams
- Numeric Streams
- Building Streams
- Infinite Streams

Collectors

Summary and References
iterate

Special Streams  Infinite Streams

```java
Stream.iterate(0, n -> n + 2)
    .limit(10)
    .forEach(System.out::println);

IntStream.iterate(0, n -> n < 100, n -> n + 4)
    .forEach(System.out::println);
```

- Needs a short-circuiting operation to terminate.
- No unbounded intermediate stateful operation (distinct and sorted).
generate

```java
Stream.generate(Math::random)
    .limit(5)
    .forEach(System.out::println);
```

- Requires a stateful (with a mutable state) supplier.
Outline

Special Streams

Collectors
  - Reducing and Summarizing
  - Other Single-Level Collectors
  - Grouping and Partitioning
  - Custom Collector

Summary and References
Collect the final result as the terminal operation.

Require a Collector object as its parameter.

A collector is designed to work on a stream.

No relation with Collection.

Several ways to define such collectors in class Collectors.

```java
stream.collect(toList());
```
Outline

Special Streams

Collectors

Reducing and Summarizing

Other Single-Level Collectors

Grouping and Partitioning

Custom Collector

Summary and References
maxBy and minBy

```java
Comparator<Dish> dishCaloriesComparator = Comparator.comparing(Dish::getCalories);
Optional<Dish> mostCalorieDish = menu.stream()
    .collect(maxBy(dishCaloriesComparator));
```

Equivalence:

```java
Optional<Dish> mostCalorieDish = menu.stream()
    .max(dishCaloriesComparator);
```
Summarizing: summingInt, counting, averagingInt and summarizingInt

```java
int totalCalories = menu.stream()
  .collect(summingInt(Dish::getCalories));
long count = menu.stream()
  .collect(counting());
double avgCalories = menu.stream()
  .collect(averagingInt(Dish::getCalories));
IntSummaryStatistics menuStatistics = menu.stream()
  .collect(summarizingInt(Dish::getCalories));
```
reducing

```java
int totalCalories = menu.stream()
    .collect(reducing(0, Dish::getCalories,
                      Integer::sum));
```

- Generalization of previous summarizing operations.
- First apply map function (e.g. `getCalories`), and then performs the reduction operation (e.g. `sum`).

Equivalence:

```java
int totalCalories = menu.stream()
    .mapToInt(Dish::getCalories)
    .reduce(0, Integer::sum);
```
Collectors Reducing and Summarizing

joining

String shortMenu = menu.stream()
   .map(Dish::getName)
   .collect(joining("", "");

Equivalence:

String shortMenu = menu.stream()
   .map(Dish::getName)
   .reduce(((a, b) -> a + "", " + b)
   .orElse("")
Outline

Special Streams

Collectors
  collect
  Reducing and Summarizing
  Other Single-Level Collectors
  Grouping and Partitioning
  Custom Collector

Summary and References
filtering

```java
List<Dish> caloricDishes = menu.stream()
    .collect(filtering(dish -> dish.getCalories() > 500,
        toList()));

Result:
[french fries, pizza, pork, beef]

Equivalence:

```java
List<Dish> caloricDishes = menu.stream()
    .filter(dish -> dish.getCalories() > 500)
    .collect(toList());
```
mapping

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

- flatMapping when generating a stream for each object instead of an object only (merge the streams).

Equivalence:

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

To apply a specific function on the result of the collection operation:

```java
Dish mostCaloric = menu.stream()
    .collect(collectingAndThen(
        maxBy(comparing(Dish::getCalories)),
        Optional::get));
```

Equivalence:

```java
Dish mostCaloric = menu.stream()
    .max(comparing(Dish::getCalories))
    .get();
```
Intermediate operation, transformation and collect

Example with reduce and reducing:

- reducing is like reduce but for a multi-level reduction.
- reduce must work on an immutable state (the accumulating and combining functions should have no side effect) because it can be parallelized.
- collect builds incrementally a mutable object (more efficient for strings because of StringBuilder).
Outline

Special Streams

Collectors
  collect
  Reducing and Summarizing
  Other Single-Level Collectors

Grouping and Partitioning
  Custom Collector

Summary and References
Example without Streams

```java
Map<Currency, List<Transaction>> transByCurr =
    new HashMap<>();
for (Transaction transaction : transactions) {
    Currency currency = transaction.getCurrency();
    List<Transaction> transForCurr =
        transByCurr.get(currency);
    if (transForCurr == null) {
        transForCurr = new ArrayList<>();
        transByCurr.put(currency, transForCurr);
    }
    transForCurr.add(transaction);
}
```
groupingBy

Map<Currency, List<Transaction>> transByCurr = transactions.stream()
  .collect(groupingBy(Transaction::getCurrency));
1. Traverse each transaction in the stream
2. Extract the transaction’s currency
3. Add the currency/transaction pair to the grouping map
Second Example

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

Next item

prawns

Apply

Classification function

Key

FISH

Classify item into list

Grouping map

FISH

salmon

MEAT

pork
beef
chicken

OTHER

pizza
rice
french fries
Grouping without Classification Function

```java
public enum CaloricLevel { DIET, NORMAL, FAT }
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream()
    .collect(groupingBy(dish -> {
        if (dish.getCalories() <= 400)
            return CaloricLevel.DIET;
        else if (dish.getCalories() <= 700)
            return CaloricLevel.NORMAL;
        else return CaloricLevel.FAT;
    }));
```
ReactiveX
filtering

Map<String, List<Dish>> caloricDishesByType =
menu.stream()
    .collect(groupingBy(Dish::getType,
        filtering(dish -> dish.getCalories() > 500,
             toList())));

Result:

{OTHER=[french fries, pizza],
 MEAT=[pork, beef],
 FISH=[]}

Note the empty group (unachievable with filter).
Collectors Grouping and Partitioning

```java
Map<String, List<String>> dishNamesByType = menu.stream()
    .collect(groupingBy(Dish::getType,
                        mapping(Dish::getName, toList())));
```
Multilevel Grouping

```java
Map<String, Map<CaloricLevel, List<Dish>>> res =
    menu.stream().collect(
        groupingBy(Dish::getType,
            groupingBy(dish -> {
                if (dish.getCalories() <= 400)
                    return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700)
                    return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
            } )
    );
```
First-level map

- FISH
  - NORMAL: salmon
  - DIET: prawns
- MEAT
  - NORMAL: pizza fries
  - DIET: fruit rice
- OTHER
  - NORMAL
  - DIET

Second-level maps

<table>
<thead>
<tr>
<th>type</th>
<th>FISH</th>
<th>MEAT</th>
<th>OTHER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIET</td>
<td>prawns</td>
<td>chicken</td>
<td>fruit rice</td>
</tr>
<tr>
<td>NORMAL</td>
<td>salmon</td>
<td>beef</td>
<td>pizza fries</td>
</tr>
<tr>
<td>FAT</td>
<td>pork</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
partitioningBy

```java
Map<Boolean, List<Dish>> partitionedMenu = 
    menu.stream().
    collect(partitioningBy(Dish::isVegetarian));
```

- Always generate lists for both `true` and `false`.
- Special map optimized for two values.
Outline

Special Streams

Collectors

collect
Reducing and Summarizing
Other Single-Level Collectors
Grouping and Partitioning
Custom Collector

Summary and References
Collector Interface

```java
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

- **T** stream element
- **A** accumulator element (temporary/intermediate result)
- **R** final returned element

Example in the following slides: `toList (A and R are both List<T>).`
collect \{ (\textcolor{red}{\textbullet}, [\textcolor{red}{\textbullet}], [\textcolor{green}{\textbullet}], [\textcolor{blue}{\textbullet}], [\textcolor{red}{\textbullet}, \textcolor{green}{\textbullet}, \textcolor{blue}{\textbullet}]) \rightarrow [\textcolor{red}{\textbullet}, \textcolor{green}{\textbullet}, \textcolor{blue}{\textbullet}] \}
Initial value for the accumulator:

```java
public Supplier<List<T>> supplier() {
    return ArrayList<T>::new;
}
```
Accumulator

Takes an element and merge it with the accumulator:

```java
public BiConsumer<List<T>, T> accumulator() {
    return List::add;
}
```
Finisher

Convert the accumulator to the final result:

```java
public Function<List<T>, List<T>> finisher() {
    return Function.identity();
}
```
A accumulator = collector.supplier().get();

collector.accumulator().accept(accumulator, next)

Are there more items in the stream?

Yes

T next = fetch next stream's item

No

R result = collector.finisher().apply(accumulator);

return result;

End
Combiner

Combine two accumulators:

```java
public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
        list1.addAll(list2);
        return list1;
    }
}
```

- Allows the parallelization: each core processes a part of the stream, which leads to a set of accumulators that are combined.
Split the stream in 2 subparts

Split the stream in 2 subparts

Split the stream in 2 subparts

Keep dividing the stream until each subpart is small enough

Process each substream in parallel using the former sequential algorithm

R r1 = collector.combiner().apply(acc1, acc2);

R r2 = collector.combiner().apply(acc3, acc4);

A accumulator = collector.combiner().apply(r1, r2);

Combine the results of the independent processing of each substream

R result = collector.finisher().apply(accumulator);

return result;
Characteristics

**UNORDERED** does not commit to preserve order

**CONCURRENT** can be parallelized (on all streams if UNORDERED, on unordered streams only otherwise)

**IDENTITY_FINISH** finisher is the identity function

```java
public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(EnumSet.of(
        IDENTITY_FINISH, CONCURRENT));
}
```
Collectors  Custom Collector

Alternative

```java
List<Dish> dishes = menuStream
    .collect(ArrayList<Dish>::new,
              List::add,
              List::addAll);
```
Demonstration

Compute the sum of all square numbers between 1 and \( n \).

```java
IntStream.rangeClosed(1, n)
    .filter(i -> {
        var x = Math.floor(Math.sqrt(i));
        return x * x == i;
    })
    .sum()
```
<table>
<thead>
<tr>
<th>Collector</th>
<th>argument</th>
<th>chaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>toList/toSet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minBy/maxBy</td>
<td>Comparator</td>
<td></td>
</tr>
<tr>
<td>summingInt</td>
<td>keyExtractor</td>
<td></td>
</tr>
<tr>
<td>averagingInt</td>
<td>keyExtractor</td>
<td></td>
</tr>
<tr>
<td>summarizingInt</td>
<td>keyExtractor</td>
<td></td>
</tr>
<tr>
<td>counting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reducing</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>joining</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>filtering</td>
<td>Predicate</td>
<td>✓</td>
</tr>
<tr>
<td>mapping</td>
<td>Function</td>
<td>✓</td>
</tr>
<tr>
<td>flatMapping</td>
<td>Function</td>
<td>✓</td>
</tr>
<tr>
<td>collectingAndThen</td>
<td>Function</td>
<td>✓</td>
</tr>
<tr>
<td>groupingBy</td>
<td>Function</td>
<td>✓</td>
</tr>
<tr>
<td>partitioningBy</td>
<td>Predicate</td>
<td>✓</td>
</tr>
</tbody>
</table>
Outline

Special Streams

Collectors

Summary and References
Official Documentation

- Documentation of interface Collector
- Documentation of class Collectors
Special Streams

- There are three primitive specializations of streams: `IntStream`, `DoubleStream`, and `LongStream`. Their operations are also specialized accordingly.
- Streams can be created not only from a collection but also from values, arrays, files, and specific methods such as `iterate` and `generate`.
- An infinite stream has an infinite number of elements (for example all possible strings). This is possible because the elements of a stream are only produced on demand. You can get a finite stream from an infinite stream using methods such as `limit`.
Collectors

- `collect` is a terminal operation that takes as argument various recipes (called collectors) for accumulating the elements of a stream into a summary result.
- Predefined collectors include reducing and summarizing stream elements into a single value, such as calculating the minimum, maximum, or average.
- Predefined collectors let you group elements of a stream with `groupingBy` and partition elements of a stream with `partitioningBy`.
- Collectors compose effectively to create multilevel groupings, partitions, and reductions.
- You can develop your own collectors by implementing the methods defined in the `Collector` interface.