Østfold University College

Java Parallel Stream

Layout

- Java Stream
- Aggregate Operations
- Mapping
- Reduction
- Mutable Reduction
- Parallel Stream
- Exercises

Java Stream (1)

- A sequence of elements supporting sequential and parallel ***aggregate operations*** (e.x, map-reduce transformation)
- Unlike a collection, it is NOT a data structure that stores elements.
- Stream carries values from a ***source*** through a ***pipeline***

Java Stream (2)

- **Aggregate operations**: Streams support SQL-like operations and common operations from functional programing languages, such as *filter, map, reduce, find, match, sorted*, etc.
- ***Source*** can be collections, arrays, I/O resources, etc.

Java Stream (3)

- A ***pipeline*** is a sequence of aggregate operations, contains:
  - A source.
  - Zero or more intermediate operations.
  - One terminal operation.

Aggregate Operations

```
for (Person p : roster) {
    System.out.println(p.getName());
}
```

```
roster
    .stream()
    .forEach(e -> System.out.println(e.getName());
```

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

```
roster
    .stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    // ...

    public int getAge() {
        // ...
    }

    public String getName() {
        // ...
    }
}
```

Mapping

- Transforming a stream by applying a given function to the elements of that stream:
  - void ***forEach***(Consumer<? super T> action)
  - Stream<T> ***filter***(Predicate<? super T> predicate)
  - Stream<T> ***sorted***(Comparator<? super T> comparator)
  - <R> Stream<R> ***map***(Function<? super T,? extends R> mapper)
  - IntStream ***mapToInt***(ToIntFunction<? super T> mapper)
  - …

Reduction (1)

- Combines the elements of the stream into a single summary result.
  - Long ***count***()
  - Optional<T> ***max***(Comparator<? super T> comparator)
  - int ***sum***()
  - OptionalDouble ***average***()
  - T ***reduce***(T identity, BinaryOperator<T> accumulator)
  - <U> U ***reduce***(U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)

Reduction (2)

T ***reduce***(T identity, BinaryOperator<T> accumulator)

```
T result = identity;
     for (T element : this stream)
         result = accumulator.apply(result, element)
     return result;
```

<U> U ***reduce***(U **identity**, BiFunction<U,? super T,U> **accumulator**, BinaryOperator<U> **combiner**)

```
     U result = identity;
     for (T element : this stream)
         result = accumulator.apply(result, element)
     return result;
```

Reduction (3)

```
Integer totalAge = roster
    .stream()
    .mapToInt(Person::getAge)
    .sum();
```

```
Integer totalAgeReduce = roster
   .stream()
   .map(Person::getAge)
   .reduce(
       0,
       (a, b) -> a + b);
```

```
double average = roster
           .stream()
           .filter(p -> p.getGender() == Person.Sex.MALE)
           .mapToInt(Person::getAge)
           .average()
           .getAsDouble();
```

```
sumOfWeights = widgets.stream()
                      .reduce(0,
                              (sum, b) -> sum + b.getWeight())
                              Integer::sum);
```

```
String concatenated = strings.reduce("", String::concat)
```

Mutable Reduction (1)

- A mutable reduction operation accumulates input elements into a mutable result container, such as a **Collection** or **StringBuilder.**
  - <R> R ***collect***(Supplier<R> *supplier*,
    BiConsumer<R,? super T> *accumulator*,
    BiConsumer<R,R> *combiner*)
  - <R,A> R ***collect***(Collector<? super T,A,R> collector)

Mutable Reduction (2)

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

```
R result = supplier.get();
for (T element : this stream)
    accumulator.accept(result, element);
return result;
```

Mutable Reduction (3)

```
ArrayList<String> strings = new ArrayList<>();
    for (T element : stream) {
        strings.add(element.toString());
    }
```

```
ArrayList<String> strings = stream.collect(() -> new ArrayList<>(),
                                           (c, e) -> c.add(e.toString()),
                                           (c1, c2) -> c1.addAll(c2));
```

```
List<String> strings = stream.map(Object::toString)
                             .collect(Collectors.toList());
```

Mutable Reduction (4)

```
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

```
Map<Person.Sex, List<String>> namesByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.mapping(
                    Person::getName,
                    Collectors.toList())));
```

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    // ...

    public int getAge() {
        // ...
    }

    public String getName() {
        // ...
    }
}
```

Mutable Reduction (5)

```
Map<Person.Sex, Integer> totalAgeByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.reducing(
                    0,
                    Person::getAge,
                    Integer::sum)));
```

```
Map<Person.Sex, Double> averageAgeByGender = roster
    .stream()
    .collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.averagingInt(Person::getAge)));
```

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    // ...

    public int getAge() {
        // ...
    }

    public String getName() {
        // ...
    }
}
```

Parallel Stream (1)

- When a stream executes in parallel, the Java runtime partitions the stream into multiple substreams. Aggregate operations iterate over and process these substreams in parallel and then combine the results.
- Invoke the operation ***Collection.parallelStream()***

Parallel Stream (2)

```
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

```
ConcurrentMap<Person.Sex, List<Person>> byGender =
    roster
        .parallelStream()
        .collect(
          Collectors.groupingByConcurrent(Person::getGender));
```

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    // ...

    public int getAge() {
        // ...
    }

    public String getName() {
        // ...
    }
}
```

Exercises (1)

- Write the following enhanced for statement as a pipeline with lambda expressions.

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

References

[1] Java Stream Tutorials

https://docs.oracle.com/javase/tutorial/collections/streams/index.html