Java Streams

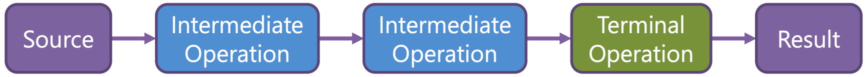
filter/map/reduce for Java

What are streams?

- Streams give us functional blocks to better process collections of data
  - We can chain these blocks together to process collections of data.
  - Streams aren't another data structure (is an Interface).
  - Streams can process an infinite list of data.
  - Streams use internal iteration meaning we don't have to code external iteration, the "how".
  - Streams support functional programming style inside the imperative Java programming language.

Structure of a Stream

- A stream consists of 3 types of things
  1. A source
  2. Zero or more intermediate operations
  3. A terminal operation



```
result = albums.stream()
               .filter(track -> track.getLength() > 300)
               .map(Track::getName)
               .collect(Collectors.toSet());
```

How they work

- The pipeline is only evaluated when the terminal operation is called.
  - The terminal operations pulls the data, the source doesn't push it.
  - Uses the stream characteristics to help identify optimisations.

  - This allows intermediate operations to be merged or parallelised (fork/join)
  - Avoiding multiple redundant passes on data
    - Short-circuit operations
    - Lazy evaluation

  - The stream takes care of the "how".
  - The stream is traversed only once
  - Can be infinite

Demo su VSCode

streams/Refactor

Intermediate Operations

- Most operations take a parameter that describes its behaviour, the "what".
- Typically using lambda expressions.
  - Must be non-interfering: stateless

The stream elements iteration is hidden in the stream library implementation

```
List<Person> persons = ...;
Stream<Person> tenPersonsOver18 = persons.stream()
                                         .filter(p -> p.getAge() > 18)
                                         .limit(10);
```

Intermediate Operations

.filter(Predicate) - excludes all elements that don't match a Predicate

.map(Function) - perform transformation of elements using a Function

.mapToInt, .mapToDouble, .mapToLong - Like map(), but producing streams of primitives

.flatMap(Function) - transform each element into a substream of zero or more elements

.peek(Consumer) - performs action on each element, useful for debugging (non terminal forEach)

.distinct - excludes all duplicate elements (based on equals(T))

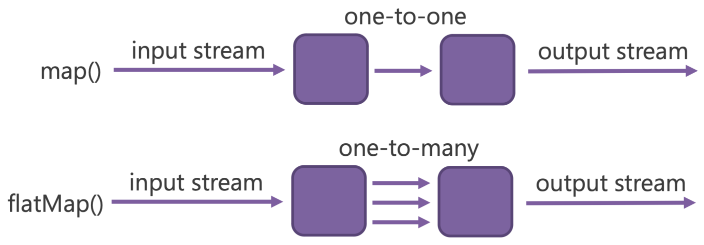
.sorted(Comparator) - return an ordered stream. With no arguments sorts by natural order.

.limit(long) - Return a stream that only contains the first n elements of the input stream

.skip(long) - Returns a stream that skips the first n elements of the input stream

Map() and FlatMap()

- Map values from a stream either as one-to-one or one- to-many, but still only produce one stream.



See example streams/operations/mapping on vscode

Terminal Operations

Terminates the pipeline of the operations on the stream.

- Only at this point is any processing performed.
- This allows for optimisation of the pipeline:
- Lazy evaluation
- Merged/fused operations
- Elimination of redundant operations
- Parallel execution
- Generates an explicit result of a side effect.

```
List<Person> persons = ..;
List<Student> students = persons.stream()
                                .filter(p -> p.getAge() > 18)
                                .map(Student::new)
                                .collect(Collectors.toList());
```

Collectors

```
List<String> list = people.stream()
 .map(Person::getName)
 .collect(Collectors.toList());

// Accumulate names into a TreeSet
Set<String> set = people.stream()
 .map(Person::getName)
 .collect(Collectors.toCollection(TreeSet::new));

// Convert elements to strings and concatenate them, separated by commas
String joined = things.stream()
 .map(Object::toString)
 .collect(Collectors.joining(", "));

// Compute sum of salaries of employee
int total = employees.stream()
 .collect(Collectors.summingInt(Employee::getSalary));
```

Collectors

```
// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
 .collect(Collectors.groupingBy(Employee::getDepartment));

// Compute sum of salaries by department
Map<Department, Integer> totalByDept = employees.stream()
 .collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.summingInt(Employee::getSalary)));

// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing = students.stream()
 .collect(Collectors.partitioningBy(s -> s.getGrade() >=
PASS_THRESHOLD));
```

Terminal Operations: collections and numeric results

Interface	Description
* collect(Collector)	Performs a mutable reduction on a stream
toArray()	Returns an array containing the elements of the stream
count()	Returns how many elements are in the stream
max(Comparator)	The maximum value element of the stream, returns an `Optional`
min(Comparator)	The minimum value element of the stream, returns an `Optional`
average()	Return the arithmetic mean of the (primitive only) stream, returns an `Optional`
sum()	Returns the sum of the stream elements
reduce(BynaryOperator)	applies an *associative* accumulation function to the stream and reduces it to an Optional

* There are a lot of built-in Collectors: toList(), toSet(), toMap(Function, Function), etc…

Terminal Operations

Interface	Description
findFirst(Predicate)	The (Optional) first element that matches predicate
findAny(Predicate)	Like `findFirst()`, but for a parallel stream
allMatch(Predicate)	True if All elements in the stream match predicate
anyMatch(Predicate)	True ifAny element in the stream matches predicate
noneMatch(Predicate)	True if No elements match the predicate
forEach(Consumer)	Performs an action on each element (ret. void)
forEachOrdered(Consumer)	Like above, but ensures order is respected when used for parallel stream

Folding to a Single Result: reduce

- The `collect` function isn't the only option.
- `reduce(BinaryOperator accumulator)` (T, T) -> T where T is the type of the stream objects
  Also called folding in Functional Programming.
- Performs a reduction on the stream using the `BinaryOperator`.
- The `accumulator` takes a partial result and the next element and returns a new partial result as an `Optional`.
- Two other versions
  - One that takes an initial value (a "Identity" value object, with respect to the applied folding).
  - One that takes an initial value and `BiFunction`. (T, U) -> U to generate a result of a different type U

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

Demo su VSCode

streams/RefactorExercise

Creating Streams

Collection Interface .stream()

- Provides a sequential stream of elements in the collection.
- parallelStream() Provides a parallel stream of elements in the collection.
- Uses the fork-join framework for implementation.
- Only Collection can provide a parallel stream directly.

Arrays Class .stream()

- Array is a collection of data, so logical to be able to create a stream.
- Provides a sequential stream.
- Overloaded methods for different types, int, double, long, Object

Stream Interface .builder() (builder Pattern…)

```
static <T> Stream.Builder<T> builder() Returns a Stream Builder.
 accept(T t) / add(T t)    Adds an element for later build.
Stream<T> build()  Builds the stream, no more adds…
```

Stream sources

- Streams from values (“of” method)
  - Stream<Integer> digits = Stream.of (3, 1, 4, 1, 5, 9);
  - Stream<String> words = Stream.of (“Mary", "had", "a", "little", "lamb");
- Empty streams
  - Stream<Student> ss = Stream.empty();
- Streams from functions
  - Random random = new Random();
  - Stream<Integer> randomNumbers = Stream.generate(random::nextInt);
- Streams from any array of objects
  - Integer[] digitArray = { 3, 1, 4, 1, 5, 9 };
  - Stream<Integer> digitStream = Stream.of(digitArray);
- Streams from collections
  - Set<Student> set = new LinkedHashSet<>();
  - Stream<Student> stream = set.stream();
- Streams from files (find, list, lines, walk).
  - Stream<String> stream = Files.lines(path);
- From other streams
  - Stream newStream = Stream.concat(stream, randomNumbers);

Primitive and infinite Streams

- IntStream, DoubleStream, LongStream
  primitive data specialisations of the Stream interface.

`range(int, int), rangeClosed(int, int)`
A stream from a start to an end value (exclusive or inclusive)

`generate(IntSupplier), iterate(int, IntUnaryOperator)`

- An infinite stream created by a given Supplier.
- `iterate` uses a seed to start the stream and an induction.

Stream.iterate

static <T> Stream<T> iterate (T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)

- Returns a sequential ordered `Stream`
- iteratively apply the given `next` function to an initial element
- The stream terminates as soon as the `hasNext` predicate returns false.
- produce the same sequence of elements as the for-loop:

```
for (T index=seed; hasNext.test(index); index = next.apply(index)) {...}
```

Example stream creations

`Stream.of(1, 2, 3)`	A stream containing the given elements. You can also pass an array.

`Collection<String> coll = ...;`
`coll.stream()`
	A stream containing the elements of a collection.

`Files.lines(path)`	A stream of the lines in the file with the given path. Use a try-with-resources statement to ensure that the underlying file is closed.
`Stream.generate(() -> 1)`	An infinite stream of ones
`Stream.iterate(0, n -> n + 1)`	An infinite stream of Integer values
`IntStream.range(0, 100)`	An IntStream of int values between 0 (inclusive) and 100 (exclusive)

`Random generator = new Random();`
`generator.ints(0, 100)`
	An infinite stream of random int values drawn from a random generator

`“Hello”.codePoints()`	An IntStream of code points of a string

Demo su VSCode

streams/StreamSources

Optional

- java.util.Optional<T>
  - a new class to help eliminate `NullPointerException`'s.
- Terminal operations like min(), max(), may not return a direct result, suppose the input stream is empty?
- It is a container for an object reference (`null`, or real object).
  - It has either 0 or 1 elements, but is never `null`.
- Doesn't stop developers from returning `null`, but an Optional tells you do maybe rather check.

Creating an Optional

<T> Optional<T> empty()	Returns an empty Optional.
<T> Optional<T> of (T value):	Returns an `Optional` containing the specified `value`. If the specified value is `null`, it throws a `NullPointerException`.
<T> Optional<T> ofNullable (T value)	Same as above, but if the specified value is null, it returns and empty Optional.

Getting Data from an Optional

<T> get()	Returns the value, or throws NoSuchElementException is value is empty.
boolean isPresent()	will return true If a value is present, false otherwise
<T> orElse(T defaultValue)	Returns the default value if value is empty.
<T> orElseGet(Supplier<? Extends T> defaultSupplier)	Same as above, but supplier gives the value.
<X extends Throwable> T orElseThrow(Supplier<? Extends T> exceptionSupplier)	If empty throw the exception from the supplier.
void ifPresent(consumer<T>)	execute a block of code with the value, if it is present.

Demo su VSCode

streams/Optional

Parallel & Sequential

```
java.util.stream
    List<Student> students = …;
    Stream stream = students.stream(); // sequential version

    // parallel version
    Stream parallelStream = students.parallelStream();
```

- Streams can switch between sequential and parallel, but all processing is either done sequential or parallel, last call wins.

```
List<Person> persons = ..;
List<Student> students = persons.stream()
         .parallel()
         .filter(p -> p.getAge() > 18)
         .sequential()
         .map(Student::new)
         .collect(Collectors.toCollection(ArrayList::new));
```

Demo su VSCode

streamLibro

Altri esempi da vedere a casa

- There are 5+21 exercises to complete at home for practice.
- There is a template source file that you should use to create the answers to the exercises called Test.java under exercises package in the test folder.
- To simplify things the lists and maps you need for the exercises have already been created.
- You just need to focus on the code to solve the exercise.
- The solutions are in the Test.java file in the solutions package, but try all the exercises first before taking a peak.

15 min. break