Programming in Java

6. Java Streams

Java Streams

- Functional programming support was implemented in Java to support stream processing
- A stream processes a collection of data objects
  - It takes input from a source of some kind without altering the source
  - All of the objects in a stream must be the same type
  - The data items move through a pipeline of transformations
  - A terminal operation ends the stream
- A stream in Java is not like a message queue
  - It can be helpful to think of it as a sequence of data objects
  - Conceptually, all the elements in a stream are processed at the same time at each step of the pipeline
  - A terminal operation either returns some collection or some single result or performs an operation (like saving to persistent storage) for each element in the stream
  - The actual pipeline is optimized before any processing takes place

Why Streams

- One of the main drivers for using streams is to handle the requirements of big data
  - Data is often streaming from multiple sources
  - We want to process the items in the stream with some sort of transformation
  - We can’t do this with OO or procedural programming
- If we use functional programming, then we have an elegant solution
  - We often need to write code that manages data streams for streaming data platforms
  - For example, Kafka has a streams capability that we program with Java streams
  - Data transformations on data streams in Spark is coded in Java, and other languages

Some Notation

- Java has a forEach(f) method that applies to collections
  - It applies the the function f to each element of the list in turn
  - Sort of a functional programming version of a for loop.
- In the examples that follow
  - We use a short hand for a lambda function
  - Assume we have a collection or stream with an iterable interface
  - Then forEach(System.out::println)
  - System.out::println is a method reference.
  - Shorthand for x -> System.out.println(x) — a lambda that prints its input.
  - Common notation in streams.

Initial Methods - Collections

- These take data source as input and return a stream
  - Any class implementing the Collection interface (e.g. List, Set) can produce a stream of its elements via the stream() method
  - The example to the right returns a sequential stream by default.
  - A List<String> can be turned into a Stream<String> for processing

```
List<String> items = Arrays.asList("apple", "banana", "cherry");

// Create a sequential stream from the list
Stream<String> stream = items.stream();

// Use stream operations (e.g., convert to uppercase and print each)
stream.map(String::toUpperCase)
      .forEach(System.out::println);
// Output: APPLE \n BANANA \n CHERRY  (each on a new line)
```

Initial Methods - Arrays

- There are two common ways to create a stream from an array
  - Using Arrays.stream(...): This method takes an array and returns a sequential stream of its elements.
  - Using Stream.of(...): This is a varargs method that can accept individual values or an array reference.

```
String[] fruits = { "apple", "banana", "cherry" };

// 1. Using Arrays.stream() on an array
Stream<String> fruitStream1 = Arrays.stream(fruits);
fruitStream1.forEach(System.out::println);
// Output: apple \n banana \n cherry

// 2. Using Stream.of() with explicit values
Stream<String> fruitStream2 = Stream.of("apple", "banana", "cherry");
fruitStream2.forEach(System.out::println);
// Output: apple \n banana \n cherry
```

Initial Methods - Files

- You can create a stream from a file, typically to process lines of text.
  - The Files.lines(Path) method (in java.nio.file.Files) returns a Stream<String> where each element is a line in the file.
  - The example on the right reads a text file line by line by turning it into a stream of Strings

```
Path path = Paths.get("data.txt");
try (Stream<String> lines = Files.lines(path)) {
    lines.filter(line -> line.contains("ERROR"))
         .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Infinite Streams

- Java streams can be infinite (unbounded).
  - The methods Stream.iterate() and Stream.generate() create streams that potentially never end
  - Typically used with a limit or a short-circuiting operation
  - Short circuit operation is one that will cause the stream to terminate, like when some test predicate fails

```
Stream<Double> randomNumbers = Stream.generate(Math::random);
randomNumbers.limit(3).forEach(System.out::println);
// Example output: 0.495... \n 0.365... \n 0.789...  (3 random doubles)
```

```
Stream<Integer> powersOfTwo = Stream.iterate(1, n -> n * 2);
powersOfTwo.limit(5).forEach(System.out::println);
// Output: 1 \n 2 \n 4 \n 8 \n 16
```


```
Stream<Integer> powersOfTwo = Stream.iterate(1, n -> n * 2);
powersOfTwo.limit(5).forEach(System.out::println);
// Output: 1 \n 2 \n 4 \n 8 \n 16
```

Parallel Streams

- A stream can be either sequential or parallel.
  - Parallel streams divide the workload across multiple threads, potentially speeding up processing on large data sets.
  - You can create a parallel stream in two ways:
    - *From a collection: use parallelStream() instead of stream() on a Collection. For example, list.parallelStream() produces a parallel stream*
    - *From an existing stream: call stream.parallel() to switch a sequential stream into parallel mode.*
- We will not cover this in this class

```
List<String> names = Arrays.asList("Alan", "Bob", "Cathy", "Doug");
names.parallelStream().forEach(System.out::println);
// Output (order may vary in parallel):
// Doug
// Cathy
// Alan
// Bob
```

Pipeline Methods

- These take a stream as input
- Return a stream as output
- Large library of methods - some of these are:
  - **map**(function) – applies the function to each element of the stream
  - **filter**(predicate) – keeps the elements that match the predicate, discards the others
  - **sorted**() – sorts the stream
- Other pipeline methods are in the java.util.streams library

Terminal Methods

- Terminal methods are methods that take an input from a stream and produce a final result
- Terminal methods mark the end of a stream – each stream can have only one terminal method
- Some terminal methods are:
  - **collect**(collection) – returns the result of the intermediate operations as a collection (e.g. list, array etc)
  - **forEach**(function) – applies the function to each element of the stream – does not produce an output stream
  - **reduce**(function) – uses function to collapse a stream into a single value

Java Streams Simple Example

```
public static void main(String[] args) {

Function<Integer,Integer> square = x -> x * x;
// Source Collection
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
List<Integer> squares =
        numbers.stream()        // creates the stream object
        .map(square)            // pipeline method applies the
                                // square function all the elements in the stream
        .collect(Collectors.toList()); // Convert the stream to a list
System.out.println(numbers);  // this as not been changed
System.out.println(squares);
}
```

Java Streams Simple Lambda Example

```
public static void main(String[] args) {

//Function<Integer,Integer> square = x -> x * x;
// Source Collection
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
List<Integer> squares =
        numbers.stream()        // creates the stream object
        .map(x -> x * x)        // pipeline Lambda method applies the
                                // square function all the elements in the stream
        .collect(Collectors.toList()); // Convert the stream to a list
System.out.println(numbers);  // this as not been changed
System.out.println(squares);
}
```

Lazy Invocation

- Streams are not executed until a terminal method is encountered
- The stream is represented as a directed acyclic graph (DAG)
- This DAG can be optimized at compile time with a number of standard rewrite rules
  - The stream can only be optimized when the whole DAG is complete
  - And that happens when a terminal operation is encountered


Lab 6-1
Basic Java Streams

Intermediate Methods

- Any stream method that returns a stream is an intermediate or pipeline method
- Some can be though of as working on individual elements
  - Specifically, they can operate on an element without reference to other elements
  - Examples - filter(), map()
  - These operations can be parallelized
- Others need to examine the relationships between stream elements
  - Examples - sorted(), distinct()
- Intermediate operations should not have side effects
  - We violated this in some of the demos to see what was happening in a stream
  - The terminal methods are where any side effects should occur

Intermediate Methods

- map(f)
  - Applies a monadic function f to each element in the stream and returns a transformed element
- filter(p)
  - Applies the predicate p to each element in the stream, if the result is false, the element is removed from the stream
- peek(f)
  - Used for debugging, it executes f on each element, when you want to see the elements as they flow past a certain point in a pipeline
- distinct()
  - removes duplicates based on the defined equality operator for the stream elements
- sorted()
  - sorts the elements based on the defined comparison operator for the stream elements

Intermediate Methods

- skip(n)
  - Omits the first n elements of a stream
- limit(n)
  - Truncates the stream after the n elements
- flatMap(stream)
  - removes levels of structure to flatten a stream
  - For example, a list of lists of integers has two levels of structure
    - *[[2, 3, 5], [7, 11, 13], [17, 19, 23]]*
  - Flattening the list removes the nested structure
    - *[2, 3, 5, 7, 11, 13, 17, 19, 23]*

Lab 6-2

Pipeline Methods

Terminal Operations

- The three basic types of terminal operations are
  - **Reducers** - returns a single value representing a computation on the stream - the stream is reduced to a single value
  - **Collectors** - returns some sort of collection
  - **Operators** - performs an operation on each element of the stream and returns void
- Terminal methods are terminal because they do not return a stream
  - They represent the end of the stream

Reducers

- There are a number of standard reducers
- count()
  - Returns the number of elements in the stream
- min(comparator), max(comparator)
  - These returns Optionals which are like futures to account for the cases where the value may or may not be returned
  - If isPresent() is true meaning that the value exists, it can be retrieved with the get() method
  - The comparator is the predicate used to determine how to order the elements
- anyMatch(p), allMatch(p), noneMatch(p)
  - Returns a Boolean if the predicate p is
    - *true for any one of the elements in the stream*
    - *true for all the elements in the stream*
    - *true for none of the elements in the stream*

Reducers

- reduce(accumulator, operator)
  - The accumlator is the last value computed (ie. from the previous element)
  - The operator is a function applied to combine the accumulator with the current element
  - Like summing an array in a loop
    - *The accumulator is the running total*
    - *The operator is adding the current element to the running total*

Collector

- The the Collectors class has a number of methods that return collections of various types
  - Eg. toList(), toMap(), toSet()
- There are other sorts of collectors that, for example:
  - Combine the stream into a single String
  - Do reduction type operations as well
  - In fact, reducers can be thought of as special cases of collectors

Lab 6-3

Terminal Methods


Java™