Stream API

- Stream is a sequence of objects that supports various sequential and parallel aggregate operations.
- These operations can be performed on the objects in a stream to produce results.
- It allows multiple intermediate operations chained together to aggregate the data, and the results are collected by applying a terminal operation on the data received.

What is the difference between java.util.streams and java.io streams?

- java.util streams meant for processing objects from the collection. i.e., it represents a stream of objects from the collection

- but java.io streams meant for processing binary and character data with respect to file. i.e it represents a stream of binary data or character data from the file.

- Hence java.io streams and java.util streams both are different.

Introduction

- Consider Java stream as a pipeline.
- That consists of 0 or more numbers of intermediate operations and terminal operation.
- Stream is not a collection or a data structure where we can store data.

- A **stream source** is a Stream instance that contains the initial data.
- **Intermediate operations** are used to perform actions on stream data and return another stream as output.
- **Terminal operations** produce the result of the stream after all the intermediate operations are applied.

- Basically, we pass input to the stream and apply zero or more intermediate operations to manipulate the data and finally, the result can be collected using a terminal operation.

- e.g.,

```
Stream.of(1, 2, 3, 4, 5)                // Stream source
   .filter(x -> x % 2 == 0)             // Intermediate operation
   .collect(Collectors.toList())        // Terminal operation
```

Different Ways to Create Streams in Java

1. Stream.empty()
2. Stream.builder()
3. Stream.of()
4. Arrays.stream()
5. Stream.concat()
6. Stream.generate()
7. Stream.iterate()

Different Operations on Streams

1. Intermediate Operations
    a. filter()
    b. map()
    c. sorted()
    d. distinct()
    e. peek()
    f. limit()
    g. skip()

2. Terminal Operations
    a. forEach()
    b. forEachOrdered()
    c. collect()
    d. count()
    e. reduce()
    f. toArray()
    g. min()
    h. max()
    i. findFirst()
    j. findAny()
    k. noneMatch()
    l. allMatch()
    m. anyMatch()

3. Stream Short-circuit Operations
    a. Intermediate short-circuit operations
        i. limit()

    b. Terminal short-circuit operations
        i. findFirst()
        ii. findAny()

iii. allMatch()
iv. anyMatch()
v. noneMatch()

4. Parallel Stream
    a. parallel()
    b. parallelStream()

Different Ways to Create Streams in Java

1. **Stream.empty()**
    - Stream.empty() creates an empty stream without any values.
    - This avoids null pointer exceptions when calling methods with stream parameters.
    - We create empty streams when we want to add objects to the stream in the program.

    - Syntax

      Stream<T> stream = Stream.empty();

2. **Stream.builder()**
    - Stream.builder() returns a builder (a design pattern that allows us to construct an object step-by-step) that can be used to add objects to the stream.
    - The objects are added to the builder using the add() method.
    - Calling the build() method on the builder creates an instance of the Stream.
    - This implementation of Stream creation is based on the famous Builder Design Pattern.
    - Syntax

      Stream.Builder<T> builder = Stream.builder();

      Stream<T> stream = builder.build();

3. **Stream.of()**
    - Stream.of() method creates a stream with the specified values.
    - This method accepts both single and multiple values.
    - it does the work of declaring and initializing a stream.
    - Syntax

// Single value
Stream<T> stream = Stream.of(T value);

// Multiple values
Stream<T> stream = Stream.of(T... values);

4. **Arrays.stream()**
   - Arrays.stream() method creates a stream instance from an existing array.
   - The resulting stream instance will have all the elements of the array.
   - Syntax

     Stream<T> stream = Arrays.stream(T[] array);

5. **Stream.concat()**
   - Stream.concat() methods combine two existing streams to produce a new stream.
   - The resultant stream will have all the elements of the first stream followed by all the elements of the second stream.
   - Syntax

Stream<T> stream = Stream.concat(Stream<T> stream1, Stream<T> stream2);

6. **Stream.generate()**
   - Stream.generate() **returns** an **infinite** sequential **unordered** stream where the **values are generated by** the provided **Supplier**.

   - **Infinite** The values in the stream are infinite (i.e.) unlimited.
   - **Unordered** Repeated execution of the stream might produce different results.
   - **Supplier** A functional interface in Java represents an operation that takes no argument and returns a result.
   - Stream.generate() is useful to create infinite values like random integers, UUIDs (Universally Unique Identifiers), constants, etc.
   - Since the resultant stream is infinite, it can be limited using the limit() method to make it run infinite time.
   - Syntax

     Stream<T> stream = Stream.generate(Supplier<T> supplier);

7. **Stream.iterate()**
   - **Stream.iterate()** returns an **infinite** sequential **ordered** stream stream where the **values are generated** by the provided **UnaryOperator**.

- **Infinite** The values in the stream are infinite (i.e. unlimited).
- **Ordered** Repeated execution of the stream produces the same results.
- **UnaryOperator** A functional interface in Java that takes one argument and returns a result that is of the same type as its argument.
- **Stream.iterate()** methods **accept two arguments**, an initial value called the seed value and a unary operator.
- The **first element of the resulting stream will be the seed value.**
- The **following elements are created** by **applying the unary operator on the previous element.**
- Syntax

Stream<T> stream = Stream.iterate(T seed, UnaryOperator<T> unaryOperator);

Different Operations on Streams

- Stream provides various operations that can be chained together to produce results.
- Stream operations can be classified into two types.
    - Intermediate Operations
    - Terminal Operations

1. Intermediate Operations

- Intermediate operations return a stream as the output.
- intermediate operations are not executed until a terminal operation is invoked on the stream.
- This is called lazy evaluation.

a. filter()

- The **filter()** method returns a stream with the stream's elements that match the given predicate.
- **Predicate is a functional interface** in Java that accepts a single input and can return a boolean value.
- Syntax:

```
list.stream()
.filter(value -> value % 2 == 0)
.collect(Collectors.toList());
```

b. map()

- The **map()** method returns a stream with the resultant elements after applying the given function on the stream elements.
- Syntax

```
list.stream()
.map(value -> value * 10)
.collect(Collectors.toList());
```

c. sorted()

- The sorted() method returns a stream with the elements of the stream sorted according to natural order or the provided Comparator.
- Syntax

```
list.stream().sorted()
.forEach(System.out::println);
```

d. distinct()

- This distinct() method returns a stream consisting of distinct elements of the stream (i.e.) it removes duplicate elements.
- Syntax

```
list.stream()
.distinct()
.collect(Collectors.toList());
```

e. peek()

- The peek() method returns a stream consisting of the elements of the stream after performing the provided action on each element.
- This is useful when we want to print values after each intermediate operation.
- Syntax

```
list.stream()
.filter(value -> value % 2 == 0)
.peek(value -> System.out.println("Filtered " + value))
.map(value -> value * 10)
.collect(Collectors.toList());
```

f. limit()

- The limit() method returns a stream with the stream elements limited to the provided size.
- Syntax

```
list.stream()
.limit(3).collect(Collectors.toList());
```

g. skip()

- This skip() method returns a stream consisting of the stream after discarding the provided first n elements.
- Syntax

```
list.stream()
.skip(2)
.collect(Collectors.toList());
```

- The first two elements are skipped using the skip(2) method.

2. Terminal Operations

- Terminal operations produce the results of the stream after all the intermediate operations are applied.
- we can no longer use the stream once the terminal operation is performed.

a. forEach()

- The forEach() method **iterates and performs the specified action** for each stream element.
- **For parallel streams**, it **doesn't guarantee to maintain the order** of the stream.
- Syntax

```
list.stream().forEach(System.out::println);
```

b. forEachOrdered()

- The forEachOrdered() method iterates and performs the specified action for each stream element.
- This is **similar to the forEach()** method, and the **only difference is that it maintains the order when the stream is parallel.**
- Syntax

```
Stream.of("A","B","C").parallel()
.forEachOrdered(x -> System.out.println( x));
```

c. collect()

- The collect() method performs a mutable reduction operation on the elements of the stream using a Collector.

Mutable Reduction

A mutable reduction is an operation in which the reduced value is a mutable (modified) result container, like an ArrayList.

Collector

A Collector is a class in Java that implements various reduction operations such as accumulating elements into collections, summarizing elements, etc.

- Syntax

```
Stream.of(1, 2, 3, 4, 5)
.filter(x -> x % 2 == 0)
.collect(Collectors.toList());
```

d. count()

- The count() method returns the total number of elements in the stream.
- Syntax

```
list.stream().count();
```

e. reduce()

- The reduce() method performs a reduction on the elements of the stream and returns the value.
- Syntax

```
list.stream().reduce(0, (value, sum) -> sum += value);
```

- The reduce() method is called with **two arguments**, an initial value (0) and the accumulator method (value, sum) -> sum += value).
- Each stream element will be added to the previous result to produce the sum.

f. toArray()

- The toArray() method returns an array that contains the elements of the stream.
- Syntax

```
Arrays.toString(stream.toArray())
```

g. min()

- The min() methods return an Optional that contains the minimum elements of the stream, according to the provided comparator.
- Syntax

  stream.**min((a, b) -> Integer.compare(a, b))**.get();

h. max()

- The max() methods return an Optional that contains the maximum elements of the stream.
- Syntax

  stream**.max((a, b) -> Integer.compare(a, b))**.get();

i. findFirst()

- The findFirst() method returns an Optional that contains the first element of the stream or an empty Optional if the stream is empty.
- Syntax

  list.stream()**.findFirst()**;

j. findAny()

- The findAny() method returns an Optional containing some element of the stream or an empty Optional if the stream is empty.
- Syntax

  list.stream()**.findAny()**;

k. noneMatch()

- When no stream elements match the specified predicate, the noneMatch() method returns true, otherwise false.
- If the stream is empty, it returns true.
- Syntax

  list.stream()**.noneMatch(value -> value == 2);**

l. **allMatch()**

- When all the stream elements meet the specified predicate, the allMatch() method returns true, otherwise false.
- If the stream is empty, it returns true.
- Syntax

  list.stream()**.allMatch(value -> value == 2)**;

m. **anyMatch()**

- When any stream element matches the specified predicate, the anyMatch() method returns true, otherwise false.
- If the stream is empty, it returns false.
- Syntax

  list.stream().**anyMatch(value -> value == 2)**;

3. **Stream Short-circuit Operations**

- Stream short-circuit operations can be better understood with Java's logical && and || operators.
- expression1 && expression2 doesn't evaluate expression2 if expression1 is false because false && anything is always false.
- expression1 || expression2 doesn't evaluate expression2 if expression1 is true because true || anything is always true.

- Stream short-circuit operations are those that can terminate without processing all the elements in a stream.

- Short-circuit operations can be classified into two types.
  1. Intermediate short-circuit operations
  2. Terminal short-circuit operations

1. Intermediate Short-Circuit Operations

- Intermediate short-circuit operations produce a finite stream from an infinite stream.
- The **intermediate short-circuit operation** is **limit()**.

2. **Terminal Short-Circuit Operations**

    - Terminal short-circuit operations are those that can produce the result before processing all the elements of the stream.
    - The terminal short-circuit operations in stream are findFirst(), findAny(), allMatch(), anyMatch(), and noneMatch().
    -

4. Parallel Stream

- By default, all stream operations are sequential
- Need to mention explicitly if parallel stream is required.

- Parallel streams are created in Java in two ways

- Calling the **parallel()** method on a sequential stream.
- Calling **parallelStream()** method on a collection.

1. Parallel()
    - parallel() method is called on the existing sequential stream to make it parallel.
    - Syntax

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

stream.parallel().forEach(System.out::println);
```

    - The stream instance is converted to parallel stream by calling stream.parallel().
    - Since the forEach() method is called on a parallel stream, the output order will not be the same as the input order because the elements are processed parallel.

2. ParallelStream()
    - parallelStream() is called on Java collections like List, Set, etc to make it a parallel stream.
    - Syntax

```
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

list.parallelStream().forEach(System.out::println);
```

5. **Lazy Evaluation**
    - Lazy evaluation (aka) call-by-need evaluation is an evaluation strategy that delays evaluating an expression until its value is needed.
    - All intermediate operations are performed on the stream only when a terminal operation is invoked on it.
    - Lazy evaluation is one of the critical characteristics of Java streams that allows significant optimizations.