**Java Streams API**

The Java Stream API, introduced in Java 8, revolutionizes the way we process data collections by providing a high-level, functional-style approach. It allows for processing sequences of elements, such as arrays and collections, in a declarative manner. The API supports operations like filtering, mapping, and reducing, which can be chained together to form complex data processing pipelines.

**Table of Contents**

1. **Introduction to Streams**
2. **Creating Streams**
3. **Intermediate Operations**
   * Filtering
   * Mapping
   * Sorting
   * Distinct
   * Limit and Skip
4. **Terminal Operations**
   * forEach
   * collect
   * reduce
   * toArray
   * count, min, max, and sum
   * anyMatch, allMatch, noneMatch, findFirst, findAny
5. **Parallel Streams**
6. **Combining Functional Programming with Streams**
7. **Advanced Examples**

**1. Introduction to Streams**

A stream is a sequence of elements that supports various methods which can be pipelined to produce the desired result. Streams are not data structures; they don't store elements. Instead, they convey elements from a source (e.g., collections, arrays, I/O channels) through a pipeline of computational operations.

**2. Creating Streams**

Streams can be created from various sources like collections, arrays, and files.

**Example: Creating Streams**

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class StreamCreationExample {

public static void main(String[] args) {

// Stream from a Collection

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

Stream<String> streamFromCollection = names.stream();

// Stream from an Array

String[] nameArray = {"Alice", "Bob", "Charlie", "David"};

Stream<String> streamFromArray = Arrays.stream(nameArray);

// Stream from individual values

Stream<String> streamFromValues = Stream.of("Alice", "Bob", "Charlie", "David");

// Printing the streams

streamFromCollection.forEach(System.out::println);

}

}

**3. Intermediate Operations**

Intermediate operations transform a stream into another stream. They are lazy, meaning they are not executed until a terminal operation is invoked.

**3.1 Filtering**

Filters elements based on a predicate (a condition).

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class FilteringExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Anna");

// Filter names that start with "A"

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("A"))

.collect(Collectors.toList());

System.out.println(filteredNames); // Output: [Alice, Anna]

}

}

**3.2 Mapping**

Transforms each element of the stream.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class MappingExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Map names to their lengths

List<Integer> nameLengths = names.stream()

.map(String::length)

.collect(Collectors.toList());

System.out.println(nameLengths); // Output: [5, 3, 7]

}

}

**3.3 Sorting**

Sorts the elements of the stream.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class SortingExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Charlie", "Alice", "David", "Bob");

// Sort names in natural order

List<String> sortedNames = names.stream()

.sorted()

.collect(Collectors.toList());

System.out.println(sortedNames); // Output: [Alice, Bob, Charlie, David]

}

}

**3.4 Distinct**

Removes duplicate elements from the stream.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class DistinctExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Alice", "David", "Bob");

// Remove duplicates

List<String> distinctNames = names.stream()

.distinct()

.collect(Collectors.toList());

System.out.println(distinctNames); // Output: [Alice, Bob, David]

}

}

**3.5 Limit and Skip**

limit(n) reduces the stream to the first n elements, and skip(n) skips the first n elements.

import java.util.Arrays;

import java.util.List;

public class LimitSkipExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva");

// Limit to the first 3 names

names.stream().limit(3).forEach(System.out::println); // Output: Alice, Bob, Charlie

// Skip the first 2 names

names.stream().skip(2).forEach(System.out::println); // Output: Charlie, David, Eva

}

}

**4. Terminal Operations**

Terminal operations produce a result or a side-effect, such as printing the elements, collecting them into a collection, or summing them.

**4.1 forEach**

Performs an action for each element of the stream.

import java.util.Arrays;

import java.util.List;

public class ForEachExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Print each name

names.stream().forEach(System.out::println); // Output: Alice, Bob, Charlie

}

}

**4.2 collect**

Collects the elements of the stream into a collection.

import java.util.Arrays;

import java.util.List;

import java.util.Set;

import java.util.stream.Collectors;

public class CollectExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Alice");

// Collect into a Set to remove duplicates

Set<String> nameSet = names.stream().collect(Collectors.toSet());

System.out.println(nameSet); // Output: [Alice, Bob, Charlie]

}

}

**4.3 reduce**

Reduces the elements of the stream to a single value.

import java.util.Arrays;

import java.util.List;

public class ReduceExample {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Sum of all numbers

int sum = numbers.stream().reduce(0, Integer::sum);

System.out.println("Sum: " + sum); // Output: Sum: 15

}

}

**4.4 toArray**

Converts the stream into an array.

import java.util.Arrays;

import java.util.List;

public class ToArrayExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Convert to an array

String[] namesArray = names.stream().toArray(String[]::new);

System.out.println(Arrays.toString(namesArray)); // Output: [Alice, Bob, Charlie]

}

}

**4.5 count, min, max, sum**

These operations return the count, minimum, maximum, and sum of the elements.

import java.util.Arrays;

import java.util.List;

public class AggregationExample {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Count the number of elements

long count = numbers.stream().count();

System.out.println("Count: " + count); // Output: Count: 5

// Find the minimum

int min = numbers.stream().min(Integer::compare).orElseThrow();

System.out.println("Min: " + min); // Output: Min: 1

// Find the maximum

int max = numbers.stream().max(Integer::compare).orElseThrow();

System.out.println("Max: " + max); // Output: Max: 5

}

}

**4.6 anyMatch, allMatch, noneMatch, findFirst, findAny**

These operations test whether any, all, or none of the stream elements match a given predicate, or find the first or any element that matches.

import java.util.Arrays;

import java.util.List;

public class MatchExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

// Check if any name starts with "A"

boolean anyMatch = names.stream().anyMatch(name -> name.startsWith("A"));

System.out.println("Any match: " + anyMatch); // Output: Any match: true

// Check if all names start with "A"

boolean allMatch = names.stream().allMatch(name -> name.startsWith("A"));

System.out.println("All match: " + allMatch); // Output: All match: false

// Check if no name starts with "Z"

boolean noneMatch = names.stream().noneMatch(name -> name.startsWith("Z"));

System.out.println("None match: " + noneMatch); // Output: None match: true

// Find the first name that starts with "C"

String firstMatch = names.stream().filter(name -> name.startsWith("C")).findFirst().orElse("Not Found");

System.out.println("First match: " + firstMatch); // Output: First match: Charlie

}

}

**5. Parallel Streams**

Parallel streams leverage multi-core processors to process data concurrently. You can create a parallel stream by calling parallelStream() on a collection or parallel() on a stream.

**Example: Parallel Stream**

import java.util.Arrays;

import java.util.List;

public class ParallelStreamExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

// Process names in parallel

names.parallelStream().forEach(name -> {

System.out.println(Thread.currentThread().getName() + " processing " + name);

});

}

}

**6. Combining Functional Programming with Streams**

Functional programming and streams go hand in hand. You can pass lambda expressions or method references as arguments to stream operations, making your code concise and readable.

**Example: Combining Functions with Streams**

import java.util.Arrays;

import java.util.List;

import java.util.function.Function;

public class FunctionalStreamExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

// Function to transform names to uppercase

Function<String, String> toUpperCase = String::toUpperCase;

// Chain the function in a stream operation

names.stream()

.map(toUpperCase)

.forEach(System.out::println); // Output: ALICE, BOB, CHARLIE, DAVID

}

}

**7. Advanced Examples**

Let's look at a few advanced use cases that combine various stream operations.

**7.1 Grouping and Partitioning**

You can group or partition data using Collectors.groupingBy() and Collectors.partitioningBy().

import java.util.Arrays;

import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;

public class GroupingPartitioningExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva", "Anna");

// Group names by their length

Map<Integer, List<String>> groupedByLength = names.stream()

.collect(Collectors.groupingBy(String::length));

System.out.println(groupedByLength);

// Output: {3=[Bob, Eva], 4=[Anna], 5=[Alice, David], 7=[Charlie]}

// Partition names by whether they start with "A"

Map<Boolean, List<String>> partitionedByA = names.stream()

.collect(Collectors.partitioningBy(name -> name.startsWith("A")));

System.out.println(partitionedByA);

// Output: {false=[Bob, Charlie, David, Eva], true=[Alice, Anna]}

}

}

**7.2 FlatMap for Flattening**

flatMap() is used to flatten nested structures.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class FlatMapExample {

public static void main(String[] args) {

List<List<String>> namesNested = Arrays.asList(

Arrays.asList("Alice", "Bob"),

Arrays.asList("Charlie", "David"),

Arrays.asList("Eva", "Frank")

);

// Flatten the nested lists

List<String> namesFlat = namesNested.stream()

.flatMap(List::stream)

.collect(Collectors.toList());

System.out.println(namesFlat);

// Output: [Alice, Bob, Charlie, David, Eva, Frank]

}

}

**Conclusion**

The Java Stream API is a powerful tool for data processing that leverages functional programming concepts. By understanding and mastering streams, you can write more concise, readable, and efficient code. The examples provided here should give you a solid foundation to explore more complex scenarios and fully harness the power of streams in Java.