Stream API

# Person Class

public class Person implements Comparable<Person> {

private String firstName;

private String lastName;

private String email;

private Gender gender;

private int age;

public Person(String firstName, String lastName, String email, Gender gender, int age) {

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

this.gender = gender;

this.age = age;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public Gender getGender() {

return gender;

}

public void setGender(Gender gender) {

this.gender = gender;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

@Override

public int compareTo(Person other) {

return Integer.compare(this.age, other.age);

}

@Override

public String toString() {

return firstName + " " + lastName + " (" + age + " years)";

}

}

# Gender Enum

public enum Gender {

MALE, FEMALE

}

# Creating a List of Person Objects

Here’s a sample list with diverse entries in names, ages, genders, and emails:

import .util.ArrayList;

import .util.List;

public class PersonList {

public static List<Person> createPersonList() {

List<Person> listPersons = new ArrayList<>();

listPersons.add(new Person("Alice", "Brown", "alice.brown@gmail.com", Gender.FEMALE, 26));

listPersons.add(new Person("Bob", "Young", "bob.young@yahoo.com", Gender.MALE, 32));

listPersons.add(new Person("Carol", "Hill", "carol.hill@gmail.com", Gender.FEMALE, 23));

listPersons.add(new Person("David", "Green", "david.green@hotmail.com", Gender.MALE, 39));

listPersons.add(new Person("Eric", "Young", "eric.young@gmail.com", Gender.MALE, 26));

listPersons.add(new Person("Frank", "Thompson", "frank.t@outlook.com", Gender.MALE, 33));

listPersons.add(new Person("Gibb", "Brown", "gibb.brown@gmail.com", Gender.MALE, 27));

listPersons.add(new Person("Henry", "Baker", "henry.baker@gmail.com", Gender.MALE, 30));

listPersons.add(new Person("Isabell", "Hill", "isabell.hill@gmail.com", Gender.FEMALE, 22));

listPersons.add(new Person("Jane", "Smith", "jane.smith@gmail.com", Gender.FEMALE, 24));

listPersons.add(new Person("Lisa", "Johnson", "lisa.j@hotmail.com", Gender.FEMALE, 45));

listPersons.add(new Person("Mark", "Evans", "mark.e@company.com", Gender.MALE, 55));

listPersons.add(new Person("Nina", "Gonzalez", "nina.gonzalez@gmail.com", Gender.FEMALE, 31));

listPersons.add(new Person("Oscar", "Black", "oscar.black@gmail.com", Gender.MALE, 60));

listPersons.add(new Person("Paul", "White", "paul.white@company.com", Gender.MALE, 48));

return listPersons;

}

}

# ****1. Overview of Stream API****

The Stream API, introduced in 8, allows us to perform functional-style operations on collections. Streams are sequences of data processed in a pipeline, and operations are performed lazily to optimize performance.

### ****2. Stream Creation and Structure****

* **Source**: Streams are created from collections, arrays, files, or I/O sources.
* **Intermediate Operations**: Operations like filter, map, and sorted are applied and return a new stream, allowing for chaining.
* **Terminal Operations**: Operations like collect, forEach, and reduce complete the pipeline, executing the entire chain.

# ****Intermediate Operations****

#### **Filter**

Filter out people based on gender and salary.

// Filter females with a salary above $50,000

List<Person> highEarners = people.stream()

.filter(p -> p.getGender() == Gender.FEMALE && p.getSalary() > 50000)

.collect(Collectors.toList());

System.out.println("Female high earners: " + highEarners);

**Explanation**: This filters the list to include only females with a salary above $50,000. The filter operation is useful for extracting elements that meet specific criteria.

#### **Map**

Extract and manipulate data, e.g., calculate monthly salary.

// Map to monthly salary by dividing annual salary by 12

List<Double> monthlySalaries = people.stream()

.map(p -> p.getSalary() / 12)

.collect(Collectors.toList());

System.out.println("Monthly Salaries: " + monthlySalaries);

**Explanation**: The map function transforms each Person object into their monthly salary, returning a Stream of Double values.

#### **FlatMap**

Example with nested lists: creating a stream of last names from nested structures.

List<List<Person>> nestedPeople = Arrays.asList(

Arrays.asList(people.get(0), people.get(1)),

Arrays.asList(people.get(2), people.get(3))

);

List<String> lastNames = nestedPeople.stream()

.flatMap(Collection::stream)

.map(Person::getLastName)

.collect(Collectors.toList());

System.out.println("Last Names: " + lastNames);

**Explanation**: flatMap is used here to flatten a stream of lists into a single stream of persons. It’s useful for processing collections within collections.

#### **Distinct**

Filter out duplicate salaries.

List<Double> distinctSalaries = people.stream()

.map(Person::getSalary)

.distinct()

.collect(Collectors.toList());

System.out.println("Distinct Salaries: " + distinctSalaries);

**Explanation**: distinct() removes duplicates from the stream, here based on each person’s salary.

#### **Sorted**

Sort persons by descending salary.

List<Person> sortedBySalary = people.stream()

.sorted(Comparator.comparingDouble(Person::getSalary).reversed())

.collect(Collectors.toList());

System.out.println("Sorted by Salary (Desc): " + sortedBySalary);

**Explanation**: sorted() sorts by a given comparator, allowing for custom sorting logic like descending order by salary.

#### **Limit**

Limit results to top three highest salaries.

List<Person> topEarners = people.stream()

.sorted(Comparator.comparingDouble(Person::getSalary).reversed())

.limit(3)

.collect(Collectors.toList());

System.out.println("Top Earners: " + topEarners);

**Explanation**: Using sorted() and limit(), we retrieve only the top 3 highest-earning people.

#### **Skip**

Skip the first two elements and print the rest.

List<Person> skippedPeople = people.stream()

.skip(2)

.collect(Collectors.toList());

System.out.println("Skipped first two people: " + skippedPeople);

**Explanation**: skip(n) is used here to ignore the first two elements in the stream, useful for pagination or batched processing.

### ****Terminal Operations****

#### **forEach**

Prints out each person with a custom message.

people.stream()

.filter(p -> p.getAge() > 25)

.forEach(p -> System.out.println(p.getFirstName() + " is over 25 years old."));

**Explanation**: forEach() is a terminal operation that performs an action on each element. In this example, it outputs persons older than 25 with a message.

#### **Collect**

Grouping data into collections, such as a Map.

// Collect females into a set

Set<Person> femaleSet = people.stream()

.filter(p -> p.getGender() == Gender.FEMALE)

.collect(Collectors.toSet());

System.out.println("Females as Set: " + femaleSet);

**Explanation**: collect accumulates stream elements into a collection like Set, List, or even Map.

#### **Reduce**

Calculate the total salary of all persons.

double totalSalary = people.stream()

.map(Person::getSalary)

.reduce(0.0, Double::sum);

System.out.println("Total Salary: $" + totalSalary);

**Explanation**: reduce reduces the stream to a single value. In this case, it adds up all salaries to calculate the total.

#### **findFirst / findAny**

Find any female over 30.

Optional<Person> anyFemaleOver30 = people.stream()

.filter(p -> p.getGender() == Gender.FEMALE && p.getAge() > 30)

.findAny();

anyFemaleOver30.ifPresent(System.out::println);

**Explanation**: findAny returns any element that matches the filter, while findFirst specifically returns the first match. It’s useful when you only need one result.

#### **Match Operations (allMatch, anyMatch, noneMatch)**

Check if everyone earns more than $50,000.

boolean allEarnAbove50k = people.stream()

.allMatch(p -> p.getSalary() > 50000);

System.out.println("Does everyone earn above $50,000? " + allEarnAbove50k);

**Explanation**: allMatch, anyMatch, and noneMatch help in testing if elements satisfy specific conditions, useful for validation.

#### **Min / Max**

Find the person with the highest salary.

Optional<Person> highestEarner = people.stream()

.max(Comparator.comparingDouble(Person::getSalary));

highestEarner.ifPresent(p -> System.out.println("Highest Earner: " + p));

**Explanation**: min and max return the minimum or maximum element based on a comparator, helpful in finding extremes.

#### **Count**

Count the number of people above 30 years old.

long countAbove30 = people.stream()

.filter(p -> p.getAge() > 30)

.count();

System.out.println("Number of people above 30: " + countAbove30);

**Explanation**: count() counts elements in the stream that match a condition, here people older than 30.

### ****Advanced Collect Examples****

#### **Grouping and Partitioning**

Group people by gender and then by age range.

Map<Gender, Map<String, List<Person>>> groupedByGenderAndAge = people.stream()

.collect(Collectors.groupingBy(

Person::getGender,

Collectors.groupingBy(p -> {

if (p.getAge() < 30) return "Young";

else return "Senior";

})

));

System.out.println("Grouped by Gender and Age Range: " + groupedByGenderAndAge);

**Explanation**: groupingBy allows multi-level grouping, which helps when you want data organized by multiple conditions.

#### **Summarizing**

Summarize salaries across the group, with average, min, max, etc.

DoubleSummaryStatistics salaryStats = people.stream()

.collect(Collectors.summarizingDouble(Person::getSalary));

System.out.println("Salary Statistics: " + salaryStats);

**Explanation**: summarizingDouble provides a summary of statistics, including average, min, max, and count.

#### **Joining**

Concatenate all email addresses separated by commas.

String emails = people.stream()

.map(Person::getEmail)

.collect(Collectors.joining(", "));

System.out.println("All Emails: " + emails);

**Explanation**: joining is useful for creating a single string from stream elements, with optional delimiters.

### ****Parallel Stream****

Parallel processing to speed up large datasets.

long femaleCount = people.parallelStream()

.filter(p -> p.getGender() == Gender.FEMALE)

.count();

System.out.println("Number of females: " + femaleCount);

**Explanation**: Parallel streams execute operations concurrently, useful for large collections where performance can benefit from multi-core processing.