Java collector api :

The Java Collector API, introduced in Java 8, is a powerful framework for performing operations on streams and collecting elements into various data structures. It provides a way to accumulate the elements of a stream into collections, summarizing statistics, grouping elements, and more. Let's explore the Collector API in depth:

Basic Usage: To use the Collector API, you typically invoke the `collect()` method on a stream and pass a collector as an argument. The collector specifies how the elements should be collected. Collectors are created using the `Collectors` class, which provides a set of predefined collectors.

Custom Collectors: You can create your own custom collectors by implementing the `Collector` interface. The `Collector` interface defines a set of methods that control the accumulation of elements and the creation of the final result.

A `Collector` interface has the following methods:

1. `supplier()`: Creates a new mutable result container.
2. `accumulator()`: Incorporates a stream element into the result container.
3. `combiner()`: Combines two result containers into one.
4. `finisher()`: Performs a final transformation on the result container.
5. `characteristics()`: Specifies the characteristics of the collector, such as whether it supports concurrent collection or maintains the order of elements.

By implementing these methods, you can define custom logic for collecting elements, merging partial results, and performing final transformations.

For example, suppose you want to collect a stream of Person objects into a Map where the keys are their ages and the values are lists of persons of that age. You can create a custom collector as follows:

```
Collector<Person, ?, Map<Integer, List<Person>>> collector =
    Collector.of(
        HashMap::new,
        (map, person) -> map.computeIfAbsent(person.getAge(), k ->
new ArrayList<>()).add(person),
        (map1, map2) -> {
            map2.forEach((age, persons) -> map1.merge(age, persons,
(list1, list2) -> { list1.addAll(list2); return list1; }));
            return map1;
        }
    );
Map<Integer, List<Person>> result =
persons.stream().collect(collector);
```

This custom collector uses the `Collector.of()` method to define the supplier, accumulator, combiner, and finisher functions.

Characteristics: Collectors can have different characteristics that specify how they behave. The `Characteristics` enum provides the following characteristics:

- `CONCURRENT`: Indicates that the collector supports parallel collection.
- `UNORDERED`: Specifies that the order of elements in the resulting collection is not important.
- `IDENTITY_FINISH`: Indicates that the finisher function is the identity function and can be skipped.

The `Collector` interface provides the `characteristics()` method, which allows you to specify the desired characteristics for your custom collector.

Using the Collector API, you can perform complex data transformations and aggregations on streams efficiently and concisely.

Collectors utility class methods :

The `Collectors` class in Java provides a variety of methods for creating collectors to perform operations on streams and accumulate elements into various data structures. Here is a comprehensive list of methods available in the `Collectors` class as of Java 17:

1. To create `List`, `Set`, or `Collection` collectors:
    - `toList()`: Collects elements into a `List`.
    - `toSet()`: Collects elements into a `Set`.
    - `toCollection(collectionFactory)`: Collects elements into a specified collection created by the given factory function.
2. To create `Map` collectors:
    - `toMap(keyMapper, valueMapper)`: Collects elements into a `Map`, where the keys and values are derived using the provided mapping functions.

- `toMap(keyMapper, valueMapper, mergeFunction)`: Collects elements into a `Map`, handling collisions with the specified merge function.
- `toConcurrentMap(keyMapper, valueMapper)`: Collects elements into a concurrent `Map`, where the keys and values are derived using the provided mapping functions.
- `toConcurrentMap(keyMapper, valueMapper, mergeFunction)`: Collects elements into a concurrent `Map`, handling collisions with the specified merge function.

3. To join elements into a `String`:
    - `joining()`: Concatenates elements into a single string.
    - `joining(delimiter)`: Concatenates elements into a string with the specified delimiter.
    - `joining(delimiter, prefix, suffix)`: Concatenates elements into a string with the specified delimiter, prefix, and suffix.
4. To calculate statistical information:
    - `counting()`: Counts the number of elements in a stream.
    - `summingInt(), summingLong(), summingDouble()`: Calculates the sum of the selected numeric property from the stream elements.
    - `averagingInt(), averagingLong(), averagingDouble()`: Calculates the average of the selected numeric property from the stream elements.
    - `minBy(comparator)`: Finds the minimum element of a stream based on the specified comparator.
    - `maxBy(comparator)`: Finds the maximum element of a stream based on the specified comparator.
    - `summarizingInt(), summarizingLong(), summarizingDouble()`: Collects statistical information about the selected numeric property from the stream elements.
5. To perform grouping and partitioning:
    - `groupingBy(classifier)`: Groups elements based on the provided classifier function.
    - `groupingBy(classifier, downstream)`: Groups elements based on the classifier function and applies a downstream collector to each group.

  - `partitioningBy(predicate)`: Partitions elements into two groups based on the provided predicate.
  - `partitioningBy(predicate, downstream)`: Partitions elements into two groups based on the predicate and applies a downstream collector to each group.
6. To create custom collectors:
  - `of(supplier, accumulator, combiner, finisher, characteristics)`: Creates a collector with the specified supplier, accumulator, combiner, finisher, and characteristics.
7. Additional utility methods:
  - `teeing(collector1, collector2, merger)`: Performs two collectors in parallel and merges their results using the specified merger.
  - `filtering(predicate, downstream)`: Applies a filtering operation before collecting elements with the specified downstream collector.
  - `mapping(mapper, downstream)`: Applies a mapping operation before collecting elements with the specified downstream collector.
  - `flatMapping(mapper, downstream)`: Applies a flat-mapping operation before collecting elements with the specified downstream collector.
  - `reducing(identity, mapper, op)`: Reduces the elements using the specified identity, mapper, and binary operator.

These methods offer a wide range of functionalities to collect and process elements from streams effectively. You can use them to aggregate data, group elements, calculate statistics, perform custom transformations, and more.

Class example

```
package collectordemo;

import java.util.Objects;
```

```java
public class Employee {
    private int id;
    private String name;
    private String deptName;
    private int age;

    public Employee(int id, String name, String deptName, int age) {
        this.id = id;
        this.name = name;
        this.deptName = deptName;
        this.age = age;
    }

    public Employee() {
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDeptName() {
        return deptName;
    }

    public void setDeptName(String deptName) {
        this.deptName = deptName;
    }
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Employee employee = (Employee) o;
        return id == employee.id && age == employee.age &&
Objects.equals(name, employee.name) && Objects.equals(deptName,
employee.deptName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, name, deptName, age);
    }

    @Override
    public String toString() {
        return "Employee{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", deptName='" + deptName + '\'' +
                ", age=" + age +
                '}';
    }
}

package collectordemo;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectorDemo {
    public static void main(String[] args) throws IOException {
        Employee e1 = new Employee(1, "abc", "dept1", 20);
        Employee e2 = new Employee(2, "def", "dept1", 25);
        Employee e3 = new Employee(3, "abc", "dept2", 30);
        Employee e4 = new Employee(4, "ijk", "dept3", 50);
        Employee e5 = new Employee(5, "nmo", "dept3", 22);
        Employee e6 = new Employee(6, "xyz", "dept3", 45);

        List<Employee> empList = List.of(e1, e2, e3, e4, e5, e6);

        //name-deptName comma separated
        String empNameList = empList.stream()
                .map(emp -> emp.getName() + "-" + emp.getDeptName())
                .collect(Collectors.joining(","));
        System.out.println(empNameList);

        //map of id=name-deptName
        Map<Integer, String> mapOfEmpId = empList.stream()
                .collect(Collectors.toMap(Employee::getId,//key function
                        emp -> emp.getName() + "-" + emp.getDeptName())//value
function
                );
        System.out.println(mapOfEmpId);

        //grouping by deptName
        Map<String, List<Employee>> empGroupByDept = empList.stream()
                .collect(Collectors.groupingBy(emp -> emp.getDeptName()));
        System.out.println(empGroupByDept);

        /*
        {
            dept1=[Employee{id=1, name='abc', deptName='dept1', age=20},
                 Employee{id=2, name='def', deptName='dept1', age=25}
                 ],
            dept2=[Employee{id=3, name='abc', deptName='dept2', age=30}],
            dept3=[Employee{id=4, name='ijk', deptName='dept3', age=50},
                 Employee{id=5, name='nmo', deptName='dept3', age=22},
                 Employee{id=6, name='xyz', deptName='dept3', age=45}
                 ]
        }
         */

        //partition employee by age >=30
        Map<Boolean, List<Employee>> partitionByAge = empList.stream()
                .collect(Collectors.partitioningBy(emp -> emp.getAge() >= 30));
        System.out.println(partitionByAge);


        /*
        {
            false=[Employee{id=1, name='abc', deptName='dept1', age=20},
                 Employee{id=2, name='def', deptName='dept1', age=25},
                 Employee{id=5, name='nmo', deptName='dept3', age=22}
                ],
            true=[Employee{id=3, name='abc', deptName='dept2', age=30},
                 Employee{id=4, name='ijk', deptName='dept3', age=50},
                 Employee{id=6, name='xyz', deptName='dept3', age=45}
                ]
          }

         */

        //count of employee in dept1
        Long count = empList.stream()
                .filter(emp -> "dept1".equals(emp.getDeptName()))
                .collect(Collectors.counting());
        System.out.println("count of dept1 employees is : "+count);



        //creating stream of file lines and finding number of words (space separated)
in stream
```

```
        Long numberOfLines = Files.readAllLines(Paths.get("sample.txt")).stream()
                .flatMap(line -> Stream.of(line.split(" ")))
                .collect(Collectors.counting());
        System.out.println(numberOfLines);

    }

}
```