



Stream API

Agenda

1 Introduction to Stream API

2 filter() and collect()

3 filter() and forEach()

4 reduce()

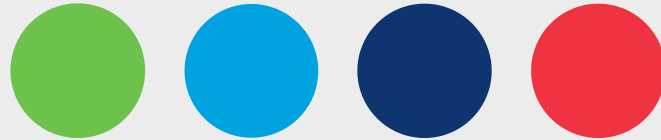
5 filter() and count()

6 max() and min()

7 limit()

8 map()

Introduction to Stream API



Introduction to Stream API

- A Stream in Java can be defined as a **sequence of elements** from a source(collections) that supports aggregate operations like filtering, finding max, min and sum etc.. on them.
- Stream elements supports sequential and parallel aggregate operations.
- It is conceptually a fixed data structure, in which elements are **computed on demand**.
- The idea is that a user will extract only the values they require from a Stream, and these elements are only produced—invisibly to the user—as and when required. This is a form of a **producer-consumer** relationship.
- Streams bring in functional programming to Java.
- **java.util.stream** package is introduced in Java 8 which contains classes and interfaces to support such functional-style operations on streams of elements.

Introduction to Stream API

- A stream pipeline consists of a source, followed by zero or more intermediate operations, and a terminal operation.
- **Example:**
 1. Source – Filter – Collect
 2. Source – Filter – Map – Collect
- **Stream Source:** Streams can be created from Collections, Lists, Sets, arrays, lines of a file etc..
- Stream operations are either intermediate or terminal.
- **Intermediate operations** such as filter, map and sorted return a new stream so that we can chain multiple operations.
- **Terminal operations** such as forEach, collect and reduce are either void or return a non-stream result.

Introduction to Stream API

Important interfaces in java.util.stream package

Interface	Description
BaseStream	Base interface for streams, which are sequences of elements supporting sequential and parallel aggregate operations.
Stream	A sequence of elements supporting sequential and parallel aggregate operations.

Important class in java.util.stream package

Class	Description
Collectors	Implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Introduction to Stream API

Important abstract methods in Stream interface

filter	map	collect
limit	forEach	reduce
max	min	count

Where are the implementation of these methods available?

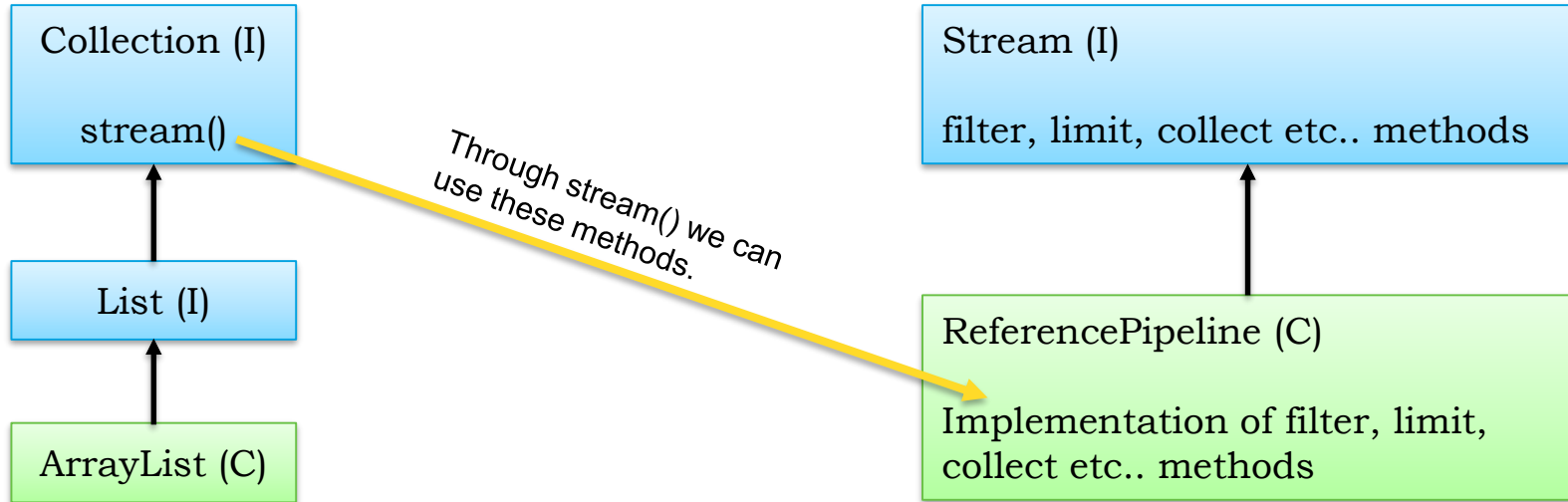
- Implementation of these abstract methods are available in the **java.util.stream.ReferencePipeline** class.
- It is an abstract class which implements the **Stream** interface.
- It is an undocumented class. We cannot find its details in Java 8 documentation.

https://github.com/JetBrains/jdk8u_jdk/blob/master/src/share/classes/java/util/stream/ReferencePipeline.java

Introduction to Stream API

Important method added to the Collection interface

- A default method **stream()** is added to the **Collection** interface which returns a sequential **Stream** with this collection as its source.
- **Signature:** default `Stream<E> stream()`



Introduction to Stream API

Steps to use the stream() method to access the Stream API methods

1. Create an ArrayList or TreeSet object and add some elements.

```
ArrayList<Integer> li = new ArrayList<Integer>();
```

```
TreeSet<String> ts = new TreeSet<String>();
```

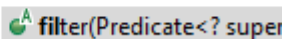
2. Call the stream() method using these objects.

```
li.stream();
```

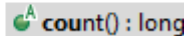
```
ts.stream();
```

3. Start using the Stream API methods.

```
li.stream().filter;
```

 filter(Predicate<? super

```
ts.stream().count;
```

 count() : long

filter() and collect()



filter() and collect()

filter()

- Returns a new stream consisting of the elements of the given stream that match the given predicate(condition).
- **Predicate**(condition) is called for each element in the stream.
- An element will be included in the resulting stream only when the predicate returns **true** for that element.

Example scenarios where this method will be useful

1. Details of the employees who are working in Chennai location needs to be filtered out.
2. Details of the vehicles whose color is black needs to be filtered out.

filter() and collect()

collect()

- This terminal method is used to perform operations like packing the elements into a data structure, applying additional logics on the elements etc..
- Functions from **Collectors** class are passed as an argument to mention how the elements has to be packed.
- For example, **Collectors.toList()** will accumulate the elements into an ArrayList.

Example scenario where this method will be useful

1. All the Employee objects whose salary is less than 20,000 needs to be filtered out and stored into an ArrayList.

filter() and collect()

Example 1 : Filtering even numbers

Code:

```
List<Integer> al = new ArrayList<Integer>();  
al.add(3);  
al.add(8);  
al.add(9);  
  
List<Integer> li = al.stream() //source  
                        .filter(x -> x % 2 == 0) //intermediate operation  
                        .collect(Collectors.toList()); //terminal operation  
  
System.out.println("Original List: " + al);  
System.out.println("Filtered List: " + li);
```

Output:

```
Original list: [3, 8, 9]  
Filtered list: [8]
```

filter() and collect()

Example 2 : Filtering negative numbers

Code:

```
List<Integer> al = new ArrayList<Integer>();  
al.add(-2);  
al.add(3);  
al.add(-5);  
  
List<Integer> li = al.stream() //source  
                        .filter(x -> x < 0) //intermediate operation  
                        .collect(Collectors.toList()); //terminal operation  
  
System.out.println("Original List: " + al);  
System.out.println("Filtered List: " + li);
```

Output:

```
Original list: [-2, 3, -5]  
Filtered list: [-2, -5]
```

filter() and collect()

Example 3 : Filtering on Strings

Code:

```
List<String> al = new ArrayList<String>();  
al.add("admin");  
al.add("user");  
al.add("customer");  
List<String> li = al.stream()    //source  
                    .filter(s -> s.contains("e"))    //intermediate operation  
                    .collect(Collectors.toList());    //terminal operation  
System.out.println(li);
```

Output:

[user, customer]

filter() and collect()

Example 4 : Filtering the Employees based on their salary

Employee.java

```
public class Employee {  
    int empNo;  
    String name;  
    int salary;  
    Employee(int empNo, String name, int salary) {  
        this.empNo = empNo;  
        this.name = name;  
        this.salary = salary;  
    }  
}
```


filter() and collect()

Example 4 continued..

Main.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
public class Main {
    public static void main(String[] args) {
        List<Employee> li=new ArrayList<Employee>();
        li.add(new Employee(100,"John", 18000));
        li.add(new Employee(200,"Varun", 10000));
        li.add(new Employee(300, "Riya", 22000));    //continued..
    }
}
```

filter() and collect()

Example 4 continued..

Main.java continued..

```
List<Employee> li2 = li.stream()  
                        .filter(emp -> emp.salary > 15000)  
                        .collect(Collectors.toList());
```

```
for (Employee employee : li2) {  
    System.out.println(employee.name);  
}  
}  
}
```

Output:

John
Riya

filter() and forEach()



filter() and forEach()

Example 1 : Filtering and printing even numbers

Code:

```
List<Integer> al = new ArrayList<Integer>();  
al.add(5);  
al.add(6);  
al.add(8);  
  
al.stream()    //source  
  
    .filter(x -> x % 2 == 0) //intermediate operation  
  
    .forEach(x -> System.out.print(x+" ")); //terminal operation
```

Output:

6 8

reduce()



reduce()

- A reduction is a terminal operation that aggregates a stream into a type or a primitive.
- **reduce()** is a general-purpose method for generating our custom reduction operations.

reduce(T identity, BinaryOperator<T> accumulator)

1. The identity element is both the initial value of the reduction and the default result if there are no elements in the stream.
2. The accumulator function takes two parameters: a partial result of the reduction and the next element of the stream.

reduce()

Example 1 : Calculating sum of numbers

Code:

```
List<Integer> al = new ArrayList<Integer>();  
al.add(10);  
al.add(20);  
al.add(30);  
int sum = al.stream().reduce(0, Integer::sum);  
System.out.println(sum);
```

Output: 60

reduce()

What is Integer::sum?

- Static method **sum** has been added to the Integer class in Java 8.
- It takes two integer arguments and returns their addition result.
- :: method reference operator is used to call a method by referring to it with the help of its class directly.
- **Signature:** public static int sum(int a, int b)

filter() and count()



filter() and count()

count()

- Returns the count of elements in the given stream which satisfies the given filter condition.

Example scenarios where this method will be useful

1. When the count of how many students have cleared the test is required.
2. When the count of how many employees are working in onsite is required.

filter() and count()

Example 1 : Count of even numbers in the list

Code:

```
List<Integer> al = new ArrayList<Integer>();  
al.add(2);  
al.add(9);  
al.add(4);  
long c = al.stream().filter(x -> x % 2 == 0).count();  
System.out.println(c);
```

Output: 2

max()



max()

- Returns the maximum element of the given stream according to the provided Comparator.
- It returns an **Optional** instance describing the maximum element of this stream, or an empty Optional if the stream is empty.
- Use the Optional class **get()** method to fetch the actual value.

max()

Example 1 : Finding the maximum number from the list

Code:

```
List<Integer> al = new ArrayList<Integer>();  
al.add(23);  
al.add(34);  
al.add(67);  
int m = al.stream().max((x, y) -> x > y ? 1 : -1).get();  
System.out.println(m);
```

Output: 67

min()



min()

- Returns the minimum element of the given stream according to the provided Comparator.
- It returns an **Optional** instance describing the minimum element of this stream, or an empty Optional if the stream is empty.
- Use the Optional class **get()** method to fetch the actual value.

min()

Example 1 : Finding the minimum number from the list

Code:

```
List<Integer> al = new ArrayList<Integer>();  
al.add(23);  
al.add(34);  
al.add(67);  
int m = al.stream().min((x, y) -> x > y ? 1 : -1).get();  
System.out.println(m);
```

Output: 23

limit()



limit()

- Returns a new stream consisting of the elements of the given stream, truncated to be no longer than the given **maxSize**.

Stream<T> limit(long maxSize)

- **maxSize**: the number of elements the stream should be limited to.

limit()

Example 1 : Printing 10 numbers by skipping one value

Code:

```
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args)
    {
        Stream.iterate(0, n -> n + 2)
            .limit(10)
            .forEach(System.out::println);
    }
}
```

Output:

0
2
4
6
8
10
12
14
16
18

limit()

Example 2 : Printing first 10 odd numbers

Code:

```
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args)
    {
        Stream.iterate(1, n -> n + 1)
            .filter(n -> n%2 != 0)
            .limit(10)
            .forEach(System.out::println);
    }
}
```

Output:

1
3
5
7
9
11
13
15
17
19

map()



map()

- The intermediate operation map converts each element in the stream into another object via the given function.
- It returns a new stream consisting of the results of applying the given function to the elements of the given stream.

map()

Example 1 : Converting the strings to uppercase format

Code:

```
List<String> li = new ArrayList<String>();  
li.add("wipro");  
li.add("technologies");  
li.add("chennai");  
li.stream()  
  .map(String::toUpperCase)  
  .forEach(System.out::println);
```

Output:

WIPRO
TECHNOLOGIES
CHENNAI

Task: Try to receive these formatted values into a new List.



Thank you