Core Java 8

Lesson 21 : Stream API

Capgemini

Lesson Objectives

After completing this lesson, participants will be able to

- Understand concept stream API
- Use stream API with collections
- Perform different stream opeations

This lesson covers new feature in Java 8, Stream API . It also covers processing collections using stream API .

Lesson outline:

Collections API in Java mainly focuses on storage and retrieval of its elements.
Many times there is need to perform advanced retrieval operations. For example,
Consider a collection of employees and the required operations to be performed:

1. Find the employees having age > 35
2. Group the employees based on their verticals and count.
3. and many more….

How many times do you find yourself re-implementing these operations using loops
over and over again?

21.1: Introduction to Streams API

Why Stream API?

Stream API allows developers process data in a declarative way.

Streams can leverage multicore architectures without writing a single line of multithread code

Enhances the usability of Java Collection types, making it easy to iterate and perform tasks against each element in the collection

Supports sequential and parallel aggregate operations

Stream API

Stream API to rescue

Streams follow the "what, not how" principle. While working with Stream API, as a developer, we only need to specify what needs to be done.

Stream can be sequential or parallel. A parallel stream is especially useful if the computer the program is running on has a multicore CPU.

The main reason for using a stream is for its supports for sequential and parallel aggregate operations. It allows developer to traverse over a collection of elements and perform aggregate operations, pipeline two or more operations, perform parallel execution, and more.

21.1: Introduction to Streams API

Stream API

Characteristics of Stream API

- Not a data structure
- Designed for lambdas
- Do not support indexed access
- Can easily be output as arrays or Lists
- Lazy
- Parallelizable
- Can be unbounded

Stream API Characteristics

Not a data structures:
Streams have no storage. They carry values from a source through a pipeline of operations. They also never modify the underlying data structure.

Designed for lambdas
All Stream operations take lambdas as arguments

Do not support indexed access
You can ask for the first element, but not the second or third or last element.

Can easily be output as arrays or Lists
Simple syntax to build an array or List from a Stream

Lazy
Many Stream operations are postponed until it is known how much data is eventually needed E.g., if you do a 10-second-per-item operation on a 100 element list, then select the first entry, it takes 10 seconds, not 1000 seconds.

Parallelizable
If you designate a Stream as parallel, then operations on it will automatically be done concurrently, without having to write explicit multi-threading code

Can be unbounded
Unlike with collections, you can designate a generator function, and clients can consume entries as long as they want, with values being generated on the fly

21.1: Introduction to Streams API

Stream Operations

Stream defines many operations, which can be grouped in two categories

- Intermediate operations

- Terminal Operations

Stream operations that can be connected are called intermediate operations. They can be connected together because their return type is a Stream.

Operations that close a stream pipeline are called terminal operations.

Intermediate operations are "lazy"

Intermediate operations

Terminal operation

Stream Operations

Some of the Stream methods perform intermediate operations and some others perform terminal operations.

Intermediate operations do not perform any processing until a terminal operation is invoked on the stream pipeline.

As shown in the slide, lets consider you want to travel by air which involves series of operations. Few of them are intermediate operations like travelling by taxi, take ticket from ticket counter, do check-in and finally you do terminal operations which is boarding the plane. The plane doesn't fly during the intermediate operations cause intermediate operations are lazy !

21.2: Working with Stream

Working with Stream: Step - 1

To perform a computation, first we need to define source of stream

To create a stream source from values, use "of " method

```
Stream<Integer> stream = Stream.of(10,20,30);
```

A stream can be obtained from sources like arrays or collections using "stream" method

To obtain steam from array, use java.util.Arrays class

- stream()

```
Integer[] values = new Integer[] {10,20,30};
Stream<Integer> stream = Arrays.stream(values);
```

To obtain stream from collections, use java.util.Collection interface

- stream()
- parallelStream()

Working with Stream: Step – 1

As stream doesn't store data, we need to define the source to perform stream operations. This is done by either creating stream or obtaining stream from array/collections.

21.2: Working with Stream
Working with Stream: Step - 2

A stream pipeline consist of source, zero or more intermediate operations and a terminal operation

A stream pipeline can be viewed as a query on the stream source

Operations on stream are categories as:

- Filter
- Map
- Reduce
- Search
- Sort

Working with Stream: Step – 2

After obtaining stream we can perform different operations on streams which are categorized into:

Filter
Map
Reduce

Kindly note a steam pipeline may consist of zero or more intermediate operations but it need only one terminal operation. A stream pipeline after terminal operation is closed automatically, hence its not available for further processing. A stream implementation may throw IllegalStateException if it detects that the stream is being reused.

21.2: Working with Stream

Stream Interface

The Stream API consists of the types in the java.util.stream package

The "Stream" interface is the most frequently used stream type

A Stream can be used to transfer any type of objects

Few important method of Stream Interface are:

Concat	Count
Collect	Filter
forEach	Limit
Map	Max
Min	Of
Reduce	Sorted

Intermediate
Terminal

Stream API

Stream API introduced in Java 8 and provided under package java.util.stream. The Stream interface is most frequently used type in Stream API. The slide list few important method of the Stream Interface.

Methods such as filter, map and sorted are examples of methods that perform intermediate operations.

Methods such as count and forEach perform terminal operations.

An intermediate operation always returns a stream, where as the terminal operations performs an action.

21.2: Working with Stream

Demo

Execute the :

- BasicStream

21.3: Stream Operations

Mapping

The Stream interface's map method maps each element of stream with the result of passing the element to a function.

Map() takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form.

The function is applied to each element, "mapping" it into a new element.

Syntax:

```
<R> Stream<R> map(java.util.function.Function<? super T, ? extends R> mapper)
```

The map method returns a new Stream of elements whose type may be different from the type of the elements of the current stream.
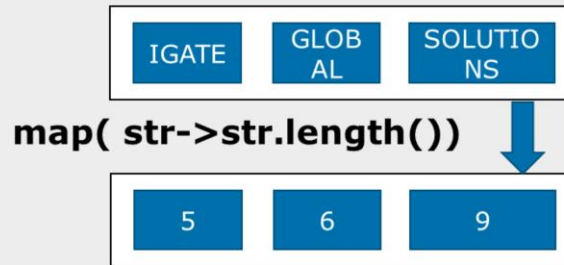
Mapping

Map method maps each element of stream with the result of passing the elements to a function.

21.3: Stream Operations
Mapping Example

```
List<String> words = Arrays.asList("IGATE","GLOBAL","SOLUTIONS");
words.stream().map(str->str.length()).forEach(System.out : : println);
```

IGATE	GLOB AL	SOLUTIO NS

map(str->str.length())

5	6	9

The mapping method returns a new stream of elements which may be different type. Here as shown in the slide example, the words stream is mapped based on length of each element. Therefore it returns a new stream of same size but the element type is integer.

Note: After any intermediate operation on stream, the processed items in resultant stream can be collected in separate collection if required.

```
List<String> words = Arrays.asList("IGATE","GLOBAL","SOLUTIONS");
List<Integer> counts = words.stream()
                            .map(str->str.length())
                            .collect(Collectors.toList());
```

21.3: Stream Operations

Filtering

There are several operations that can be used to filter elements from a stream:

Operation	What ?
filter(Predicate)	Takes a predicate (java.util.function.Predicate) as an argument and returns a stream including all elements that match the given predicate
distinct	Returns a stream with unique elements (according to the implementation of equals for a stream element)
limit(n)	Returns a stream that is no longer than the given size n
skip(n)	Returns a stream with the first n number of elements discarded

Filtering

A stream can be filtered out by using filter() method on Stream interface. Its takes predicate as argument which is usually a criteria for filtering, and returns new filtered stream containing elements which satisfies the criteria.

21.3: Stream Operations

Filtering Examples

filter(predicate)

> 11 44 66 33
> 44

```
List<Integer> listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().filter(num -> num > 10).forEach(num->System.out.println(num));
```

distinct()

```
List<Integer> listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().distinct().forEach(System.out :: println);
```

> 11 3 44 5
> 66 33

limit(size)

```
List<Integer> listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().limit(4).forEach(System.out :: println);
```

> 11 3 44 5

As shown in the slide examples, the filer operation filters a stream and reduces elements which less than 10.

The distinct() method removes duplicate from the stream.

The limit() method takes the new size of stream as argument and reduces the stream.

Note: All these operations are intermediate operations, whereas the forEach is terminal operation.
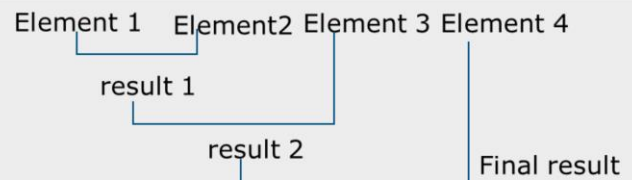
21.3: Stream Operations

Reducing

The reduce operation on streams, which repeatedly applies an operation on each element until a result is produced.

It's often called a fold operation in functional programming.

The reduce () method takes a BinaryOperator as argument and returns an Optional instance

Syntax:
java.util.Optional<T> reduce(java.util.function.BinaryOperator<T> accumulator))

Element 1   Element2 Element 3 Element 4

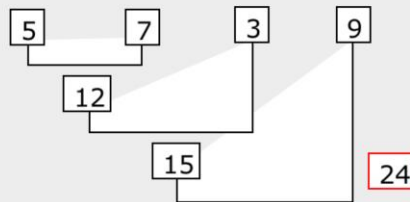result 1

result 2

Final result

21.3: Stream Operations

Reducing Example

```
List<Integer> intList = Arrays.asList(5,7,3,9);
Optional<Integer> result = intList.stream().reduce((a,b)->a+b);
if(result.isPresent()) {
        System.out.println("Result:"+result.get());
}
```

Reduction of elements by adding them

Result: 24

21.3: Stream Operations

Demo

Execute the :

- StreamMap
- StreamFiter
- StreamReduce

21.5: Stream API

Lab

Lab 11: Lambda Expressions and Stream API

Summary

> In this lesson, you have learnt:

- Working with Stream API
- Using Stream Operations on Collections

Review Question

Question 1 : Which of the following stream reduce call is valid to find max of given stream?

- **Option 1 :** stream.reduce((a,b)->a>b?a:b)

- **Option 2 :** stream.max()

- **Option 3 :** stream.map((a,b)->a>b)

Question 2 : _____ is a pipe for transferring data.

Question 3 : Resource-intensive tasks can be done efficiently by using parallel stream.

- True/False