Stream API

Lesson Objectives

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

Why Stream API?

How to find the most senior employee?

What is the count of employees joined this year?

Send meeting Invite to only Java Programmers

Group of Employees (Collections)

Manager

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

- 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 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

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 - 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

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

Mapping

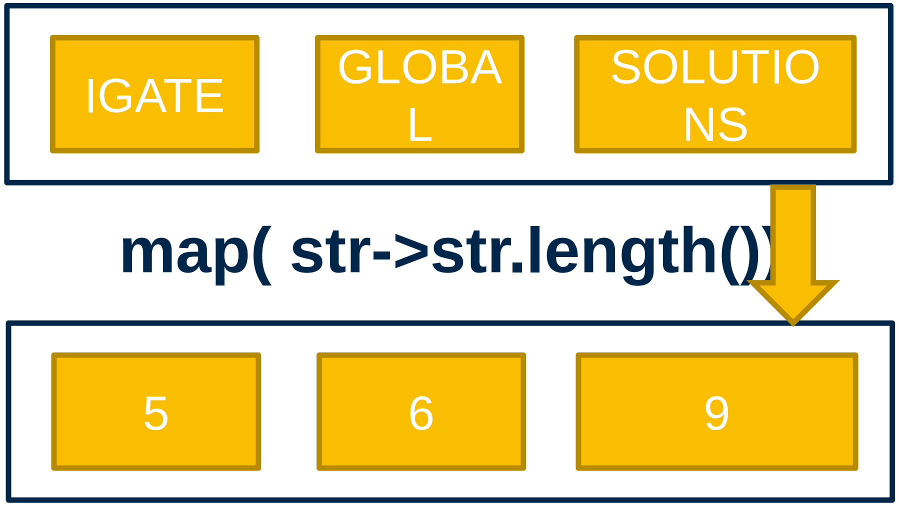
- The Stream interface's map method maps each element of stream with the result of passing the element to a function.
- Map() takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form.
- The function is applied to each element, “mapping” it into 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 Example

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



IGATE

GLOBA
L

SOLUTIO
NS

map(str->str.length())

5

6

9

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 Examples

- filter(predicate)

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

11 44 66 33
44

- 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);
```

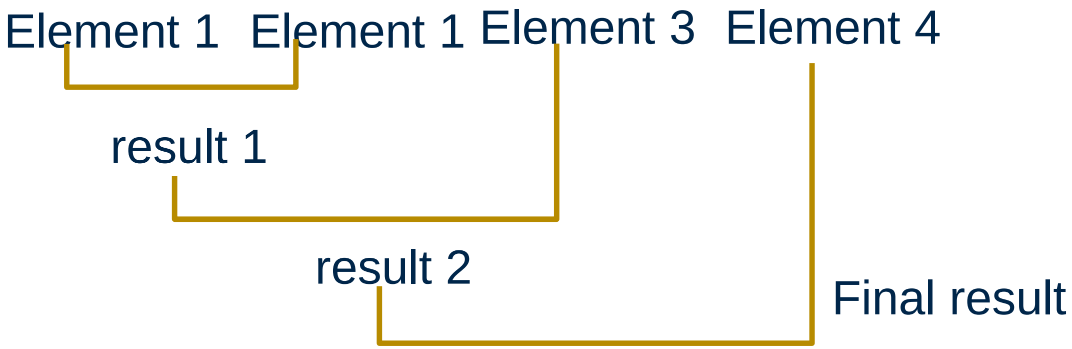
11 3 44 5

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
- Syntax:

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

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

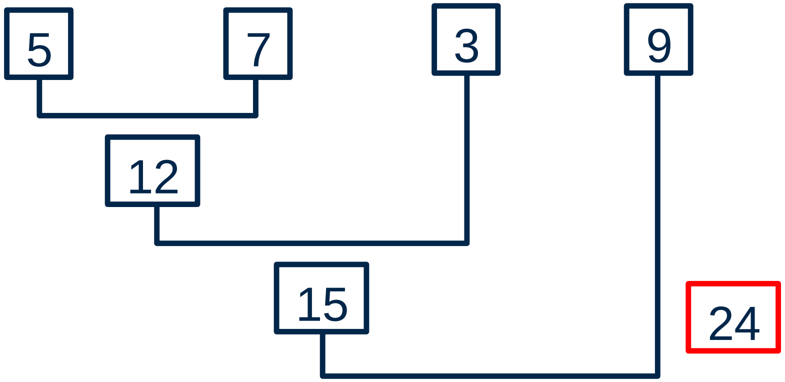


Reducing Example

```
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



Thank You!