Session: 16

Stream API

Objectives

- ❖Describe the Stream API
- ❖Outline the differences between collections and streams
- ❖Explain the classes and interfaces in Stream API
- ❖Describe how to use functional interfaces with Stream API
- ❖Describe the Optional class and Spliterator interface
- ❖Explain stream operations
- ❖Discuss the limitations of Stream API

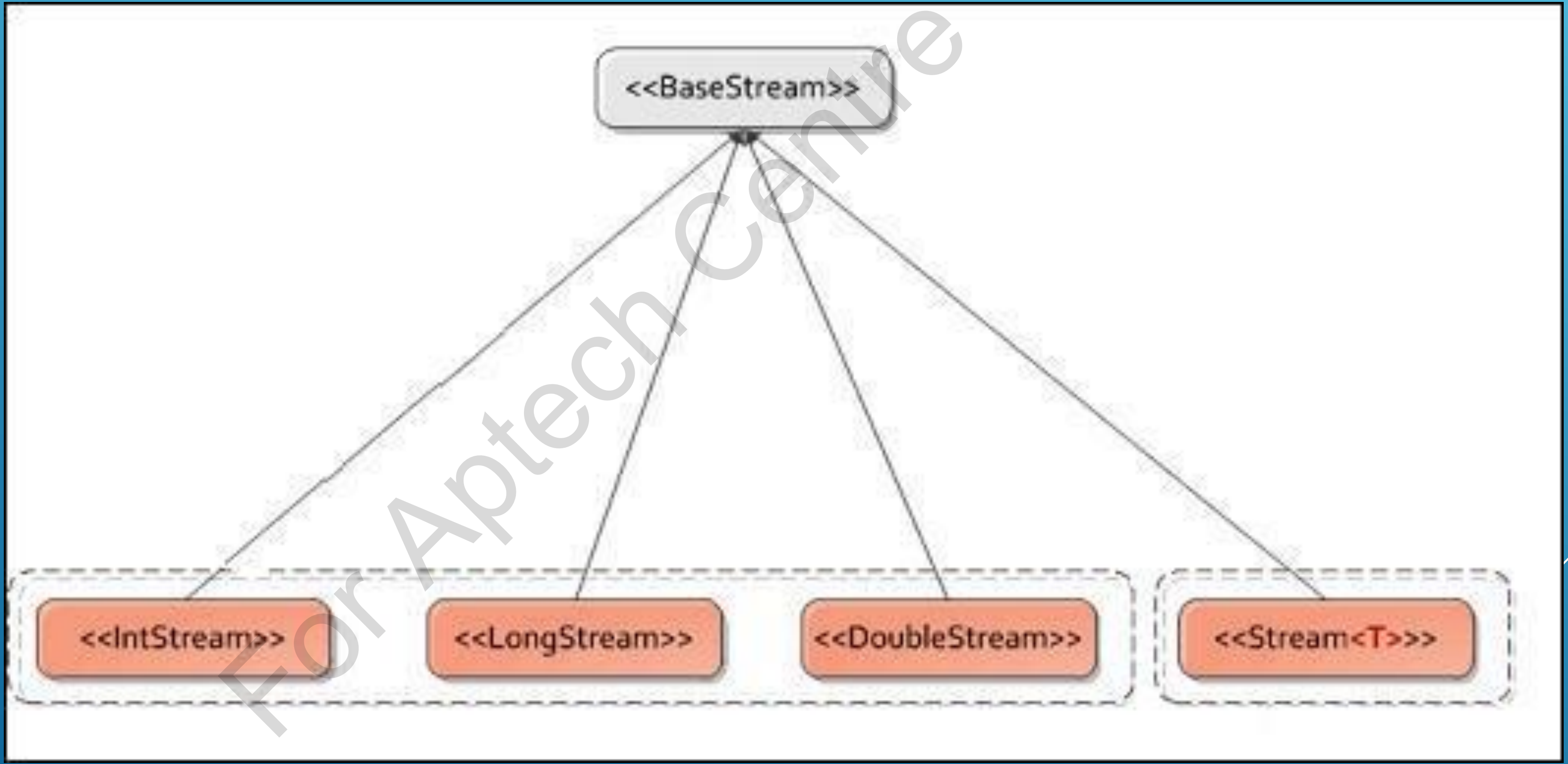
Stream API 1/2

Stream API is a notable Java 8 inclusion that allows parallel processing and helps to express efficient, SQL-like queries and manipulations on data.

Stream API 2/2

Stream interface and `Collectors` class forms the basic foundation of the Stream API.

`java.util.stream` package contains all the Stream API interfaces and classes. `IntStream`, `LongStream`, and `DoubleStream` are few interfaces of Stream API.



Collections and Streams

A stream is a series or set of elements that support sequential and parallel aggregate operations.

A collection is a set of data in the form of objects or elements.

Major differences between Streams and Collections are:

Streams	Collections
Fixed structures computed on-demand	In-memory data structure to store values
Operates on user demand basis	Focus on holding data
Data storage not available	Collections are actual data structures
Supports pipelining	May not support pipelining
Do not iterate	Iterate explicitly
Functional interface friendly with slow processing time	Functional Interface friendly with faster processing time

Generating Streams 1/6

There are many options available to generate a Stream in Java 8.

`stream()`: Is used to get a sequential `Stream` with the collection as its source.

Code Snippet shows the usage of `stream()`.

```
Stream<String> str = list.stream();
```

`parallelStream()`: Is used to get a possibly-parallel `Stream` lateral to the collection given as its source.

Code Snippet shows an example.

```
Stream parStr = list.parallelStream();
```

Generating Streams 2/6

BufferedReader class of java.io package includes the lines() method that returns a Stream, as shown in Code Snippet.

```
try (FileReader sampleFR = new FileReader ("D:\\random_file.txt");
BufferedReader sampleBR = new BufferedReader(sampleFR))
   {
       sampleBR.lines().forEach(System.out::println);
   }
...
```

Here, SampleBR is created as a BufferedReader instance that uses lines() method for a simple stream operation - reading and displaying data from a text file.

Generating Streams 3/6

Code Snippet shows how to read a file as a `java.util.stream.Stream` object using `Files.lines(Path filePath)`.

```
try (Stream sampleST = Files.lines(Paths.get("D:\\random_file.txt")))
{
   sampleST.forEach(System.out::println);
}
```

Generating Streams 4/6

Static methods on the Files class assist in navigating file trees using a Stream. Some of these are listed in the table.

Method	Explanation
`static Stream<Path> list(Path dir)`	Retrieves a `Stream`, whose elements include files in the specific directory.
`static Stream<Path> walk(Path dir, FileVisitOption options)`	Retrieves a Stream that is created by traversing the file tree starting at a specific file. `FileVisitOption` is an enumeration that defines file tree traversal options.
`static Stream<Path> walk(Path dir, int maxDepth, FileVisitOption options)`	Retrieves a Stream that is created by traversing the file tree depth-first starting at a specific file.

Generating Streams 5/6

Text patterns can be streamed using the `Pattern` class that contains a method, `splitAsStream(CharSequence)` to generate a stream.

```
import java.util.regex.Pattern;// to use Pattern class
public class TextPatterns
{
  public static void main( String args[] )
  {
      // Creating a pattern
      Pattern createPatt = Pattern.compile(",");// adding a comma
      // to pass a set of names
      createPatt.splitAsStream("Nathan, Ethan, Hank, Dennis, Sarah")
      .forEach(System.out::println);//
  }
}// result as stream
```

Generating Streams 6/6

- The example in previous slide generates a `Stream` from a simple text pattern that contains a comma as separator and separates the text into a Stream by using the `splitAsStream()` method.

- Then, each element in the `Stream` is printed out using a `forEach` loop. In practical scenarios, a similar code to match and display large collections of strings can be used.

Output:

```
Nathan
Ethan
Hank
Dennis
Sarah
```

Infinite Streams

An infinite stream is a sequence or collection of elements that has no limit.

Following Code Snippet shows an infinite quantity of objects being created using `generate()` method.

```
Stream.generate(()→"*").forEach(System.out::println); //
```

Stream Range

- The newly included primitive stream called `IntStream` can be used for `Stream` range calculation.

- Code Snippet describes the usage of static method, `range()` on the `IntStream` interface.

```
IntStream.range(2,18)// to produce Stream range
 .forEach(System.out::println);// result
```

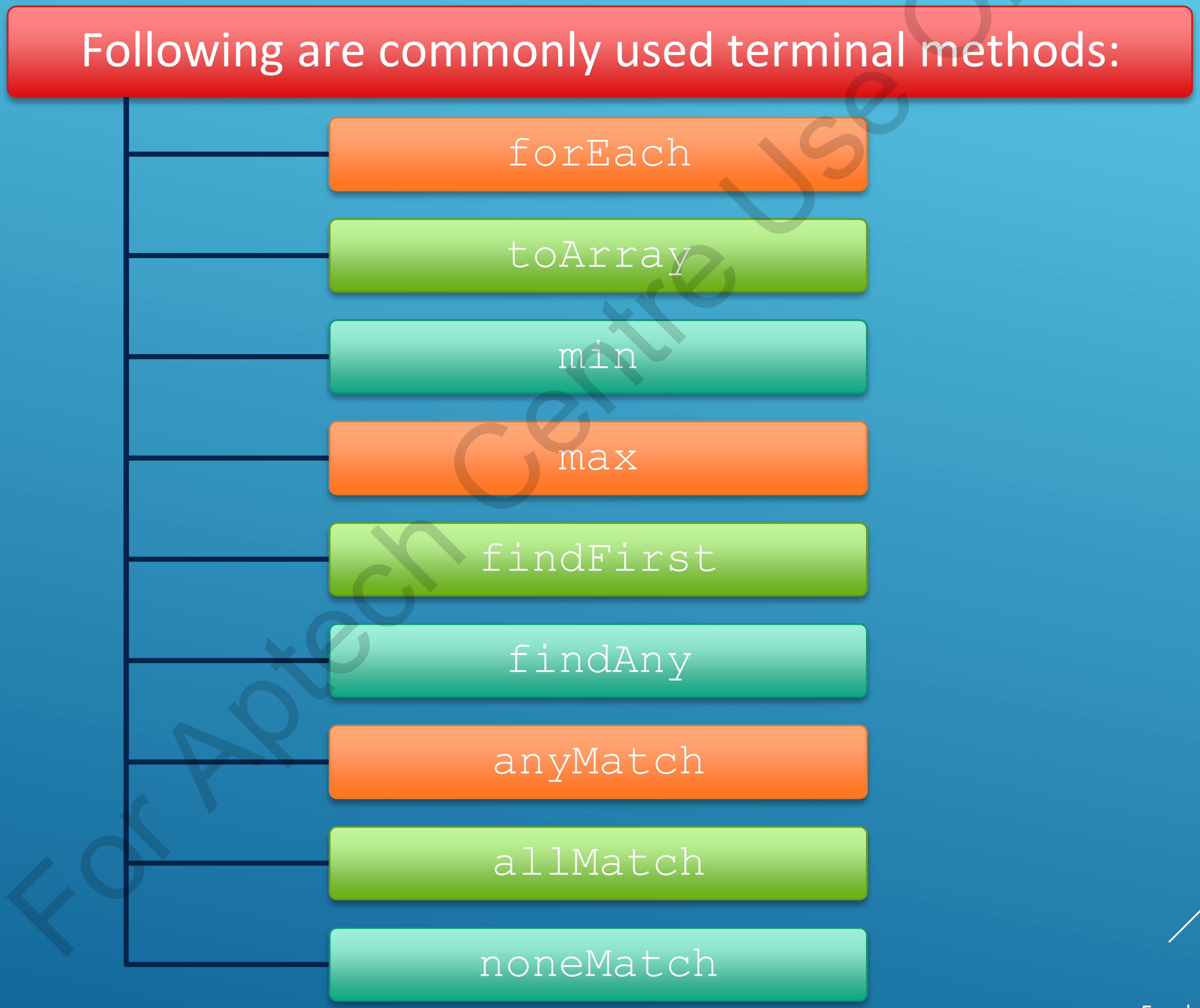
Operations on Streams 1/3

Intermediate Operations

- In intermediate operations, operators (intermediate operators) apply logic thus, the inbound `Stream` generates another `stream`.
- A `Stream` can contain `'n'` number of intermediate operators, which has no limitations.
- Intermediate operators can start a pipeline of `Stream` elements to execute the process further.

Operations on Streams 2/3

Terminal Operations



Following are commonly used terminal methods:

- forEach
- toArray
- min
- max
- findFirst
- findAny
- anyMatch
- allMatch
- noneMatch

Operations on Streams 3/3

Short-Circuiting Operations

Not Standalone Operations

Operation generating finite `Stream` from infinite `Stream` is defined as Short-circuiting

Map/Filter/Reduce with Streams

Map/Filter/Reduce methods implementations are allowed in lambda expressions.

Map:
This method is applied for mapping all the elements to its output.

Filter:
Choosing a set of element and eliminating other elements based on the instructions is the basic feature of Filter.

Reduce:
Reduce method is applied to reduce the elements based on the given instructions.

```
String outcome = scores.stream()
.reduce((acc, score) → acc + " " + score)
.get();
```

Streams and Parallel Array

`Array` class of Java contains functionalities for various array operations such as sort.

Following Code Snippet shows the usage of the parallel approach.

```
Arrays.parallelSort(sampleArray);
```

Limit

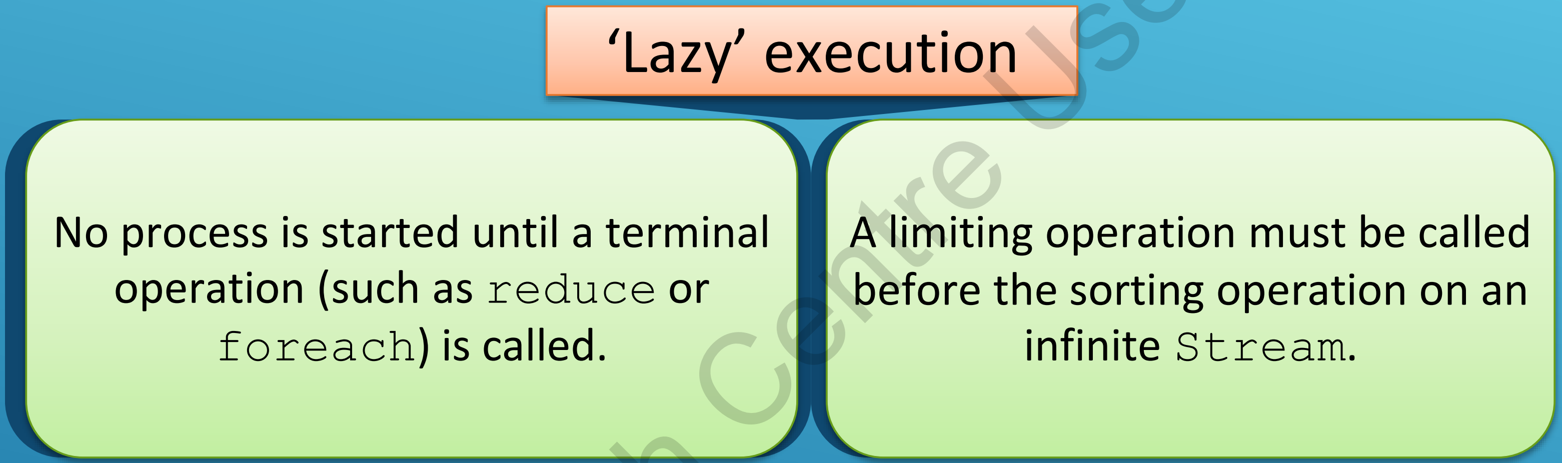
To limit a Stream to a specified number of elements, `limit()` method can be applied.

```
Random sampleRand = new Random();
sampleRand.ints().limit(12)
      .forEach(System.out::println);// to display the results
```

Here, `sampleRand` is used to return random integer values and `limit()` method is applied to limit the numbers. The code is limited to display only 12 random numbers.

Sort

`sorted()` method is another method within Stream API that helps to sort the Stream.



‘Lazy’ execution

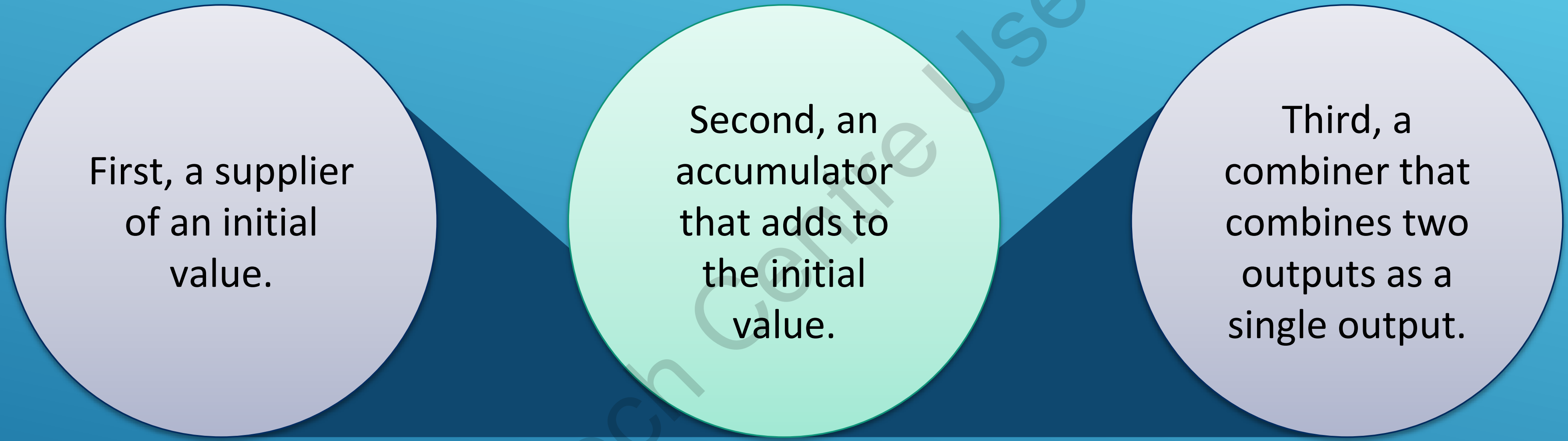
No process is started until a terminal operation (such as `reduce` or `foreach`) is called.

A limiting operation must be called before the sorting operation on an infinite `Stream`.

```
sampleRand.ints().limit(12).sorted()//limit before sort
.forEach(System.out::println);//to display output
```

Collectors

There are three different elements in a collector:



First, a supplier of an initial value.

Second, an accumulator that adds to the initial value.

Third, a combiner that combines two outputs as a single output.

There are two methods to implement this:

- `collect(supplier, accumulator, combiner)`
- `collect(Collector)`

Grouping and Partitioning 1/3

Grouping

Grouping (`groupingBy`) collector groups elements based on a given function.

```
// Grouping using first letter
List<Tiger>tigers = getTigers();
Map<Character, List<Tiger>> map = tigers.stream()
.collect(groupingBy(tiger →tiger.getName().charAt(0)));
// first letter
```

Names from the `tigers` list is grouped based on the first letter of each name.

Grouping and Partitioning 2/3

Partitioning

Partitioning `(partitioningBy)` method is parallel to Grouping method that creates a map with a boolean key.

```
Map<Boolean,List<Tiger>> map = tigers.stream()
.collect(partitioningBy(Tiger::isWhite));// white or not
```

Groups the elements based on whether the Tiger is white or not

Grouping and Partitioning 3/3

Parallel Grouping

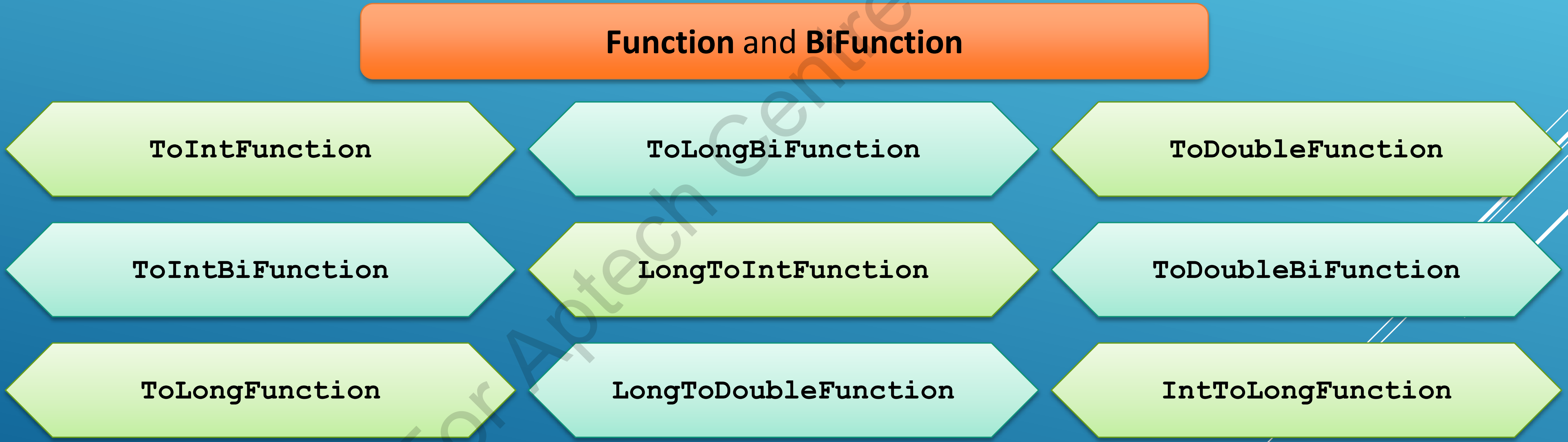
Parallel Grouping `(groupingByConcurrent)` executes grouping in parallel (without ordering).

```
tigers.parallelStream().unordered().
collect(groupingByConcurrent(Tiger::getColor));//parallel grouping
```

Using Functional Interfaces with Stream API 1/4

Function and BiFunction

- Functional interfaces can be used with several new APIs in Java 8.
- Commonly used functional interfaces are:



Function and BiFunction

ToIntFunction

ToLongBiFunction

ToDoubleFunction

ToIntBiFunction

LongToIntFunction

ToDoubleBiFunction

ToLongFunction

LongToDoubleFunction

IntToLongFunction

Using Functional Interfaces with Stream API 2/4

Predicate and BiPredicate

They denote a predicate against which arguments of the `Stream` are tested.

Following are the `Stream` methods in which `Predicate` or `BiPredicate` methods are used:

- `boolean noneMatch(Predicate<? super T> predicate)//`to filter no match
- `boolean anyMatch(Predicate<? super T> predicate)//`to filter any match
- `Stream<T> filter(Predicate<? super T> predicate)//`filter in Stream
- `boolean allMatch(Predicate<? super T> predicate)//`to filter all matches

Using Functional Interfaces with Stream API 3/4

Consumer and Biconsumer

They denote operations that accept a single input element and produce no output.

Example demonstrates Consumer and Biconsumer functions.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
public class SampleDemo {
  public static void main(String[] args) {
       List<Employee> employees = Arrays.asList(
           new Employee("John Simmons", 350000),
           new Employee("Mark Smith", 413000),
           new Employee("Jane Weston", 344000),
           new Employee("Gillian Bush", 690000)
       );
       displayAllEmployee(employees, e → {
       e.salary *= 1.5;
      );
      System.out.println("Salaries after increment:");
      displayAllEmployee(employees, e →
           System.out.println(e.empname + ": " + e.salary));
   }
```

Using Functional Interfaces with Stream API 4/4

```
  public static void displayAllEmployee(List<Employee> emp,
    Consumer<Employee> printer) {
      for (Employee e : emp) {
          printer.accept(e);
      }
   }
  public static void display(List<Employee> emp,
     Consumer<Employee> printer) {
   }
}
class Employee {
 public String empname;
 public long salary;
 Employee(String name, long sal) {
     this.empname = name;
     this.salary = sal;
 }
}
```

Optional and Spliterator API

`Optional` is a container object that optionally contains a value (non-null).
If it contains a value, `isPresent()` shows true and `get()` returns the value.
Following are `Stream` terminal operations that return an `Optional` object:

- `Optional<T> min(Comparator<? super T> comparator)`
  `// minimum`
- `Optional<T> max(Comparator<? super T> comparator)`
  `// maximum`
- `Optional<T> reduce(BinaryOperator<T> accumulator)`
  `// to reduce`
- `Optional<T>findFirst()`
  `// to find first`
- `Optional<T>findAny()`
  `// to find any`

`Spliterator` interface is used to support the parallel `execution.Spliterator(trySplit)` method which produces a new `Spliterator` that manages a subset of the elements of the original `Spliterator`.

Parallelism

- Parallelism is splitting a task into its sub-tasks, and then simultaneously running these tasks to merge their outputs.

- Adding a `parallel()` method to the `Stream` instructs the library to deal with the complexities of threading. Thus, the library controls the process of forking.

Executing Streams in Parallel 1/4

- ❖ Aggregate operations are implemented to combine the results.
- ❖ This process is known as concurrent reduction.
- ❖ Following conditions must be true for performing a collect operation in the process:
  - The `Stream` must be parallel.
  - The parameter of the collect operation, the collector, contains the characteristic `Collector.Characteristics.CONCURRENT`.
  - Stream must be unordered or the collector must contain the `Collector.Characteristics.UNORDERED`.

Executing Streams in Parallel 2/4

Following example shows a complete program with various Stream API operations:

```
import java.util.Arrays;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;
public class AptechJavaStreamAPI {
   public static void main(String args[]) {
     List<String> clientList = Arrays.asList("Flipkart",
     "Snapdeal", "PayTm", "King","","", "MaBeats", "Miniclip");
     System.out.println("^The new Client List: " + clientList);
     System.out.println("Result2:no. of clients with name length > 5: " + lengthCount);
     //To receive the client name starts with letter 'A' and display count
     long startCount = clientList.stream().filter(x -> x.startsWith("M")).count();
     System.out.println("Result3:no. of clients which name starts with letter M: " +
     startCount);
     // To eliminate all empty Strings from List
     List<String>removeEmptyStrings =
     clientList.stream().filter(x →!x.isEmpty()).collect(Collectors.toList());
     System.out.println("Result4:no. New Client List without empty list" +
     removeEmptyStrings);
     // To display the client names with > 8 characters
```

Executing Streams in Parallel 3/4

```
    List<String>newList = clientList.stream().filter(x ->x.length() >
    8).collect(Collectors.toList());
    System.out.println("Result5: New client list with letter count > 8: " + newList + "\n");
    List<Integer>aptechInt = Arrays.asList(77,66,888, 22, 33,7, 121, 89,55);
    IntSummaryStatistics aptechStats = aptechInt.stream().mapToInt((x) ->
    x).summaryStatistics();
    System.out.println("^ A list of Random numbers: " + aptechInt);
    System.out.println("Highest number in the lot -" + aptechStats.getMax());
    System.out.println("Lowest number in the lot -" + aptechStats.getMin());
    System.out.println("Combined value of All: " + aptechStats.getSum());
    System.out.println("Average value of all numbers: " + aptechStats.getAverage() + "\n");
    // To convert a Message in UPPERCASE and join them using space
    List<String>aptechTips = Arrays.asList("java8", "has", "some", "great", "features");
    String joinList = aptechTips.stream().map(x ->
    x.toUpperCase()).collect(Collectors.joining(" "));
    System.out.println("- To Join and Display the message with UPPERCASE: " + joinList);
    // To display the cube value of the numbers
    List<Integer> numbers = Arrays.asList(5,10,15,20,25);
    List<Integer> cubes = numbers.stream().map(myInt ->myInt *
    myInt * myInt).distinct().collect(Collectors.toList());
    System.out.println("- Display the cube value of the numbers : " + cubes + "\n");
  }
}
```

Executing Streams in Parallel 4/4

Output:

```
$javac AptechJavaStreamAPI.java 2>&1
^The new Client List: [Flipkart, Snapdeal, PayTm, King, , , MaBeats, Miniclip]
Result1:no. of Empty Strings: 2
Result2:no. of clients with name length > 5: 4
Result3:no. of clients which name starts with letter M: 2
Result4:no. New Client List without empty list[Flipkart, Snapdeal, PayTm, King,
MaBeats, Miniclip]
Result5: New client list with letter count > 8: []
^ A list of Random numbers: [77, 66, 888, 22, 33, 7, 121, 89, 55]
Highest number in the lot -888
Lowest number in the lot -7
Combined value of All: 1358
Average value of all numbers: 150.88888888888889
- To Join and Display the message with UPPERCASE: JAVA8 HAS SOME GREAT FEATURES
- Display the cube value of the numbers : [125, 1000, 3375, 8000, 15625]
```

Limitations of Java Stream API

- Once a Stream is consumed, it cannot be used later.

- Learning is time-consuming and cumbersome due to overloaded Stream APIs.

Summary

- The new Stream API in Java 8 supports many sequential and parallel aggregate operations
- Stream API interfaces and classes are contained within `java.util.stream` package
- The foundation of the Stream API is `Stream` interface and `Collectors` class
- Some of the interfaces in the API include `IntStream`, `LongStream`, and `DoubleStream`
- Streams are lazily implemented and support parallel operation
- Function denotes a function that gets one type of element and produces another type of element
- The Optional class and Spliterator interface defined in `java.util` package can be used with Stream API
- Commonly used functional interfaces with Stream API include Function and BiFunction, Predicate and BiPredicate, Consumer and BiConsumer, and Supplier