BridgeLabz

Employability Delivered

Java8 Core Concepts

Section 2: Stream API

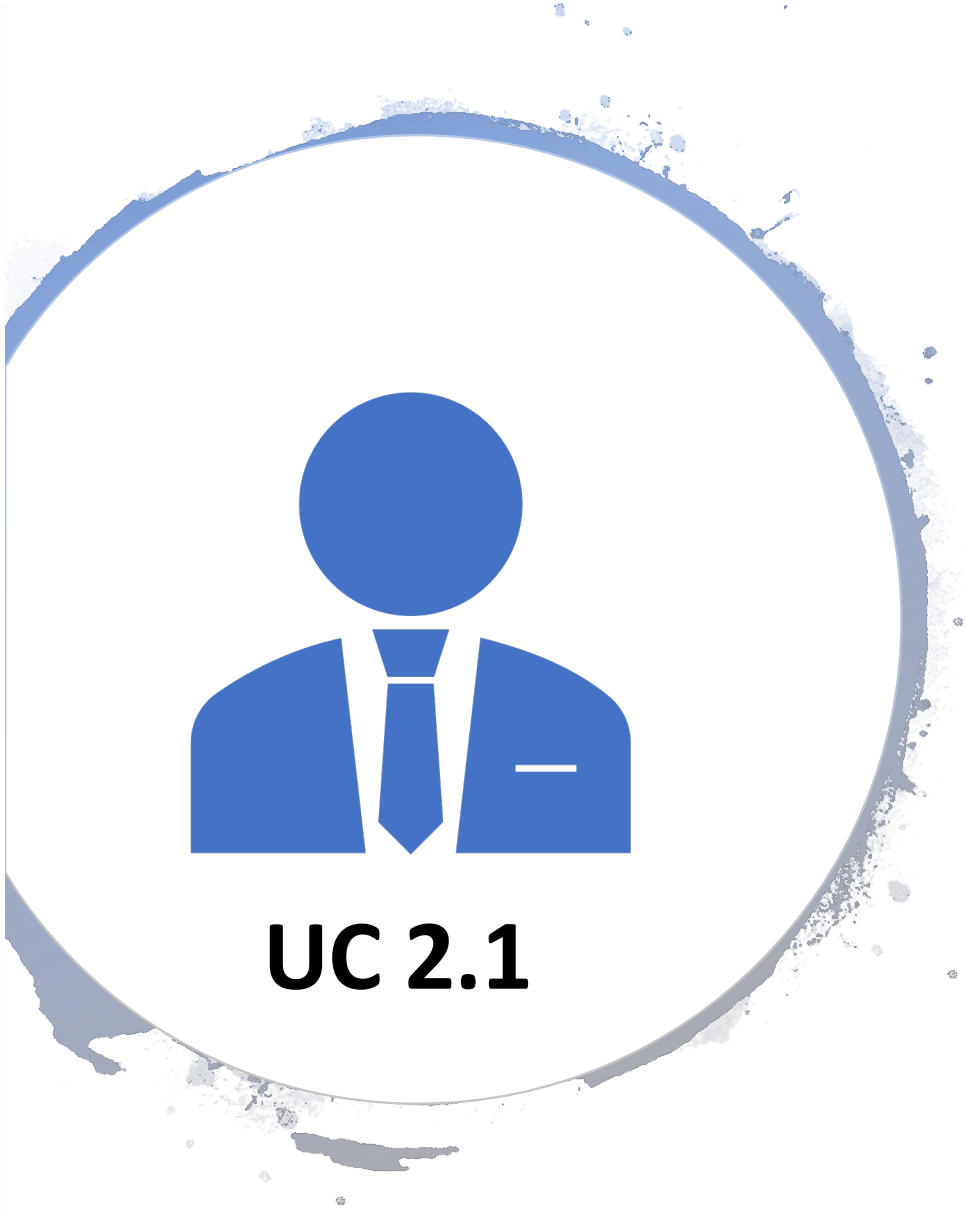
Java Streams API

- Java Streams can be visualized as **streaming/sequencing** of data from a Source. A stream is **not a data structure** and does not store elements.
- The **Source** of data here refers to a Collection or Array or File IO or any IO Stream that provides data to the Stream.
- Stream keeps the **Ordering** of the data elements the same as the ordering in the source.
- Streams supports **Operations** on the data like filter, map, reduce, find, etc. to manipulate the Stream Elements quickly and easily.
- The source of the stream remains unmodified after the operations are performed on it. It essentially produces a **new Stream**.
- This helps in creating a chain of stream operations which is essentially called as **Pipe-lining**.

Multiple ways to Generate Stream

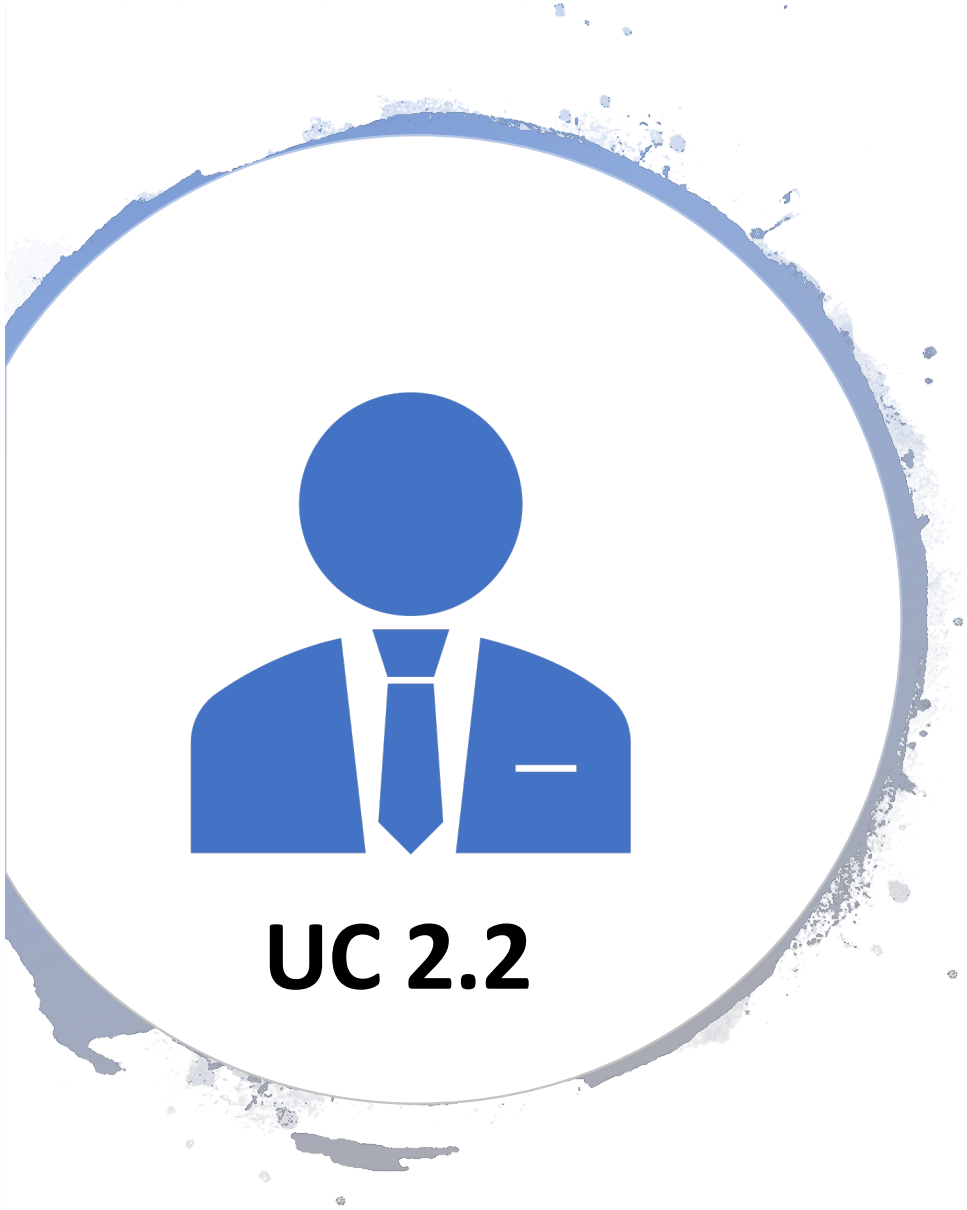
BridgeLabz
Employability Delivered

```
// Mth1: Creating Stream from Array
case "Mth1":
    return Arrays.stream(arrayOfEmps);
// Mth2: Creating Stream from List
case "Mth2":
    List<Employee> employeeList = Arrays.asList(arrayOfEmps);
    return employeeList.stream();
// Mth2: Creating Stream from elements using stream.of
case "Mth3":
    return Stream.of(arrayOfEmps);
// Mth2: Creating List from StreamBuilder
case "Mth4":
    Stream.Builder<Employee> empStreamBuilder = Stream.builder();
    empStreamBuilder.accept(arrayOfEmps[0]);
    empStreamBuilder.accept(arrayOfEmps[1]);
    empStreamBuilder.accept(arrayOfEmps[2]);
    return empStreamBuilder.build();
```



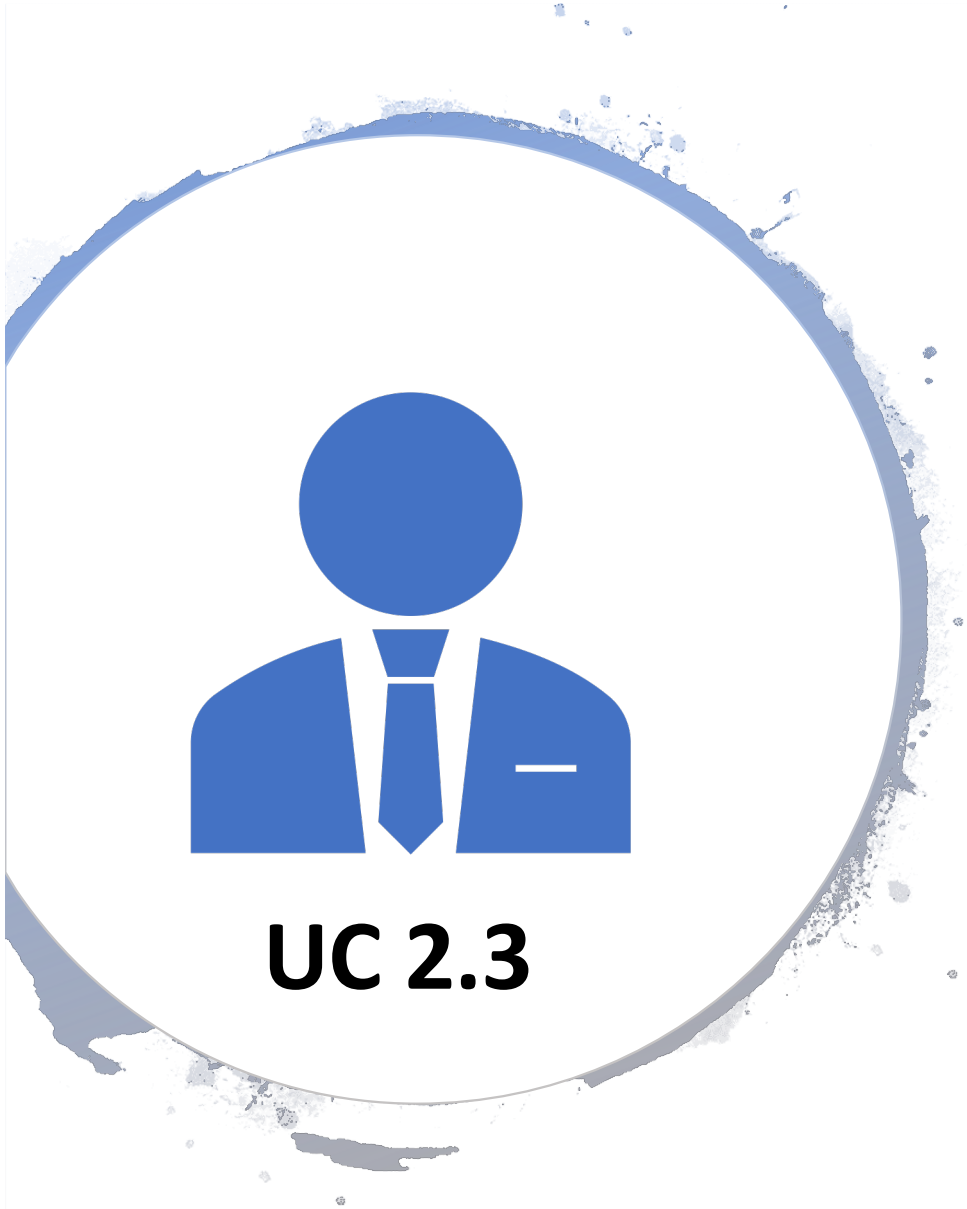
Ability to Create a Stream and Iterate to show each element of the stream

- Use stream.forEach



Ability to Transform each element to double and store the result

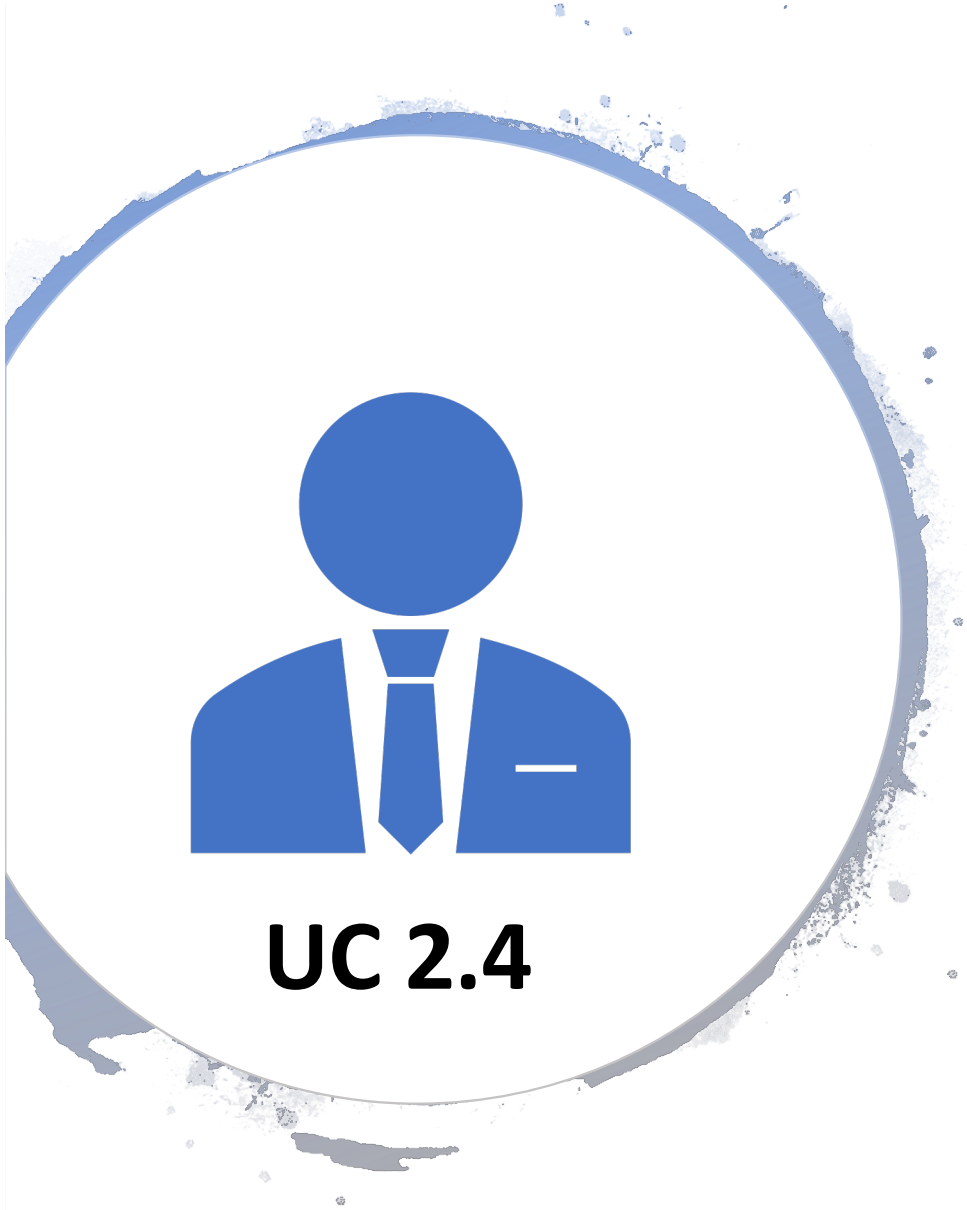
- Use stream.map function



UC 2.3

Ability to store the Transformed double value into a new List

- Use stream.collect function



Ability to filter the even numbers from the number Stream and store the result

- Use stream.filter function

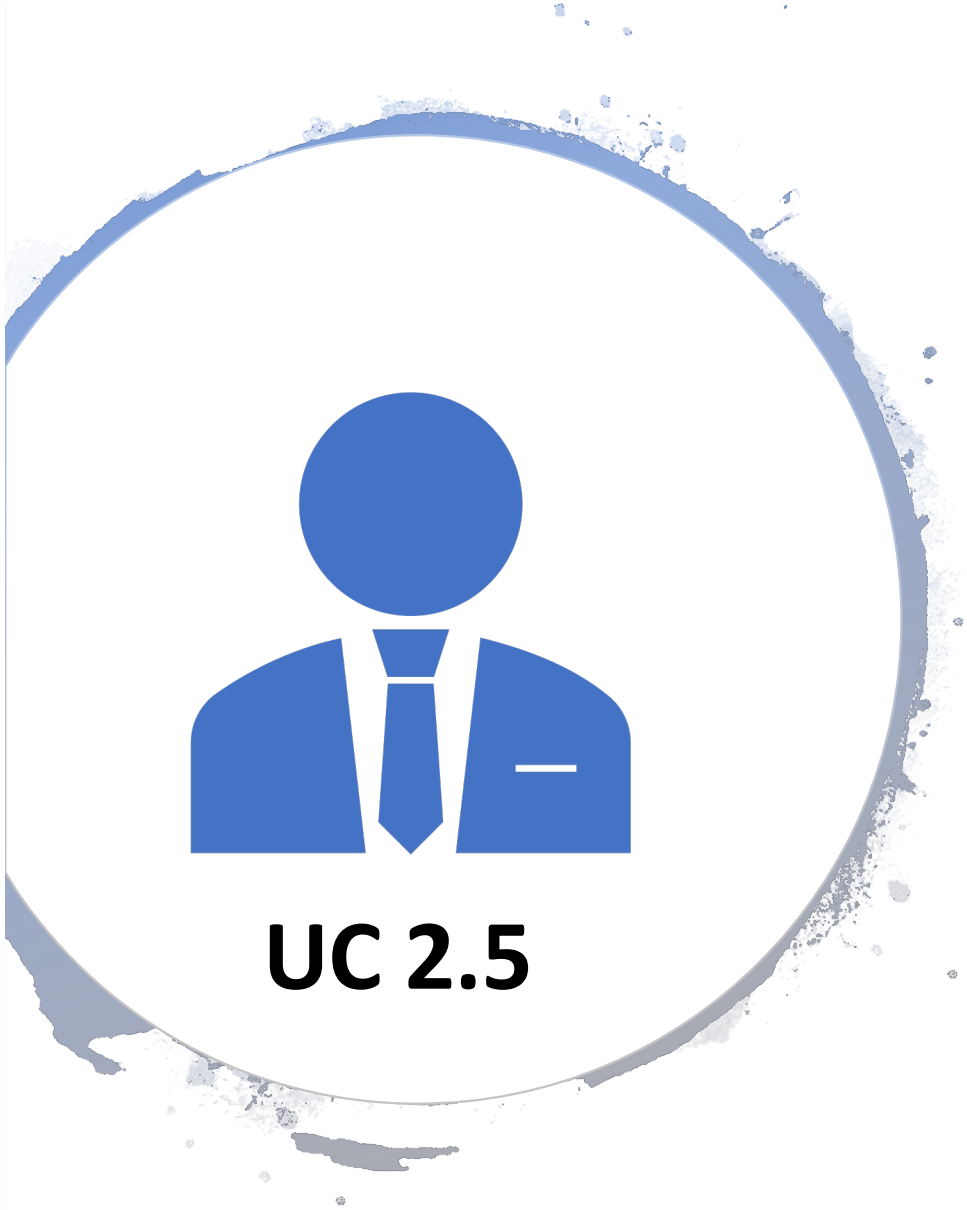
Java Streams Operations Quick Intro

BridgeLabz
Employability Delivered

- **forEach –** Like List forEach, the Stream forEach also loops over each element. The Stream forEach() is a **Terminal Operation**, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used.
- **map –** As the name indicates this operation maps each element into a new data and produces a New Stream. The map() is an **Intermediate Operation**, which means after this Operation further operations can be performed in the pipeline.
- **collect –** The collect function is one of the common ways to get stuff out of the stream once all the Intermediate Operations are done.
- **filter –** This operation produces a new stream that contains elements of the original stream that pass a given test (specified by a Predicate).
- **toArray –** As the name indicates if we need to get an array out of the stream, we can simply use toArray()

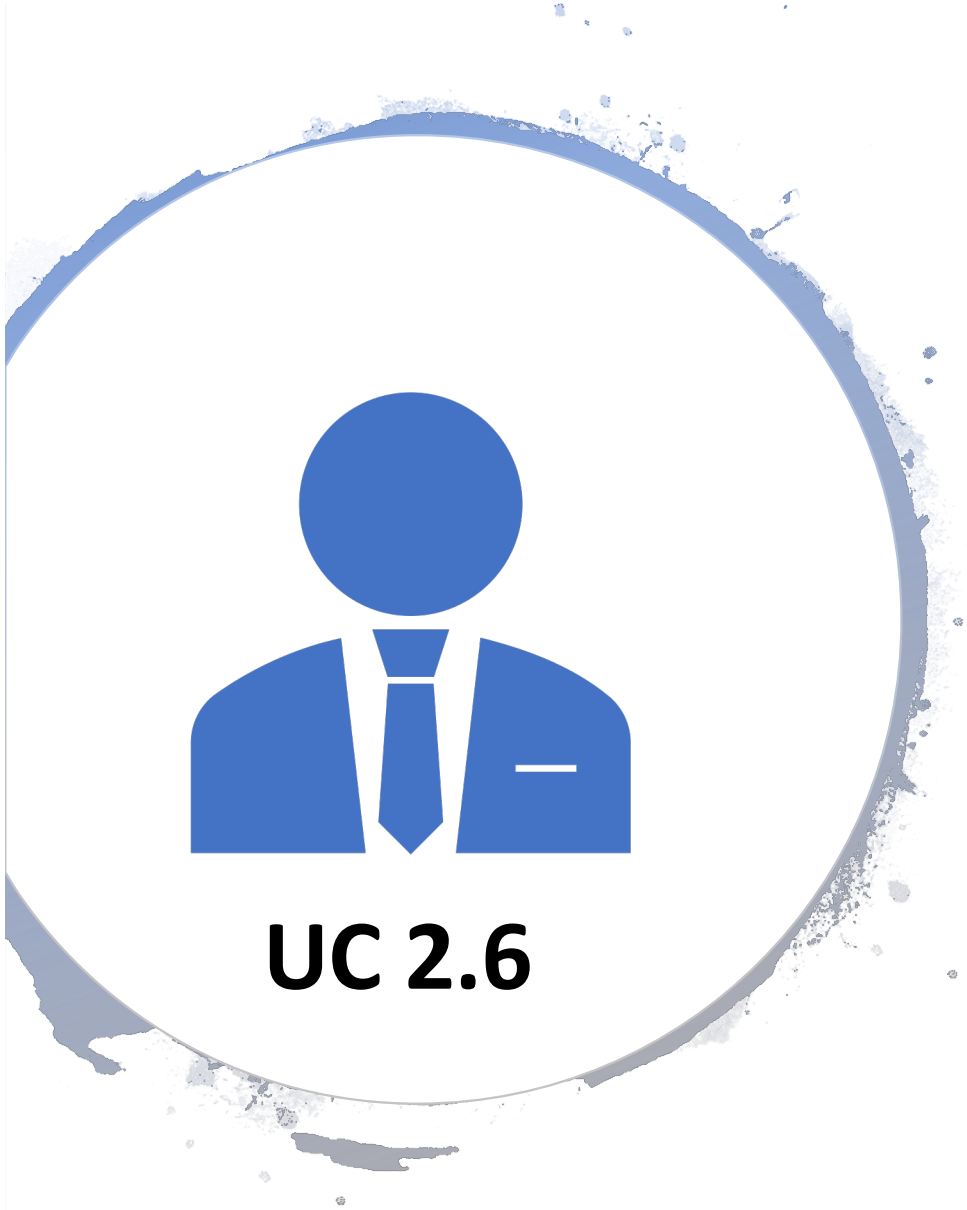
Java Streams Operations Sample


```java
//Method 5: Implicit Lambda Function
myList.forEach(n-> {
    System.out.println("Mth5: forEach Lambda impl Value::" + n);
});
myList.stream().forEach( n -> {
    System.out.println("Mth5: Stream forEach Value::" + n);
});

//Method 6: Implicit Lambda Function to print double value
Function<Integer,Double> toDoubleFunction = Integer::doubleValue;
myList.forEach(n-> {
    System.out.println("Mth5: forEach Lambda double Value::" +
                       toDoubleFunction.apply(n));
});
myList.stream().map(toDoubleFunction).forEach(System.out::println);
List<Double> doubleList = myList.stream()
                          .map(toDoubleFunction)
                          .collect(Collectors.toList());
System.out.println("Printing Double List: " + doubleList);

//Method 7: Implicit Lambda Function to check even
Predicate<Integer> isEvenFunction = n -> n%2 == 0;
myList.forEach(n-> {
    System.out.println("Mth5: forEach value of: "+n+
                       " check for Even: " + isEvenFunction.test(n));
});
List<Integer> evenList = myList.stream()
                         .filter(isEvenFunction)
                         .collect(Collectors.toList());
System.out.println("Printing Even List: "+evenList);
```



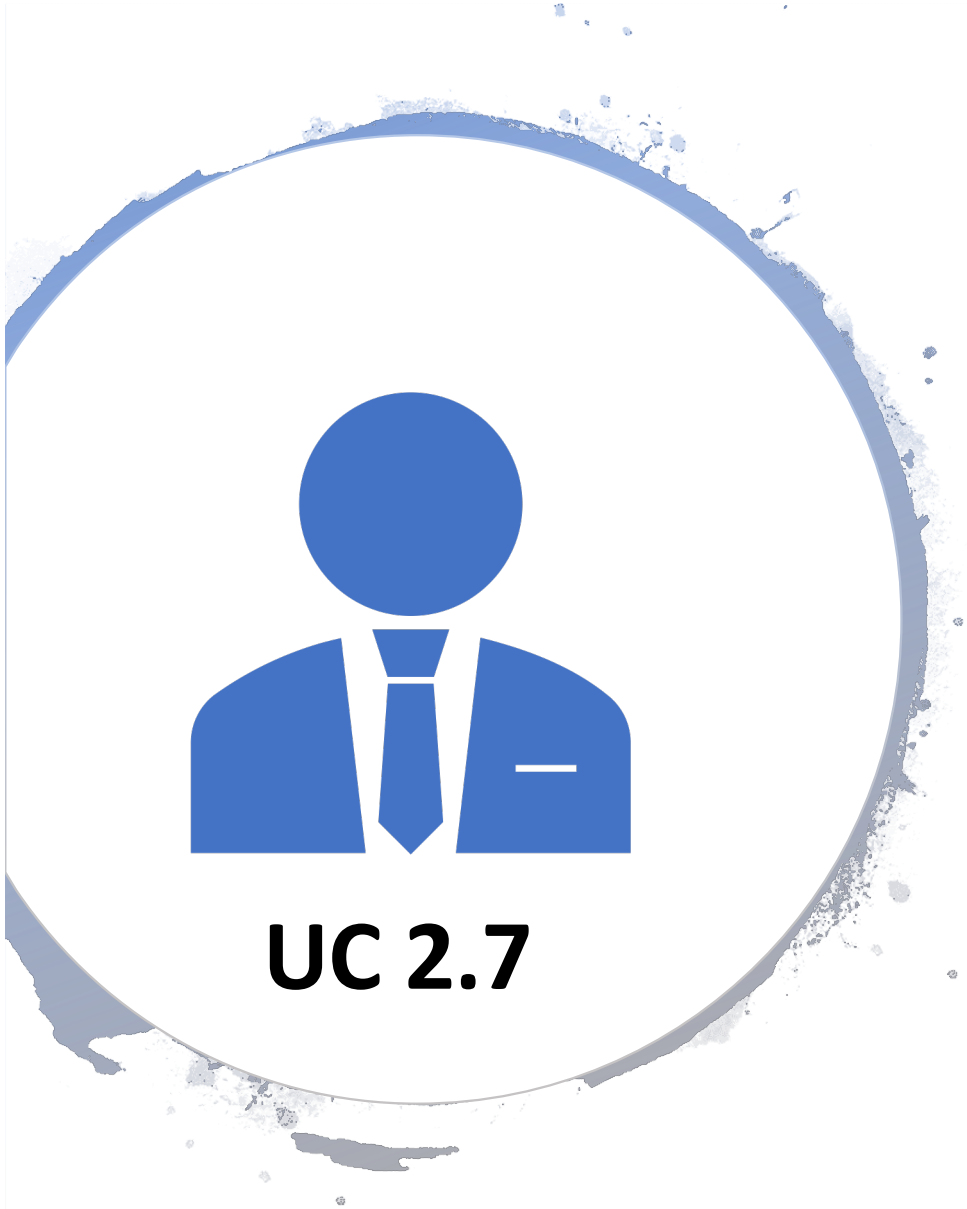
Ability to peak and show the first even number in the number stream

- Use stream.findFirst function



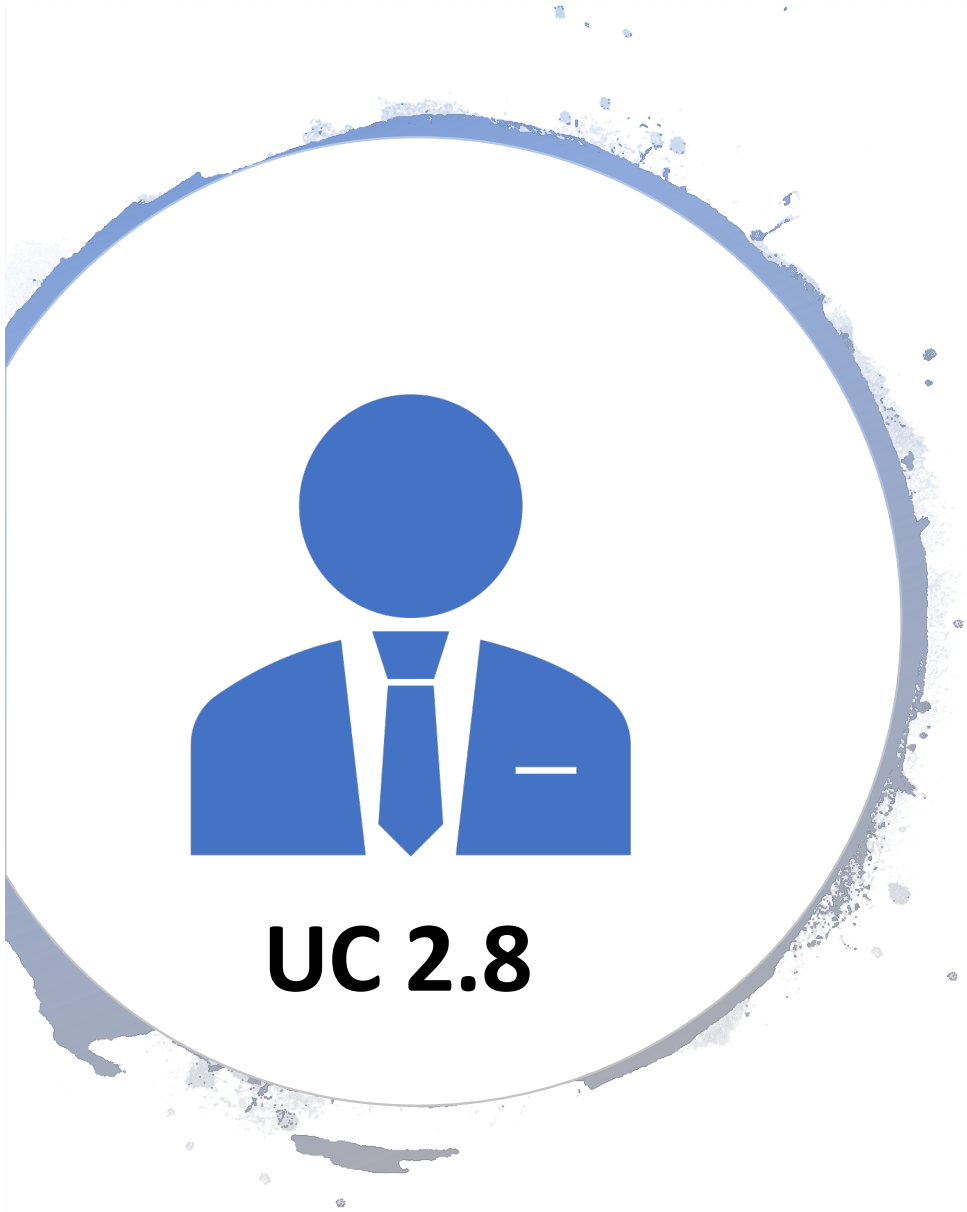
Ability to find min and max even number in the number stream

- Use stream.min and max function



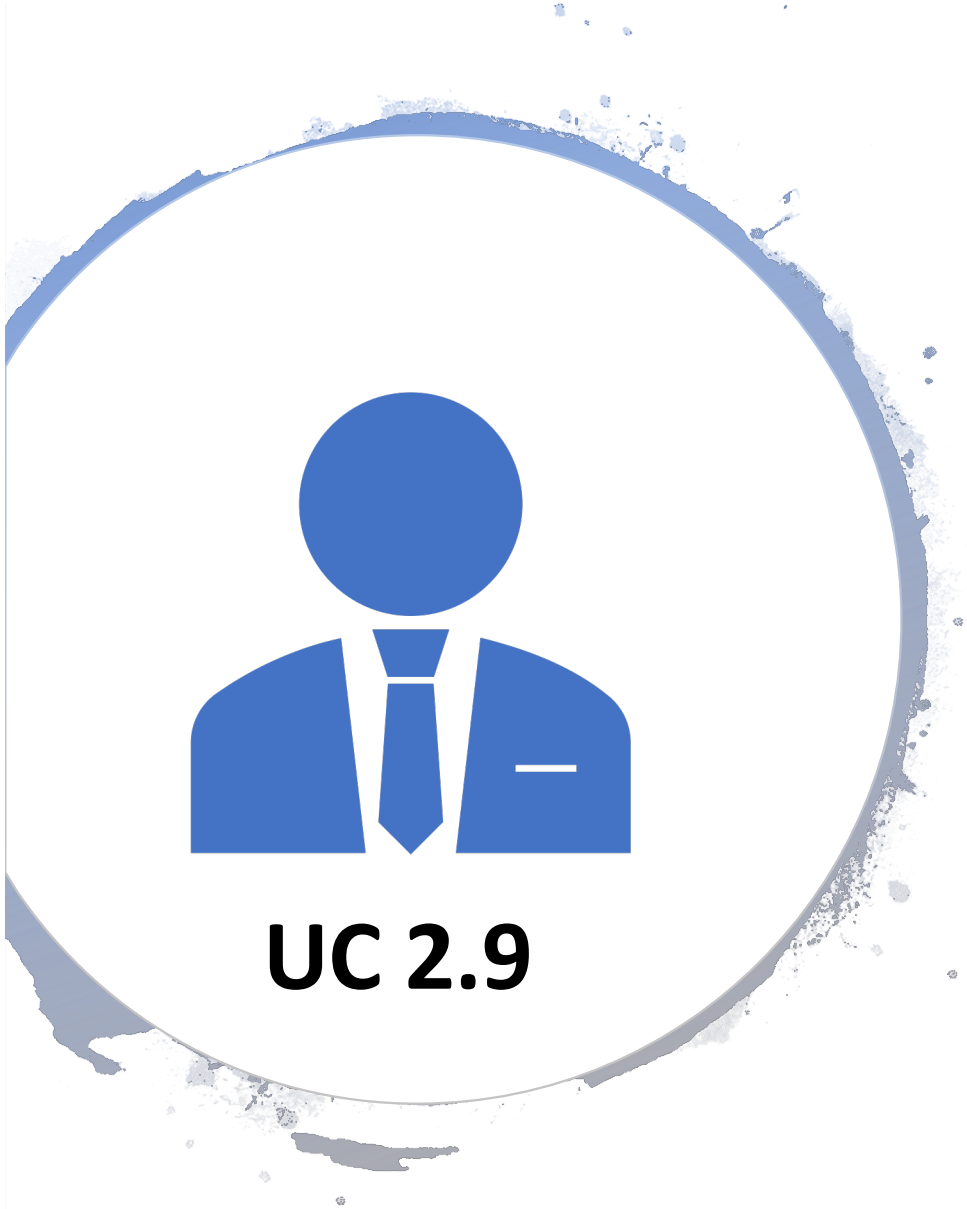
Ability to find the sum and the average in the number stream

- Use stream.min and max function



Ability to find if all the number or atleast one number is even in the number stream

- Use stream.allMatch and anyMatch function



Ability to sort the number stream in Ascending Order

- Use stream.sort function

Java Streams Operations Sample


```java
// Method 8: Listing the first Even
Integer first = myNumberList.stream().filter(isEvenFunction)
                .peek(n -> System.out.println("Peak Even Number: "+n))
                .findFirst()
                .orElse( other: null);
System.out.println("Mth8: First Even: "+first);

// Method 9: Minimum Even Numbers
Integer min = myNumberList.stream().filter(isEvenFunction)
              .min((n1, n2) -> n1-n2).orElse( other: null);
System.out.println("Mth9: Min Even: "+min);

// Method 10: Maximumj Even Numbers
Integer max = myNumberList.stream().filter(isEvenFunction)
              .max(Comparator.comparing(Integer::intValue))
              .orElse( other: null);
System.out.println("Mth10: Max Even: "+max);

// Method 11: Sum, Count and Average of numbers
Integer sum = myNumberList.stream().reduce( identity: 0, Integer::sum);
long count = myNumberList.stream().count();
System.out.println("Mth11: Avg of "+sum+"/"+count+" = "+sum/count);

// Method 12: Checking all even, single even or none are divisible by 6
boolean allEven = myNumberList.stream().allMatch(isEvenFunction);
boolean oneEven = myNumberList.stream().anyMatch(isEvenFunction);
boolean noneMultOfSix = myNumberList.stream().noneMatch(i -> i > 0 && i % 6 == 0);
System.out.println("Mth12: allEven: "+allEven+" oneEven: "+oneEven+
                   " noneMultOfSix: "+noneMultOfSix);

// Method 13: Sort the number in Ascending Order
List<Integer> sortedList = myNumberList.stream()
                           .sorted((n1, n2) -> n2.compareTo(n1))
                           .collect(Collectors.toList());
System.out.println("Mth13: SortedList: "+sortedList);
```

Java Streams Operations Q&A

BridgeLabz
Employability Delivered

Answer is the following operation is Intermediate or Terminal Operations

- collect
- filter
- map
- forEach
- toArray
- distinct
- findFirst
- flatMap
- peek
- sorted
- min and max
- allMatch, anyMatch and noneMatch

Method Types and Pipelines

- As seen Stream operations are divided into intermediate and terminal operations.
- Intermediate operations such as filter(), map(), etc return a new stream on which further processing can be done.
- Terminal operations, such as forEach(), mark the stream as consumed, after which point it can no longer be used further.
- A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

Lazy Evaluation

BridgeLabz
Employability Delivered

- One of the most important characteristics of streams is that they allow for significant optimizations through lazy evaluations.
- Computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.
- All intermediate operations are lazy, so they’re not executed until a result of a processing is actually needed.
- This concept gives rise to significant programming benefits. 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.

Java Stream vs Collection

BridgeLabz
Employability Delivered

- All of us have watched online videos on Youtube. When we start watching a video, a small portion of the file is first loaded into the computer and starts playing. we don't need to download the complete video before we start playing it. This is called streaming.
- At a very high level, we can think of that small portions of the video file as a stream, and the whole video as a Collection.
- A Collection is an in-memory data structure, which holds all the values that the data structure currently has. Every element in the Collection has to be computed before it can be added to the Collection.
- While a Stream is a conceptually a pipeline, in which elements are computed on demand.

BridgeLabz

Employability Delivered

Thank You