**INTERNITY FOUNDATION**

**TASK-8**

**Submitted By:**

**Amisha Singhal**

**Java Batch**

**Java Stream API  
Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method**

**Ans-** Peek is an intermediate operation that returns a stream consisting of the elements of the original stream, and performing the provided action on each element.

**Stream<T> peek(Consumer<? super T> action)**

This method exists mainly to support debugging, where we want to see the elements as they flow past a certain point in a pipeline:

**Stream.of("the", "good", "bad", "ugly")**

**.filter(e -> e.length() > 3)**

**.peek(e -> System.out.println("Filtered value: " + e))**

**.map(String::toUpperCase)**

**.peek(e -> System.out.println("Mapped value: " + e))**

**.collect(Collectors.toList());**

Map is an intermediate operation that returns a stream consisting of the results of applying the given function to the elements of this stream. Since we're talking about a Function, it's possible to convert the items in the stream to other objects. In other words, for each item you create a new object based on that item.

Eg-To convert all strings in the stream to uppercase:

**Stream.of("a", "b", "c", "d").map(s -> s.toUpperCase()).forEach(s -> System.out.print(s));**

map() includes primitive versions:

**IntStream mapToInt(ToIntFunction<? super T> mapper)**

**LongStream mapToLong(ToLongFunction<? super T> mapper)**

**DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)**

**Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch**

**Ans- findFirst():**For ordered streams you may wish to find the first element

Eg- given a list of integers, finds the first number that is divisible by 7:

**List<Integer> ints = List.of(1, 6, 22, 21, 35, 36);**

**Optional<Integer> result = ints.stream().filter(i -> i % 7 == 0).findFirst();**

**result.ifPresentOrElse(System.out::print, () -> System.out.print("No results found"));**

**findAny():**The findAny() method returns an arbitrary element of the current stream.

IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);  
stream.findFirst()  
    .ifPresent(System.out::println); // 1  
  
IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);  
stream2.findAny()  
    .ifPresent(System.out::println); // 1

**anyMatch()** returns true if any of the elements in a stream matches the given predicate:

**IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);**

**System.out.println(**

**stream.anyMatch(i -> i%3 == 0)**

**); // true**

**allMatch()** returns true only if **ALL** elements in the stream match the given predicate:

**IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);**

**System.out.println(**

**stream.allMatch(i -> i > 0)**

**); // true**

**IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);**

**System.out.println(**

**stream2.allMatch(i -> i%3 == 0)**

**); // false**

**noneMatch()** is the opposite of allMatch(), it returns true if **NONE** of the elements in the stream match the given predicate:

**IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);**

**System.out.println(**

**stream.noneMatch(i -> i > 0)**

**); // false**

**IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);**

**System.out.println(**

**stream2.noneMatch(i -> i%3 == 0)**

**); // false**

**IntStream stream3 = IntStream.of(1, 2, 3, 4, 5, 6, 7);**

**System.out.println(**

**stream3.noneMatch(i -> i > 10)**

**Develop code that uses the Optional class**

**Ans-** Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

**Eg-**

import java.util.Optional;

public class OptionalExample {

    public static void main(String[] args) {

        String[] str = new String[10];

        str[5] = "JAVA OPTIONAL CLASS EXAMPLE";// Setting value for 5th index

        Optional<String> checkNull = Optional.ofNullable(str[5]);

        if(checkNull.isPresent()){  // It Checks, value is present or not

            String lowercaseString = str[5].toLowerCase();

            System.out.print(lowercaseString);

        }else

            System.out.println("String value is not present");

    }

}

**OUTPUT-**

JAVA OPTIONAL CLASS EXAMPLE

**Develop code that uses Stream data methods and calculation methods**

**Ans-  
Sort a collection using Stream API**

**Ans-Stream sorted()** returns a stream consisting of the elements of this stream, sorted according to natural order. For ordered streams, the sort method is stable but for unordered streams, no stability is guaranteed. It is a **stateful intermediate operation** i.e, it may incorporate state from previously seen elements when processing new elements.

**Syntax :**

Stream<**T**> sorted()

Where, Stream is an interface and **T**

is the type of stream elements.

import java.util.\*;

class Demo {

    public static void main(String[] args)

    {

        // Creating a list of integers

        List<Integer> list = Arrays.asList(-9, -18, 0, 25, 4);

        System.out.println("The sorted stream is : ");

        // displaying the stream with elements

        // sorted in natural order

        list.stream().sorted().forEach(System.out::println);

    }

}

**Output:**

The sorted stream is:

-18

-9

0

4

25

**Save results to a collection using the collect method and group/partition data using the Collectors class**

**Ans-  
Use flatMap() methods in the Stream API**

**Ans- flatMap()** is the combination of a map and a flat operation i.e, it applies a function to elements as well as flatten them.The **flatMap()** operation has the effect of applying a one-to-many transformation to the elements of the stream and then flattening the resulting elements into a new stream.

**Syntax :**

**<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**

where, R is the element type of the new stream.

Stream is an interface and T is the type

of stream elements. mapper is a stateless function

which is applied to each element and the function

returns the new stream.

**flatMap() = Flattening + map()**

**Eg-**

public class Main

{

    public static void main(String[] args)

    {

        List<Integer> list1 = Arrays.asList(1,2,3);

        List<Integer> list2 = Arrays.asList(4,5,6);

        List<Integer> list3 = Arrays.asList(7,8,9);

        List<List<Integer>> listOfLists = Arrays.asList(list1, list2, list3);

        List<Integer> listOfAllIntegers = listOfLists.stream()

                            .flatMap(x -> x.stream())

                            .collect(Collectors.toList());

        System.out.println(listOfAllIntegers);

    }}

OUTPUT: **[1, 2, 3, 4, 5, 6, 7, 8, 9]**