**Creating Methods**

In this exercise, we’ll create four methods that will use the functional programming intermediate operations like filter, map, distinct and sorted

**Method 1:** This method will take as input a list of integers and will return a string consisting of even numbers more than four from the input list, each number separated by a comma.

**Step 1:** We’ll begin by creating a method named getAllEvensMoreThanFour that must accept a List<Integer> of nos as an input and returns back a string consisting of all the even numbers more than four from the input list, separated by a comma in the final string. Name the method getAllEvensMoreThanFour and this would be a static method with public access. Create this method below the main method in the StreamIntermediateOpsRunner class.  
  
public static String getAllEvensMoreThanFour(List<Integer> nos) {

}

**Step 2:**  While looping over the numbers from the input list, we will be selectively picking only even numbers more than four from the list and concatenating them in the final String (along with the comma!), that needs to be returned from the method. For this purpose, we will create a StringBuilder instance

public static String getAllEvensMoreThanFour(List<Integer> nos) {

final StringBuilder builder = new StringBuilder();

}

**Step 3:** Now, we selectively filter every number from the input list, using the functional programming construct `filter` that is available to be called on the underlying Stream object. It takes a lambda expression as the only argument. We write the filtering condition of the number being even and more than four, in the lambda expression

public static String getAllEvensMoreThanFour(List<Integer> nos) {

final StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 == 0 && no > 4);

}

**Step 4:** Next, we chain the output of the filter operation with `forEach` functional programming construct that will loop through every element in the output of the filter operation and concatenate them using the StringBuilder instance. Also don't forget to concatenate the comma.

public static String getAllEvensMoreThanFour(List<Integer> nos) {

final StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 == 0 && no > 4).forEach(no -> {

builder.append(no);

builder.append(',');

});

}

**Step 5:** After having finished looping through each number in the filter output using `forEach`, check whether the StringBuilder is empty or not. It can be empty if the input list passed is empty or does not have even numbers more than four. If it is empty, directly return an empty string from here.

public static String getAllEvensMoreThanFour(List<Integer> nos) {

final StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 == 0 && no > 4).forEach(no -> {

builder.append(no);

builder.append(',');

});

if (builder.length() == 0) {

return "";

}

}

**Step 6:** Finally, if the StringBuilder is not empty, extract the entire concatenated String from the StringBuilder and yes, do not forget to remove the trailing comma using String manipulation.

public static String getAllEvensMoreThanFour(List<Integer> nos) {

final StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 == 0 && no > 4).forEach(no -> {

builder.append(no);

builder.append(',');

});

if (builder.length() == 0) {

return "";

}

return builder.toString().substring(0, builder.length() - 1);

}

That’s it! You’ve just built a method that can take any input list of integers and returns a string of even numbers more than four using functional programming construct `filter`

**Method 2:** This method will take as input a list of integers and will return a string consisting of all numbers from the input list, but each number deducted by one and separated by a comma.

**Step 1:** We’ll begin by creating a method named getAllDeductedByOne that must accept a List<Integer> of nos as an input and returns back a string consisting of all the numbers from the input list, with each number deducted by one and separated by a comma in the final string. Name the method getAllDeductedByOne and this would be a static method with public access. Create this method below the main method in the StreamIntermediateOpsRunner class.  
  
public static String getAllDeductedByOne(List<Integer> nos) {

}

**Step 2:**  While looping over the numbers from the input list, we will be deducting 1 from every element from the list and concatenating them in the final String (along with the comma!), that needs to be returned from the method. For this purpose, we will create a StringBuilder instance

public static String getAllDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

}

**Step 3:** Now, we deduct one from every element from the input list, using the functional programming construct `map` that is available to be called on the underlying Stream object. It takes a lambda expression as the only argument. We write the mapping operation of deduction by one, to be applied to every element, in the lambda expression. We also check if the number is 0, we do not deduct, else we deduct by 1, as part of the mapping operation.

public static String getAllDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().map(no -> no > 0 ? no - 1 : no);

}

**Step 4:** Next, we chain the output of the map operation with `forEach` functional programming construct that will loop through every element in the output of the map operation and concatenate them using the StringBuilder instance. Also don't forget to concatenate the comma.

public static String getAllDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().map(no -> no > 0 ? no - 1 : no).forEach(no -> {

builder.append(no);

builder.append(',');

});

}

**Step 5:** After having finished looping through each number in the map output using `forEach`, check whether the StringBuilder is empty or not. It can be empty if the input list passed is empty. If it is empty, directly return an empty string from here.

public static String getAllDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().map(no -> no > 0 ? no - 1 : no).forEach(no -> {

builder.append(no);

builder.append(',');

});

if (builder.length() == 0) {

return "";

}

}

**Step 6:** Finally, if the StringBuilder is not empty, extract the entire concatenated String from the StringBuilder and yes, do not forget to remove the trailing comma using String manipulation.

public static String getAllDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().map(no -> no > 0 ? no - 1 : no).forEach(no -> {

builder.append(no);

builder.append(',');

});

if (builder.length() == 0) {

return "";

}

return builder.toString().substring(0, builder.length() - 1);

}

That’s it! You’ve just built a method that can take any input list of integers and returns a string of numbers, with each number from the list deducted by one, using functional programming construct `map`

**Method 3:** This method will take as input a list of integers and will return a string consisting of all odd numbers from the input list, but each number deducted by one and separated by a comma.

**Step 1:** We’ll begin by creating a method named getAllOddsDeductedByOne that must accept a List<Integer> of nos as an input and returns back a string consisting of all the odd numbers from the input list, with each number deducted by one and separated by a comma in the final string. Name the method getAllOddsDeductedByOne and this would be a static method with public access. Create this method below the main method in the StreamIntermediateOpsRunner class.  
  
public static String getAllOddsDeductedByOne(List<Integer> nos) {

}

**Step 2:**  While looping over the numbers from the input list, we will be deducting 1 from odd elements from the list and concatenating them in the final String (along with the comma!), that needs to be returned from the method. For this purpose, we will create a StringBuilder instance

public static String getAllOddsDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

}

**Step 3:** Now, we filter only odd elements from the input list, using the functional programming construct `filter` that is available to be called on the underlying Stream object. It takes a lambda expression as the only argument.

public static String getAllOddsDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 != 0);

}

**Step 4:** Next, we chain the output of the previous filter operation with the map functional programming construct. It takes a lambda expression as an argument. The mapping operation of deducting 1 from every number from the output of the filter operation, is done inside the lambda expression passed to the map function

public static String getAllOddsDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 != 0).map(no -> no - 1);

}

**Step 5:** Next, we chain the output of the map operation with `forEach` functional programming construct that will loop through every element in the output of the map operation and concatenate them using the StringBuilder instance. Also don't forget to concatenate the comma.

public static String getAllOddsDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 != 0).map(no -> no - 1).forEach(no -> {

builder.append(no);

builder.append(',');

});

}

**Step 6:** After having finished looping through each number in the map output using `forEach`, check whether the StringBuilder is empty or not. It can be empty if the input list passed is empty or there were only even elements in the input list. If it is empty, directly return an empty string from here.

public static String getAllOddsDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 != 0).map(no -> no - 1).forEach(no -> {

builder.append(no);

builder.append(',');

});

if (builder.length() == 0) {

return "";

}

}

**Step 7:** Finally, if the StringBuilder is not empty, extract the entire concatenated String from the StringBuilder and yes, do not forget to remove the trailing comma using String manipulation.

public static String getAllOddsDeductedByOne(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().filter(no -> no % 2 != 0).map(no -> no - 1).forEach(no -> {

builder.append(no);

builder.append(',');

});

if (builder.length() == 0) {

return "";

}

return builder.toString().substring(0, builder.length() - 1);

}

That’s it! You’ve just built a method that can take any input list of integers and returns a string of numbers, with odd number from the list deducted by one, using functional programming construct `map` and `filter`

**Method 4:** This method will take as input a list of integers and will return a string consisting of all distinct elements from the input list, sorted in the descending order and separated by comma in the final string

**Step 1:** We’ll begin by creating a method named getDistinctDescendingOrder that must accept a List<Integer> of nos as an input and returns back a string consisting of all the distinct numbers from the input list, sorted in the descending order and separated by a comma in the final string. Name the method getDistinctDescendingOrder and this would be a static method with public access. Create this method below the main method in the StreamIntermediateOpsRunner class.  
  
public static String getDistinctDecendingOrder(List<Integer> nos) {

}

**Step 2:**  While looping over the numbers from the input list, we will be finding distinct elements, sorting them and concatenating them in the final String (along with the comma!), that needs to be returned from the method. For this purpose, we will create a StringBuilder instance

public static String getDistinctDecendingOrder(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

}

**Step 3:** Now, we find distinct elements from the input list, using the functional programming construct `distinct` that is available to be called on the underlying Stream object.

public static String getDistinctDecendingOrder(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().distinct();

}

**Step 4:** Next, we chain the output of the previous distinct operation with the sorted functional programming construct. It takes a lambda expression as an argument. The lambda expression is a simple Integer value comparator implemented in the descending order.

public static String getDistinctDecendingOrder(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().distinct().sorted((n1, n2) -> n2.compareTo(n1));

}

**Step 5:** Next, we chain the output of the sorted operation with `forEach` functional programming construct that will loop through every element in the output of the sorted operation and concatenate them using the StringBuilder instance. Also don't forget to concatenate the comma.

public static String getDistinctDecendingOrder(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().distinct().sorted((n1, n2) -> n2.compareTo(n1)).forEach(no -> {

builder.append(no);

builder.append(',');

});

}

**Step 6:** After having finished looping through each number in the map output using `forEach`, check whether the StringBuilder is empty or not. It can be empty if the input list passed is empty or there were only even elements in the input list. If it is empty, directly return an empty string from here.

public static String getDistinctDecendingOrder(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().distinct().sorted((n1, n2) -> n2.compareTo(n1)).forEach(no -> {

builder.append(no);

builder.append(',');

});

if (builder.length() == 0) {

return "";

}

}

**Step 7:** Finally, if the StringBuilder is not empty, extract the entire concatenated String from the StringBuilder and yes, do not forget to remove the trailing comma using String manipulation.

public static String getDistinctDecendingOrder(List<Integer> nos) {

StringBuilder builder = new StringBuilder();

nos.stream().distinct().sorted((n1, n2) -> n2.compareTo(n1)).forEach(no -> {

builder.append(no);

builder.append(',');

});

if (builder.length() == 0) {

return "";

}

return builder.toString().substring(0, builder.length() - 1);

}

That’s it! You’ve just built a method that can take any input list of integers and returns a string of numbers, with distinct numbers from the list, sorted in the descending order using functional programming construct `distinct` and `sorted`