**Lambda Expressions and Java Stream API Exercises**

1. Write the following methods that *return a lambda expression* performing a specified action:

PerformOperation isOdd(): The lambda expression must return  if a number is odd or  if it is even.

PerformOperation isPrime(): The lambda expression must return  if a number is prime or  if it is composite.

PerformOperation isPalindrome(): The lambda expression must return  if a number is a palindrome or  if it is not.

|  |
| --- |
|  |
|  | import java.io.\*;  import java.util.\*; |
|  | public interface PerformOperation { |
|  | boolean check(int a); |
|  | } |
|  | public class MyMath { |
|  | public static boolean checker(PerformOperation p, int num) { |
|  | return p.check(num); |
|  | } |
|  | // Write your code here |
|  |  |
|  | public static PerformOperation isOdd(){ |
|  | return a -> (a % 2 != 0)? true:false; |
|  | } |
|  |  |
|  | public static PerformOperation isPrime(){ |
|  | return a ->{ for (int i = 2; i \* i <= a; i++) { |
|  | if (a % i == 0) { |
|  | return false; |
|  | } |
|  | } |
|  | return true; |
|  | }; |
|  | } |
|  |  |
|  | public static PerformOperation isPalindrome() { |
|  | return a -> a == Integer.parseInt(new StringBuilder(String.valueOf(a)).reverse().toString()) ? true : false; |
|  | } |
|  | } |
|  | public class Solution { |
|  |  |
|  | public static void main(String[] args) throws IOException { // don’t throw the exception like these, instead use try .. catch |
|  | MyMath ob = new MyMath(); |
|  | BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); |
|  | int T = Integer.parseInt(br.readLine()); |
|  | PerformOperation op; |
|  | boolean ret = false; |
|  | String ans = null; |
|  | while (T--> 0) { |
|  | String s = br.readLine().trim(); |
|  | StringTokenizer st = new StringTokenizer(s); |
|  | int ch = Integer.parseInt(st.nextToken()); |
|  | int num = Integer.parseInt(st.nextToken()); |
|  | if (ch == 1) { |
|  | op = ob.isOdd(); |
|  | ret = ob.checker(op, num); |
|  | ans = (ret) ? "ODD" : "EVEN"; |
|  | } else if (ch == 2) { |
|  | op = ob.isPrime(); |
|  | ret = ob.checker(op, num); |
|  | ans = (ret) ? "PRIME" : "COMPOSITE"; |
|  | } else if (ch == 3) { |
|  | op = ob.isPalindrome(); |
|  | ret = ob.checker(op, num); |
|  | ans = (ret) ? "PALINDROME" : "NOT PALINDROME"; |
|  |  |
|  | } |
|  | System.out.println(ans); |
|  | } |
|  | } |
|  | } |

1. Use lambda expression to iterate over a List and perform some action on list items

List<String> list = **new** ArrayList();

list.add("1");

list.add("2");

list.forEach( p -> { System.out.println(p); } );

1. Write a method that converts all strings in a list to their upper case.

public List<String> upperCase(List<String> list) {

// your code goes here

}

Solution:

return list.stream()

.map(String::toUpperCase)

.collect(Collectors.toList());

1. Write a method that returns the average of a list of integers.

public Double average(List<Integer> list) {

// your code goes here

}

Solution:

return list.stream()

.mapToInt(i -> i)

.average()

.getAsDouble();

1. Given a list of strings, write a method that returns a list of all strings that start with the letter 'a' (lower case) and have exactly 3 letters.

public List<String> search(List<String> list) {

// your code goes here

}

Solution:

return list.stream()

.filter(s -> s.startsWith("a"))

.filter(s -> s.length() == 3)

.collect(Collectors.toList());

1. Write a method that returns a comma-separated string based on a given list of integers. Each element should be preceded by the letter 'e' if the number is even, and preceded by the letter 'o' if the number is odd. For example, if the input list is (3,44), the output should be 'o3,e44'.

public String getString(List<Integer> list) {

// your code goes here

}

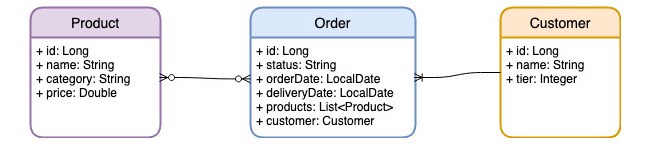
Solution:

return list.stream()

.map(i -> i % 2 == 0 ? "e" + i : "o" + i)

.collect(joining(","));

1. The exercises are based on a data model — customer, order and product. Refer to the entity relationship diagram below, customers can place multiple orders and so it is a one-to-many relationship while the relationship between products and orders is many-to-many



Tier one customers are **the lowest level of customers**. When companies classify someone as a tier one customer, then they consider the person a low-grade customer, who can easily cost the company as much as the company can make off of the customer.

## tier one(1)

Tier one customers are the lowest level of customers. When companies classify someone as a tier one customer, then they consider the person a low-grade customer, who can easily cost the company as much as the company can make off of the customer. Tier one customers are often customers who repeatedly call in with fake complaints in an attempt to get discounts or free stuff from the company or harass the company in other ways.

## tier two(2)

Tier two customers are the bulk of most companies' business. Businesses say that the hardest sale is to first-time customers, which is why tier two customers are so important. Tier two customers are customers who return to time and again to make both large and small purchases. Businesses sometimes name tier two customers the loyal customers, and businesses usually spend most of their time and assets trying to appeal to this group of customers.

## tier three(3)

Tier three customers usually make up a very small percentage of a business’s customer base as these are the lifetime customers: customers who are not only loyal, but have no problem spending or investing in your company. Tier three customers are often eager to expand their existing services, and buy brand new products from your company.

Write down Product, Order, and Customer classes and do the following exercises.

## Exercise 1 — Obtain a list of products belongs to category “Books” with price > 100

Write a class ProductRepository which will be having a code like shown below

List<Product> products = new ArrayList<>();

products.add(new Product(101, “A text book”, “Books”, 250.00);

“

“

“

“

“

“

“

“

“

“

And write a method findAll() which returns the products list.

|  |
| --- |
| List<Product> result = productRepo.findAll() |
|  | .stream() |
|  | .filter(p -> p.getCategory().equalsIgnoreCase("Books")) |
|  | .filter(p -> p.getPrice() > 100) |
|  | .collect(Collectors.toList()); |

## Exercise 2 — Obtain a list of order with products belong to category “Baby”

|  |
| --- |
| List<Order> result = orderRepo.findAll() |
|  | .stream() |
|  | .filter(o -> |
|  | o.getProducts() |
|  | .stream() |
|  | .anyMatch(p -> p.getCategory().equalsIgnoreCase("Baby")) |
|  | ) |
|  | .collect(Collectors.toList()); |

## Exercise 3 — Obtain a list of product with category = “Toys” and then apply 10% discount

|  |
| --- |
| List<Product> result = productRepo.findAll() |
|  | .stream() |
|  | .filter(p -> p.getCategory().equalsIgnoreCase("Toys")) |
|  | .map(p -> p.withPrice(p.getPrice() \* 0.9)) |
|  | .collect(Collectors.toList()); |

# Exercise 4 — Obtain a list of products ordered by customer of tier 2 between 01-Feb-2021 and 01-Apr-2021

|  |
| --- |
| List<Product> result = orderRepo.findAll() |
|  | .stream() |
|  | .filter(o -> o.getCustomer().getTier() == 2) |
|  | .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 2, 1)) >= 0) |
|  | .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 4, 1)) <= 0) |
|  | .flatMap(o -> o.getProducts().stream()) |
|  | .distinct() |
|  | .collect(Collectors.toList()); |

# Exercise 5 — Get the cheapest products of “Books” category

|  |
| --- |
| Optional<Product> result = productRepo.findAll() |
|  | .stream() |
|  | .filter(p -> p.getCategory().equalsIgnoreCase("Books")) |
|  | .sorted(Comparator.comparing(Product::getPrice)) |
|  | .findFirst(); |

## Exercise 6 — Get the 3 most recent placed order

|  |
| --- |
| List<Order> result = orderRepo.findAll() |
|  | .stream() |
|  | .sorted(Comparator.comparing(Order::getOrderDate).reversed()) |
|  | .limit(3) |
|  | .collect(Collectors.toList()); |

## Exercise 7 — Get a list of orders which were ordered on 15-Mar-2021, log the order records to the console and then return its product list

|  |
| --- |
| List<Product> result = orderRepo.findAll() |
|  | .stream() |
|  | .filter(o -> o.getOrderDate().isEqual(LocalDate.of(2021, 3, 15))) |
|  | .peek(o -> System.out.println(o.toString())) |
|  | .flatMap(o -> o.getProducts().stream()) |
|  | .distinct() |
|  | .collect(Collectors.toList()); |

## Exercise 8 — Calculate total lump sum of all orders placed in Feb 2021

|  |
| --- |
| Double result = orderRepo.findAll() |
|  | .stream() |
|  | .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 2, 1)) >= 0) |
|  | .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 3, 1)) < 0) |
|  | .flatMap(o -> o.getProducts().stream()) |
|  | .mapToDouble(p -> p.getPrice()) |
|  | .sum(); |

## Exercise 9 — Calculate order average payment placed on 14-Mar-2021

|  |
| --- |
| Double result = orderRepo.findAll() |
|  | .stream() |
|  | .filter(o -> o.getOrderDate().isEqual(LocalDate.of(2021, 3, 15))) |
|  | .flatMap(o -> o.getProducts().stream()) |
|  | .mapToDouble(p -> p.getPrice()) |
|  | .average().getAsDouble(); |

## Exercise 10 — Obtain a collection of statistic figures (i.e. sum, average, max, min, count) for all products of category “Books”

|  |
| --- |
| DoubleSummaryStatistics statistics = productRepo.findAll() |
|  | .stream() |
|  | .filter(p -> p.getCategory().equalsIgnoreCase("Books")) |
|  | .mapToDouble(p -> p.getPrice()) |
|  | .summaryStatistics(); |
|  |  |
|  | System.out.println(String.format("count = %1$d, average = %2$f, max = %3$f, min = %4$f, sum = %5$f", |
|  | statistics.getCount(), statistics.getAverage(), statistics.getMax(), statistics.getMin(), statistics.getSum()))); |

## Exercise 11 — Obtain a data map with order id and order’s product count

|  |
| --- |
| Map<Long, Integer> result = orderRepo.findAll() |
|  | .stream() |
|  | .collect( |
|  | Collectors.toMap( |
|  | order -> order.getId(), |
|  | order -> order.getProducts().size() |
|  | ) |
|  | ); |

## Exercise 12 — Produce a data map with order records grouped by customer

|  |
| --- |
| Map<Customer, List<Order>> result = orderRepo.findAll() |
|  | .stream() |
|  | .collect( |
|  | Collectors.groupingBy(Order::getCustomer) |
|  | ); |

## Exercise 13 — Produce a data map with order record and product total sum

|  |
| --- |
| Map<Order, Double> result = orderRepo.findAll() |
|  | .stream() |
|  | .collect( |
|  | Collectors.toMap( |
|  | Function.identity(), |
|  | order -> order.getProducts().stream() |
|  | .mapToDouble(p -> p.getPrice()).sum() |
|  | ) |
|  | ); |

## Exercise 14 — Obtain a data map with list of product name by category

|  |
| --- |
| Map<String, List<String>> result = productRepo.findAll() |
|  | .stream() |
|  | .collect( |
|  | Collectors.groupingBy( |
|  | Product::getCategory, |
|  | Collectors.mapping(product -> product.getName(), Collectors.toList())) |
|  | ); |

## Exercise 15 — Get the most expensive product by category

|  |
| --- |
| Map<String, Optional<Product>> result = productRepo.findAll() |
|  | .stream() |
|  | .collect( |
|  | Collectors.groupingBy( |
|  | Product::getCategory, |
|  | Collectors.maxBy(Comparator.comparing(Product::getPrice))) |