Sofia University
Department of Mathematics and Informatics

Course : OO Programming Java

Date: December 13, 2018

Student Name:

Lab No. 10a

Problem 1

Use the **`class Invoice`** provided in the **`Lab`** folder to create an array of **`Invoice`** objects. Use the sample data shown below.

Part number	Part description	Quantity	Price
83	Electric sander	7	57.98
24	Power saw	18	99.99
7	Sledge hammer	11	21.50
77	Hammer	76	11.99
39	Lawn mower	3	79.50
68	Screwdriver	106	6.99
56	Jig saw	21	11.00
3	Wrench	34	7.50

Class **`Invoice`** includes four properties- a **`PartNumber`** (type int), a **`PartDescription`** (type **`String`**), a **`Quantity`** of the item being purchased (type **`int`**) and a **`Price`** (type **`double`**). Perform the following queries on the array of **`Invoice`** objects and display the results:

a) Use lambdas and streams to sort the **`Invoice`** objects by **`PartDescription`**, then display the results.

b) Use lambdas and streams to sort the **`Invoice`** objects by **`Price`**, then display the results.

c) Use lambdas and streams to **`map`** each **`Invoice`** to its **`PartDescription`** and **`Quantity`**, sort the results by **`Quantity`**, then display the results.

d) Use lambdas and streams to **map** each **Invoice** to its **PartDescription** and the value of the Invoice (i.e., **Quantity * Price**). Order the results by **Invoice** value.

e) Modify Part (d) to select the **Invoice** values in the range **$200** to **$500**.

f) Group the **Invoice** values into two sets of Invoices- Invoices with values (**Quantity * Price**) below of equal to **$300** and Invoices with values above **$300**.

g) Find the invoice with the maximum quantity

Problem 2

Write a program that inputs a sentence from the user (assume no punctuation), then determines and displays the unique words in alphabetical order. Treat uppercase and lowercase letters the same.

Problem 3

Write a program that inserts **30** random letters into a **List<Character>**. Perform the following operations and display your results:

a) Sort the **List** in ascending order.

b) Sort the **List** in descending order.

c) Display the **List** in ascending order with duplicates removed. Write a program that inserts **30** random letters into a **List<Character>**.

Problem 4

Write a Stream application to roll a die 6,000,000 times and display a table with the frequencies each side of the die has occurred in that sequence as in the following sample output

```
Face   Frequency
1      999549
2      1001189
3      999596
4      999672
5      998597
6      1001397
```

Use grouping by the side number and count the occurrences in each group

Problem 5

The lambda you pass to a stream's reduce method should be *associative*- that is, regardless of the order in which its subexpressions are evaluated, the result should be the same. The underlined lambda expression in the following code is *not* associative.

```
import java.util.Arrays;
import java.util.stream.IntStream;

public class IntStreamOperations
{
   public static void main(String[] args)
   {
      int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

      // display original values
      System.out.print("Original values: ");
      IntStream.of(values)
               .forEach(value -> System.out.printf("%d ", value));
      System.out.println();

      // count, min, max, sum and average of the values
      System.out.printf("%nCount: %d%n",
         IntStream.of(values).count());
      System.out.printf("Min: %d%n",
         IntStream.of(values).min().getAsInt());
      System.out.printf("Max: %d%n",
         IntStream.of(values).max().getAsInt());
      System.out.printf("Sum: %d%n", IntStream.of(values).sum());
      System.out.printf("Average: %.2f%n",
         IntStream.of(values).average().getAsDouble());

      // sum of values with reduce method
      System.out.printf("%nSum via reduce method: %d%n",
         IntStream.of(values)
                  .reduce(0, (x, y) -> x + y));

      // sum of squares of values with reduce method
      System.out.printf("Sum of squares via reduce method: %d%n",
```

```
            IntStream.of(values).parallel()
                     .reduce(0, (x, y) -> x + y * y));

         // product of values with reduce method
         System.out.printf("Product via reduce method: %d%n",
            IntStream.of(values)
                     .reduce(1, (x, y) -> x * y));

         // even values displayed in sorted order
         System.out.printf("%nEven values displayed in sorted order: ");
         IntStream.of(values)
                  .filter(value -> value % 2 == 0)
                  .sorted()
                  .forEach(value -> System.out.printf("%d ", value));
         System.out.println();

         // odd values multiplied by 10 and displayed in sorted order
         System.out.printf(
            "Odd values multiplied by 10 displayed in sorted order: ");
         IntStream.of(values)
                  .filter(value -> value % 2 != 0)
                  .map(value -> value * 10)
                  .sorted()
                  .forEach(value -> System.out.printf("%d ", value));
         System.out.println();

         // sum range of integers from 1 to 10, exlusive
         System.out.printf("%nSum of integers from 1 to 9: %d%n",
            IntStream.range(1, 10).sum());

         // sum range of integers from 1 to 10, inclusive
         System.out.printf("Sum of integers from 1 to 10: %d%n",
            IntStream.rangeClosed(1, 10).sum());
      }
   } // end class IntStreamOperations
```

When you create parallel streams using the **`parallel()`** operation with that lambda, you might get incorrect results for the sum of the squares, depending on the order in which the subexpressions are evaluated. The proper way to implement the summation of the squares would be *first* to map each int value to the square of that value, *then* to reduce the stream to the sum of the squares. Modify the above program to implement the summation of squares with **`reduce()`** in this manner.

Problem 6

Assume the following classes are given

```
public class Employee {

    private Integer employeeNumber;
    private String employeeFirstName;
    private String employeeLastName;
    private LocalDate hireDate;

    //.. getter and setter methods, constructor and toString()
}

public class EmployeeTest {

    List<Employee> employees;

    @Before
    public void setup() {

        employees = new ArrayList<>();
        employees.add(new Employee(123, "Jack", "Johnson", LocalDate.of(1988,
Month.APRIL, 12)));
        employees.add(new Employee(345, "Cindy", "Bower", LocalDate.of(2011,
Month.DECEMBER, 15)));
        employees.add(new Employee(567, "Perry", "Node", LocalDate.of(2005,
Month.JUNE, 07)));
        employees.add(new Employee(467, "Pam", "Krauss", LocalDate.of(2005,
Month.JUNE, 07)));
        employees.add(new Employee(435, "Fred", "Shak", LocalDate.of(1988,
Month.APRIL, 17)));
        employees.add(new Employee(678, "Ann", "Lee", LocalDate.of(2007,
Month.APRIL, 12)));
    }

    //..
}
```

1. Write a method in class `EmployeeTest` that outputs the list `employees` sorted in ascending order of the `Employee number` using the Stream API.
2. Write a method in class `EmployeeTest` that outputs the list `employees` sorted in descending order of the `Employee hireDate` using the Stream API.
3. Write a method in class `EmployeeTest` that outputs the longest tenured employee the list `employees` using the Stream API. Let the method return a Optional<Employee> reference
4. Write a method in class `EmployeeTest` that outputs the list `employees` sorted in ascending order of the first name and descending order of the last name of the `Employee` using the Stream API.