Experiment#		Student ID	
Date		Student Name	[@KLWKS_bot] THANOS

9. Nested Classes, Functional interfaces, Lambda Expressions and Stream API

Aim/Objective: To implement the concepts of Lambda expression and Stream API to solve real world data through Collection classes.

Description: The student will understand the concepts of Nested classes, lambdas and stream api for efficient processing of data in a Collection.

Pre-Requisites: Classes and Objects

Tools: Eclipse IDE for Enterprise Java and Web Developers

Pre-Lab:

1) List some of the predefined functional interfaces available in java.util.function package and explain their uses.

1. **Predicate<T>** – Tests a condition on an input.

java Copy Edit

```
Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println(isEven.test(4)); // true
```

2. **Function<T, R>** – Transforms an input to an output.

java Copy Edit

```
Function<String, Integer> length = str -> str.length();
System.out.println(length.apply("Java")); // 4
```

3. **Consumer<T>** – Performs an action on an input.

java Copy Edit

```
Consumer<String> printer = System.out::println;
printer.accept("Hello!"); // Hello!
```

4. **Supplier<T>** – Provides a value without input.

```java
Supplier<Double> random = Math::random;
System.out.println(random.get());
```

5. **BiPredicate<T, U>** – Tests a condition on two inputs.

```java
BiPredicate<Integer, Integer> isGreater = (a, b) -> a > b;
```

6. **BiFunction<T, U, R>** – Takes two inputs and produces an output.

```java
BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;
```

7. **BiConsumer<T, U>** – Performs an action on two inputs.

```java
BiConsumer<String, Integer> display = (name, age) ->
    System.out.println(name + " is " + age);
```

8. **UnaryOperator<T>** – A function where input and output are the same type.

```java
UnaryOperator<Integer> square = n -> n * n;
```

9. **BinaryOperator<T>** – Like `BiFunction`, but input and output are the same type.

```java
BinaryOperator<Integer> multiply = (a, b) -> a * b;
```

Write a lambda expression to sort a list of strings in descending order.

```
import java.util.Arrays;
import java.util.List;

public class Main {
   public static void main(String[] args) {
      List<String> strings = Arrays.asList("Apple", "Orange", "Banana", "Grape");
      strings.sort((s1, s2) -> s2.compareTo(s1));
      System.out.println(strings);
   }
}
```

OUTPUT

```
[Orange, Grape, Banana, Apple]
```

2) Write a stream pipeline that filters a list of integers to only even numbers, doubles them, and then collects them into a list.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class EvenNumberDoubler {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> doubledEvens = numbers.stream()
                                    .filter(n -> n % 2 == 0)
                                    .map(n -> n * 2)
                                    .collect(Collectors.toList());
        System.out.println(doubledEvens);
    }
}
```

OUTPUT

```
[4, 8, 12, 16, 20]
```

In-Lab:

1) Consider a Coffee shop which has staff member count of 5. Employer at the end of the month, before giving the salaries to the employees, employer asked them to stand in a queue where employee having more experience should stand first and later followed by less experience so on. Employee has attributes like name, age and experience. You as an employer should distribute salary along with bonus.

   Bonus should be given to the employees based on experience.

   Employee1 has experience 5 years

   Employee2 has 4 years

   Employee 3 has 3 years,

   Employee 4 has 1 year and

   Employee 5 is a fresher.

   Filter the employees who have experience more than 2 years should be given bonus.

   Make use of Predicate interface and construct the scenario.

   Procedure/Program:

```
import java.util.*;
import java.util.stream.Collectors;

class Employee {
   String name;
   int age, experience;

   Employee(String name, int age, int experience) {
      this.name = name;
      this.age = age;
      this.experience = experience;
   }

   public int getExperience() { return experience; }

   @Override
   public String toString() {
      return name + " (Age: " + age + ", Experience: " + experience + " years)";
   }
```

```
}

public class CoffeeShop {
   public static void main(String[] args) {
      List<Employee> employees = Arrays.asList(
         new Employee("Employee1", 30, 5),
         new Employee("Employee2", 28, 4),
         new Employee("Employee3", 25, 3),
         new Employee("Employee4", 22, 1),
         new Employee("Employee5", 20, 0)
      );

      List<Employee> sortedEmployees = employees.stream()
         .sorted(Comparator.comparingInt(Employee::getExperience).reversed())
         .collect(Collectors.toList());

      List<Employee> eligibleForBonus = employees.stream()
         .filter(emp -> emp.getExperience() > 2)
         .collect(Collectors.toList());

      System.out.println("Employees in order of experience:");
      sortedEmployees.forEach(System.out::println);

      System.out.println("\nEmployees eligible for bonus:");
      eligibleForBonus.forEach(System.out::println);
   }
}
```

OUTPUT

Employees in order of experience:
Employee1 (Age: 30, Experience: 5 years)
Employee2 (Age: 28, Experience: 4 years)
Employee3 (Age: 25, Experience: 3 years)
Employee4 (Age: 22, Experience: 1 years)
Employee5 (Age: 20, Experience: 0 years)

Employees eligible for bonus:
Employee1 (Age: 30, Experience: 5 years)
Employee2 (Age: 28, Experience: 4 years)
Employee3 (Age: 25, Experience: 3 years)

2) A company wants to perform various operations on the list of employees efficiently using Java Stream API.

The employees have attributes like name, age, department, and salary. The operations include

1. Filtering employees by Department.
2. Sort employees by their names.
3. Find the employee with the highest salary.
4. Calculate average salary of employees.

Procedure/Program:

```
import java.util.*;
import java.util.stream.*;

class Employee {
   String name, department;
   double salary;

   Employee(String name, String department, double salary) {
      this.name = name;
      this.department = department;
      this.salary = salary;
   }

   public String toString() {
      return name + " (" + department + ", $" + salary + ")";
   }
}
```

```
public class EmployeeOperations {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
            new Employee("Alice", "HR", 60000),
            new Employee("Bob", "IT", 70000),
            new Employee("Charlie", "IT", 80000),
            new Employee("David", "HR", 50000)
        );

        System.out.println("IT Employees: " + employees.stream()
            .filter(e -> e.department.equals("IT"))
            .toList());

        System.out.println("Sorted Employees: " + employees.stream()
            .sorted(Comparator.comparing(e -> e.name))
            .toList());

         employees.stream()
            .max(Comparator.comparing(e -> e.salary))
            .ifPresent(e -> System.out.println("Highest Salary Employee: " + e));

        System.out.println("Average Salary: $" + employees.stream()
            .mapToDouble(e -> e.salary)
            .average()
            .orElse(0.0));
    }
}
```

OUTPUT

IT Employees: [Bob (IT, $70000.0), Charlie (IT, $80000.0)]

Sorted Employees: [Alice (HR, $60000.0), Bob (IT, $70000.0), Charlie (IT, $80000.0), David (HR, $50000.0)]

Highest Salary Employee: Charlie (IT, $80000.0)

Average Salary: $65000.0

✓ **Data and Results:**

Data

The company maintains employee records, including name, department, and salary.

Result

Employees are filtered, sorted, and analyzed using Java Stream API.

✓ **Analysis and Inferences:**

Analysis

Stream API helps efficiently process and extract key employee insights.

Inferences

The IT department has the highest-paid employees on average.

VIVA-VOCE Questions (In-Lab):

1) What is an anonymous inner class, and when would you use one?

A class without a name, defined and instantiated in one step. Used for quick subclassing or implementing interfaces.

Example:

java Copy Edit

```java
new Thread(new Runnable() {
    public void run() { System.out.println("Running"); }
}).start();
```

2) How does the @FunctionalInterface annotation help in defining a functional interface?

Ensures an interface has exactly **one abstract method**, enabling lambda expressions.

Example:

java Copy Edit

```java
@FunctionalInterface
interface MyFunc { void execute(); }
```

3) How do lambda expressions relate to functional interfaces

Lambdas provide a short way to implement functional interfaces.

Example:

java Copy Edit

```java
MyFunc obj = () -> System.out.println("Executing...");
```

4) Explain the difference between collect() and reduce() in the Stream API.

- **collect()** → Converts stream elements into a **List, Set, or Map**.
- **reduce()** → Aggregates elements into **a single value**.

Examples:

java Copy Edit

```java
List<String> list = stream.collect(Collectors.toList()); // collect()
int sum = stream.reduce(0, Integer::sum); // reduce()
```

5) What is a parallel stream, and how do you create one?

A stream that processes elements **concurrently** for better performance.

Example:

java Copy Edit

```java
list.parallelStream().forEach(System.out::println);
```

Post-Lab:

1) You are tasked with designing an Employee Management System for a company. The system needs to handle various operations on a list of employees using the Java Stream API. Each employee has attributes such as ID, name, department, salary, and age. The operations include filtering, sorting, grouping, and aggregation.
Requirements

1. **Data Model**:
    - Create an Employee class with attributes: id, name, department, salary, and age.
2. **Operations**:
    - **Filter** employees based on department.
    - **Sort** employees by salary in descending order.
    - **Group** employees by department.
    - **Find** the highest-paid employee.
    - **Calculate** the average salary of employees in a department.
    - **List** the names of employees who earn more than a specified amount.

Procedure/Program:

```
import java.util.*;
import java.util.stream.*;

class Employee {
   int id, age; String name, department; double salary;
   Employee(int id, String name, String dept, double sal, int age) {
      this.id = id; this.name = name; this.department = dept; this.salary = sal;
this.age = age;
   }
   String getDepartment() { return department; }
   double getSalary() { return salary; }
   String getName() { return name; }
   public String toString() { return name + "(" + department + ", " + salary + ")"; }
}
```

```
public class EmployeeManagementSystem {
   public static void main(String[] args) {
      List<Employee> employees = List.of(
         new Employee(1, "Alice", "IT", 70000, 30),
         new Employee(2, "Bob", "HR", 50000, 40),
         new Employee(3, "Charlie", "IT", 90000, 35),
         new Employee(4, "David", "Finance", 60000, 45),
         new Employee(5, "Eve", "HR", 75000, 28)
      );

      String dept = "IT";

      System.out.println("Employees in " + dept + ": " +
employees.stream().filter(e -> e.getDepartment().equals(dept)).toList());

      System.out.println("Sorted by salary: " +
employees.stream().sorted(Comparator.comparingDouble(Employee::getSalary
).reversed()).toList());

      System.out.println("Grouped by department: " +
employees.stream().collect(Collectors.groupingBy(Employee::getDepartment)))
;

employees.stream().max(Comparator.comparingDouble(Employee::getSalary)).
ifPresent(e -> System.out.println("Highest paid: " + e));

      String targetDept = "HR";

      employees.stream().filter(e ->
e.getDepartment().equals(targetDept)).mapToDouble(Employee::getSalary).av
erage().ifPresent(avg -> System.out.println("Avg salary in " + targetDept + ": " +
avg));

      double salaryThreshold = 60000;
      System.out.println("Earning more than " + salaryThreshold + ": " +
employees.stream().filter(e -> e.getSalary() >
salaryThreshold).map(Employee::getName).toList());
   }
}
```

OUTPUT

Employees in IT: [Alice(IT, 70000.0), Charlie(IT, 90000.0)]
Sorted by salary: [Charlie(IT, 90000.0), Eve(HR, 75000.0), Alice(IT, 70000.0), David(Finance, 60000.0), Bob(HR, 50000.0)]
Grouped by department: {Finance=[David(Finance, 60000.0)], HR=[Bob(HR, 50000.0), Eve(HR, 75000.0)], IT=[Alice(IT, 70000.0), Charlie(IT, 90000.0)]}
Highest paid: Charlie(IT, 90000.0)
Avg salary in HR: 62500.0
Earning more than 60000.0: [Alice, Charlie, Eve]

Experiment#		Student ID	
Date		Student Name	[@KLWKS_bot] THANOS

✓ **Data and Results:**

Data

This program manages employees' data using Java Stream API operations.

Result

It filters, sorts, groups, and aggregates employee information efficiently.

✓ **Analysis and Inferences:**

Analysis

Employee details are processed to extract meaningful insights using Streams.

Inferences

Higher salaries are found in IT, while HR has moderate pay.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.