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

A) Here are some predefined functional interfaces from java.util.function:

1) Function<T, R>: Takes one argument of type T and returns a result of type R.

```
Function<Integer, String> intToString = (i) -> "Number: " + i;
```

2) Predicate<T>: Takes one argument and returns a boolean

```
Predicate<Integer> isGreaterThan10 = (n) -> n > 10;
```

3)

Experiment#		Student ID	
Date		Student Name	

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

```
import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class SortStringsDescending {

    public static void main(String[] args) {

        List<String> fruits = new ArrayList<>();

        fruits.add("Apple");

        fruits.add("Orange");

        fruits.add("Banana");

        fruits.add("Mango");


        Collections.sort(fruits, (s1, s2) -> s2.compareTo(s1));

        System.out.println("Fruits in descending order: "+fruits);

    }

}
```

4) 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 StreamPipelineExample {
    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("Doubled even numbers:"+doubledEvens);
    }
}
```

Experiment#		Student ID	
Date		Student Name	

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.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.function.Predicate;
Class Employee{
    String name;
    int age;
    int experience;
    public Employee(String name, int age, int experience) {
        this.name=name;
        this.age= age;
        this.experience=experience;
}
```

```
public void displaySalaryWithBonus(boolean bonusEligible) {
        String bonus = bonusEligible ? "with bonus" : "without bonus";
        System.out.println(name + "(Experience: " + experience + " years)
                                receives salary " + bonus);
}
}
public class Coffeeshop{
    public static void main(String[] args) {
        List<Employee> employees = new Arraylist<>();
        employees.add(new Employee("Employee1", 30, 5));
        employees.add(new Employee("Employee2", 28, 4));
        employees.add(new Employee("Employee3", 25, 3));
        employees.add(new Employee("Employee4", 22, 1));
        employees.add(new Employee("Employee5", 20, 0));
        Collections.sort(employees, Comparator.comparingInt(e -> -e.experience));
        Predicate<Employee> bonusEligibility = e -> e.experience > 2;
        for (Employee employee : employees) {
            boolean isEligibleForBonus = bonusEligibility.test(employee);
            employee.displaySalaryWithBonus(isEligibleForBonus);
        }
    }
}
```

Experiment#		Student ID	
Date		Student Name	

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.Collectors;

class Employee{
    String name;
    int age;
    String department;
    double salary;

  public Employee(String name, int age, String department, double salary){
        this.name=name;
        this.age= age;
        this.department=department;
        this.salary=salary;
}public String toString(){
        return "Name: "+name+", Age: "+age+", Department: "+department+
        "; Salary: $"+salary;
    }
}
```

```
Public class CompanyOperations {

    public static void main (String[] args) {

        List<Employees> employees = new ArrayList<>();

        employees.add(new Employee("Alice", 30, "HR", 5000));

        employees.add(new Employee("Bob", 25, "IT", 7000));

        employees.add(new Employee("Charlie", 28, "Finance", 6000));

        employees.add(new Employee("David", 35, "IT", 8000));

        employees.add(new Employee("Eve", 32, "Finance", 6500));

        System.out.println("Employees in IT department:");

        employees.stream()

                .filter(e->e.department.equals("IT"))

                .forEach(System.out::println);

        System.out.println("\nEmployees sorted by name:");

        employees.stream()

                .sorted(Comparator.comparing(e->e.name))

                .forEach(System.out::println);

        Employee highestSalary = employees.stream(1

                                        .max(Comparator.comparingDouble

                                        (e->e.salary)).

                                        .orElse(null);

        System.out.println("\nHighest salary : "+ highestSalary);
```

```
double averageSalary = employees.stream()
                .mapToDouble(e -> e.salary)
                .average()
                .orElse(0.0);

System.out.println("\nAverage salary: $" +averageSalary);
    }
}
```

O/P:

```
Employees in IT Department:
Name: Bob, Age: 25, Department: IT, Salary: $7000.0
Name: David, Age: 35, Department: IT, Salary: $8000.0

Employees sorted by name:
Name: Alice, Age: 30, Department: HR, Salary: $5000.0
Name: Bob, Age: 25, Department: IT, Salary: $7000.0
Name: Charlie, Age: 28, Department: Finance, Salary: $6000.0
Name: David, Age: 35, Department: IT, Salary: $8000.0
Name: Eve, Age: 32, Department: Finance, Salary: $6500.0

Employee with highest salary:
Name: David, Age: 35, Department: IT, Salary: $8000.0

Average salary of employees: $6500.0
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

✓ **Analysis and Inferences:**

* The IT department includes Bob and David, with the highest salary going to David ($8000).

* The employees are sorted alphabetically by name.

* The average salary across all employees is $6500, showing moderate salary distribution.

Stream API efficiently filters, sorts and calculates necessary information.

Experiment#		Student ID	
Date		Student Name	

VIVA-VOCE Questions (In-Lab):

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

An anonymous inner class is a nameless class used to create objects for one-time use, typically for overriding methods or implementing interfaces. It's useful for small, local implementations without the need for a separate class.

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

The @FunctionalInterface annotation ensures an interface has only one abstract method, suitable for lambda expressions. It provides compile-time checks to prevent adding extra abstract methods.

3) How do lambda expressions relate to functional interfaces?

Lambda expressions provide a concise way to implement the single abstract method of a functional interface, replacing anonymous inner classes for simpler syntax.

Experiment#		Student ID	
Date		Student Name	

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

collect(): Transforms a stream into a collection or other structure (eg; List, Set).

reduce(): Combines stream elements into a single result using a binary operator. (e.g.) summing values).

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

A parallel stream, processes elements concurrently across multiple threads to improve performance. You create it using parallelStream() on a collection or stream().parallel().

Experiment#		Student ID	
Date		Student Name	

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.Collectors;

class Employee{

    private int id;

    private String name;

    private String department;

    private double salary;

    private int age;

    public Employee(int id, String name, String department, double salary,
                    int age) {

    this.id=id;

    this.name=name;

    this.department=department;

    this.salary=salary;
```

```
    this.age = age;
}public String toString() {
        return "ID: "+id+", Name: "+name+", Department: "+department+
            ", Salary: $"+salary+", Age : "+age;
    }
} public class EmployeeManagementSystem {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee(1, "Alice", "HR", 5000, 30);
            new Employee(2, "Bob", "IT", 7000, 28);
            new Employee(3, "Charlie", "Finance", 6000, 40);
            new Employee(4, "David", "IT", 8000, 35);
            new Employee(5, "Eve", "Finance", 6500, 32);
    };
    System.out.println("Employees in IT department:");
    employees.stream()
            .filter(e->e.department.equals("IT")) :
            .forEach(System.out::println);
    System.out.println("\n Employees sorted by salary (descending):");
    employees.stream();
            .sorted(Comparator.comparingDouble(Employee::getSalary).
                                                reversed())
            .forEach(System.out::println);
```

```
System.out.println("\n Employees grouped by department:");

Map<String, List<Employee>> groupedByDept = employees.stream()
                                                .collect(Collectors.groupingBy
                                                        (e->e.department));
groupedByDept.forEach(dept, emplist) -> {
    System.out.println(dept+": "+emplist);
});

Employee highestPaid = employees.stream()
                                .max(Comparator.comparingDouble(Employee::getSalary))
                                .orElse(null);
System.out.println("\nHighest paid employee: "+highestPaid);

double avgSalary = employees.stream()
                            .filter(e->e.department.equals("Finance"))
                            .mapToDouble(Employee::getSalary)
                            .average()
                            .orElse(0.0);
System.out.println("\nAverage Salary in Finance department: $"+avgSalary);
System.out.println("\nEmployees earning more than
```