IT Skill Test GIC Myanmar

Duration: 30 Minutes   Total Questions: 8

---

1. You are working on a Java application that processes a list of employee objects. You need to find all employees who are older than 30 and have a salary greater than 50000. Which of the following Stream API operations would be most efficient for this task?

   A.employees.stream().filter(e -> e.getAge() > 30).filter(e -> e.getSalary() > 50000)

   B.employees.stream().filter(e -> e.getAge() > 30 && e.getSalary() > 50000)

   C.employees.parallelStream().filter(e -> e.getAge() > 30 || e.getSalary() > 50000)

   D.employees.stream().map(e -> e.getAge() > 30).filter(e -> e.getSalary() > 50000)

2. Consider the following code snippet:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .reduce(0, (a, b) -> a + b);
System.out.println(sum);
```

   What will be the output of this code?

   A.20

   B.30

   C.55

   D.15

3. In a Java application, you need to implement a method that takes a list of strings and returns a new list containing only the unique strings, sorted in alphabetical order. Which of the following Stream API operations would you use?

   A..stream().distinct().sorted().collect(Collectors.toList())

   B..stream().sorted().distinct().collect(Collectors.toList())

   C..stream().collect(Collectors.toSet()).stream().sorted().collect(Collectors.toList()
)

   D..stream().sorted().collect(Collectors.toCollection(LinkedHashSet::new))

4. You are working on a Java application that processes customer orders. You have a List<Order> and need to find the total value of all orders placed by customers from a specific country. Which of the following Stream API operations would be most appropriate?

   A.orders.stream().filter(o -> o.getCustomer().getCountry().equals(country)).ma
   pToDouble(Order::getValue).sum()

   B.orders.stream().filter(o -> o.getCustomer().getCountry().equals(country)).ma
   p(Order::getValue).reduce(0.0, Double::sum)

   C.orders.parallelStream().filter(o ->
   o.getCustomer().getCountry().equals(country)).map(Order::getValue).sum()

   D.orders.stream().map(o -> o.getCustomer().getCountry().equals(country) ?
   o.getValue() : 0).reduce(0.0, Double::sum)

5. You are reviewing code and come across the following snippet:

```
public class StringProcessor {
    public static List<String> processStrings(List<String> input) {
        return input.stream()
            .filter(s -> s != null && !s.isEmpty())
            .map(String::toUpperCase)
            .sorted((s1, s2) -> s2.compareTo(s1))
            .collect(Collectors.toList());
    }
}
```

What improvement could be made to this code to make it more readable and maintainable?

A.Use method references instead of lambda expressions

B.Replace the stream operations with a for-loop

C.Use parallel stream for better performance

D.Extract the comparator to a separate method or use a predefined comparator

6. You are working on a legacy system that processes customer orders. The system currently uses a for-loop to filter and transform a list of orders. Your task is to refactor this code using Java 8 features to improve readability and performance. Which of the following implementations correctly uses Lambda expressions and Stream API to achieve the same result?

A.orders.stream().filter(o -> o.getStatus().equals("PENDING")).map(Order::getCustomerId).collect(Collectors.toList());

B.orders.parallelStream().filter(o -> o.getStatus() == "PENDING").map(o -> o.getCustomerId()).toList();

C.orders.stream().filter(o -> o.getStatus() == "PENDING").forEach(o -> o.getCustomerId()).collect(Collectors.toList());

D.orders.stream().map(Order::getCustomerId).filter(o -> o.getStatus().equals("PENDING")).collect(Collectors.toList());

7. Debug the following code snippet:

```
public class EmployeeProcessor {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("Alice", 30, "HR"),
            new Employee("Bob", 25, "IT"),
            new Employee("Charlie", 35, "HR")
        );

        Map<String, Long> departmentCount = employees.stream()
            .collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.counting()));

        System.out.println(departmentCount);
    }
}

class Employee {
    private String name;
    private int age;
    private String department;

    // Constructor and getters omitted for brevity
}
```

What will be the output of this program?

A.{HR=2, IT=1}

B.{IT=1, HR=2}

C.{HR=1, IT=1}

D.The code will throw a NullPointerException

8. You're developing a system to process sensor data. Each sensor reading is represented by a SensorReading object with attributes timestamp and value. You need to implement a method that finds the average value of readings within a specific time range. Which of the following implementations correctly uses Java 8 features to achieve this?

A.
```
public double getAverageInRange(List<SensorReading> readings, long
   startTime, long endTime) {
       return readings.stream()
           .filter(r -> r.getTimestamp() >= startTime && r.getTimestamp() <=
   endTime)
           .mapToDouble(SensorReading::getValue)
           .average()
           .orElse(0.0);
   }
```

B.public double getAverageInRange(List<SensorReading> readings, long startTime, long endTime) {

```
    return readings.stream()
        .filter(r -> r.getTimestamp() >= startTime && r.getTimestamp() <= endTime)
        .map(SensorReading::getValue)
        .reduce(0.0, Double::sum) / readings.size();
}
```

C.public double getAverageInRange(List<SensorReading> readings, long startTime, long endTime) {

```
    return readings.parallelStream()
        .filter(r -> r.getTimestamp() >= startTime && r.getTimestamp() <= endTime)
        .mapToDouble(r -> r.getValue())
        .sum() / readings.size();
}
```

D.public double getAverageInRange(List<SensorReading> readings, long startTime, long endTime) {

```
    return readings.stream()
        .filter(r -> r.getTimestamp() >= startTime && r.getTimestamp() <= endTime)
        .collect(Collectors.averagingDouble(SensorReading::getValue));
}
```