IT Skill Test GIC Myanmar

Duration: 30 Minutes Total Questions: 8

---

1. You are working on a legacy system that processes customer orders. The system currently uses a for-loop to iterate through a list of orders and apply a discount. Your task is to refactor this code using Java 8 features. Which of the following options correctly implements this using Lambda expressions and the Stream API?

   A.orders.stream().forEach(order -> order.setDiscount(0.1));

   B.orders.parallelStream().map(order -> order.setDiscount(0.1)).collect(Collectors.toList());

   C.orders.stream().filter(order -> order.getTotal() > 100).forEach(order -> order.setDiscount(0.1));

   D.orders.forEach(order -> { if(order.getTotal() > 100) order.setDiscount(0.1); });

2. In a financial application, you need to implement a method that calculates the total value of a customer's investments. The investments are stored in a List<Investment> where each Investment has a getValue() method. Which of the following implementations correctly uses the Stream API to sum the values?

   A.investments.stream().map(Investment::getValue).sum();

   B.investments.stream().mapToDouble(Investment::getValue).sum();

   C.investments.parallelStream().reduce(0, (sum, inv) -> sum + inv.getValue());

   D.investments.stream().collect(Collectors.summingDouble(Investment::getValue));

3. You are developing a user management system and need to implement a method to find all users with premium accounts. The User class has a boolean isPremium() method. Which of the following correctly implements this using the Stream API and stores the result in a new list?

   A. List<User> premiumUsers = users.stream().filter(User::isPremium).collect(Collectors.toList());

   B. List<User> premiumUsers = users.parallelStream().filter(u -> u.isPremium()).toList();

   C. List<User> premiumUsers = users.stream().map(u -> u.isPremium() ? u : null).collect(Collectors.toList());

   D. List<User> premiumUsers = users.stream().filter(u -> u.isPremium() == true).collect(Collectors.toList());

4. You are working on a text processing application and need to implement a method that counts the occurrence of each word in a given text. Which of the following implementations correctly uses the Stream API and Lambda expressions to achieve this?

   A. Arrays.stream(text.split(" ")).collect(Collectors.groupingBy(w -> w, Collectors.counting()));

   B. Arrays.stream(text.split(" ")).collect(Collectors.toMap(w -> w, w -> 1, Integer::sum));

   C. text.split(" ").stream().collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));

   D. Arrays.stream(text.split(" ")).reduce(new HashMap<String, Long>(), (map, word) -> { map.put(word, map.getOrDefault(word, 0L) + 1); return map; }, (m1, m2) -> m1);

5. You are developing a method to process a list of transactions. Each transaction has an amount and a type (DEBIT or CREDIT). You need to calculate the total amount of CREDIT transactions. Which of the following correctly implements this using the Stream API?

   A. transactions.stream().filter(t -> t.getType() ==

TransactionType.CREDIT).mapToDouble(Transaction::getAmount).sum();

B.transactions.parallelStream().filter(t -> t.getType() == TransactionType.CREDIT).map(Transaction::getAmount).reduce(0.0, Double::sum);

C.transactions.stream().collect(Collectors.groupingBy(Transaction::getType, Collectors.summingDouble(Transaction::getAmount))).get(TransactionType.CREDIT);

D.transactions.stream().filter(t -> t.getType() == TransactionType.CREDIT).collect(Collectors.summingDouble(Transaction::getAmount));

6. You're implementing a caching mechanism for a web application. You want to use a Map to store cached items, but need to ensure that entries are automatically removed after a certain time period. Which of the following Java classes would be most appropriate for this use case?

A.HashMap<String, CachedItem>

B.ConcurrentHashMap<String, CachedItem>

C.LinkedHashMap<String, CachedItem>

D.WeakHashMap<String, CachedItem>

7. You're developing a method to process a list of employees and calculate their average salary. The method should ignore employees with less than 2 years of experience. Which of the following implementations correctly achieves this using OOP principles and Stream API?

A.
```
public double calculateAverageSalary(List<Employee> employees) {
    return employees.stream()
        .filter(e -> e.getYearsOfExperience() >= 2)
        .mapToDouble(Employee::getSalary)
        .average()
        .orElse(0.0);
}
```

B.
```
public double calculateAverageSalary(List<Employee> employees) {
    return employees.stream()
        .filter(e -> e.getYearsOfExperience() >= 2)
        .map(Employee::getSalary)
```

```
            .reduce(0.0, Double::sum) / employees.size();
    }
```

C.
```
public double calculateAverageSalary(List<Employee> employees) {
    double sum = 0.0;
    int count = 0;
    for (Employee e : employees) {
        if (e.getYearsOfExperience() >= 2) {
            sum += e.getSalary();
            count++;
        }
    }
    return count > 0 ? sum / count : 0.0;
}
```

D.
```
public double calculateAverageSalary(List<Employee> employees) {
    return employees.parallelStream()
        .filter(e -> e.getYearsOfExperience() >= 2)
        .mapToDouble(Employee::getSalary)
        .sum() / employees.size();
}
```

8. You are debugging a Java application that processes financial transactions. The following code snippet is supposed to sum all transactions above $1000, but it's not working as expected. What is the issue and how can it be fixed?

```
List<Transaction> transactions = getTransactions();
double sum = transactions.stream()
    .filter(t -> t.getAmount() > 1000)
    .map(Transaction::getAmount)
    .reduce(0, (a, b) -> a + b);

System.out.println("Sum of large transactions: $" + sum);
```

A.The reduce operation is incorrect. It should be: .reduce(0.0, Double::sum)

B.The filter condition is wrong. It should be: .filter(t -> t.getAmount() >= 1000)

C.The map operation is unnecessary. Remove it and use:

.mapToDouble(Transaction::getAmount).sum()

D.There's a type mismatch in the reduce operation. Use: .map(t ->
t.getAmount().doubleValue())