Play with Java Lambda

Lab #6

COMP3021 2022 Spring

ChengPeng Wang(cwangch@cse.ust.hk)
Yiyuan Guo(yguoaz@cse.ust.hk)
Bowen Zhang(bzhangbr@cse.ust.hk)
Heqing Huang(hhuangaz@cse.ust.hk)

- **Objectives of this lab**

  Write Lambda Expression

  AND/OR operations on Predicate

  Higher order function

  Reduce/Map on List

- **Prior Knowledges**
  - Lambda Expression
    - a -> a + 1
    - (<parameter list>) -> (<statements>)
    - If we only write 1 line in <statements>, then Java regards this as a return statement
  - Predicate<T>
    - A set of lambda expressions
    - Input: an object of type T
    - Output: a Boolean value
  - Higher order function
    - A function that receives functions as parameters
    - A function that returns a new function
    - Both …
  - Reduce/Map
    - Reduce: aggregation on List(sum, max, min, …)
    - Map: modify each elements in List using same pattern.
    - Recall Excel functions

- **Background**

We have an **Account** Class, representing bank accounts for customers.

```
public class Account {
    public int id;
    public int balance;

    public Account(int id, int balance) {
        this.id = id;
        this.balance = balance;
    }
```

- **Task#1**

Implement **add100** by writing a lambda expression.

Its functionality:
It receives an Account object, and increase the balance by 100.
`Account.add100.accept(a1);`

```
// TODO: Task1
// replace the null with a lambda expression
public static Consumer<Account> add100 = null;
```

```
public static void test1() {
    output("Task1");
    Account a1 = new Account(1, 500);
    Account.add100.accept(a1);

    if(a1.balance == 600) {
        output("Success");
    } else {
        output("Fail");
    }
}
```

- **Task#2**

We defined 2 Predicates to check whether a bank account is legal.

```
public static Predicate<Account> lowerBound = a -> a.balance >=0;
public static Predicate<Account> upperBound = a -> a.balance <=10000;
```

This is how we used them.

```
Account a1 = new Account(1, -100);
if(lowerBound.test(a1) && upperBound.test(a1)) { // can't pass lowerBound's test
    System.out.println("Legal");
} else {
    System.out.println("Illegal");
}
```

- **Task#2**

However, what if we want to “combine” this 2 Predicates into 1?
Your task is to define **checkBound** using BOTH LowerBound AND upperBound.

Replace the **null** with an expression.

```
public static Predicate<Account> checkBound = null;
```

```
public static void test2() {
    output("Task1");
    Account a1 = new Account(1, 500);
    Account a2 = new Account(2, -100);
    Account a3 = new Account(3, 12000);

    boolean b1 = Account.checkBound.test(a1);
    boolean b2 = Account.checkBound.test(a2);
    boolean b3 = Account.checkBound.test(a3);
    if(b1 && !b2 && !b3) {
        output("Success");
    } else {
        output("Fail");
    }
}
```

- **Task#3**

Remember in Task#1 we implement an add100. Which can increase the balance of an account by 100. But we want to make it more flexible.

Define **maker** which has Type AddMaker. It takes a number N as input, and returns a function that behaves like addN.

```
interface AddMaker {
    Consumer<Account> make(int N);
}

// TODO: Task3
// replace the null with a lambda expression
public static AddMaker maker = null;
```

With maker, we can define add1000 and sub1000 like this:

```
Consumer<Account> add1000 = Account.maker.make(1000);
Consumer<Account> sub1000 = Account.maker.make(-1000);
```

- **Task#4**

Complete **getMaxAccountID** method. It takes a List of Account objects, and returns the id of the object with maximum balance.

```
// You can assume that all the Account in acconts have positive balances.
public static int getMaxAccountID(List<Account> accounts) {
    // TODO: Task4
    // replace the null with a lambda expression
    Account maxOne = accounts.stream().reduce(new Account(0, -100), null);

    return maxOne.id;
}
```

- **Test your code**

Use testLab6.java to test your code. By reading it you can understand the tasks better.

END OF LAB #6

Don’t forget to commit and push your code.