CS209A – Lambda

Key Content

- Understand the concept that passing code to methods with behavior parameterization.
- Understand the necessity of method reference and lambda.
- Learn how to use important functional interfaces.

1. Before Exercise

Here is a question:

Sort

Description:
Given an array, you should output the indices of elements according their values from small to large. For example, here is an array [2, 1, 3, 4], element 1 is the smallest one, its index is 1, so the first output is 1, element 2 is the second smallest, its index is 0, so the second output is 0, and so on. The output of array [2,1,3,4] is [1,0,2,3]. If some elements have the same value, according their indices from small to large. For example, the output of array [2,2,2,2] is [0,1,2,3]. When you print the result, you should follow the format according to the **Output** description.

Input:
The first line will be an integer T (1<=T<=20), which is the number of test cases.
For each test case, the first line will be the size of the array N (1 <= N<= 1000). The second line contains N integers, each integer a_i is in range: [-2^30,2^30].

Output:
The indices in output should be separated by a space.

Sample Input:
1
9
5 3 66 22 9 1 77 88 99

Sample Output:
5 1 0 4 3 2 6 7 8

And here is a reference code:

```
import java.util.Scanner;

public class Sort {
    public static void main(String[] args) {
```

```
        Scanner in = new Scanner(System.in);
        int total = in.nextInt();

        while (total-- > 0) {
            int n = in.nextInt();
            int array[] = new int[n];
            int index[] = new int[n];
            for (int i = 0; i < n; i++) {
                array[i] = in.nextInt();
                index[i] = i;
            }
            //Bubble sort the index according the value
            int temp;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n - i - 1; j++) {

                    if (array[index[j]] > array[index[j+1]]){
                        temp = index[j + 1];
                        index[j + 1] = index[j];
                        index[j] = temp;

                    }
                }
            }

            for (int i = 0; i < n; i++) {
                System.out.printf("%d", index[i]);
                if (i != n - 1)
                    System.out.print(" ");
            }
            System.out.println();
        }
        in.close();
    }
}
```

1. Rewrite above code using Collections.sort().
2. If we want to output the indices of elements according their values from large to small, what we should do?

Sample Input:
1
9
5 3 66 22 9 1 77 88 99

Sample Output:
8 7 6 2 3 4 0 1 5

3. If add an input argument, 0 means to arrange the element from small to large, 1 means to arrange the element from large to small, what we should do?

Sample Input:
2
0 9
5 3 66 22 9 1 77 88 99
1 9
5 3 66 22 9 1 77 88 99

Sample Output:
5 1 0 4 3 2 6 7 8
8 7 6 2 3 4 0 1 5

Reference code:
1. Sort1.java
2. Sort2.java
3. Sort3.java

2. Behavior parameterization

From the above example, we know in software engineering a well-known problem is that no matter what you do, user requirements will change.

Behavior parameterization is a software development pattern that lets you handle frequent requirement changes. In a nutshell, it means taking a block of code and making it available without executing it. This block of code can be called later by other parts of your programs, which means that you can defer the execution of that block of code. For instance, you could pass the block of code as an argument to another method that will execute it later. As a result, the method's behavior is parameterized based on that block of code. For example, if you process a collection, you may want to write a method that:

- Can do "something" for every element of a list
- Can do "something else" when you finish processing the list
- Can do "yet something else" if you encounter an error

This is what behavior parameterization refers to.

3. Lambda Expression

3.1 Lambda definition

Lambda has the following features:

- **Anonymous**— We say anonymous because it doesn't have an explicit name like a method would normally have: less to write and think about!
- **Function**— We say function because a lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.
- **Passed around**— A lambda expression can be passed as an argument to a method or stored in a variable.

- **Concise**— You don't need to write a lot of boilerplate like you do for anonymous classes.

using a lambda expression you can create a custom Comparator object in a more concise way.

Before:

```
Collections.sort(arrayList, new Comparator<Element>() {
        @Override
        public int compare(Element o1, Element o2) {
                return o1.value - o2.value;
        }
});
```

After:

```
Collections.sort(arrayList, (e1, e2)->e1.value - e2.value);
```

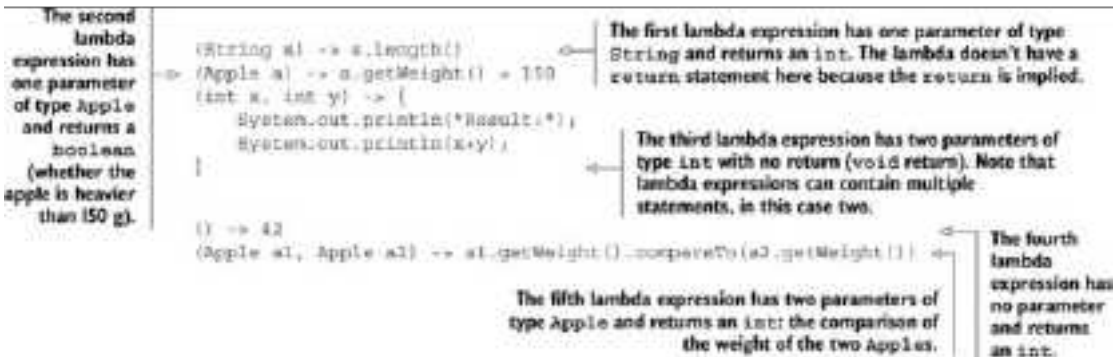
The lambda has three parts, as shown in the following figure:

- **A list of parameters**— In this case it mirrors the parameters of the compare method of a Comparator—two Elements.

- **An arrow**— The arrow -> separates the list of parameters from the body of the lambda.

- **The body of the lambda**— Co**mpare two Elements using their values. The expression is considered the lambd**a's return value.

To illustrate further, the following listing shows five examples of valid lambda expressions in Java 8.



The basic syntax of a lambda is either

(parameters) -> expression

or (note the curly braces for statements)

(parameters) -> { statements; }

3.2 Functional interface

The following table provides a list of example lambdas with examples of use cases.

Use case	**Examples of lambdas**
A boolean expression	(List<String> list) -> list.isEmpty()
Creating objects	() -> new Integer(10)
Consuming from an object	(e) -> { System.out.println(e); }
Select/extract from an object	(String s) -> s.length()
Combine two values	(int a, int b) -> a * b
Compare two objects	(e1, e2)->e1.value - e2.value

But where you're allowed to use lambda expressions?

In sort2 and sort3, you assigned a lambda to a variable of type Comparator<Element>, then it is passed to Collections.sort() as second argument.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

You could parameterize the behavior of the sort method using the interface Comparator<T> in sort2 and sort3. The interface Comparator<T> is a functional interface.

A functional interface is an interface that specifies **exactly one abstract method**.

Some common functional interfaces:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```
java.util.Comparator

```
public interface Runnable{
    void run();
}
```
java.lang.Runnable

```
public interface ActionListener extends EventListener{
    void actionPerformed(ActionEvent e);
}
```
java.awt.event.ActionListener

```
public interface Callable<V>{
    V call();
}
```
java.util.concurrent.Callable

```
public interface PrivilegedAction<V>{
    T run();
}
```
java.security.PrivilegedAction

Note:

Interfaces can also have **default methods** (that is, a method with a body that provides some default implementation for a method in case it isn't implemented by a class). An interface is still a functional interface if it has many default methods as long as it specifies only one abstract method.

3.3 Important functional interfaces

Interface name	Arguments	Returns	Example
`Predicate<T>`	T	`boolean`	Has this album been released yet?
`Consumer<T>`	T	`void`	Printing out a value
`Function<T,R>`	T	R	Get the name from an `Artist` object
`Supplier<T>`	None	T	A factory method
`UnaryOperator<T>`	T	T	Logical not (`!`)
`BinaryOperator<T>`	(T, T)	T	Multiplying two numbers (`*`)

3.3.1 Predicate<T>

```
@FunctionalInterface
public interface Predicate<T>{

    boolean test(T t);

}
```

The java.util.function.Predicate<T> interface defines an abstract method named test that accepts an object of generic type T and returns a boolean. You might want to use this interface when you need to represent a boolean expression that uses an object of type T.

For example, you can define a lambda that accepts String objects, only return the nonempty strings.

```
public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for (T s : list) {
        if (p.test(s)) {
            results.add(s);
        }
    }
    return results;
}
public static void main(String[] args) {
    List<String> listOfStrings = new ArrayList<>();
    listOfStrings.add("");
    listOfStrings.add("abc");
    listOfStrings.add("\n");
    listOfStrings.add("e");
    Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
    List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
    System.out.println(nonEmpty);
}
```

3.3.2 Consumer<T>

```
@FunctionalInterface
public interface Consumer<T>{

    void accept(T t);

}
```

The java.util.function.Consumer<T> interface defines an abstract method named accept that takes an object of generic type T and returns no result (void). You might use this interface when you need to access an object of type T and perform some operations on it. For example, you can use it to create a method forEach, which takes a list of Integers and applies an operation on each element of that list. In the following listing you use this forEach method combined with a lambda to print all the elements of the list.

```
public static <T> void forEach(List<T> list, Consumer<T> c) {
    for (T s : list) {
        c.accept(s);
    }
}
public static void main(String[] args) {
```

```
        List<String> listOfStrings = new ArrayList<>();
        listOfStrings.add("");
        listOfStrings.add("abc");
        listOfStrings.add("\n");
        listOfStrings.add("e");
        Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
        List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
        System.out.println(nonEmpty);

        forEach(listOfStrings, (s)-> System.out.println(s));
        forEach(nonEmpty, (s)-> System.out.println(s));

}
```

3.3.3 Function<T, R>

```
public interface Function<T, R> {

        R apply(T t);

}
```

The java.util.function.Function<T, R> interface defines an abstract method named apply that takes an object of generic type T as input and returns an object of generic type R. You might use this interface when you need to define a lambda that maps information from an input object to an output (for example, extracting the weight of an apple or mapping a string to its length). In the listing that follows we show how you can use it to create a method map to transform a list of Strings into a list of Integers containing the length of each String.

```
public static <T, R> List<R> map(List<T> list, Function<T,R> f){
        List<R> results = new ArrayList<>();
        for(T s : list){
                results.add(f.apply(s));
        }
        return results;
}
public static void main(String[] args) throws InvalidKeyException {
        List<String> listOfStrings = new ArrayList<>();
        listOfStrings.add("");
        listOfStrings.add("abc");
        listOfStrings.add("\n");
        listOfStrings.add("e");
        Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
        List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
        System.out.println(nonEmpty);

        forEach(listOfStrings, (s)-> System.out.println(s));
        forEach(nonEmpty, (s)-> System.out.println(s));
```

```
        List<Integer> listStrLen1 = map(listOfStrings, (s)->s.length());
        List<Integer> listStrLen2 = map(nonEmpty, (s)->s.length());

        forEach(listStrLen1, (s)-> System.out.println(s));
        forEach(listStrLen2, (s)-> System.out.println(s));

}
```

UnaryOperator<T> and BinaryOperator<T> , similarly, you can read the java doc or Baidu some tutorial yourself to learn how to use.