Lecture 3 & Lab 3

Functional Programming

1. What is functional programming?

   a programming paradigm -- the ways of thinking about things and solving problems

2. we have known OOP and procedure programing before, and Fp is introduced here

3. characteristics:

   1. functions are treated as first-class citizens(Basic unit of computation)

      functions are treat like any other variables

      ```
      // javascript
      const greet = function(){
          console.log('J')
      }
      ```

      higher order functions

      1. takes another function as **argument**
      2. **return** another function as result

   2. no side effects.

      1. **Side effects**: events cause in the scope affect outter scope

         some examples:

         ```
         writeFile(filename)
         updateDatabase(table)
         sendAjaxRequest(request)
         ```

      2. **pure functions**: **no sides effects**, and produce same output for the same input(**determinative**).

   3. immutablility

      1. Varibles will never change its value once defined
      2. Therefore, no assignment in Fp.

   4. recursion

      1. To avoid increment and decrement in the loop, we use recursive to do repetitive work.
      2. No `while, for` in the Fp

   5. advantages:

      Easy to debug, test, and parallelize

      Architectural simplicity

   6. Fp in Java

1. Lambda expression
2. Streams API (will be introduced later)

Lambda expression(since Java 8)

1. Difference to normal function
    1. is an **anonymous** function(**no name/type declaration/identifier**)
    2. can be create without belonging to any class(**like a anonymous object**)
    3. can be passed as a **parameter** to another function or **assigned to a variable**
    4. are callable anywhere in the scope(**Once assigned, use anywhere**)

    ```
    Comparator<student> cp = (e1, e2) -> e1.id - e2.id;
    Collections.sort(student_array, cp);
    // another way
    Collections.sort(student_array, (e1, e2) -> e1.id - e2.id);
    ```

2. Lambda Syntax

    ```
    (para1, para2) -> {lambda expressions} // not argument1, argument2...
    ```

    left part:

    No function name

    `()` could be omitted for a **single** parameter

    `,` is used to seperate different parameter

    `()` cannot be omitted for **no parameter**

    right part

    `{}` can be omitted for a **single** expression

    multi-expression are seperate by `;`

    can have a return statement

    `return` can be omitted for a **single** expression(i.e. the function body only have a statemetn, **that is, the return statement**)

    local assignment and control struture are allow but less common

3. Lambda Usage
    1. It's a short cut to define an implementation of a **functional interface**
    2. **Functional interface**: an interface has **and only has one abstract method**, but default and static method is not limitted.
    3. example: `Comparator<T>` has only an abstract method `int compare(T o1, T o2);`
4. Lambda Matching

```
collections.sort(strList,
                 (s1, s2) -> Integer.compare(s1.length, s2.length()));
```

1. Compiler can inference that `(s1, s2) -> Integer.compare(s1.length, s2.length())` must be a `Comparator<? super T> cp`
2. Because `Comparator` only have one abstract method, so that is the inplementation.
3. Therefore, `s1, s2` are parameter in order and the return value is `Integer.compare(s1.length, s2.length())`
4. It looks like **"an instantiation(an object)"** of the interface
5. with the usage of the lambda expression, a class implements Comparater<T> seems not neccessary. In other words, when we need an object only to use a special method, may be lambda expression is more convenient.

5. type inference

```
collections.sort(strList,
                 (s1, s2) -> Integer.compare(s1.length, s2.length()));
```

1. Because strlist is a `List<String>`, that is, `T` is String.
2. Thus, `(s1, s2) -> Integer.compare(s1.length, s2.length())` must implement the `Comparator<? super T>`
3. That's why we should avoid using raw type

6. tricky details
    1. can not redeclare variables in the **same or outter scopes**

    ```
    String s1 = "";
    Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();
    // compilation error: s1 redeclared
    ```

    2. Variable **defined in the scope** should be **final or effectively final**

    ```
    String str = "";
    Comparator<String> comp = (s1, s2) -> { str = str + " test";
        return s1.length() - s2.length();};
    //compilation error: str is not declared final
    ```

    Variable **used** in the scope should be **at least effective final**

    effective final: after using it **without modification**, it could not be modified even it's not declared **final**.

```
int x = 10;
Comparator<Integer> cp = (i, j) -> {
    // compilation error: disobey "using it without modification"
    x = 10;
    return i - j;
}
```

```
int x = 10;
Comparator<Integer> cp = (i, j) -> {
    // ok
    i += x;
    return i - j;
}
```

```
int x = 10;
Comparator<Integer> cp = (i, j) -> {
    // ok
    i += x;
    return i - j;
}
//compilation error: x could not be modified though not declared final
x += 15;
```

7. More use cases
    1. “instantiate” a functional interface **to use repetively**

```
public interface MyInterface{
    // abstract method
    double getPiValue();
}
MyInterface ref = () -> 3.1415; // we can treat ref as an instantiation of MyInterface with a special implementation
System.out.println("Pi = " + ref.getPiValue());}
```

    2. executing the same operation when iterating elements

```
// In list<E>
default void farEach(Consumer<? super T> action);

//Comsumer Interface
//Indicate the extra operation we want to do in the iteration
public interface Consumer<T>{
    void accept(T t);
}

//we want to print
strList.forEach(e -> System.out.println(e));
```