

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 programming before, and Fp is introduced here

3. characteristics:

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

functions are treated like any other variables

```
1 // javascript
2 const greet = function(){
3     console.log('J')
4 }
```

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 outer scope

some examples:

```
1 writeFile(filename)
2 updateDatabase(table)
3 sendAjaxRequest(request)
```

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

3. immutability

1. Variables 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**)

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

2. Lambda Syntax

```
1 (type1 para1, type2 para2) -> {lambda expressions} // not argument1,  
   argument2...
```

left part:

No function name

- ☐ could be omitted for a **single** parameter
- ☒ is used to separate different parameter
- ☐ cannot be omitted for **no parameter**

types can be either specified or omitted

right part:

- ☒ can be omitted for a **single** expression

multi-expression are separate 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 limited.
3. example: `Comparator<T>` has only an abstract method `int compare(T o1, T o2);`

4. Lambda Matching

```
1 collections.sort(strList,
2                 (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 implementation.
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 `Comparator<T>` seems not necessary. In other words, when we need an object only to use a special method, may be lambda expression is more convenient.

5. type inference

```
1 collections.sort(strList,
2                 (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**

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

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

```
1 String str = "";
2 Comparator<String> comp = (s1, s2) -> { str = str + " test";
3     return s1.length() - s2.length();};
4 //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**.

```

1  int x = 10;
2  Comparator<Integer> cp = (i, j) -> {
3      // compilation error: disobey "using it without modification"
4      x = 10;
5      return i - j;
6  }

```

```

1  int x = 10;
2  Comparator<Integer> cp = (i, j) -> {
3      // ok
4      i += x;
5      return i - j;
6  }

```

```

1  int x = 10;
2  Comparator<Integer> cp = (i, j) -> {
3      // ok
4      i += x;
5      return i - j;
6  }
7  //compilation error: x could not be modified though not declared final
8  x += 15;

```

7. More use cases

1. "instantiate" a functional interface **to use repetively**

```

1  public interface MyInterface{
2      // abstract method
3      double getPValue();
4  }
5  MyInterface ref = () -> 3.1415; // we can treat ref as an instantiation
  // of MyInterface with a special implementation
6  System.out.println("Pi = " + ref.getPValue());}

```

2. executing the same operation when iterating elements

```

1  // In list<E>
2  default void forEach(Consumer<? super T> action);
3
4  //Consumer Interface
5  //Indicate the extra operation we want to do in the iteration
6  public interface Consumer<T>{
7      void accept(T t);
8  }
9
10 //we want to print
11 strList.forEach(e -> System.out.println(e));

```

Method reference

1. condition when we use it

1. a lambda expression does nothing but call an existing method

```
1 | strList.forEach(e -> System.out.println(e));
```

2. It seems we don't need **e** actually

3. Method reference allow us to **refer to existing method by name**, which is often easier to read

2. example

```
1 | MyInter ref = s -> System.out.println(s);  
2 |  
3 | MyInter ref = System.out::println;
```

3. method types that we can referenced

1. static method:

declared with `static`, can be called without a object

e.g. `Integer.parseInt`

2. Instance method (Bound)
3. Instance method (Unbound)
4. Constructor

4. static method

1. syntax:

```
1 | Typename::staticMethod
```

2. example

```
1 | Integer::parseInt
```

3. equivalent lambda

```
1 | str -> Integer.parseInt(str)
```

5. Instance method(Bound)

1. **what is bound?**

If we convert the **method reference** to a **lambda expression**, we will find the method we use need a instance outside the lambda expression.

i.e. We say this method should bind to another instance instead of parameters.

```
1 | (args) -> expr.instanceMethod(args);  
2 | // This is Bound
```

2. syntax

```
1 | InstanceName::MethodName
```

3. example

```
1 | // System.out is an instance of PrintStream  
2 | System.out::println
```

4. equivalent lambda

```
1 | // without instance "System.out", we cannot realize the function of the  
   | println  
2 | str -> System.out.println(str)
```

5. when the paras and return values are the same, we can use the method reference even if it's not the library function

```
1 | public interface i1{  
2 |     public int foo(String s1)  
3 | }
```

6. Instance method(Unbound)

1. what is unbound?

If we convert the **method reference** to a **lambda expression**, we will find the method we use **don't** need a instance outside the lambda expression.

```
1 | (arg0, rest) -> arg0.instanceMethod(rest);  
2 | // This is unBound
```

2. syntax

```
1 | ClassName::MethodName
```

3. example

```
1 | Transformer tr = String::toLowerCase;
```

4. equivalent lambda

```
1 | Transformer tr = (s) -> s.toLowerCase();
```

Reference about bound and unbound

[lambda - Java 8: Difference between method reference Bound Receiver and UnBound Receiver - Stack Overflow](#)

7. Construter(Something like **Unbound**)

1. syntax

```
1 | ClassName::new
```

2. example

```
1 | Supplier<String> s = String::new
```

3. equivalent lambda

```
1 | Supplier<String> s = () -> new String();
```

Java functional Interface

1. In the `java.util.function` package

2. It's well defined

All have only one abstract method

Lambda can be used wherever these interfaces are used

They are used extensively in Java class libraries, especially with the Streams API (later)

3. **Consumer<T>**

1. represents a function that **takes an argument of type T** and **returns nothing** (consume it)

2. abstract method in `Consumer<T>`

```
1 | void accept(T t);
```

3. used example

```
1 | public void forEach(Consumer<? super T> action)
```

4. **Supplier<T>**

1. represents a function that **takes no argument** and **returns (supplies) a value of type T**

2. abstract method in `Supplier<T>`

```
1 | T get();
```

3. used example

```
1 | Supplier<T> su = Math::random;
2 | System.out.println(su.get())
```

5. Predicate<T>

1. represents a function that **takes a value of type T** and **returns a boolean**
2. The Predicate interface has one abstract method

```
1 | boolean test(T t);
```

3. example in **List<T>**

```
1 | // remove all of the element that satisfy the given predicate
2 | default boolean removeIf(Predicate<? super T> filter)
```

6. Operator Interfaces

1. Represent functions whose **result** and **argument types** are the **same**
2. Abstract method in `UnaryOperator<T>`

```
1 | T apply(T t);
```

Abstract method in `BinaryOperator<T>`

```
1 | T apply(T t1, T t2);
```

3. examples will be shown in the next lecture

7. Function Interface

1. Represents functions whose **result** and **argument** types could **differ**, or we say maps.
2. Abstract method in `Function<T, R>`

```
1 | R apply(T t);
```

3. examples will be shown in the next lecture

More used cases

Reference

[Lambda Expressions \(The Java™ Tutorials > Learning the Java Language > Classes and Objects\).](#)
([oracle.com](#))