Lecture

Lambda expression in Java

Nguyen Anh Tu

October 21st, 2019

NAZARBAYEV UNIVERSITY

Outline

- Introduce Java Lambda Expressions
- Map-reduce style data-processing
- Java Streams

Anonymous Inner Class Style

```
Thread threadinner = new Thread(new Runnable() {
    public void run() {
        System.out.println("Inner Classs Thread");
    }
});
```

- In Java, anonymous inner classes allow us to pass in methods as if they were parameters
  - Anonymous inner classes are "ad hoc" implementations
- For example, when defining a new thread all we usually care about is defining the *run()* method

First Class Functions

- In Java, methods are always attached to objects
- The *first class citizens* of a language are the things that can be passed as parameters
- Java does not have first-class functions although some common languages like Javascript and the functional languages do.

Functional Programming

- Programming with *pure functions*, functions in the mathematical sense
  - Take arguments and return values
  - No *side effects*
  - Values treated as *immutable*
  - Looping constructs handled with recursion
- Some people claim that functional programming is ‘safer’ and more suitable to applications that involve concurrency

- We can get some of the benefits of a functional style in Java by using lambdas, streams, and by making as many things final as possible.

Example of Pure functions

- **Example of pure function:**

```
public class ObjectWithPureFunction{

    public int sum(int a, int b) {
        return a + b;
    }
}
```

- **Example of impure function:**

```
public class ObjectWithNonPureFunction{
    private int value = 0;

    public int add(int nextValue) {
        this.value += nextValue;
        return this.value;
    }
```

Thread in a Functional Style

```
Thread t = new Thread(() -> {System.out.println("Lambda Thread");});

t.start();
```

- Using functional style Java with Lambda expressions we can get the same behavior with less verbosity
- Notice the arrow notation $\rightarrow$
- Comes from mathematical notation for expressing a mapping for example $f(x) : x \rightarrow x^2$
- Unlike casual mathematics we have to explicitly give the types

What is a lambda?

- Term comes from λ-Calculus
  - Formal logic introduced by Alonzo Church in the 1930's
  - Everything is a function!
  - Equivalent in power and expressiveness to Turing Machine

- A lambda (λ) is an *anonymous* function
  - A function without a corresponding identifier (name)

Java 8

- λ generally signifies the lambda abstraction operator that is used when discussing the lambda calculus
- Lambda calculus can be thought of as a very elementary programming language based on the notion of functional composition, in the mathematical sense $h(x) = f\,(g(x))$
- Style of coding has become popular because of the map-reduce style of data processing
- Java 8 introduced a number of constructs that enable a more functional style of programming

Lambda Expressions

- One of the major functional ideas introduced in Java 8 are lambda expressions or simply lambdas
- Lambdas are indicated by the syntax −> (minus sign followed by a greater than sign) which is meant to express the idea of a function or mapping
- Lambdas are like anonymous functions, they don’t have a name but they have parameters and a return type

```
(String s) -> System.out.println(s)
```

Lambda Syntax

Syntax	Example
parameter -> expression	`x -> x * x`
parameter -> block	`s -> { System.out.println(s); }`
(parameters) -> expression	`(x, y) -> Math.sqrt(x*x + y*y)`
(parameters) -> block	`(s1, s2) ->` `{ System.out.println(s1 + "," + s2); }`
(parameter decls) -> expression	`(double x, double y) -> Math.sqrt(x*x + y*y)`
(parameters decls) -> block	`(List<?> list) ->` `{ Arrays.shuffle(list); Arrays.sort(list); }`

Functional Interfaces

- Lambdas are used in the context of *Functional Interfaces*, interfaces that declare a single abstract method
- Basically a lambda expression can be used wherever a functional interface is expected, as long as the types match
- For example, runnable is a functional interface since it only has one abstract method *run()* and has empty input type and returns a void
- Code below is valid syntax in Java 8

```
Runnable r = () -> System.out.println("Runnable via lambda expression");
```

Functional Interfaces (cont'd)

- Optionally annotated with @FunctionalInterface
- A functional interface can have any number of default methods.
- Some functional interfaces you know
  - java.lang.Runnable
  - java.util.concurrent.Callable
  - java.util.Comparator
  - java.awt.event.ActionListener
  - Many, many more in package java.util.function

Example of Lambdas

```java
public class Test
{
    // operation is implemented using lambda expressions
    interface FuncInter1
    {
        int operation(int a, int b);
    }
    // sayMessage() is implemented using lambda expressions
    interface FuncInter2
    {
        void sayMessage(String message);
    }

    // Performs FuncInter1's operation on 'a' and 'b'
    private int operate(int a, int b, FuncInter1 fobj)
    {
        return fobj.operation(a, b);
    }

    public static void main(String args[])
    {
        FuncInter1 add = (int x, int y) -> x + y;
        FuncInter1 multiply = (int x, int y) -> x * y;

        Test tobj = new Test();

        // Add two numbers using lambda expression
        System.out.println("Addition is " + tobj.operate(6, 3, add));

        // Multiply two numbers using lambda expression
        System.out.println("Multiplication is " + tobj.operate(6, 3, multiply));

        // lambda expression for single parameter
        // This expression implements 'FuncInter2' interface
        FuncInter2 fobj = message ->System.out.println("Hello " + message);
        fobj.sayMessage("CSCI360");
    }
}
```

More examples at: https://www.javatpoint.com/java-lambda-expressions

Practical Aspects

- In practice, the functional style is often useful when we want to deal with lists or collections
- For instance, a common programming task is to find a subset of a list of objects based on some conditional expression
- *Example*: Take a list of strings and select all the entries that begin with a capital letter

```
List<String> users = Arrays.asList("alice","bob","Charlie");
List<String> filteredUsers = filter(users,
        (String s) -> Character.isUpperCase(s.charAt(0)));
System.out.println(filteredUsers.size());
```

Typing

- The compiler infers the typing from the context

- In the previous slide the function filter was declared with one of its arguments as a functional interface, in this case Predicate<T>

```
List<String> users = Arrays.asList("alice","bob","Charlie");
List<String> filteredUsers = filter(users,
        (String s) -> Character.isUpperCase(s.charAt(0)));
System.out.println(filteredUsers.size());
```

```
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;
}
```

Functional Interfaces in Java 8

- A number of common functional interfaces have been included in Java 8
- Access these with java.util.function.*
- The Predicate takes a generic type and returns a Boolean
- Consumer<T>: T –> void
- Supplier<T>:() –> T
- Function<T,R> : T –> R

Function interfaces in java.util.function

```
BiConsumer<T,U>
BiFunction<T,U,R>
BinaryOperator<T>
BiPredicate<T,U>
BooleanSupplier
Consumer<T>
DoubleBinaryOperator
DoubleConsumer
DoubleFunction<R>
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function<T,R>
IntBinaryOperator
IntConsumer
IntFunction<R>
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction<R>
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
Predicate<T>
Supplier<T>
ToDoubleBiFunction<T,U>
ToDoubleFunction<T>
ToIntBiFunction<T,U>
ToIntFunction<T>
ToLongBiFunction<T,U>
ToLongFunction<T>
UnaryOperator<T>
```

Composing Functions

- Notice that the Function interface is something like a generic function with a domain and codomain.

- This interface can be used to compose functions, suppose we want to compose $f(x) = x + 1$ with $g(x) = x^2$

- There are two ways $f(g(x))$ and $g(f(x))$
  - Are these the same?

- From a more theoretical point of view this is relevant because the category theory that is currently driving computer science is all about the abstract algebra of function composition

Composing Functions Example

- Function interface has methods *andThen()* and *compose()* for doing functional composition

- What is the main concern when you are doing something like this?

- For example:

```
Function<Integer, Integer> f = x –> x + 1;
Function<Integer, Integer> g = x –> x * 2;

Function<Integer, Integer> h1 = f.compose(g);
Function<Integer, Integer> h2 = f.andThen(g);

Integer result1 = h1.apply(3);
System.out.println(result1);

Integer result2 = h2.apply(3);
System.out.println(result2);
```