Lambda expression in Java

How are lambda function done in Java

Because Java likes everything to be under a class or an interface, so the prerequisite of writing a lambda function is that it must be written according to an interface.

The specific interface must have one interface method in it, it cannot have other interface method! This will be called the **functional interface**. You are allowed to have other methods with implementation in the interface, however, it cannot have any other interface method.

The way you would write a lambda expression in Java is as follows

```
parameter -> expression
// or
(parameter1, parameter2) -> expression
// or
parameter -> {
// multi-line code
    // with explicit return
}
```

> Just like in JavaScript you would use the => (in Java is ->) to indicate the expression of the lambda function

What lambda function solves

Let's go through an example of what Lambda function is used to solve and how things work without it.

Let's say we are writing an interface called **StringFunction.java** it contain one interface method called **returnString** which will take in one parameter of String. The specification of **returnString** basically allows the implementer of the StringFunction interface to define a class that contains the returnString method to the string with the implementer's modification. For example, append a String to the parameter string.

```
interface StringFunction {
    public String returnString(String s);
}

class LmaoStr implements StringFunction {
    public String returnString(String s) {
        return s + "lmao";
    }
}

public class Test {
    public static void main(String[] args) {
        LmaoStr ls = new LmaoStr();
            System.out.println(ls.returnString("ayy "));
    }
}
```

As you can see to use this interface, you would have to first define a class that implements it, then instantiate an object of type `LmaoStr` then finally call the method to use it. This is just so much work to just implement a simple interface no? Lambda function aim to solve that.

With lambda function, the above code can be shorten to just:

```
interface StringFunction {
    public String returnString(String s);
}

public class Test {
    public static void main(String[] args) {
        StringFunction sf = (s) -> s + "lmao";
            System.out.println(sf.returnString("ayy "));
    }
}
```

You no longer have to define a explicit class just to implement one simple method. You can just write the lambda function and it will be the implementation of `returnString`.

Other example of functional interface

If you use the List.forEach method it uses the Consumer interface which allows you to write lambda function like such:

```
ArrayList<Integer> arr = new ArrayList<>();
arr.add(1);
arr.add(2);
arr.add(3);
arr.add(4);
arr.forEach((i) -> System.out.println(i));
```

```
// Consumer.class
public interface Consumer<T> {
void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
      Objects.requireNonNull(after);
      return (T t) -> { accept(t); after.accept(t); };
    }
}
```

As you can see the Consumer class is an interface in order for you to define the lambda function. `andThen` is not an interface method because it has implementation so it is fine.

Instance method method references

Method references can be a bit confusion because of the way it is used. So method reference are again also only be with used the functional interface context.

It is a reference to the body of the function and use it to implement the interface, so it has even a shorter syntax than a lambda function, since you are just borrowing a method to implement it instead of writing your own.

The one that I want to go over in particular is instance method method references because the rest of the ones the input type parameter matches and is self-explanatory.

```
Function<Cat, String> f1 = Cat::getName;
System.out.println(f1.apply(cat1));

Function<Cat, String> f1 = in -> in.getName();
System.out.println(f1.apply(cat1));
```

Here is an example. As you can see the first and second way of writing using lambda function are exactly the same. The output is the same. `getName` is an instance method, so how does instance method matches the interface of Function? Which is

```
public interface Function<T, R> {
R apply(T t);
}
```

While the getName method is

```
class Cat {
public String getName() {
        return name;
    }
}
```

When you make a reference to an instance method of a class, it will automatically assume that the interface method you are implementing for, have its first parameter as this class type. So in this example it will assume that for **apply** the first parameter is **Cat t**. Because you are referencing an instance method of a class. Then it will invoke the actual instance method on that first parameter, which is how it works despite not having the same headers with the interface.

To drive the point home:

You can see that **getName** doesn't take any parameter, but it can still be used as an method reference for Function, because the method will be called on the actual object passed into apply (the first parameter of apply), and not used **independently**.

Revision #5
Created 2023-04-09 22:05:36 UTC by Tamarine
Updated 2023-05-08 02:28:59 UTC by Tamarine