Section 16: Lambda Expressions

15 July 2021 19:23

Section 16: Lambda Expressions

Every lambda expressions got three parts :-

1) Argument list
2) Arrow token
3) Body

Because the compiler needs to match the lamnda expression to a method, lambda expressions can only be used with interfaces that contain only one method that has to be implemented. So these interfaces are also referred to as functional interfaces.

Using **lambda Expression to create a thread** :-

```
new Thread(()-> {
        System.out.println("Printing from the Runnable");
        System.out.println("Line 2");
        System.out.format("This is line %d\n", 3);
    }).start();
```

Interface Comparator <T>
https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Local variable has to be declared as Final when we use them within an anonymous call.
Because the local variable doesn't belong to the anonymous class instance.

Lambda expressions are treated as nested blocks. So there within the enclosing block scope so what that should mean is that were able to use the local variable.

When working with lambdas if we want to use the local variables in enclosing block they have to be effectively final so that the runtime will know what values to use when the lambda expression is evaluated.

Functional Programming :-
https://en.wikipedia.org/wiki/Functional_programming

Consumer Document :-
https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

Functional Interface & Predicated :-

Function :-

```
  private static void printEmployeesByAge(List<Employee> employees,
                                          String ageText,
                                          Predicate<Employee> ageCondition) {

    System.out.println(ageText);
    System.out.println("==================");
    for(Employee employee : employees) {
      if (ageCondition.test(employee)) {
        System.out.println(employee.getName());
      }
```

```
    }
  }
```

Calling from Main :-

```
printEmployeesByAge(employees, "Employees over 30", employee -> employee.getAge() > 30);
printEmployeesByAge(employees, "\nEmployees 30 and under", employee ->employee.getAge() <=
30);
```

According to age condition it will print the employee names from employees list.

We can do the above same thing using anonymous class also. Like this :-

```
printEmployeesByAge(employees, "\nEmployees younger than 25", new Predicate<Employee>() {
        @Override
        public boolean test(Employee employee) {
           return employee.getAge() < 25;
        }
    });
```

IntPredicate greaterThan15 = i -> i > 15;

```
System.out.println(greaterThan15.test(10));       // output : false

IntPredicate lessThan100 = i -> i < 100;
System.out.println(greaterThan15.and(lessThan100).test(50));       // output : true
```

Function Lambda Expression for getting last from a list of employees :- (Employee is class)

```
Function<Employee, String> getLastName = (Employee employee) -> {
        return employee.getName().substring(employee.getName().indexOf(' ') + 1);
    };

String lastName = getLastName.apply(employees.get(1));
    System.out.println(lastName);
```

Upper case Employee first name :-

```
Function<Employee, String> upperCase = employee -> employee.getName().toUpperCase();
Function<String, String> firstName = name -> name.substring(0, name.indexOf(' '));
Function chainedFunction = upperCase.andThen(firstName);
System.out.println(chainedFunction.apply(employees.get(0)));
```

Package java.util.function :-

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Stream :-

In practice a stream is a set of object references. The stream method which was added to the collections class of object references. The stream method which was added to the collections class in Java 8 creates a stream from a collection. Now each object references in the stream corresponds to an object in the collection and the ordering of the object reference matched the ordering of the collection.

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Example :-

```
 List<String> someBingoNumbers = Arrays.asList(
            "N40", "N36",
            "B12", "B6",
            "G53", "G49", "G60", "G50", "g64",
            "I26", "I17", "I29",
            "O71");
```

```
someBingoNumbers
        .stream()
        .map(String::toUpperCase)  // :: colon colon notation is called method reference
        .filter(s->s.startsWith("G"))
        .sorted()
        .forEach(System.out::println);
```

Method References :-
https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

```
Stream<String> ioNumberStream = Stream.of("I26", "I17", "I29", "O71");
Stream<String> inNumberStream = Stream.of("N40", "N36", "I26", "I17", "I29", "O71");
Stream<String> concatStream = Stream.concat(ioNumberStream, inNumberStream);
System.out.println("----------------------");
System.out.println(concatStream
        .distinct()
        .peek(System.out::println)
        .count());
```

We may want to map a single object more than one object and we can use a **flat map method** to actually achieve that. The method accepts a function that returns a stream value so we can pass an object as the function argument and returns a string containing several objects which that were effectively mapping one object too many.

```
public class Employee {
   private String name;
   private int age;
……..
}

public class Department {
   private String name;
   private List<Employee> employees;
…….
}
departments.stream()
        .flatMap(department -> department.getEmployees().stream())
        .forEach(System.out::println);
```

Interface Callable<V> :-
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html

Interface Comparator<T> :-
https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

The stream chains are evaluated lazily. Nothing happens until a terminal operation is added to the chain. At that point, the chain is executed.

Interface Stream<T> :-
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html