PER SCHOLAS

Introduction to Java Programming

Unit 3 - Introduction to Lambda and exploring the uses with Functional interfaces

Java Basics

- Students should be able to understand:
  - Lambda Expressions Introduction
  - Lambda Expressions Nested Blocks
  - Scope and Functional Programming
  - Functional Interfaces & Predicates

Java Basics

Intro to Lambda

- Lambda expression is a new feature which is introduced in Java 8. A lambda expression is an anonymous function. A function that doesn't have a name and doesn't belong to any class. The concept of lambda expression was first introduced in LISP programming language.
- Parts of Lambda
  - A lambda expression in Java has these main parts:
  - Lambda expression only has body and parameter list.
  - No name – function is anonymous so we don't care about the name
  - Parameter list
  - Body – This is the main part of the function.
  - No return type – The java 8 compiler is able to infer the return type by checking the code. you need not to mention it explicitly.

Java Basics

Lambda Components

- The Arrow Operator
  - Lambda expressions introduce the new arrow operator **->** into Java. It divides the lambda expressions in two parts:
- The Parameters
  - A lambda expression can receive zero, one or more parameters.
  - The type of the parameters can be explicitly declared or it can be inferred from the context.
  - Parameters are enclosed in parentheses and separated by commas.
  - Empty parentheses are used to represent an empty set of parameters.
  - When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.
- The Body
  - The body of the lambda expression can contain zero, one or more statements.
  - When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
  - When there is more than one statement then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

Java Basics

Use of Lambda

- To use lambda expression, you need to either create your own functional interface or use the pre defined functional interface provided by Java. An interface with only single abstract method is called functional interface(or Single Abstract method interface), for example: Runnable, callable, ActionListener etc.
- Functional interface
  - A functional interface is an interface that contains one and only one abstract method.
  - In Java, the lambda expressions are represented as objects, and so they must be bound to a particular object type known as a functional interface. This is called the *target type*.
  - Since a functional interface can only have a single abstract method, the types of the lambda expression parameters must correspond to the parameters in that method, and the type of the lambda body must correspond to the return type of this method. Additionally, any exceptions thrown in the lambda body must be allowed by the throws clause of this method in the functional interface.

Java Basics

Java Lambda Expression with no parameter

```
3  public class FuncExample
4  {
5      public static void main(String[] args)
6      {
7          MyFunctionalInterface msg = () -> {
8              return "Hello";
9          };
10         System.out.println(msg.sayHello());
11     }
12 }
```

```
3  @FunctionalInterface
4  public interface MyFunctionalInterface
5  {
6      public String sayHello();
7  }
```

Java Basics

Java Lambda Expression with single parameter

```
3  public class FuncExample2
4  {
5      public static void main(String[] args)
6      {
7          MyFunctionalInterface2 f = (num) -> num+5;
8          System.out.println(f.incrementByFive(22));
9      }
10 }
```

```
3  public interface MyFunctionalInterface2
4  {
5      public int incrementByFive(int a);
6  }
7
```

Java Basics

Java Lambda Expression with multiple parameters

```
3  public interface StringConcat
4  {
5      public String sconcat(String a, String b);
6  }
3  public class FuncExample3
4  {
5      public static void main(String args[]) {
6          // lambda expression with multiple arguments
7          StringConcat s = (str1, str2) -> str1 + str2;
8          System.out.println("Result: "+s.sconcat("Hello ", "World"));
9      }
10 }
```

Java Basics

Java Lambda

- Write a method that generates a Comparator(String) that can be normal and no-space or reversed and with-space. Your method should return a lambda expression.

```
3 public interface ComparatorInterface
4 {
5     public String comparator(String str);
6 }
```

Java Basics

Java Lambda

```
3  public class Generator {
4      public static void main(String[] args) {
5          ComparatorInterface reverser_space = (str) -> {
6              String sub_str = "";
7            for(int i = str.length() - 1; i >= 0; i--) {
8                sub_str += str.charAt(i);
9            }
10           return sub_str;
11         };
12         ComparatorInterface normal_noSpace = (str) -> {
13             String sub_str = "";
14             for(int i = 0; i < str.length(); i++) {
15                 String s = Character.toString(str.charAt(i));
16                 if(!s.equals(" ")) {
17                     sub_str += s;
18                 }
19             }
20             return sub_str;
21         };
22         System.out.println(reverser_space.comparator("Hello class of JD"));
23         System.out.println(normal_noSpace.comparator("Hello class of JD"));
24     }
25 }
```

Java Basics

Java Lambda Exercise

- Write a program that can sum, add, multiply or divide two numbers. Follow these steps in order to complete this program:
  - Create a functional interface that is generic (<T>). The would be only one abstract method called: compute. This method return T as the generic type and as parameters: T num1, T num2 and String operator
  - Create a class called: MyCalc that is generic. MyCalc has four member variables.
    - Calculator<Integer> sum
    - Calculator<Integer> subtract
    - Calculator<Float> multiply
    - Calculator<Float> divide
      - These four variable are initialize to a Lambda expression that takes three arguments: num1, num2 and operator and perform the operation that the name implies.
  - Create a class called: CalcMain. This class reads in an expression from the user and using a switch statement, it figures out which implementation of the method compute to call.

Java Basics

forEach

- Introduced in Java 8, the *forEach* loop provides programmers a new, concise and interesting way for iterating over a collection.
- In java, the *Collection* interface has *Iterable* as its super interface – and starting with Java 8 this interface has a new API:
  - **void forEach(Consumer<? super T> action)**
- forEach performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.

Java Basics

forEach - Consumer Example

```
6  public class LambdaConsumer {
7      public static void main(String[] args) {
8          List<String> names = new ArrayList<String>();
9          names.add("Larry");
0          names.add("Steve");
1          names.add("James");
2          names.add("Conan");
3          names.add("Ellen");
4
5          names.forEach(name -> {
6              System.out.println(name);
7          });
8      }
9  }
```

Java Basics

Stream

- A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements:

```
 6 public class LambdaConsumer {
 7     public static void main(String[] args) {
 8         List<String> names = new ArrayList<String>();
 9         names.add("Larry");
10         names.add("Steve");
11         names.add("James");
12         names.add("Conan");
13         names.add("Ellen");
14
15         names.stream().forEach(name -> {
16             System.out.println(name);
17         });
18     }
19 }
```

Java Basics

Stream

```
6  public class LambdaConsumer {
7      public static void main(String[] args) {
8          List<String> names = new ArrayList<String>();
9          names.add("Larry");
10         names.add("Steve");
11         names.add("James");
12         names.add("Conan");
13         names.add("Ellen");
14
15         names
16         .stream()
17         .filter(s -> s.startsWith("C") || s.startsWith("S"))
18         .map(String::toUpperCase)
19         .sorted()
20         .forEach(s -> {
21             System.out.println(s);
22         });
23     }
24 }
```

Java Basics

Stream

- Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example filter, map and sorted are intermediate operations whereas forEach is a terminal operation.
- Calling the method stream() on a list of objects returns a regular object stream. But we don't have to create collections in order to work with streams as we see in the next code sample:

Java Basics

Stream

```
5  public class StreamEx1 {
6      public static void main(String[] args) {
7          Stream.of("a1", "a2", "a3")
8          .findFirst()
9          .ifPresent(s -> System.out.println(s));
10     }
11 }
```

Java Basics

Stream

- When executing this code snippet, nothing is printed to the console. That is because intermediate operations will only be executed when a terminal operation is present.

```
5  public class StreamEx2 {
6      public static void main(String[] args) {
7          Stream.of("d2", "a2", "b1", "b3", "c")
8          .filter(s -> {
9              System.out.println("filter: " + s);
10             return true;
11         });
12     }
13 }
```

Java Basics

Stream

```
5  public class StreamEx2 {
6      public static void main(String[] args) {
7          Stream.of("d2", "a2", "b1", "b3", "c")
8          .filter(s -> {
9              System.out.println("filter: " + s);
10             return true;
11         })
12         .forEach(s -> System.out.println("forEach: " + s));
13     }
14 }
```

Java Basics

Stream

- Consider the order of the chained methods:
  - The next example consists of two intermediate operations map and filter and the terminal operation forEach.
  - Both map and filter are called five times for every string in the underlying collection whereas forEach is only called once.

```
 5 public class StreamEx2 {
 6     public static void main(String[] args) {
 7         Stream.of("d2", "a2", "b1", "b3", "c")
 8         .map(s -> {
 9             System.out.println("map: " + s);
10             return s.toUpperCase();
11         })
12         .filter(s -> {
13             System.out.println("filter: " + s);
14             return s.startsWith("A");
15         })
16         .forEach(s -> System.out.println("forEach: " + s));
17     }
18 }
```

Java Basics

Stream

- Reduce the actual number of executions if we change the order of the operations, moving filter to the beginning of the chain:
- Now, map is only called once so the operation pipeline performs much faster for larger numbers of input elements.

```
 5 public class StreamEx2 {
 6     public static void main(String[] args) {
 7         Stream.of("d2", "a2", "b1", "b3", "c")
 8         .filter(s -> {
 9             System.out.println("filter: " + s);
10             return s.startsWith("a");
11         })
12         .map(s -> {
13             System.out.println("map: " + s);
14             return s.toUpperCase();
15         })
16         .forEach(s -> System.out.println("forEach: " + s));
17     }
18 }
```

Java Basics

Stream

- IntStreams can replace the regular for-loop utilizing IntStream.range()

```
 7 public class Calculator {
 8     public Object CalcExpression(String ex) {
 9         List<String> expression = Arrays.asList(ex.split(""));
10 
11         IntStream.range(0, expression.size()).forEach((s) -> {
12             System.out.println(s);
13         });
14         return null;
15     }
16 }
```