COURSEWARE

Lambda Expressions

Contents

Overview

A lambda expression is a function which can be created without belonging to any class.
They express instances of functional interfaces (an interface with a single abstract method is called a functional interface) since lambda expressions implement the only abstract function they are implementing the functional interface.
A lambda expression can be passed around as if it were an object and executed on demand.

Tutorial

Defining a Lambda Expression

Lambda expressions have three parts to them, they have the argument list, the arrow token, and then the body of the expression.

```
(arg1, arg2) -> {System.out.println("Two arguments " + arg1 + " and " + arg2);}
```

Above we have our argument list in the first set of parentheses, then we have the arrow token which is the "->" part, and finally our expressions body.

The number of parameters can vary, we may have zero, one, or multiple parameters.
If we only have the one parameter, then the initial parentheses are not necessary if the type of that variable can be inferred from context.

Using Lambda Expressions

Print All Elements

We can use lambda functions to iterate over an ArrayList and print out the values stored within it.

```
public static void main(String[] args) {
    ArrayList<Integer> arrayList = new ArrayList<>();
    arrayList.add(1);
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(4);

    arrayList.forEach(n -> System.out.println(n));
}
```

In the above example, we are declaring and instantiating an ArrayList of Integers in the variable arrayList.
We are then adding the values 1, 2, 3, and 4 to our arrayList.

Finally, we run the forEach method against the arrayList, passing in a lambda expression with "n" as our argument, then the arrow token, then our expressions body which is to print out the value stored as "n".

Using Logic Within a Lambda

We can expand on the previous example by adding some logic to the lambda function.

```
public static void main(String[] args) {
    ArrayList<Integer> arrayList = new ArrayList<>();
    arrayList.add(1);
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(4);

    arrayList.forEach(n -> {if (n % 2 == 0) System.out.println(n);});
}
```

In the above example, we are declaring and initialising an Integer ArrayList and adding the values 1, 2, 3, and 4 to it.
Then we are running the forEach method against the arrayList, passing in our lambda function which has an argument of "n", then the arrow token, then the expression body.
The expression body within our lambda is running a conditional check to see if the value modulus 2 is equal to 0; if it is it will print the value, if not it will move onto the next iteration.

Lambda With Multiple Arguments

```
public class LambdaExpressions {

    interface FuncInterface {
        int operation(int a, int b);
    }

    private int operate(int a, int b, FuncInterface funcObj) {
        return funcObj.operation(a, b);
    }

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

        LambdaExpressions lambda = new LambdaExpressions();
        System.out.println(lambda.operate(5, 7, add));
    }
}
```

In the above example, we have a functional interface called FuncInterface that has one method.
Within the class, we have the private method "operate" which takes two integer values and a FuncInterface value.
In the main method, we are instantiating a variable with the type FuncInterface and setting the value to our lambda expression.

We then instantiate an instance of the class we are working inside so that we can run the methods from the main without the need for making them static.
Finally, we are calling our private method and passing it the values of 5, 7, and our lambda function; which is using the lambda function to call the functional interface and passing it the values of 5 and 7, which then get added together and we print the result.

Exercises

There are no exercises for this module.