Core Java 8 and Development Tools

Lesson 21 : Lambda Expressions

Capgemini

Lesson Objectives

After completing this lesson, participants will be able to

- Understand the concept of Lambda expressions
- Work with lambda expressions
- Use method references and functional interfaces

21.1: Introduction to Functional Interface
Functional Interface

Functional Interface is an interface having exactly one abstract method

Such interfaces are marked with optional @FunctionalInterface annotation

```
@FunctionalInterface
interface xyz {
        //single abstract method
}
```

21.1: Introduction to Functional Interface
Functional Interface : Example

```
@FunctionalInterface
public interface MaxFinder {
        //single abstract method to find max between two numbers
        public int maximum(int num1,int num2);
}
```

How to implement this interface?

21.1: Introduction to Functional Interface
Functional Interface : Implementation

Class Implementation:

```
public class MaxFinderImpl implements MaxFinder {
        @Override
        public int maximum(int num1, int num2) {
                return num1>num2?num1:num2;
        }
}
```

```
MaxFinder finder = new MaxFinderImpl();
int result = finder.maximum(10, 20);
```

Want to know more concise way for implementation?

21.1: Introduction to Functional Interface
Functional Interface : Implementation

Lambda Expression:

```
public class MaxFinderImpl implements MaxFinder {
        @Override
        public int maximum(int num1, int num2) {
                return num1>num2?num1:num2;
        }
}
```

```
MaxFinder finder = (num1,num2) -> num1>num2?num1:num2;
int result = finder.maximum(10, 20);
```

Return type of "λE" is Functional Interface!

21.2: Writing Lambda Expressions
Lambda Expression

Lambda expression represents an instance of functional interface

A lambda expression is an anonymous block of code that encapsulates an expression or a block of statements and returns a result

Syntax of Lambda expression:

```
(argument list) -> { implementation }
```

The arrow operator -> is used to separate list of parameters and body of lambda expression

21.2: Writing Lambda Expressions
Lambda Expression

Sample Lambda Expressions

Functional Method	Lambda Expression
int fun(int arg);	(int num) -> num + 10
int fun(int arg0,int arg1);	(int num1,int num2) -> num1+num2

int fun(int arg0,int arg1);	(int num1,int num2) -> {
	int min = num1>num2?num2:num1;
	return min; }

String fun();	() -> “Hello World!”
void fun();	() -> { }
int fun(String arg);	(String str) -> str.length()
int fun(String arg);	str -> str.length()

21.2: Lambda Expressions and Functional Interface Demo

Execute the :

- CalculatorDemo

21.3: Builtin Functional Interfaces
Built-in Functional Interfaces

Java SE 8 provides a rich set of 43 functional interfaces

All these interfaces are included under package java.util.function

This set of interfaces can be utilized to implement lambda expressions

All functional interfaces are categorized into four types:

- Supplier
- Consumer
- Predicate
- Function

21.3: Builtin Functional Interfaces
Supplier

A Supplier<T> represents a function that takes no argument and returns a result of type T.

This is an interface that doesn't takes any object but provides a new one

```
@FunctionalInterface
public interface Supplier<T> {
        T get();
}
```

List of predefined Suppliers:

- BooleanSupplier
- IntSupplier
- LongSupplier
- DoubleSupplier etc.

21.3: Builtin Functional Interfaces
Consumer

A Consumer<T> represents a function that takes an argument and returns no result

A BiConsumer<T,U> takes two objects which can be of different type and returns nothing

```
@FunctionalInterface
public interface Consumer<T> {
        void accept(T t);
}
```

```
@FunctionalInterface
public interface BiConsumer<T,U> {
        void accept(T t, U,u);
}
```

List of predefined Consumer:

- IntConsumer
- LongConsumer
- ObjIntConsumer
- ObjLongConsumer etc.

21.3: Builtin Functional Interfaces
Predicate

A Predicate<T> represents a function that takes an argument and returns true or false result

A BiPredicate<T,U> takes two objects which can be of different type and returns result as either true or false

```
@FunctionalInterface
public interface Predicate<T> {
        boolean test(T t);
}
```

```
@FunctionalInterface
public interface BiPredicate<T,U> {
        boolean test(T t, U,u);
}
```

List of predefined Predicates:

- IntPredicate
- LongPredicate
- DoublePredicate etc.

21.3: Builtin Functional Interfaces
Function

A Function<T> represents a function that takes an argument and returns another object

A BiFunction<T,U> takes two objects which can be of different type and returns one object

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

```
@FunctionalInterface
public interface BiFunction<T,U,R> {
        R apply(T t, U,u);
}
```

List of predefined Functions:

- DoubleFunction<R>
- IntFunction<R>
- IntToDoubleFunction
- DoubleToIntFunction
- DoubleToLongFunction etc.

21.4 : Builtin Functional Interfaces and Lambda Expressions
Lambda Expression for Function Interfaces

Writing Lambda Expressions for Predefined Functional Interfaces

Functional Interface	Functional Method	Lambda Expression
Supplier<String>	String get();	() -> “Hello World”;
BooleanSupplier	Boolean get();	() -> { return true; }
Consumer<String>	void accept(String str);	(msg) -> syso(msg);
IntConsumer	void accept(Integer num);	(num) -> syso(num);
Predicate<Integer>	boolean test(Integer num);	(num) -> num>0;
Function<String,Integer>	Integer apply(String str);	(str) -> str.length;
UnaryOperator<Integer>	Integer apply(Integer num);	(num) -> num +10;
BiFunction<String,String, Boolean>	Boolean apply(String user,String pass);	(user,pass) -> { //functionality to validate user }

21.4 : Builtin Functional Interfaces and Lambda Expressions
Using Built-in Functional Interfaces

```
Consumer<String> consumer = (String str)-> System.out.println(str);
consumer.accept("Hello LE!");
Supplier<String> supplier = () -> "Hello from Supplier!";
consumer.accept(supplier.get());
//even number test
Predicate<Integer> predicate = num -> num%2==0;
System.out.println(predicate.test(24));
System.out.println(predicate.test(21));
//max test
BiFunction<Integer, Integer, Integer> maxFunction = (x,y)->x>y?x:y;
System.out.println(maxFunction.apply(25, 14));
```

21.3: Functional Interface Demo

Execute the :

- FunctionalInterfaces

21.5: Method Reference
Method References

Method reference is a shorthand way to write lambda expressions

It is a new way to refer a method by its name instead of calling it directly

Consider the below lambda expression, which call println method of System.out object:

```
Consumer<String> consumer = (String str)-> System.out.println(str);
```

Such lambda expressions are candidate for method references as it just calling a method for some functionality

The same expression can be written as with method reference:

```
Consumer<String> consumer = System.out :: println;
```

21.5: Method reference Demo

Execute the :

- MethodReference

Summary

In this lesson, you have learnt:

- Writing Lambda Expressions
- Functional Interfaces
- Method reference

Review Question

Question 1 : Which of the following Lambda expressions are valid to perform addition of two numbers?

- Option 1 **:** (x, y) -> x +y;
- Option 2 : (Integer x,Integer y) - > {return x+y;}
- Option 3 : (Integer x, Integer y) -> (x + y);
- Option 4 : All of above

Question 2 : _____ is a predicate of two arguments.

Question 3 : A method reference is shorthand to create a lambda expression using an existing method.

- True/False