Lesson

Functional Programming in Java:  
Commanding All the Laws of Nature from the Source

Wholeness of the Lesson: The declarative style of functional programming makes it possible to write methods (and programs) just by declaring *what* is needed, without specifying the details of *how* to achieve the goal. Including support for functional programming in Java makes it possible to write parts of Java programs more concisely, in a more readable way, in a more threadsafe way, in a more parallelizable way, and in a more maintainable way, than ever before.   
  
Maharishi’s Science of Consciousness: Just as a king can simply *declare* what he wants – a banquet, a conference, a meeting of all ministers – without having to specify the details about how to organize such events, so likewise can one who is awake to the home of all the laws of nature, the “king” among laws of nature, command those laws and thereby fulfill any intention. The royal road to success in life is to bring awareness to the home of all the laws of nature, through the process of transcending, and live life established in this field.

Representing Functors (Implementation of a Functional Interface) with Lambda Expressions

*//compare in Comparator* (*Employee e1, Employee e2)*

*{*

*if(method == SortMethod.BYNAME) {*

*return e1.name.compareTo(e2.name);*

*} else {*

*if(e1.salary == e2.salary) return 0;*

*else if(e1.salary < e2.salary) return -1;*

*else return 1;*

*}*

*}  
  
  
 //the “accept” method in Consumer*

*(String str) System.out.println(str);*

*//the “handle” method in EventHandler:*

*(ActionEvent evt) ->*

*System.out.println("Hello " + (username != null ? username : "World") + "!");*

*//the “apply” method in BiFunction*

*(x,y) -> 2\*x - y*

*//the “apply” method in TriFunction  
 //(a user defined functional interface)  
 (x,y,z) -> x + y + z*

These lambda expressions can be used wherever a matching functional interface is expected. But now we can think of these expressions as *functions* rather than as *objects.* In this way, lambdas upgrade the status of functions (at least in a certain context) to first-class citizens.

1. Important: Lambda expressions do not, on their own, have a unique type. Their type is *inferred* from the context. Inferring type from context is called *target typing.*Examples: Context in both cases below tells us that this lambda expression should be converted to a *Comparator<Employee>*

*//explicitly typed*

*Comparator<Employee> empNameComp* = (*Employee e1, Employee e2)*

*{*

*if(method == SortMethod.BYNAME) {*

*return e1.name.compareTo(e2.name);*

*} else {*

*if(e1.salary == e2.salary) return 0;*

*else if(e1.salary < e2.salary) return -1;*

*else return 1;*

*}*

*};  
  
//compiler is expecting a Comparator in the second argument(lambda expression here) of sort*

*Collections.sort(emps,* (*Employee e1, Employee e2)*

*{*

*if(method == SortMethod.BYNAME) {*

*return e1.name.compareTo(e2.name);*

*} else {*

*if(e1.salary == e2.salary) return 0;*

*else if(e1.salary < e2.salary) return -1;*

*else return 1*