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.

The Functional Style of Programming

1. Programs are declarative (“what”) rather than imperative (“how”). Makes code more *self-documenting* – the sequence of function calls mirrors precisely the requirements
2. Functions have *referential transparency* ­– two calls to the same method are guaranteed to return the same result (For Example Do Not use variables Outside the method (JL)).
3. Functions do not cause a change of state; in an OO language, this means that functions do not change the state of their enclosing object (by modifying instance variables). In general, functions do not have *side effects;* they compute what they are asked to compute and return a value, without modifying their environment (modifying the environment is a *side effect*).
4. Functions are *first-class citizens.* This means in particular that it is possible to use functions in the same way objects are used in an OO language: They can be passed as arguments to other functions and can be the return value of a function.

How Java SE 7 Approximates “Functions As First-Class Citizens”

Example: Suppose we want to sort a list of Employee objects.

*class Employee {  
 String name;  
 int salary;  
 public Employee(String n, int s) {  
 this.name = n;  
 this.salary = s;  
 }  
}*

Suppose we have a function *compare* that tells us how to compare two *Employee* objects:

*int compare(Employee e1, Employee e2) {  
 return e1.name.compareTo(e2.name);  
 }*

It would be nice to be able to make a call like this in order to sort the list by name:  
 *Collections.sort(list, compare)*  
Since functions are not first-class citizens, this cannot be done. But it can almost be done.

How Java SE 7 Approximates “Functions As First-Class Citizens”:  
The *Comparator* Interface and a Functorial Realization

The *Comparator* interface is a *declarative wrapper* for the function *compare*, described in the last slide.

*interface Comparator<T> {  
 int compare(T o1, T o2);*

*}*

It is called a *functional interface* because it has just one (abstract) method\*. So a class that implements it will have in effect just one implemented function; it will be an object that acts like a function.

An implementation of a functional interface is called a *functor.* Example:

*Public class EmployeeNameComparator implements Comparator<Employee> {*

*@Override*

*public int compare(Employee e1, Employee e2) {*

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

*}*

*}*

NOTE: Though *EmployeeNameComparator* is a class, it is essentially just a function that associates to each pair *(e1,e2)* of *Employees* an integer (indicating an ordering for *e1, e2).*\*NOTE: In reality, *Comparator* declares *two* abstract methods: *compare* and *equals*. However, *equals* already has an implementation in the *Object* class. The precise rule to determine whether an interface is a *functional* interface is that it must have exactly one abstract method, not counting any methods from *Object* that have been re-declared. See <https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html>

(‘equals’ will be inherited from the Object class.)

Functional interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the functional method for that functional interface, to which the lambda expression's parameter and return types are matched or adapted.

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

How Java SE 7 Approximates “Functions As First-Class Citizens”:  
Using Local Inner Classes As Closures

The implementation of the *Comparator* interface shown in the previous slide has a limitation: If the way the *compare* method acts depends on the state of the class that is attempting to sort *Employee* objects, our *Comparator* implementation will never be aware of this fact. (This is not a big problem in this case but can be in more complex settings.)  
  
Example: If we want to have the choice of sorting by name or by salary, we will need two different Comparators.

*Public class EmployeeSalary Comparator implements Comparator<Employee> {*

*@Override*

*public int compare(Employee e1, Employee e2) {*

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

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

*else return 1;*

*}*

*}*

*public class EmployeeNameComparator implements Comparator<Employee> {*

*@Override*

*public int compare(Employee e1, Employee e2) {*

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

*}*

*}*

*EmployeeInfo* Class

*Public class EmployeeInfo {*

*Static enum SortMethod {BYNAME, BYSALARY};*

*SortMethod method;*

*Public EmployeeInfo(SortMethod method) {*

*this.method = method;*

*}*

*//The Comparators are unaware of the choice of sort method*

*public void sort(List<Employee>emps) {*

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

*Collections.sort(emps, new EmployeeNameComparator());*

*}*

*else if (method == SortMethod.BYSALARY) {*

*Collections.sort(emps, new EmployeeSalaryComparator());*

*}*

*}*

*public static void main(String[] args) {*

*List<Employee>emps = new ArrayList<>();*

*emps.add(new Employee("Joe", 100000));*

*emps.add(new Employee("Tim", 50000));*

*emps.add(new Employee("Andy", 60000));*

*EmployeeInfo ei = new*

*EmployeeInfo(EmployeeInfo.SortMethod.BYNAME);*

*ei.sort(emps);*

*System.out.println(emps);*

*ei = new EmployeeInfo(EmployeeInfo.SortMethod.BYSALARY);*

*ei.sort(emps);*

*System.out.println(emps);*

*}*

*}*

Creating a *Comparator* Closure

A closure is a functor embedded inside another class, that is capable of remembering the state of its enclosing object. In Java 7, instances of member, local, and anonymous inner classes are (essentially) closures, since they have full access to their enclosing object’s state.  
  
Implementing an *EmployeeComparator* using a local inner class allows us to use just one *Comparator*, embedded in the *sort* method itself:

*public class EmployeeInfo {*

*static enum SortMethod {BYNAME, BYSALARY};*

*public void sort(List <Employee> emps, final SortMethod method) {*

*class EmployeeComparator implements Comparator<Employee> {*

*//SortMethod method argument is outside*

*//of the EmployeeComparator class (JL).*

*@Override //Comparator is now aware of sort method*

*public int compare(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;*

*}*

*}*

*}*

*Collections.sort(emps, new EmployeeComparator());*

*}*

*public static void main(String[] args) {*

*List<Employee>emps = newArrayList<>();*

*emps.add(new Employee("Joe", 100000));*

*emps.add(new Employee("Tim", 50000));*

*emps.add(new Employee("Andy", 60000));*

*EmployeeInfo ei = new EmployeeInfo();*

*ei.sort(emps, EmployeeInfo.SortMethod.BYNAME);*

*System.out.println(emps);*

*//same instance*

*ei.sort(emps, EmployeeInfo.SortMethod.BYSALARY);*

*System.out.println(emps);*

*}*

*}*

NOTE: In Java 7 and before, the method argument *SortMethod method* must be declared *final* since it is referenced in the method body. In Java 8, this is no longer necessary but still the argument may not be modified in the method body.

*Historical Note*: The “orginator” of the Java language – James Gosling – hoped to include real closures in Java from the beginning; inner classes were introduced as an approximation to closures. A quote from Gosling:

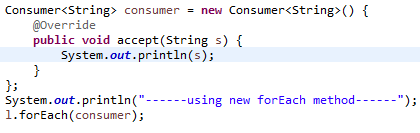
Closures were left out of Java initially more because of time pressures than anything else. In the early days of Java the lack of closures was pretty painful, and so inner classes were born: an uncomfortable compromise that attempted to avoid a number of hard issues. But as is normal in so many design issues, the simplifications didn't really solve any problems, they just moved them.

Another Functional Interface :*Consumer*

*public interface Consumer<T> {  
 public void accept(T t);  
 }*

The *Consumer* interface, like *Comparator*, has just one abstract method, so it is also a functional interface. It can likewise be implemented with a local or anonymous inner class to obtain a closure:

1 is a list below (JL).

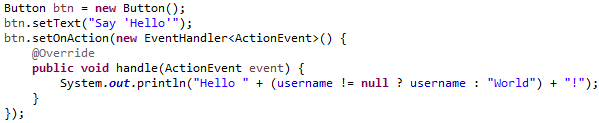


This is another example of a closure, though in this case, the accept method did not make special use of the state of its environment.

Another Functional Interface (JavaFX): EventHandler<T>

*public interface EventHandler<T extends Event> {* *public void handle(T evt); //typically, T is ActionEvent*}

One of the primary event handlers in JavaFX is EventHandler, another functional interface. From the JavaFX Lesson, we have:



This is also a closure, and *username* is a variable that is part of the state of the environment.

(See page 323 in Core Java, 10thed.)

Introducing Lambda Expressions

Lambda notation was an invention of the mathematician A. Church in his analysis of the concept of “computable function,” long before computers had come into existence (in the 1930s).   
  
Several equivalent ways of specifying a (mathematical) function:

f(x, y) = 2x – y //this version gives the function a name – namely ‘f’

(x,y) *↦* 2x – y //in mathematics, this is called “maps to” notation  
  
λxy.2x – y //Church’s lambda notation

*(x,y) 2\*x – y // Java SE 8 lambda notation*

NOTE: In Church’s lambda notation, the function’s arguments are specified to the left of the dot, and output value to the right.

Example: the Function (x,y) 2\*x – y

Java SE 8 offers new functional interfaces to support the majority of lambda expressions that could arise (though not all).

The *BiFunction<S,T,R>* interface has as its unique abstract method *apply(),* which returns the result of applying a function (apply) to it first two arguments (of type *S, T*) to produce a result (of type *R*).

*public interface BiFunction<S,T,R> {  
 R apply(S s, T t);  
}*

This code uses lambda notation to express functional behavior.

*public static void main(String[] args) {*

*BiFunction<Integer, Integer, Integer> f =*

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

*System.out.println(f.apply(2, 3)); //output: 1*

*}*

One way to accomplish the same thing without lambdas would be like this:

*public static void main(String[] args) {*

*class MyBiFunction implements BiFunction<Integer, Integer, Integer> {*

*public Integer apply(Integer x, Integer y) {*

*return 2 \* x.intValue() - y.intValue();*

*}*

*}*

*MyBiFunction f = new MyBiFunction();*

*System.out.println(f.apply(2, 3)); // output 1*

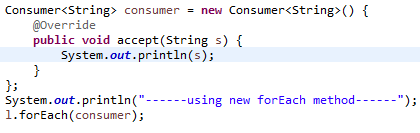
*}*

Using a Lambda Expression for a Consumer

Recall the Consumer interface

*public interface Consumer<T> {  
 public void accept(T t);  
 }*

and the application



This *forEach* code can be rewritten using lambdas as follows (syntax rules will be provided later):

*l.forEach(s ->System.out.println(s));*

Note how we are, in effect, simply passing the *accept* method of an anonymously defined (not a defined class (JL)) *Consumer* to the *forEach*  method.

Example: Creating Your Own Functional Interface

*@FunctionalInterface*

*public interface TriFunction<S,T,U,R> {*

*R apply(S s, T t, U u); // You have to write this new apply method*

*}*

*public static void main(String[] args) {*

*TriFunction<Integer, Integer, Integer, Integer> f =*

*(x,y, z) -> x + y + z;*

*System.out.println(f.apply(2, 3, 4)); //output: 9*

*}*

Notes

1. The *@FunctionalInterface* annotation is checked by the compiler – if the interface does not contain exactly one abstract method, there is a compiler error.
2. It is not necessary to use this annotation when providing a type for a lambda expression, but, like other annotations (@Override for example) it is a best practice because it allows a compiler check that would otherwise be overlooked until runtime.

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.

MAIN POINT 1

In Java, before Java SE 8, functions were not first-class citizens, which made the functional style difficult to implement. Prior to Java SE 8, Java approximated a function with a functional interface; when implemented as an inner class, objects of this type were close approximations to functions. In Java SE 8, these inner class approximations can be replaced by lambda expressions, which capture their essential functional nature: *Arguments mapped to outputs*. With lambda expressions, it is now possible to reap many of the benefits of the functional style while maintaining the OO essence of the Java language as a whole.  
  
The “purification” process that made it possible to transform “noisy” one-method inner classes into simple functional expressions (lambdas) is like the purification process that permits a noisy nervous system to have a chance to operate smoothly and at a higher level. This is one of the powerful benefits of the transcending process.

A Sample Application of Lambdas

Task: Extract from a list of names (*Strings*) a sublist containing those names that begin with a specified character, and transform all letters in such names to upper case.  
  
Imperative Style (Java 7)

*public List<String> findStartsWithLetterToUpper (List<String>list, char c) {*

*List<String> startsWithLetter = new ArrayList<String>();*

*for(String name : list) {*

*if(name.startsWith(""+c)) {*

*startsWithLetter.add(name.toUpperCase());*

*}*

*}*

*returnstartsWithLetter;*

*}*

Using Lambdas and Streams (Java 8)

*public List<String> findStartsWithLetter (List<String> list, String letter) {*

*return*

*list.stream() //convert list to stream*

*.filter(name ->name.startsWith(letter)) //returns filtered stream*

*.map(name ->name.toUpperCase())//maps each string to upper case string*

*.collect(Collectors.toList()); //organizes into a list*

*}*

*//parallel processing  
public List<String> findStartsWithLetter (List<String>list, String letter) {*

*return*

*list.parallelStream() //convert list to a parallel stream*

*.filter(name ->name.startsWith(letter)) //returns filtered stream*

*.map(name ->name.toUpperCase())//maps each string to upper case string*

*.collect(Collectors.toList()); //organizes into a list*

*}*

Anatomy of a Lambda Expression

A lambda expression has three parts:

*parameters* [zero or more]  
  *code block*  [if more than one statement, enclosed in curly braces { . . . } ]  
 [may contain *free variables*; values for these supplied by local or instance variables]

Examples

*//compare in Comparator: two parameters e1, e2; 1 free variable method*

(*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;*

*}*

*}*

*//accept in Consumer: one parameter str; no free variables*

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

*//handle in EventHandler: one parameter evt, one free variable, username*

*(ActionEvent evt) ->*

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

Free Variables and Closures - DO THIS!!!!

1. Free variables are variables that are *not* parameters and *not* defined inside the block of code (on the right hand side of the lambda expression)
2. In order for a lambda expression to be evaluated, values for the free variables need to be supplied (either by the method in which the lambda expression appears or in the enclosing class). These values are said to be *captured by the lambda expression.*
3. A *closure* in Java can be defined to be a lambda expression, together with the values of the free variables that are captured by the lambda expression*.*[Note that this is the same definition of closure as was given before since lambda expressions can always be interpreted as inner classes that are aware of the state of their enclosing class.]

Naming Lambda Expressions

1. We want to be able to reuse lambda expressions rather than rewriting the entire expression each time. To do so, we need to give it a name and a type.
2. Every object in Java has a type; the same is true of lambda expressions.

The type of a lambda expression is any functional interface for which   
the lambda expression is an implementation

Example: The lambda expression can be assigned the type *Comparator<Employee>*. The lambda expression can be viewed as a shorthand for a local or anonymous inner class that implements this interface. (Java doesn’t actually implement lambdas this way, but this viewpoint is accurate enough.)

(*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;*

*}*

*}*

1. *Naming a lambda expression* is done by using an appropriate functional interface as its type, like naming any other object:

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

*{*

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

*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;*

*}*

*}*

*. . .*

*public void sort(List <Employee> emps, final SortMethod method) {*

*Collections.sort(emps, empNameComp); //Closure here!!*

*}*

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*