Lecture 8: Functional Programming in Java 8

Commanding All the Laws of Nature from the Source

Wholeness Statement

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.

Features of Functional 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

```
int x = 0;
int mutate() {
    ++x;
    return x;
}
```

lacks referential transparency; produces a *side effect*

Features of Functional Programming

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*). [See example, previous slide]

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.

The Functional Style of Programming (cont.)

Demos show examples of adopting a Functional Programming style within Java SE 7. See ***lesson8.lecture.functionalprogramming***.

These are not true functional programming examples because they rely on simpler methods that are not purely functional. But these examples illustrate the functional style at the top level. In Java SE 8, these techniques are supported in a truly functional (and much more efficient) way.

Benefits of the Functional Style

- Programs are more compact, easier to write, and easier to read/understand
- Programs are thread-safe
- Easier to demonstrate correctness of functional programs
- Easier to test; less likely that a test of a subroutine will fail tomorrow if it passed today since there are no side effects

Functional Programming in Java

- Java designers – in agreement with many experts – decided it was time to incorporate the many benefits of the functional style of programming into Java
- Java *will remain an OO language*, with all the benefits of OO for building and maintaining large-scale applications
- For specialized needs – for instance, to extract data from a large collection using SQL-like queries – a functional approach would be more efficient, easier to read, and less error-prone
- Another potential benefit: Significant increase in performance because of automatic support for parallelizing code.

How Java SE 7 Approximates "Functions As First-Class Citizens"

- Problem: Sort a list of Employee objects.

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

- To sort, we need a criterion for comparing two Employee objects. We can define a function `compare` that tells us how to do this:

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

How Java SE 7 Approximates "Functions As First-Class Citizens"

- Problem: Sort a list of Employee objects.

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

- To sort, we need a criterion for comparing two Employee objects. We can define a function `compare` that tells us how to do this:

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

- Natural thing to try to do:

```
Collections.sort(list, compare)
```

How Java SE 7 Approximates "Functions As First-Class Citizens"

- Problem: Sort a list of Employee objects.

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

- To sort, we need a criterion for comparing two Employee objects. We can define a function `compare` that tells us how to do this:

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

- Natural thing to try to do:

```
Collections.sort(list, compare)
```

BUT, *functions are not first-class citizens in Java*, so this cannot be done.

How Java SE 7 Approximates "Functions As First-Class Citizens"

- The `Comparator` interface is a *declarative wrapper* for the function `compare`.

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

How Java SE 7 Approximates "Functions As First-Class Citizens"

- The `Comparator` interface is a *declarative wrapper* for the function `compare`.

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

- Although we cannot pass the function `compare` as an argument to the `sort` method, we can pass an instance of `Comparator`.

```
public class EmployeeNameComparator
        implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.name.compareTo(e2.name);
    }
}
```

How Java SE 7 Approximates "Functions As First-Class Citizens"

- The `Comparator` interface is a *declarative wrapper* for the function `compare`.

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

- Although we cannot pass the function `compare` as an argument to the `sort` method, we can pass an instance of `Comparator`.

```
public class EmployeeNameComparator
        implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.name.compareTo(e2.name);
    }
}
```

Now we can pass in the `Comparator` to sort a list of `Employee`s:

Collections.sort(list, new EmployeeNameComparator())

Functional Interfaces

- A `Comparator` is called a *functional interface* because it has just one (abstract) method. A class that implements it will have just one implemented function; it will be an object that acts like a function.

- **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)).

Functional Interfaces

- A `Comparator` is called a *functional interface* because it has just one (abstract) method. A class that implements it will have just one implemented function; it will be an object that acts like a function.

- **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)).

(e1, e2) $\rightarrow$ -1 (or any value < 0)	means	e1 comes before e2
(e1, e2) $\rightarrow$ 1 (or any value > 0)	means	e1 comes after e2
(e1, e2) $\rightarrow$ 0	means	e1 are equal

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.

```
Consumer<String> consumer = new Consumer<String>() {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
};
System.out.println("------using new forEach method------");
l.forEach(consumer);
```

Another Functional Interface: 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 Lesson 6, we have:

```
Button btn = new Button();
btn.setText("Say 'Hello'");
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello " + (username != null ? username : "World") + "!");
    }
});
```

Functor and Closure

- 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);
        }
}
```

- 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.

An example of Closure

```
public class EmployeeInfo {
    private final String sortMethod = "BYNAME";
    public void sort(List<Employee> emps, final String method) {
        class EmployeeNameComparator implements Comparator<Employee> {
            @Override
            // Comparator is now aware of sortMethod
            public int compare(Employee e1, Employee e2) {
                if (sortMethod.equals(method)) {
                    return e1.name.compareTo(e2.name);
                } else {
                    throw new IllegalArgumentException("don't know how to sort");
                }
            }
        }
        Collections.sort(emps, new EmployeeNameComparator());
    }
    public static void main(String[] args) {
        List<Employee> emps = new ArrayList<>();
        emps.add(new Employee("Joe", 100000));
        emps.add(new Employee("Andy", 60000));
        EmployeeInfo ei = new EmployeeInfo();
        ei.sort(emps, "BYNAME");
        System.out.println(emps);
        //ei.sort(emps, "BYSALARY");
    }
}
```

Closure and Lambda Expressions

- Lambda Expressions are syntax shortcuts for closures.
- Lambdas upgrade the status of functions (at least in a certain context) to first-class citizens.

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'

$(x,y) \mapsto 2x - y$ //in mathematics, this is called "maps to" notation

$\lambda xy.2x - y$ //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.

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]

Using a Lambda Expression for a Consumer

Recall the Consumer interface and the application

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

```
Consumer<String> consumer = new Consumer<String>() {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
};
System.out.println("------using new forEach method------");
l.forEach(consumer);
```

- 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 Consumer to the forEach method.

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 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 - y;
            }
    }
    MyBiFunction f = new MyBiFunction();
    System.out.println(f.apply(2, 3)); // output 1
}
```

More Examples:

```
public void sort(List<Employee> emps, final String method) {
    class EmployeeNameComparator implements Comparator<Employee> {
        @Override
        // Comparator is now aware of sortMethod
        public int compare(Employee e1, Employee e2) {
            if (sortMethod.equals(method)) {
                return e1.name.compareTo(e2.name);
            } else {
                throw new IllegalArgumentException("don't know how to sort");
            }
        }
    }
    Collections.sort(emps, new EmployeeNameComparator());
}
```

```
//compare in Comparator: two parameters e1, e2; two free variables sortMethod and method
(Employee e1, Employee e2) ->
 {
        if (sortMethod.equals(method)) {
           return e1.name.compareTo(e2.name);
        } else
           throw new IllegalArgumentException("don't know how to sort");
 }
//handle in EventHandler: one parameter evt, one free vble username
(ActionEvent evt) -> System.out.println("Hello " + (username != null ? username :
"World") + "!");
```

Free Variables

- 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)

- 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*.

Creating Your Own Functional Interface:

Demo: lesson8.lecture.lambdaexamples.trifunction

```
@FunctionalInterface
public interface TriFunction<S,T,U,R> {
        R apply(S s, T t, U u);
}
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:

- The @FunctionalInterface annotation is checked by the compiler – if the interface does not contain exactly one abstract method, there is a compiler error.
- 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.

Exericses: What happens when we attempt to create these interfaces? Does the code compile? Are these functional interfaces?

```
public interface Example1 {
    String toString();
}
@FunctionalInterface
public interface Example2 {
  String toString();
  void act();
}
```

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 uppercase.

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());
        }
    }
    return startsWithLetter;
}
```

A Sample Application of Lambdas (cont.)

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 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
}
```

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!

Naming Lambda Expressions (cont.)

Example: Consider this lambda expression:

```
(Employee e1, Employee e2) -> e1.getName().compareTo(e2.getName());
```

What functional interface has been implemented here?

Naming Lambda Expressions (cont.)

Example: Consider this lambda expression:

```
(Employee e1, Employee e2) -> e1.getName().compareTo(e2.getName());
```

What functional interface has been implemented here?

Analysis. The lambda accepts two input arguments – e1 and e2 – and has a return value of type Integer. We must find (or create) a functional interface whose encapsulated function has these characteristics.

Naming Lambda Expressions (cont.)

Example: Consider this lambda expression:

```
(Employee e1, Employee e2) -> e1.getName().compareTo(e2.getName());
```

What functional interface has been implemented here?

Analysis. The lambda accepts two input arguments – e1 and e2 – and has a return value of type Integer. We must find (or create) a functional interface whose encapsulated function has these characteristics.

How to Search. The new functional interfaces in Java are contained in the package `java.util.function`. We can look at the JDK API docs at https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html to locate a matching interface.

Naming Lambdas (continued)

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the o
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of boolean-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.

Naming Lambdas (continued)

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of boolean-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.

First Candidate

```
BiFunction<Employee, Employee, Integer> lambda
   =  (Employee e1, Employee e2) -> e1.getName().compareTo(e2.getName());
```

Naming Lambdas (continued)

- There is another possibility that we have already seen in the examples. Recall that a `Comparator` accepts two arguments and returns an integer – positive integer means "greater than", negative integer means "less than".

Second Candidate

```
Comparator<Employee> lambda
     =  (Employee e1, Employee e2) -> e1.getName().compareTo(e2.getName());
```

- The Second Candidate is also correct. The lambda can be typed in either way.

Naming Lambdas (continued)

Question: Which of the two candidates should be used?

Naming Lambdas (continued)

Question: Which of the two candidates should be used?

Answer: It depends on how you intend to use the lambda. In this case, it is likely that the lambda is designed as a `Comparator` for sorting. Only the `Comparator` type can be used for this purpose

```
Comparator<Employee> lambda1
  = (e1, e2) -> e1.getName().compareTo(e2.getName());
BiFunction<Employee, Employee, Integer> lambda2
  = (e1, e2) -> e1.getName().compareTo(e2.getName());
```

Naming Lambdas (continued)

Question: Which of the two candidates should be used?

Answer: It depends on how you intend to use the lambda. In this case, it is likely that the lambda is designed as a `Comparator` for sorting. Only the `Comparator` type can be used for this purpose

```
Comparator<Employee> lambda1
  = (e1, e2) -> e1.getName().compareTo(e2.getName());
BiFunction<Employee, Employee, Integer> lambda2
  = (e1, e2) -> e1.getName().compareTo(e2.getName());
```

```
public List<Employee> sort(List<Employee> list) {
    Collections.sort(list, lambda1);
    return list;
}
```

This code works

```
public List<Employee> sort2(List<Employee> list) {
    Collections.sort(list, lambda2);
    return list;
}
```

This code does *not* work

Naming Lambdas (Continued)

- [Optional] NOTE: Although every lambda is a realization of a functional interface, the way in which the Java compiler translates a lambda into a realization of such an interface is *not* obvious. Historically, the possibility of simply translating the lambda into an anonymous inner class was considered by the Java engineers, but was rejected for a number of reasons. One reason is performance – inner classes have to be loaded separately by the class loader. Another is that tying lambdas to such an implementation would limit the possibility for evolution of new features of lambdas in future releases. You can verify that lambdas and anonymous inner classes are fundamentally different (even though very similar) by considering how the implicit object reference `this' is interpreted by each type: In an anonymous inner class, `this' refers to the inner class; in a lambda, `this' refers to the surrounding class. See http://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood

Syntax Shortcuts via Target Typing

1. If parameter types can be inferred, they can be omitted

```
Comparator<Employee> empNameComp = (e1, 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;
      }
}
//sort expects a Comparator; since types in emps list are Employee, infer type Comparator<Employee>
Collections.sort(emps, (e1, 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;
      }
   }
));
```

See DEMO: ***lesson8.lecture.lambdaexamples.comparator3***

Syntax Shortcuts via Target Typing(cont.)

2. If a lambda expression has a single parameter with an inferred type, the parentheses around the parameter can be omitted.

```
Consumer consumer = str -> {System.out.println(str);};

EventHandler<ActionEvent> handler  = evt ->
        {System.out.println("Hello World");};
```

Syntax Shortcuts via Target Typing(cont.)

3. *Method References.* (See Lesson 9 for a fourth type of method reference – *constructor reference*)

A. Type: *object::instanceMethod.* Given an object ob and an instance method meth() in ob, the lambda expression

```
x -> ob.meth(x)
```

can be written as

```
ob::meth
```

Example (see SimpleButton demo in lesson8.lecture.methodreferences.objinstance.print)

Rewrite

```
button.setOnAction(evt -> p.print(evt));
```

as

```
button.setOnAction(p::print);
```

Another Example: The 'this' implicit object can be captured in a method reference in the same way: For instance the method reference `this::equals` is equivalent to the lambda expression `x -> this.equals(x)`.

Syntax Shortcuts via Target Typing(cont.)

B. Type: *Class::staticMethod*. Given a class ClassName and one of its static methods meth(), the lambda expression

```
x -> ClassName.meth(x)
```

//or (x,y) -> ClassName.meth(x,y) if meth accepts two args

can be rewritten as

```
ClassName::meth
```

Example

(see MethodRefMath demo in lesson8.lecture.methodreferences.classmethod.math)

Rewrite

```
BiFunction<Integer, Integer, Double> f
                    = (x,y) -> Math.pow(x, y);
```

As

```
BiFunction<Integer, Integer, Double> f = Math::pow;
```

Syntax Shortcuts via Target Typing(cont.)

C. Type: *Class*::*instanceMethod*. Given a class ClassName and one of its instance methods meth(), the lambda expression

```
(x,y) -> x.meth(y)
```

can be rewritten as

```
ClassName::meth
```

Example:

```
(str1, str2) -> str1.compareToIgnoreCase(str2)
```

can be written as

```
String::compareToIgnoreCase
```

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Declarative programming and command of all the laws of nature

1. In Java SE 7, the only first-class citizens are objects, created from classes. The valuable techniques of functional programming and a declarative style can be approximated using functional interfaces.
2. In Java SE 8, functions – in the form of lambda expressions – have become first-class citizens, and can be passed as arguments and occur as return values. In this new version, the advantage of functional programming with its declarative style is now supported in the language

---

3. **Transcendental Consciousness**: TC, which can be experienced in the stillness of one's awareness through transcending, is where the laws of nature begin to operate – it is the *home of all the laws of nature*
4. **Impulses Within the Transcendental Field:** As TC becomes more familiar, more and more, intentions and desires reach fulfillment effortlessly, because of the hidden support of the laws of nature.
5. **Wholeness moving within Itself**: In Unity Consciousness, one finally recognizes the universe in oneself – that all of life is simply the impulse of one's own consciousness. In that state, one effortlessly commands the laws of nature for all good in the universe.