**Lesson 8**

**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
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.

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.  
  
Demo Code:

* FactorialImperative, FactorialFunctional
* MapImperative, MapFunctional
* LackReferentialTransparency

**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 than any methods from Object that have been re-declared. See <https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.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** EmployeeSalaryComparator **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> {

@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 = **new** ArrayList<>();

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.

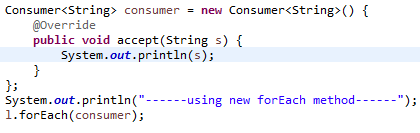
*Historical Note*: The “orginator” of the Java language – James Gosling – hoped to include real closures in Java from the beginning; inner classes were introduced as an approximation to closures. A quote from Gosling:

*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:

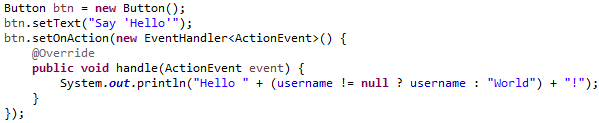


This is another example of a closure, though in this case, the accept method did not make special use of the state of its environment.

**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 Lesson 6, we have:



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

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

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

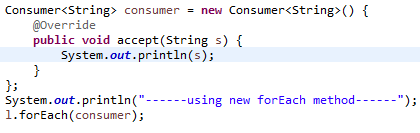
}

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 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);

}

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

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

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

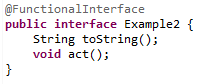
System.***out***.println(f.apply(2, 3, 4)); //output: 9

}

**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.

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

**Representing Functors 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());

}

}

**return** startsWithLetter;

}

**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

}

**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 vbles]

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 vbles**

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

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

(ActionEvent evt) ->

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

**Free Variables and Closures**

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***) {

**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**);

}

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 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;

}

}

));

NOTE: The following is another valid way to type this lambda expression, but this type cannot be used for sorting.  
  
Bifunction<Employee, Employee, Integer> bifunction =  
 (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;

}

}

TECHNICAL 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

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”);}

4. *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).

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*;

C. Type: *Class*::*instanceMethod.* Given a class ClassName and one of its instance methods meth(), the lambda expression (Unbounded receiver call as method call belongs to one of parameter provided to call method i.e. x)

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

can be rewritten as

ClassName::meth  
 Example (Comparator interface):   
 (str1, str2) -> str1.compareToIgnoreCase(str2)   
 can be written as

String::compareToIgnoreCase

[IN-CLASS EXERCISE]

**Syntax Rules Concerning Closures: The View from Java SE 7**(lesson8.lecture.closures.java7)

**public** **class** EmployeeInfo {

**static** **enum** SortMethod {***BYNAME***, ***BYSALARY***};

**private** **boolean** ignoreCase = **true**;

**public** **void** setIgnoreCase(**boolean** b) {

ignoreCase = b;

}

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

**class** EmployeeComparator **implements** Comparator<Employee> {

@Override

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

//local variable method must be final

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

//instance vble ignoreCase does not need to be final

**if**(ignoreCase) **return** e1.name.compareToIgnoreCase(e2.name);

**else** **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());

}  
}

1. Local and anonymous inner classes have access to instance variables of the enclosing class; they may also use local variables only if they are *final.*
2. *Best Practice*: Never modify instance variables from a method of a local inner class (for example, because of thread safety)

**Syntax Rules Concerning Closures: The View from Java SE 8**lesson8.lecture.closures.java8

**public** **class** EmployeeInfo {

**static** **enum** SortMethod {***BYNAME***, ***BYSALARY***};

**private** **boolean** ignoreCase = **true**;

**public** **void** setIgnoreCase(**boolean** b) {

ignoreCase = b;

}

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

Collections.*sort*(emps, (e1,e2) ->

{

//local variable method must be *effectively final*,

//but not necessarily final

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

//instance vble ignoreCase does not need to be effectively final

//but should not be modified either

**if**(ignoreCase) **return** e1.name.compareToIgnoreCase(e2.name);

**else** **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. Lambda expressions have access to instance variables of the enclosing class; they may also use local variables only if they are *effectively* *final* – this means that the value of the variable never changes (this is compiler-checked). This is now also the rule for local and anonymous inner classes.
2. *Best Practice*: Never modify instance variables inside a a lambda expression (for example, for thread safety)

**New Techniques: Filtering a List Using stream() and filter()**

**The Task:** Efficiently extract from a list a sublist satisfying certain criteria. (See Demos in package lesson8.lecture.filter)

1. Pre-Java 8 approach (see Demo class Weak): Use the usual for loop to pull out strings from a list that meet the criteria.
   1. For loop is part of imperative thinking, not declarative thinking.
   2. Elements are arranged into the return list in the same order they were read out. Maybe this is desirable, maybe not.
2. Good approach with Java 8 (see Demo class Good):

List<String> startsWithLetter =

list.stream()

.filter(name -> name.startsWith(letter))

.map(name -> name.toUpperCase())

.collect(Collectors.*toList*());

* 1. Convert the list to a Stream, which permits new operations, like filtering.
  2. The filter operation on a stream accepts a Java 8 Predicate<T>, whose only method is

boolean test(T t). Filter operations examine each element and applies the test method. Here, test method is name.startsWith(letter). The output of filter is another Stream – those elements for which test returned true.

* 1. The map operation accepts a Java 8 Function<T,R>, whose only method is R apply(T t). Map operations transform each element by using apply. Here, apply is name.toUpperCase, T is String, R is String.
  2. The collect method, with argument Collectors.toList() is a way to organize a stream back into a list.
  3. Can make even more compact.

1. Even more functional style version (see Demo class Better):
2. Idea:   
    Folks.***friends***.stream()

.filter(<<find the right Predicate>>).count();

1. If we have a particular letter ‘N’ in mind, this predicate would work:

Folks.***friends***.stream()

.filter(name -> name.startsWith(“N”)).count();

but then we have to duplicate the code to handle “B” or “S”, etc.

1. Solution: Create a function that associates with each possible letter *a lambda expression* representing a Predicate. This can be done using the Function interface whose only method is

R apply(T t)

1. Here is the concrete implementation of the Function interface we will use:

Function<String, Predicate<String>> startsWithLetter

= letter -> name -> name.startsWith(letter);

The lambda expression name -> name.startsWith(letter) is a Predicate, which returns a boolean, and which depends on the input value letter.

Then letter -> name -> name.startsWith(letter) is a lambda expression for a Function; when apply(letter) is invoked, the predicate   
 name -> name.startsWith(letter)

will be usable by the filter. This is an example of a *higher-order function*, which maps input to another function.

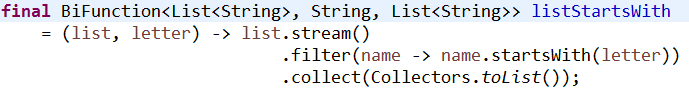
1. Using this Function, we create an atomic expression for the core of the computation:

**final** **long** countFriendsStartN =

Folks.***friends***.stream()

.filter(startsWithLetter.apply("N"));

1. (Advanced technique) We can make an even more general lambda expression, with wider applicability, like this:



Apply this expression as follows:

**final** List<String> friendsStartN   
= listStartsWith.apply(Folks.***friends***, "N");

**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.