
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Dr. Shafqat Ali Shad

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 thread-safe way, in a more parallelizable way, and in a more maintainable way, than ever before.

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.

The Functional Style of Programming (cont.)

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

Demo Code:

- *FactorialImperative*, *FactorialFunctional*
- *MapImperative*, *MapFunctional*
- *LackReferentialTransparency*

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.

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

Behavior Parameterization

- *Behavior parameterization* is a software development pattern that lets you handle frequent requirement changes. In a nutshell, it means taking a block of code and making it available without executing it.

- Requirement:

  A farmer needs to build a farm-inventory application, you're asked to implement a functionality to filter *green* apples from a list.

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

Behavior Parameterization (cont.)

1. ***First attempt: filtering green apples***

```
/**
 * First Attempt: filtering green apples
 *
 * @param inventory
 * @return
 */
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> apples = new ArrayList<>();
    for (Apple apple : inventory) {
        if ("green".equals(apple.getColor())) { // this part is to select
                                                // apple based on color.
            apples.add(apple);
        }
    }
    return apples;
}
```

But now the farmer changes his mind and wants to also filter *red* apples. What can you do? Write another method to filter Red?

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

Behavior Parameterization (cont.)

2. ***Second attempt: parameterizing the color***

```
public static List<Apple> filterApplesByColor(List<Apple> inventory, String color) {
    List<Apple> apples = new ArrayList<>();
    for (Apple apple : inventory) {
        if (apple.getColor().equals(color)) {
            apples.add(apple);
        }
    }
    return apples;
}
```

```
List<Apple> greenApples = filterApplesByColor(inventory, "green");
List<Apple> redApples = filterApplesByColor(inventory, "red");
```

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

Behavior Parameterization (cont.)

- The farmer comes back to you and says, "It would be really cool to differentiate between light apples and heavy apples. Heavy apples typically have a weight greater than 150 g."

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory, int weight) {
    List<Apple> apples = new ArrayList<>();
    for (Apple apple : inventory) {
        if (apple.getWeight() > weight) {
            apples.add(apple);
        }
    }
    return apples;
}
```

- Breaks **DRY** (don't repeat yourself) principle.
- If you want to alter the filter traversing to enhance performance?

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

Behavior Parameterization (cont.)

3. ***Third attempt: filtering with every attribute you can think of***

```
public static List<Apple> filterApples(List<Apple> inventory, String color, int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if ((flag && apple.getColor().equals(color)) || (!flag && apple.getWeight() > weight)) {
            result.add(apple);
        }
    }
    return result;
}
```

```
List<Apple> greenApples1 = filterApples(inventory, "green", 0, true);
List<Apple> heavyApples = filterApples(inventory, "", 150, false);
```

- the client code looks terrible.
- What do true and false mean?
- Doesn't cope well with changing requirements.

How Java SE 7 Approximates "Functions As First-Class Citizens": Behavior Parameterization (cont.)

4. ***Fourth attempt: filtering by abstract criteria***

   Find a better level of abstraction

```
public interface ApplePredicate {

    public boolean test(Apple apple);

}
```

```
public class AppleHeavyWeightPredicate implements ApplePredicate {
    @Override
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
```

```
public class AppleGreenColorPredicate implements ApplePredicate {

    @Override
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }

}
```

How Java SE 7 Approximates "Functions As First-Class Citizens": Behavior Parameterization (cont.)

behavior parameterization:

- the ability to tell a method to *take* multiple behaviors (or strategies) as parameters and use them internally to *accomplish* different behaviors.
- the behavior of the `filterApples` method depends on the *code you pass* to it via the `ApplePredicate` object. In other words, you've parameterized the behavior of the `filterApples` method!

```
public static List<Apple> filterApples(List<Apple> inventory, ApplePredicate predicate) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (predicate.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

```
List<Apple> heavyApples2 = filterApples(inventory, new AppleHeavyWeightPredicate());
List<Apple> greenApples2 = filterApples(inventory, new AppleGreenColorPredicate());
```

Verbosity?

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

Anonymous class

5. ***Fifth attempt: using an anonymous class***

```
List<Apple> redApples2 = filterApples(inventory, new ApplePredicate() {

    @Override
    public boolean test(Apple apple) {
        return "red".equals(apple.getColor());
    }

});

List<Apple> heavyApples3 = filterApples(inventory, new ApplePredicate() {

    @Override
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }

});
```

How Java SE 8 Does"Functions As First-Class Citizens":

Lambda Expression

6. ***Sixth attempt: using a lambda expression***

```
List<Apple> result = filterApples(inventory, (Apple apple) -> "red".equals(apple.getColor()));
```

At the moment, the `filterApples` method works only for `Apple`. But you can also abstract on the List type to go beyond the problem domain you're thinking of right now:

How Java SE 8 Does "Functions As First-Class Citizens":

Lambda Expression and Generics

7. ***Seventh attempt: abstracting over List type***

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

```
public static <T> List<T> filter(List<T> list, Predicate<T> predicate) {
    List<T> result = new ArrayList<>();
    for (T t : list) {
        if (predicate.test(t)) {
            result.add(t);
        }
    }
    return result;
}
```

```
List<Apple> redApples4 = filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));

List<Integer> numbers = Arrays.asList(1, 5, 7, 8, 10);
List<Integer> evenNumbers = filter(numbers, (Integer i) -> i % 2 == 0);
```

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.

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

- 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:

Creating a Comparator *Closure(cont.)*

```
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);
        }
}
```

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

a. [if more than one statement, enclosed in curly braces { . . . }]

b. [may contain *free variables*; values for these supplied by local or instance variables]

Free Variables and Closures

- 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*.

- 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.]

Representing Functors with Lambda Expressions

```
//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") + "!");
```

Exercise: convert to lambda

```
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2) {
        return a1.getWeight().compareTo(a2.getWeight());
    }
};
```

```
Comparator<Apple> byWeightLambda =
        (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

```
List<Apple> heavyApples3 = filterApples(inventory, new ApplePredicate() {

    @Override
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }

});
```

Functional Interface

- ***Where and how to use lambdas?***
  - You can use a lambda expression in the context of a functional interface.

- ***What is functional interface?***
  - A *functional* interface is that it **must have exactly one abstract method, not counting than any methods from Object that have been re-declared**.

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:

```
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 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:

```
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") + "!");
    }
});
```

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

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);
}
```

Example: the Function (x,y) -> 2 * x – y (cont.)

- 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 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: 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
}
```

- 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.

Example: Creating Your Own Functional Interface (cont.)

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();
}
```

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") + "!");
```

Representing Functors with Lambda Expressions (cont.)

```
//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.

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
}
```

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.

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: 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;
    }
}
```

Naming Lambda Expressions (cont.)

3. *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);
}
```

Naming Lambda Expressions (cont.)

4. 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;
        } }
));
```

Naming Lambda Expressions (cont.)

- **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***

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<String> 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 - *Method References* (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 - *Method References* (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 (Comparator interface):

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

can be written as

```
String::compareToIgnoreCase
```

Syntax Rules Concerning Closures: The View from Java SE 7

(lesson8.lecture.closures.java7)

- 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*.

- *Best Practice*: Never modify instance variables from a method of a local inner class (for example, because of thread safety)

Demo: EmployeeInfo .java

Syntax Rules Concerning Closures: The View from Java SE 8

lesson8.lecture.closures.java8

- 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.

- *Best Practice*: Never modify instance variables inside a a lambda expression (for example, for thread safety)

Demo: EmployeeInfo .java

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.
   a. For loop is part of imperative thinking, not declarative thinking.
   b. Elements are arranged into the return list in the same order they were read out. Maybe this is desirable, maybe not.

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

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());
```

a. Convert the list to a Stream, which permits new operations, like filtering.
b. 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.
c. 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.
d. The collect method, with argument Collectors.toList() is a way to organize a stream back into a list.
e. Can make even more compact.

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

3. Even more functional style version (see Demo class Better):

a. Idea:

```
Folks.friends.stream().filter(<<find the right
Predicate>>).count();
```

b. 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.

c. 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)
```

New Techniques: Filtering a List Using stream() and filter()-Even more functional style version (cont.)

d. 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.

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

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

```
final long countFriendsStartN =
Folks.friends.stream().filter(startsWithLetter.apply
("N"));
```

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

```
final BiFunction<List<String>, String, List<String>> listStartsWith
    = (list, letter) -> list.stream()
                            .filter(name -> name.startsWith(letter))
                            .collect(Collectors.toList());
```

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.