



# Lambda Expressions in Java 8: Part 4 – Higher Order Functions

Originals of slides and source code for examples: <http://courses.coreservlets.com/Course-Materials/java.html>  
Also see Java 8 tutorial: <http://www.coreservlets.com/java-8-tutorial/> and many other Java EE tutorials: <http://www.coreservlets.com/>  
Customized Java training courses (onsite or at public venues): <http://courses.coreservlets.com/java-training.html>

Slides © 2016 Marty Hall, [hall@coreservlets.com](mailto:hall@coreservlets.com)



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.



For customized training related to Java or JavaScript, please email [hall@coreservlets.com](mailto:hall@coreservlets.com)  
Marty is also available for consulting and development support

The instructor is author of several popular Java EE books, two of the most popular Safari videos on Java and JavaScript, and this tutorial.

Courses available at public venues, or custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JSF 2.3, PrimeFaces, Java programming (using Java 8, for those new to Java), Java 8 (for Java 7 programmers), JavaScript, jQuery, Angular 2, Ext JS, Spring Framework, Spring MVC, Android, GWT, custom mix of topics.
  - Java 9 training coming soon.
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Spark, Hibernate/JPA, HTML5, RESTful Web Services

Contact [hall@coreservlets.com](mailto:hall@coreservlets.com) for details



## Topics in This Section

- **Big idea: methods that return lambdas**
- **From Predicate: and, or, negate, isEqual**
- **From Function: andThen, compose, identity**
- **From Consumer: andThen**
- **Custom methods**

5

**coreservlets.com** – custom onsite training



# Higher Order Functions: Methods that Return Functions

Slides © 2016 [Marty Hall](http://www.coreservlets.com), [hall@coreservlets.com](mailto:hall@coreservlets.com)



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

## Big Idea

- **Methods can return “functions”**

- Really objects that implement functional interfaces
- You can also have a lambda that returns another lambda
  - Predicate, Function, and Consumer have builtin methods that return lambdas

- **Benefit**

- It is nothing new in Java to have a method return an object that implements an interface. But, by thinking of these return values as functions, you can have methods that compose functions, negate functions, chain functions, and so forth.

- **Syntax example**

```
Predicate<Employee> isRich = e -> e.getSalary() > 200000;  
Predicate<Employee> isEarly = e -> e.getEmployeeId() <= 10;  
empList = allMatches(employees, isRich.and(isEarly));
```

7

## Example: combinedOp

```
@FunctionalInterface  
public interface Op {  
    void runOp();  
  
    static void timeOp(Op operation) {  
        // Unchanged from last example  
    }  
  
    default Op combinedOp(Op secondOp) {  
        return(() -> { runOp();  
                        secondOp.runOp(); });  
    }  
}
```

8

# Higher Order Functions in Predicate

Slides © 2016 Marty Hall, hall@coreservlets.com



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

## Methods from Predicate

- **and**
  - Given a Predicate as an argument, produces a new Predicate whose test method is true if both the original Predicate and the argument Predicate return true for the given argument. Default method.
- **or**
  - Given a Predicate as an argument, produces a new Predicate whose test method is true if either the original Predicate or the argument Predicate return true for the given argument. Default method.
- **negate**
  - Takes no arguments: returns a Predicate whose test method returns the opposite of whatever the original Predicate returned. Default method.
- **isEqual**
  - Given an Object as an argument, produces a Predicate whose test method returns true if the Predicate argument is equals to the Object. Static method.

## Test Code: allMatches

- **Idea**

- Given a list and a predicate, returns new list of all the entries in the old list that passed the test.

- Very similar to filter method of Stream, which we will cover later

- **Code**

```
public static <T> List<T> allMatches(List<T> candidates,
                                     Predicate<T> matchFunction) {
    List<T> matches = new ArrayList<>();
    for(T possibleMatch: candidates) {
        if(matchFunction.test(possibleMatch)) {
            matches.add(possibleMatch);
        }
    }
    return(matches);
}
```

11

## Examples of and, or, negate, isEqual

```
private static final List<Employee> employees = EmployeeSamples.getSampleEmployees();

public static void predicateExamples() {
    Predicate<Employee> isRich = e -> e.getSalary() > 200000;
    Predicate<Employee> isEarly = e -> e.getEmployeeId() <= 10;
    System.out.printf("Rich employees: %s.%n", allMatches(employees, isRich));
    System.out.printf("Employees hired early: %s.%n", allMatches(employees, isEarly));
    System.out.printf("Employees that are rich AND hired early: %s.%n",
        allMatches(employees, isRich.and(isEarly)));
    System.out.printf("Employees that are rich OR hired early: %s.%n",
        allMatches(employees, isRich.or(isEarly)));
    System.out.printf("Employees that are NOT rich: %s.%n",
        allMatches(employees, isRich.negate()));
    Employee polly = employees.get(1);
    Predicate<Employee> isPolly = Predicate.isEqual(polly);
    System.out.printf("Employees in list that are 'equals' to Polly Programmer: %s.%n",
        allMatches(employees, isPolly));
}
```

12

## Results

Rich employees:

```
[Harry Hacker [Employee#1 $234,567], Polly Programmer [Employee#2 $333,333],  
Desiree Designer [Employee#14 $212,000]].
```

Employees hired early:

```
[Harry Hacker [Employee#1 $234,567], Polly Programmer [Employee#2 $333,333],  
Cody Coder [Employee#8 $199,999]].
```

Employees that are rich AND hired early:

```
[Harry Hacker [Employee#1 $234,567], Polly Programmer [Employee#2 $333,333]].
```

Employees that are rich OR hired early:

```
[Harry Hacker [Employee#1 $234,567], Polly Programmer [Employee#2 $333,333],  
Cody Coder [Employee#8 $199,999], Desiree Designer [Employee#14 $212,000]].
```

Employees that are NOT rich:

```
[Cody Coder [Employee#8 $199,999], Devon Developer [Employee#11 $175,000],  
Archie Architect [Employee#16 $144,444], Tammy Tester [Employee#19 $166,777],  
Sammy Sales [Employee#21 $45,000], Larry Lawyer [Employee#22 $33,000],  
Amy Accountant [Employee#25 $85,000]].
```

Employees in list that are 'equals' to Polly Programmer:

```
[Polly Programmer [Employee#2 $333,333]].
```

13

## Extending allMatches

```
public static <T> Predicate<T> combinedPredicate(Predicate<T>... tests) {  
    Predicate<T> result = e -> true;  
    for(Predicate<T> test: tests) {  
        result = result.and(test);  
    }  
    return(result);  
}
```

```
public static <T> List<T> allMatches2(List<T> candidates,  
                                     Predicate<T>... matchFunctions) {  
    Predicate<T> combinedTest = combinedPredicate(matchFunctions);  
    return(allMatches(candidates, combinedTest));  
}
```

The combinedPredicate method and a method similar to allMatches2 is discussed in much more detail in the lecture on File I/O.

14

## Using allMatches2

```
private static List<String> words =
    Arrays.asList("hi", "hello", "hola", "bye", "goodbye", "adios");

public static void allMatch2Examples() {
    List<String> hWords = allMatches2(words, word -> word.contains("h"));
    System.out.printf("Words with h: %s.%n", hWords);
    List<String> hlWords = allMatches2(words, word -> word.contains("h"),
                                       word -> word.contains("l"));
    System.out.printf("Words with h and l: %s.%n", hlWords);
    List<String> hlShortWords = allMatches2(words, word -> word.contains("h"),
                                             word -> word.contains("l"),
                                             word -> word.length() <= 4);
    System.out.printf("Words with h and l and length <= 4: %s.%n",
                      hlShortWords);
}
```

```
Words with h: [hi, hello, hola].
Words with h and l: [hello, hola].
Words with h and l and length <= 4: [hola].
```

15

coreservlets.com – custom onsite training



# Higher Order Functions in Function

Slides © 2016 Marty Hall, hall@coreservlets.com



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

## Methods from Function

- **compose**

- `f1.compose(f2)` means to first run `f2`, then pass the result to `f1`. Default method.
  - Of course, you cannot really “call” lambdas. So, strictly speaking, `f1.compose(f2)` means to produce a Function whose apply method, when called, first passes the argument to the apply method of `f2`, then passes the result to the apply method of `f1`.

- **andThen**

- `f1.andThen(f2)` means to first run `f1`, then pass the result to `f2`. So, `f2.andThen(f1)` is the same as `f1.compose(f2)`. Math people usually think of “compose” instead of “andThen”. Default method.

- **identity**

- `Function.identity()` creates a function whose apply method just returns the argument unchanged. Static method.

17

## Test Code: transform

- **Idea**

- Given a list and a function, returns new list by passing all the entries in the old list through the function
  - Very similar to map method of Stream, which we will cover later

- **Code**

```
public static <T,R> List<R> transform(List<T> origValues,
                                     Function<T,R> transformer) {
    List<R> transformedValues = new ArrayList<>();
    for(T value: origValues) {
        transformedValues.add(transformer.apply(value));
    }
    return(transformedValues);
}
```

18



## Using transform

```
public static void transformExamples() {
    System.out.printf("Original words: %s.%n", words);
    Function<String,String> makeUpperCase = String::toUpperCase;
    List<String> upperCaseWords = transform(words, makeUpperCase);
    System.out.printf("Uppercase: %s.%n", upperCaseWords);
    Function<String,String> makeExciting = word -> word + ": Wow!";
    List<String> excitingWords = transform(words, makeExciting);
    System.out.printf("Exciting: %s.%n", excitingWords);
    Function<String,String> makeBoth1 = makeExciting.compose(makeUpperCase);
    List<String> excitingUpperCaseWords1 = transform(words, makeBoth1);
    System.out.printf("Exciting uppercase[1]: %s.%n", excitingUpperCaseWords1);
    Function<String,String> makeBoth2 = makeUpperCase.andThen(makeExciting);
    List<String> excitingUpperCaseWords2 = transform(words, makeBoth2);
    System.out.printf("Exciting uppercase[2]: %s.%n", excitingUpperCaseWords2);
}
```

f1.compose(f2)  
same as  
f2.andThen(f1)

19

## Results

Original words: [hi, hello, hola, bye, goodbye, adios].

Uppercase: [HI, HELLO, HOLA, BYE, GOODBYE, ADIOS].

Exciting:

[hi: Wow!, hello: Wow!, hola: Wow!, bye: Wow!, goodbye: Wow!, adios: Wow!].

Exciting uppercase[1]:

[HI: Wow!, HELLO: Wow!, HOLA: Wow!, BYE: Wow!, GOODBYE: Wow!, ADIOS: Wow!].

Exciting uppercase[2]:

[HI: Wow!, HELLO: Wow!, HOLA: Wow!, BYE: Wow!, GOODBYE: Wow!, ADIOS: Wow!].

f1.compose(f2)  
same as  
f2.andThen(f1)

20

## Goal: Chained Function Composition

- **Idea**

- Modify the transform method so that it takes any number of Functions, instead of just one. It will compose all the functions, then use the result to transform entries.

- **Example usage**

```
List<String> words = ...;
Function<String,String> makeUpperCase = String::toUpperCase;
Function<String,String> makeExciting = word -> word + ": Wow!";
List<String> excitingUpperCaseWords =
    transform2(words, makeExciting, makeUpperCase);
```

21

## Extending transform: composeAll

```
public static <T> Function<T,T> composeAll
    (Function<T,T>... functions) {
    Function<T,T> result = Function.identity();
    for(Function<T,T> f: functions) {
        result = result.compose(f);
    }
    return(result);
}
```

22

## Extending transform: transform2

```
public static <T> List<T> transform2
    (List<T> origValues,
     Function<T,T>... transformers) {
    Function<T,T> composedFunction =
        composeAll(transformers);
    return(transform(origValues, composedFunction));
}
```

23

## Using transform2

```
public static void transform2Examples() {
    System.out.printf("Original words: %s.%n", words);
    Function<String,String> makeUpperCase = String::toUpperCase;
    List<String> upperCaseWords = transform2(words, makeUpperCase);
    System.out.printf("Uppercase: %s.%n", upperCaseWords);
    Function<String,String> makeExciting = word -> word + ": Wow!";
    List<String> excitingWords = transform2(words, makeExciting);
    System.out.printf("Exciting: %s.%n", excitingWords);
    List<String> excitingUpperCaseWords =
        transform2(words, makeExciting, makeUpperCase);
    System.out.printf("Exciting uppercase: %s.%n", excitingUpperCaseWords);
}
```

```
Original words: [hi, hello, hola, bye, goodbye, adios].
Uppercase: [HI, HELLO, HOLA, BYE, GOODBYE, ADIOS].
Exciting: [hi: Wow!, hello: Wow!, hola: Wow!, bye: Wow!, goodbye: Wow!, adios: Wow!].
Exciting uppercase: [HI: Wow!, HELLO: Wow!, HOLA: Wow!, BYE: Wow!, GOODBYE: Wow!, ADIOS: Wow!].
```

24

## Typing Issues with compose and Method References

- **Legal: two steps**

```
Function<Double,Double> round = Math::rint;  
transform(nums, round.compose(Math::sqrt));
```

- **Illegal: one step**

```
transform(nums, Math::rint.compose(Math::sqrt));
```

- **Question: why?**

- It looks like the same code both times

- **Answer: typing**

- Math::rint does not have a type until it is assigned to a variable or passed to a method. *Any* interface that has a single (abstract) method that can take two doubles could be the target for Math::rint.

- But, those other interfaces do not have “compose” method

25

coreservlets.com – custom onsite training



# Higher Order Functions in Consumer

Slides © 2016 Marty Hall, hall@coreservlets.com



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

## Method from Consumer

- **andThen**
  - `f1.andThen(f2)` produces a Consumer that first passes argument to `f1` (i.e., to its accept method), then passes argument to `f2`
  - Default method
- **Difference between andThen of Consumer and of Function**
  - With `andThen` from Consumer, the argument is passed to the accept method of `f1`, then *that same argument* is passed to the accept method of `f2`
  - With `andThen` from Function, the argument is passed to the apply method of `f1`, then *the result of apply* is passed to the apply method of `f2`

27

## Test Code: processEntries

- **Idea**
  - Given a list and a function, passes each list entry to the function, but does not return anything
    - Very similar to `forEach` method of Stream and of List, which we will cover later

- **Code**

```
public static <T> void processEntries(List<T> entries,
                                     Consumer<T> operation) {
    for (T e : entries) {
        operation.accept(e);
    }
}
```

28

## Using processEntries

```
public static void consumerExamples() {
    List<Employee> myEmployees =
        Arrays.asList(new Employee("Bill", "Gates", 1, 200000),
            new Employee("Larry", "Ellison", 2, 100000));
    System.out.println("Original employees:");
    processEntries(myEmployees, System.out::println);
    Consumer<Employee> giveRaise = e -> e.setSalary(e.getSalary() * 11 / 10);
    System.out.println("Employees after raise:");
    processEntries(myEmployees, giveRaise.andThen(System.out::println));
}
```

```
Original employees:
Bill Gates [Employee#1 $200,000]
Larry Ellison [Employee#2 $100,000]
Employees after raise:
Bill Gates [Employee#1 $220,000]
Larry Ellison [Employee#2 $110,000]
```

29

coreservlets.com – custom onsite training



# Higher Order Functions in Comparator

Slides © 2016 Marty Hall, hall@coreservlets.com



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

## Methods from Comparator

- **comparing**

- Static method that takes function that returns a key and builds a Comparator from it  
`Arrays.sort(words, Comparator.comparing(String::length));`

- **reversed**

- Default method that imposes the reverse ordering  
`Arrays.sort(words, Comparator.comparing(String::length)  
 .reversed());`

- **thenComparing**

- Default method that specifies how to break ties in the initial comparison  
`Arrays.sort(employees,  
 Comparator.comparing(Employee::getLastName)  
 .thenComparing(Employee::getFirstName));`

31

## Sorting: Comparing Approaches

- **Sorting with explicit Comparator**

```
Arrays.sort(words, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return(s1.length() - s2.length());  
    }  
});
```

- **Sorting with explicit lambda**

```
Arrays.sort(words, (s1, s2) -> s1.length() - s2.length());
```

- **Sorting with method that returns lambda**

```
Arrays.sort(words, Comparator.comparing(String::length));
```

32

## More Examples

```
private static Employee[] employees = { new Employee("John", "Doe", 1, 234_567),
                                         new Employee("Jane", "Doe", 2, 333_333),
                                         new Employee("Sammy", "Smith", 3, 99_000),
                                         new Employee("Sally", "Smith", 4, 99_000) };

private static String[] words =
    { "hi", "hello", "hola", "bye", "goodbye", "adios" };

public static void main(String[] args) {
    System.out.printf("Words before sorting: %s.%n", Arrays.asList(words));
    Arrays.sort(words, Comparator.comparing(String::length));
    System.out.printf("After sorting by length: %s.%n", Arrays.asList(words));
    Arrays.sort(words, Comparator.comparing(String::length).reversed());
    System.out.printf("After sorting by length (reversed): %s.%n", Arrays.asList(words));
    System.out.printf("Employees before sorting: %s.%n", Arrays.asList(employees));
    Arrays.sort(employees, Comparator.comparing(Employee::getLastName));
    System.out.printf("After sorting by last name: %s.%n", Arrays.asList(employees));
    Arrays.sort(employees, Comparator.comparing(Employee::getLastName).thenComparing(Employee::getFirstName));
    System.out.printf("After sorting by last name and then first name: %s.%n", Arrays.asList(employees));
    Arrays.sort(employees, Comparator.comparing(Employee::getSalary).thenComparing(Employee::getEmployeeId));
    System.out.printf("After sorting by salary and then ID: %s.%n", Arrays.asList(employees));
}
}
```

Entries that are tied by last name remain in their original relative order

Entries that are tied by last name are sorted by their first name

33

## Results

Words before sorting: [hi, hello, hola, bye, goodbye, adios].  
After sorting by length: [hi, bye, hola, hello, adios, goodbye].  
After sorting by length (reversed): [goodbye, hello, adios, hola, bye, hi].  
Employees before sorting:  
[John Doe [Employee#1 \$234,567], Jane Doe [Employee#2 \$333,333],  
Sammy Smith [Employee#3 \$99,000], Sally Smith [Employee#4 \$99,000]].  
After sorting by last name:  
[John Doe [Employee#1 \$234,567], Jane Doe [Employee#2 \$333,333],  
Sammy Smith [Employee#3 \$99,000], Sally Smith [Employee#4 \$99,000]].  
After sorting by last name and then first name:  
[Jane Doe [Employee#2 \$333,333], John Doe [Employee#1 \$234,567],  
Sally Smith [Employee#4 \$99,000], Sammy Smith [Employee#3 \$99,000]].  
After sorting by salary and then ID:  
[Sammy Smith [Employee#3 \$99,000], Sally Smith [Employee#4 \$99,000],  
John Doe [Employee#1 \$234,567], Jane Doe [Employee#2 \$333,333]].

34



# Higher Order Functions in Your Own Code

Slides © 2016 Marty Hall, hall@coreservlets.com



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

## Custom Methods that Return Lambdas

- **Idea**
  - Return a Predicate, Function, or other lambda from a method. But, embed a local variable before returning it. For example, return a Predicate that tests if an employee’s salary is above a certain cutoff. Pass the cutoff to the method.
- **Syntax options**
  - Use a regular method
    - Use normal “return” syntax, but have a lambda as the return value
  - Use a Function
    - Technically, this is a regular method (apply). But, you can use “double” lambda syntax: a lambda whose expression is another lambda

## Building a Predicate to Test for Salary Above a Cutoff

- **Regular method**

```
public static Predicate<Employee> buildIsRichPredicate(double salaryLowerBound) {  
    return(e -> e.getSalary() > salaryLowerBound);  
}
```

- Then call `buildIsRichPredicate(salary)` to get a Predicate

- **Function**

```
Function<Integer, Predicate<Employee>> makeIsRichPredicate =  
    salaryLowerBound -> (e -> e.getSalary() > salaryLowerBound);
```

- Then call `makeIsRichPredicate.apply(salary)` to get a Predicate

37

## Using the Tests

```
public static Predicate<Employee> buildIsRichPredicate(double salaryLowerBound) {  
    return(e -> e.getSalary() > salaryLowerBound);  
}  
  
public static void customHigherOrderFunctionExamples() {  
    List<Employee> richEmployees1 =  
        allMatches(employees, buildIsRichPredicate(200_000));  
    System.out.printf("Rich employees [via method that returns Predicate]: %s.%n",  
        richEmployees1);  
    Function<Integer, Predicate<Employee>> makeIsRichPredicate =  
        salaryLowerBound -> (e -> e.getSalary() > salaryLowerBound);  
    List<Employee> richEmployees2 =  
        allMatches(employees, makeIsRichPredicate.apply(200_000));  
    System.out.printf("Rich employees [via Function that returns Predicate]: %s.%n",  
        richEmployees2);  
}
```

38

## Results

Rich employees [via method that returns Predicate]:

```
[Harry Hacker [Employee#1 $234,567],  
 Polly Programmer [Employee#2 $333,333],  
 Desiree Designer [Employee#14 $212,000]].
```

Rich employees [via Function that returns Predicate]:

```
[Harry Hacker [Employee#1 $234,567],  
 Polly Programmer [Employee#2 $333,333],  
 Desiree Designer [Employee#14 $212,000]].
```

39

coreservlets.com – custom onsite training



# Wrap-Up

Slides © 2016 Marty Hall, hall@coreservlets.com



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

## Summary

- **Predicate**
  - Default methods: and, or, negate
  - Static method: isEqual
- **Function**
  - Default methods: andThen, compose
  - Static method: identity
- **Consumer**
  - Default method: andThen
- **Comparator**
  - Default methods: reversed, thenComparing
  - Static method: comparing
- **Custom higher-order functions**
  - Regular method that returns lambda or Function that returns lambda

41

coreservlets.com – custom onsite training



# Questions?

More info:

<http://courses.coreservlets.com/Course-Materials/java.html> – General Java programming tutorial

<http://www.coreservlets.com/java-8-tutorial/> – Java 8 tutorial

<http://courses.coreservlets.com/java-training.html> – Customized Java training courses, at public venues or onsite at your organization

<http://coreservlets.com/> – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training  
Many additional free tutorials at [coreservlets.com](http://coreservlets.com) (JSF, Android, Ajax, Hadoop, and lots more)

Slides © 2016 Marty Hall, [hall@coreservlets.com](mailto:hall@coreservlets.com)



For additional materials, please see <http://www.coreservlets.com/>. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.