



# Lambda Expressions in Java 8: Part 3

Originals of slides and source code for examples: <http://www.coreservlets.com/java-8-tutorial/>

Also see the general Java programming tutorial – <http://courses.coreservlets.com/Course-Materials/java.html>  
and customized Java training courses (onsite or at public venues) – <http://courses.coreservlets.com/java-training.html>



**Customized Java EE Training:** <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.



**For live Java-related training,  
email [hall@coreservlets.com](mailto:hall@coreservlets.com)**

**Marty is also available for consulting and development support**



**Taught by lead author of Core Servlets & JSP, co-author of Core JSF (4<sup>th</sup> Ed), & this tutorial. Available at public venues, or customized versions can be held on-site at your organization.**

- Courses developed and taught by Marty Hall
  - JSF 2.2, PrimeFaces, servlets/JSP, Ajax, jQuery, Android, Java 7 or 8 programming, GWT, custom mix of topics
  - Courses available in any state or country. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring MVC, Core Spring, Hibernate/JPA, Hadoop, HTML5, RESTful Web Services

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



# Topics in This Section

- **Variable scoping**
  - Lambdas are lexically scoped
- **Method references: details**
  - `Class::staticMethod`
  - `variable::instanceMethod`
  - `Class::instanceMethod`
  - `Class::new`
- **New features in Java 8 interfaces**
  - Default methods
  - Static methods
- **Methods that return lambdas**
  - From Predicate: `and`, `or`, `negate`, `isEqual`
  - From Function: `andThen`, `compose`, `identity`
  - From Consumer: `andThen`
  - Custom methods

4

© 2015 Marty Hall



## Variable Scoping



**Customized Java EE Training: <http://courses.coreservlets.com/>**

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

# Main Points

- **Lambdas are lexically scoped**
  - They do not introduce a new level of scoping
- **Implications**
  - The “this” variable refers to the outer class, not to the anonymous inner class that the lambda is turned into
  - There is no “OuterClass.this” variable
    - Unless lambda is inside a normal inner class
  - Lambdas cannot introduce “new” variables with same name as variables in method that creates the lambda
    - However, lambdas can refer to (but not modify) local variables from the surrounding code
  - Lambdas can still refer to (and modify) instance variables from the surrounding class

6

# Examples

- **Illegal: repeated variable name**  
`double x = 1.2;  
someMethod(x -> doSomethingWith(x));`
- **Illegal: repeated variable name**  
`double x = 1.2;  
someMethod(y -> { double x = 3.4; ... });`
- **Illegal: lambda modifying local var from the outside**  
`double x = 1.2;  
someMethod(y -> x = 3.4);`
- **Legal: modifying instance variable**  
`private double x = 1.2;  
public void foo() { someMethod(y -> x = 3.4); }`
- **Legal: local name matching instance variable name**  
`private double x = 1.2;  
public void bar() { someMethod(x -> x + this.x); }`

7



# Method References: Details



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Big Idea

- **Method references can be used for lambdas**
  - If there is a method whose signature matches the signature of the method of a functional (SAM) interface, then you can use `Class::method` (or something similar), rather than an explicit lambda.
- **Four variations**

Method Ref Type	Example	Equivalent Lambda
<code>SomeClass::staticMethod</code>	<code>Math::cos</code>	<code>x -&gt; Math.cos(x)</code>
<code>someVariable::instanceMethod</code>	<code>someString::toUpperCase</code>	<code>() -&gt; someString.toUpperCase()</code>
<code>SomeClass::instanceMethod</code>	<code>String::toUpperCase</code>	<code>s -&gt; s.toUpperCase()</code>
<code>SomeClass::new</code>	<code>Employee::new</code>	<code>() -&gt; new Employee()</code>



## var::instanceMethod vs. Class::instanceMethod

- **someVariable::instanceMethod**

- Produces a lambda that takes *exactly as many* arguments as the method expects.

```
String test = "PREFIX:";
```

```
List<String> result1 = transform(strings, test::concat);
```

- The concat method takes one arg
- This lambda takes one arg
- Lambda calls test.concat(entryFromList)

- **SomeClass::instanceMethod**

- Produces a lambda that takes *one more* argument than the method expects. The first argument is the object on which the method is called; the rest of the arguments are the parameters to the method.

```
List<String> result2=
```

```
transform(strings, String::toUpperCase);
```

- The toUpperCase method takes zero args
- This lambda takes one arg
- Lambda calls entryFromList.toUpperCase()

10

## Preview: Constructor References and toArray

- **Will soon see how to turn Stream into array**

- Employee[] employees =  
employeeStream.toArray(Employee[]::new);

- **This is a special case of a constructor ref**

- It takes an int as an argument, so you are calling “new Employee[n]” behind the scenes. This builds an empty Employee array, and then toArray fills it in with the elements of the Stream.

- **Most general form**

- toArray takes a lambda or method reference to anything that takes an int as an argument and produces an array of the right type and right length.
  - That array will then be filled in by toArray.

11

# Method For Testing

```
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);
}
```

This is basically a less-powerful version of the map method of Stream (which will be shown later).  
Used on next slide to test the use of method references.

12

# Method Reference Examples

```
List<Integer> mixedNums = Arrays.asList(1, -1, 2, -2);
List<Integer> positiveNums = transform(mixedNums, Math::abs);
System.out.println("Positive nums [via Class::staticMethod]: " +
                    positiveNums);

List<String> names = Arrays.asList("Joe", "John", "Jane");
String test = "PREFIX:";
List<String> mangledNames = transform(names, test::concat);
System.out.println("Mangled names [via variable::instanceMethod]: " +
                    mangledNames);

List<String> upperCaseNames = transform(names, String::toUpperCase);
System.out.println("Uppercase names [via Class::instanceMethod]: " +
                    upperCaseNames);

Supplier<Person> maker1 = () -> new Person("John", "Doe");
Supplier<Person> maker2 = Person::new;
System.out.printf("Person [via lambda for Supplier]: %s.%n",
                  maker1.get());
System.out.printf("Person [via constructor ref for Supplier]: %s.%n",
                  maker2.get());
```

13

# Results

```
Positive nums [via Class::staticMethod]:  
    [1, 1, 2, 2]  
Mangled names [via variable::instanceMethod]:  
    [PREFIX:Joe, PREFIX:John, PREFIX:Jane]  
Uppercase names [via Class::instanceMethod]:  
    [JOE, JOHN, JANE]  
Person [via lambda for Supplier]:  
    John Doe (Person).  
Person [via constructor ref for Supplier]:  
    Brad Carson (Person).
```

14

© 2015 Marty Hall



## Java 8 Interfaces: Default Methods and Static Methods



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Aside: Default Methods

- **Question: multiple methods in SAM interfaces**
  - Function has “apply”, so how can it also have compose, andThen, and identity?
  - Isn’t the whole point of functional (SAM) interfaces that they have a *single* method?
- **Answer: default and static methods**
  - Functional interfaces have a single *abstract* method (one that must be defined in class that implements interface). That is the method that the lambda specifies. But, interfaces in Java 8 can have default methods that have real method bodies and are inherited. Java 8 interfaces can also have static methods.
    - This makes Java 8 interfaces more like abstract classes. Now we can have multiple inheritance of implementation (mixins). This also means that, to avoid the “diamond problem”, you can no longer necessarily implement as many interfaces as you want. In particular, if a class implements two interfaces that have default methods with the same name, it must override that method or it won’t compile. In your own code, you can also use `InterfaceName.super.methodName(...)` to disambiguate.

16

## Source Code for Function Interface

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose
        (Function<? super V, ? extends T> before) {
        // Real code here
    }

    default <V> Function<T, V> andThen(...) { ... }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

The is the one and only method specification. As with interface methods in previous Java versions, it must be defined in classes that implement the interface.

These are default methods, a new feature of Java 8. Classes that implement the interface do not need to implement these methods, but will inherit both their names and their behavior. The classes can also override them. Interfaces are prohibited from having default methods for methods of Object (e.g., toString, equals).

Interfaces in Java 8 can now have static methods.

17





# Higher Order Functions: Methods that Return Lambdas



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## 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;  
allMatches(employees, isRich.and(isEarly))
```

# 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.

20

## Examples

```
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));
```

21

# 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]].
```

22

# 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.compose(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.

23

# 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.

- **Code for transform2**

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

- **Helper method**

```
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);  
}
```

24

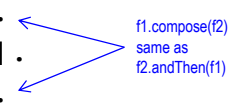
## Examples

```
List<Double> nums = Arrays.asList(2.0, 4.0, 6.0, 8.0);  
System.out.printf("Original nums:  %s.%n", nums);  
Function<Double,Double> square = x -> x * x;  
Function<Double,Double> half = x -> x / 2;  
System.out.printf("square.compose(half): %s.%n",  
    transform(nums, square.compose(half)));  
System.out.printf("square.andThen(half): %s.%n",  
    transform(nums, square.andThen(half)));  
System.out.printf("half.andThen(square): %s.%n",  
    transform(nums, half.andThen(square)));  
System.out.printf("square.compose(half) by transform2: %s.%n",  
    transform2(nums, square, half));  
System.out.printf("identity: %s.%n",  
    transform(nums, Function.identity()));  
Function<Double,Double> round = Math::rint;  
System.out.printf("round of square roots: %s.%n",  
    transform(nums, round.compose(Math::sqrt)));
```

25

# Results

```
Original nums: [2.0, 4.0, 6.0, 8.0].  
square.compose(half): [1.0, 4.0, 9.0, 16.0].  
square.andThen(half): [2.0, 8.0, 18.0, 32.0].  
half.andThen(square): [1.0, 4.0, 9.0, 16.0].  
square.compose(half) by transform2: [1.0, 4.0, 9.0, 16.0].  
identity: [2.0, 4.0, 6.0, 8.0].  
round of square roots: [1.0, 2.0, 2.0, 3.0].
```



A blue arrow points from the text "f1.compose(f2) same as f2.andThen(f1)" to the two lines: `square.compose(half): [1.0, 4.0, 9.0, 16.0].` and `half.andThen(square): [1.0, 4.0, 9.0, 16.0].`

26

## 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

27



# 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.
- **Note the 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`, and then *the result of apply* is passed to the `apply` method of `f2`.

28

# Examples

```
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));
```

29

# Results

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]

30

# 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

31

# Building a Predicate to Test for Salary Above a Cutoff

- **Regular method**

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

- **Function**

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

32

## Examples

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

---

```
List<Employee> mediumRichEmployees1 =  
    allMatches(employees, buildIsRichPredicate(150000));  
System.out.printf("Medium rich employees [via method that " +  
    "returns Predicate]: %s.%n",  
    mediumRichEmployees1);  
  
Function<Integer, Predicate<Employee>> makeIsRichPredicate =  
    salaryLowerBound -> (e -> e.getSalary() > salaryLowerBound);  
List<Employee> mediumRichEmployees2 =  
    allMatches(employees, makeIsRichPredicate.apply(150000));  
System.out.printf("Medium rich employees [via Function that " +  
    "returns Predicate]: %s.%n",  
    mediumRichEmployees2);
```

33

# Results

Medium rich employees [via method that returns Predicate]:

```
[Harry Hacker [Employee#1 $234,567],  
Polly Programmer [Employee#2 $333,333],  
Cody Coder [Employee#8 $199,999],  
Devon Developer [Employee#11 $175,000],  
Desiree Designer [Employee#14 $212,000],  
Tammy Tester [Employee#19 $166,777]].
```

Medium rich employees [via Function that returns Predicate]:

```
[Harry Hacker [Employee#1 $234,567],  
Polly Programmer [Employee#2 $333,333],  
Cody Coder [Employee#8 $199,999],  
Devon Developer [Employee#11 $175,000],  
Desiree Designer [Employee#14 $212,000],  
Tammy Tester [Employee#19 $166,777]].
```

34

© 2015 Marty Hall



## Wrap-Up



Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop

Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

# Variable Scoping: No New Level (“this” is Surrounding Class)

- **Illegal: repeated variable name**  
`double x = 1.2;  
someMethod(x -> doSomethingWith(x));`
- **Illegal: repeated variable name**  
`double x = 1.2;  
someMethod(y -> { double x = 5.6; ... });`
- **Illegal: lambda modifying local var from the outside**  
`double x = 1.2;  
someMethod(y -> x = 3.4);`
- **Legal: modifying instance variable**  
`private double x = 1.2;  
public void foo() { someMethod(y -> x = 3.4); }`
- **Legal: local name matching instance variable name**  
`private double x = 1.2;  
public void bar() { someMethod(x -> x + this.x); }`

36

## Method References

Method Ref Type	Example	Equivalent Lambda
<code>SomeClass::staticMethod</code>	<code>Math::cos</code>	<code>x -&gt; Math.cos(x)</code>
<code>someVariable::instanceMethod</code>	<code>someString::toUpperCase</code>	<code>() -&gt; someString.toUpperCase()</code>
<code>SomeClass::instanceMethod</code>	<code>String::toUpperCase</code>	<code>s -&gt; s.toUpperCase()</code>
<code>SomeClass::new</code>	<code>Employee::new</code>	<code>() -&gt; new Employee()</code>

- The only tricky one is `SomeClass::instanceMethod`. This builds a lambda where the first argument to the lambda is the target of the method and the other arguments are the parameters to the method.

37



# Methods that Return Lambdas

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

38

© 2015 Marty Hall



## 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)



39

**Customized Java EE Training: <http://courses.coreservlets.com/>**

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop

Developed and taught by well-known author and developer. At public venues or onsite at *your* location.