Lambda Expressions in Java 8: Part 1 – Basics

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

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.

λ

more SERVLETS and JAVASERVER PAGES
MARTY HALL
core SERVLETS and JAVASERVER PAGES
2ND EDITION
Volume 1: Core Technologies
MARTY HALL · LARRY BROWN

Topics in This Section

- **Intro**
  - Motivation
  - Quick summary of big idea
- **New option: lambdas**
  - Interpretation
  - Most basic form
  - Type inferencing
  - Expression for body
  - Omitting parens
  - Comparing lambda approaches to alternatives
- **Examples**
  - Numerical integration
  - Timing utilities

Motivation and Overview

Many Languages Let You Pass Functions Around

- **Dynamically (and usually weakly) typed**
  - JavaScript, Lisp, Scheme, etc.
- **Strongly typed**
  - Ruby, Scala, Clojure, ML, etc.
- **Functional approach proven concise, flexible, and parallelizable**
  - JavaScript sorting

```
var testStrings = ["one", "two", "three", "four"];
testStrings.sort(function(s1, s2) {
                   return(s1.length - s2.length);});
testStrings.sort(function(s1, s2) {
                   return(s1.charCodeAt(s1.length - 1) –
                          s2.charCodeAt(s2.length - 1));});
```

Why Lambdas in Java Now?

- **Concise syntax**
  - More succinct and clear than anonymous inner classes
- **Deficiencies with anonymous inner classes**
  - Bulky, confusion re “this” and naming in general, no access to non-final local vars, hard to optimize
- **Convenient for new streams library**
  - shapes.forEach(s -> s.setColor(Color.RED));
- **Programmers are used to the approach**
  - Callbacks, closures, map/reduce idiom

Surface Advantage of Lambdas: Concise and Expressive

- **Old**

```
button.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    doSomethingWith(e);
  }
});
```

- **New**

```
button.addActionListener(e -> doSomethingWith(e));
```

"Vigorous writing is concise... This requires not that the writer make all sentences short, or avoid all details and treat subjects only in outline, but that every word should tell." – Strunk and White, *The Elements of Style*.

Underlying Advantages: Support New Way of Thinking

- **Encourage functional programming**
  - When functional programming approach is used, many classes of problems are easier to solve and result in code that is clearer to read and simpler to maintain
    - Functional programming does not replace object-oriented programming in Java 8. OOP is still the main approach for representing types. But functional programming augments and improves many methods and algorithms.
- **Support streams**
  - Streams are wrappers around data sources (arrays, collections, etc.) that use lambdas, support map/filter/reduce, use lazy evaluation, and can be made parallel automatically.
    - Can*not* be made parallel automatically

      for(Employee e: employees) { e.giveRaise(1.15); }
    - *Will* automatically be run in parallel

      employees.stream().parallel().forEach(e -> e.giveRaise(1.15));

coreservlets.com – custom onsite training

Lambdas: Syntax

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.

Main Points

- **You write what looks like a function**
  - `Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());`
  - `taskList.execute(() -> downloadSomeFile());`
  - `someButton.addActionListener(event -> handleButtonClick());`
  - `double d = MathUtils.integrate(x -> x*x, 0, 100, 1000);`
- **You get an instance of a class that implements the interface that was expected in that place**
  - The expected type must be an interface that has *exactly one* (abstract) method
    - Called “Functional Interface” or “Single Abstract Method (SAM) Interface”
      - The designation of a single ABSTRACT method is not redundant, because in Java 8 interfaces can have concrete methods, called “default methods”. Java 8 interfaces can also have static methods.

Example: Sorting Strings by Length

- **Java 7 example**

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

- **Java 8 alternative**

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

The above is simply by replacing the anonymous inner class with a lambda, and no new capabilities are needed other than lambdas. But, Java 8 also added several sorting-related methods, one of which is Comparator.comparing.
So, the above could also be: Arrays.sort(testStrings, Comparator.comparing(String::length));
Method references like String::length are discussed in the next lambda section. Comparator.comparing and similar methods are discussed in section on lambdas and higher-order functions.

Getting There: Step 1 – Drop Interface and Method Names

- **Idea**
  - From the API, Java already knows the second argument to Arrays.sort is a Comparator, so you do not need to say it is a Comparator. Comparator has only one method, so you do not need to say that method name is compare.
  - Add “->” (i.e., dash then greater-than sign) between method params and method body
- **Java 7 example**

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

- **Java 8 alternative (legal, but not ideal)**

```
Arrays.sort(testStrings,
  (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

Getting There: Step 2 – Drop Parameter Type Declarations

- **Idea**
  - By looking at the first argument (testStrings), Java can infer that the type of the second argument is Comparator<String>. Thus, parameters for compare are both Strings. Since Java knows this, you do not need to say so.
  - Java is still doing strong, compile-time type checking. The compiler is just inferring types. Somewhat similar to how Java infers types for the diamond operator.
    - `List<String> words = new ArrayList<>();`
  - In a few cases, types are ambiguous, and compiler will warn you that it cannot infer the types. In that case, you can*not* drop the types declarations as below.
- **Previous version**

```
Arrays.sort(testStrings,
  (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

(In the previous version, both occurrences of ~~String~~ are struck through.)

- **Improved version (legal, but still not ideal)**

```
Arrays.sort(testStrings,
  (s1, s2) -> { return(s1.length() – s2.length()); });
```

Getting There: Step 3 – Use Expression Instead of Block

- **Idea**
  - If method body can be written as a single return statement, you can drop the curly braces and “return”, and just put the return value as an expression.
  - This cannot always be done, especially if you use loops or if statements. However, lambdas are most commonly used when the method body is short, so this can usually be done. If not, leaving curly braces and “return” is legal, but if body of lambda is long, you might want to reconsider and use normal inner class instead.
- **Previous version**

```
Arrays.sort(testStrings,
  (s1, s2) -> { return(s1.length() – s2.length()); });
```

(Struck through: ~~{ return(~~ and ~~); }~~)

- **Improved version (ideal)**

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

Optional Step 4 – Omit Parens When there is Exactly One Param

- **Idea**
  - If the method of the interface has *exactly* one parameter, the parens are optional
- **Java 7**

```
button.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    doSomethingWith(e);
  }
});
```

- **Java 8 with parens**

```
button.addActionListener((e) -> doSomethingWith(e));
```

- **Java 8 without parens**

```
button.addActionListener(e -> doSomethingWith(e));
```

Summary: Lambda Syntax

- **Omit interface and method names**

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

replaced by

```
Arrays.sort(testStrings, (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

- **Omit parameter types**

```
Arrays.sort(testStrings, (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

replaced by

```
Arrays.sort(testStrings, (s1, s2) -> { return(s1.length() – s2.length()); });
```

- **Use expressions instead of blocks**

```
Arrays.sort(testStrings, (s1, s2) -> { return(s1.length() – s2.length()); });
```

replaced by

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

- **Drop parens if single param to method**

```
button1.addActionListener((event) -> popUpSomeWindow(...));
```

replaced by

```
button1.addActionListener(event -> popUpSomeWindow(...));
```

Java 7 vs. Java 8

- **Java 7**

```
taskList.execute(new Runnable() {
   @Override
   public void run() {
     processSomeImage(imageName);
   }
 });
 button.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent event) {
     doSomething(event);
   }
 });
```

- **Java 8**

```
taskList.execute(() -> processSomeImage(imageName));
button.addActionListener(event -> doSomething(event));
```

Java vs. JavaScript

- **Java**

```
String[] testStrings = {"one", "two", "three", "four"};
Arrays.sort(testStrings,
            (s1, s2) -> s1.length() - s2.length());
Arrays.sort(testStrings,
            (s1, s2) -> s1.charAt(s1.length() - 1) –
                        s2.charAt(s2.length() - 1));
```

- **JavaScript**

```
var testStrings = ["one", "two", "three", "four"];
testStrings.sort(function(s1, s2) {
                   return(s1.length - s2.length);});
testStrings.sort(function(s1, s2) {
                 return(s1.charCodeAt(s1.length - 1) -
                        s2.charCodeAt(s2.length - 1));
                 });
```

Thinking About Lambdas

Under the Hood

- **You do this**
  - `Arrays.sort(testStrings, (s1, s2) -> s1.length() – s2.length());`
- **What really is happening**
  - You used a shortcut way of representing an instance of a class that implements Comparator<T>. You provided the body of the compare method after the "->".
- **How you usually think about it**
  - You passed in the comparison function
- **Function types**
  - Java 8 does *not* technically have function types, since under the hood, lambdas become instances of classes that implement whatever interface was expected. Nevertheless, you normally think of lambdas as functions.

Where Can Lambdas Be Used?

- **Find <u>any</u> variable or parameter that expects an interface that has one method**
    - Technically 1 abstract method, but in Java 7 there was no distinction between a 1-method interface and a 1-abstract-method interface. These 1-method interfaces are called “functional interfaces” or “SAM (Single Abstract Method) interfaces”.
  - public interface Blah { String foo(String someString); }
- **Code that <u>uses</u> interface is the same**
  - public void someMethod(**Blah** b) { … b.**foo**(…) …}
    - Code that uses the interface must still know the real method name in the interface
- **Code that <u>calls</u> the method that expects the interface can supply lambda**
  - someMethod(**s -> s.toUpperCase() + "!"**);

Example: Numerical Integration

Example: Numerical Integration

- **Goals**
  - Simple numerical integration using rectangle (mid-point) rule

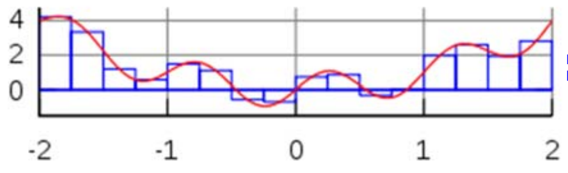


Diagram from
http://en.wikipedia.org/wiki/Numerical_integration

  - Want to use lambdas to make it convenient and succinct to supply the function that will be intetgrated
    - Need to define a functional (SAM) interface with a “double eval(double x)” method to specify function to be integrated

Interface

```
public interface Integrable {
  double eval(double x);
}
```

In later sections, we will upgrade this example.
- First, we will add the optional but useful @FunctionalInterface annotation.
- Second, we will observe that there is already a compatible interface built into the java.util.function package, and use it instead.

Numerical Integration Method

```
public static double integrate(Integrable function,
                               double x1, double x2,
                               int numSlices){
  if (numSlices < 1) {
    numSlices = 1;
  }
  double delta = (x2 - x1)/numSlices;
  double start = x1 + delta/2;
  double sum = 0;
  for(int i=0; i<numSlices; i++) {
    sum += delta * function.eval(start + delta * i);
  }
  return(sum);
}
```

Method for Testing

```
public static void integrationTest(Integrable function,
                                   double x1, double x2) {
  for(int i=1; i<7; i++) {
    int numSlices = (int)Math.pow(10, i);
    double result =
      MathUtilities.integrate(function, x1, x2, numSlices);
    System.out.printf("  For numSlices =%,10d result = %,.8f%n",
                      numSlices, result);
  }
}
```

Testing Results

```
MathUtilities.integrationTest(x -> x*x, 10, 100);
MathUtilities.integrationTest(x -> Math.pow(x,3), 50, 500);
MathUtilities.integrationTest(x -> Math.sin(x), 0, Math.PI);
MathUtilities.integrationTest(x -> Math.exp(x), 2, 20);
```

Output

```
Estimating integral of x^2 from 10.000 to 100.000.
Exact answer = 100^3/3 - 10^3/3.
                                 ~= 333,000.00000000
  For numSlices =        10 result = 332,392.50000000
  For numSlices =       100 result = 332,993.92500000
  For numSlices =     1,000 result = 332,999.93925000
  For numSlices =    10,000 result = 332,999.99939250
  For numSlices =   100,000 result = 332,999.99999393
  For numSlices = 1,000,000 result = 332,999.99999994

... // Similar for other three integrals
```

General Lambda Principles

- **Interfaces in Java 8 are same as in Java 7**
  - Integrable was the same as it would be in Java 7, except that you can (should) optionally use @FunctionalInterface
    - This annotation is covered in the next section
- **Code that *uses* interfaces is the same in Java 8 as in Java 7**
  - E.g., the definition of integrate is exactly the same as you would have written it in Java 7. The author of integrate must know that the real method name is eval.
- **Code that *calls* methods that expect 1-method interfaces can now use lambdas**

```
MathUtilities.integrate(x -> Math.sin(x), 0, Math.PI, …);
```

Instead of new Integrable() { public void eval(double x) { return(Math.sin(x)); } }

Making a Reusable Timing Utility

Timing

- **Goals**
  - Pass in a “function”
  - Run the function
  - Print elapsed time
- **Problem: Java evaluates args on the call**
  - TimingUtils.timeOp(someSlowCalculation());
    - The calculation is computed *before* timeOp is called!
- **Solution: use lambdas**
  - TimingUtils.timeOp(() -> someSlowCalculation());
    - timeOp can run the calculation internally
- **Could be done with inner classes**
  - And in fact we did so in fork-join section
  - But, code that called timeOp was long, cumbersome, and obtuse

The Op Interface

```
public interface Op {
  void runOp();
}
```

In later sections, we will upgrade this example.

- First, we will add the optional but useful @FunctionalInterface annotation.
- Second, since Java 8 interfaces can have static methods, we will move the static timeOp method from TimingUtils into this interface.
- Third, since Java 8 interfaces can have regular concrete methods ("default methods"), we will add a method called combinedOp that will let us turn two separate Ops into a single one.

The TimingUtils Class

```
public class TimingUtils {
  private static final double ONE_BILLION = 1_000_000_000;

  public static void timeOp(Op operation) {
    long startTime = System.nanoTime();
    operation.runOp();
    long endTime = System.nanoTime();
    double elapsedSeconds = (endTime - startTime)/ONE_BILLION;
    System.out.printf("  Elapsed time: %.3f seconds.%n", elapsedSeconds);
  }
}
```

Main Testing Code

```
public class TimingTests {
  public static void main(String[] args) {
    for(int i=3; i<8; i++) {
      int size = (int)Math.pow(10, i);
      System.out.printf("Sorting array of length %,d.%n", size);
      TimingUtils.timeOp(() -> sortArray(size));
    }
  }
```

Output

```
Sorting array of length 1,000.
  Elapsed time: 0.002 seconds.
Sorting array of length 10,000.
  Elapsed time: 0.004 seconds.
Sorting array of length 100,000.
  Elapsed time: 0.020 seconds.
Sorting array of length 1,000,000.
  Elapsed time: 0.148 seconds.
Sorting array of length 10,000,000.
  Elapsed time: 1.339 seconds.
```

Supporting Testing Code

```
  public static double[] randomNums(int length) {
    double[] nums = new double[length];
    for(int i=0; i<length; i++) {
      nums[i] = Math.random();
    }
    return(nums);
  }

  public static void sortArray(int length) {
    double[] nums = randomNums(length);
    Arrays.sort(nums);
  }
```

Final Lambda Examples

A Few More Samples

- As arguments to methods

```
Arrays.sort(testStrings,
            (s1, s2) -> s1.length() - s2.length());
taskList.execute(() -> downloadSomeFile());
button.addActionListener(event -> handleButtonClick());
double d = MathUtils.integrate(x -> x*x, 0, 100, 1000);
```

A Few More Samples (Continued)

- As variables (makes real type more obvious)

```
AutoCloseable c = () -> cleanupForTryWithResources();
Thread.UncaughtExceptionHandler handler =
  (thread, exception) -> doSomethingAboutException();
Formattable f =
  (formatter, flags, width, precision) -> makeFormattedString();
ContentHandlerFactory fact =
  mimeType -> createContentHandlerForMimeType();
CookiePolicy policy =
  (uri, cookie) -> decideIfCookieShouldBeAccepted();
Flushable toilet = () -> writeBufferedOutputToStream();
TextListener t = event -> respondToChangeInTextValue();
```

Wrap-Up

Summary: Big Ideas

- **Yay! We have lambdas**
  - Concise and succinct
  - Retrofits into existing APIs
  - Familiar to developers that know functional programming
  - Fits well with new streams API
  - Also have method references and prebuilt functional interfaces

- **Boo! We do not have full functional programming (?)**
  - Type of a lambda is class that implements interface, not a "real" function
    - Must create or find interface first, must know method name
  - Cannot use mutable local variables

Summary: Syntax

- **Replace this use of an anonymous inner class**

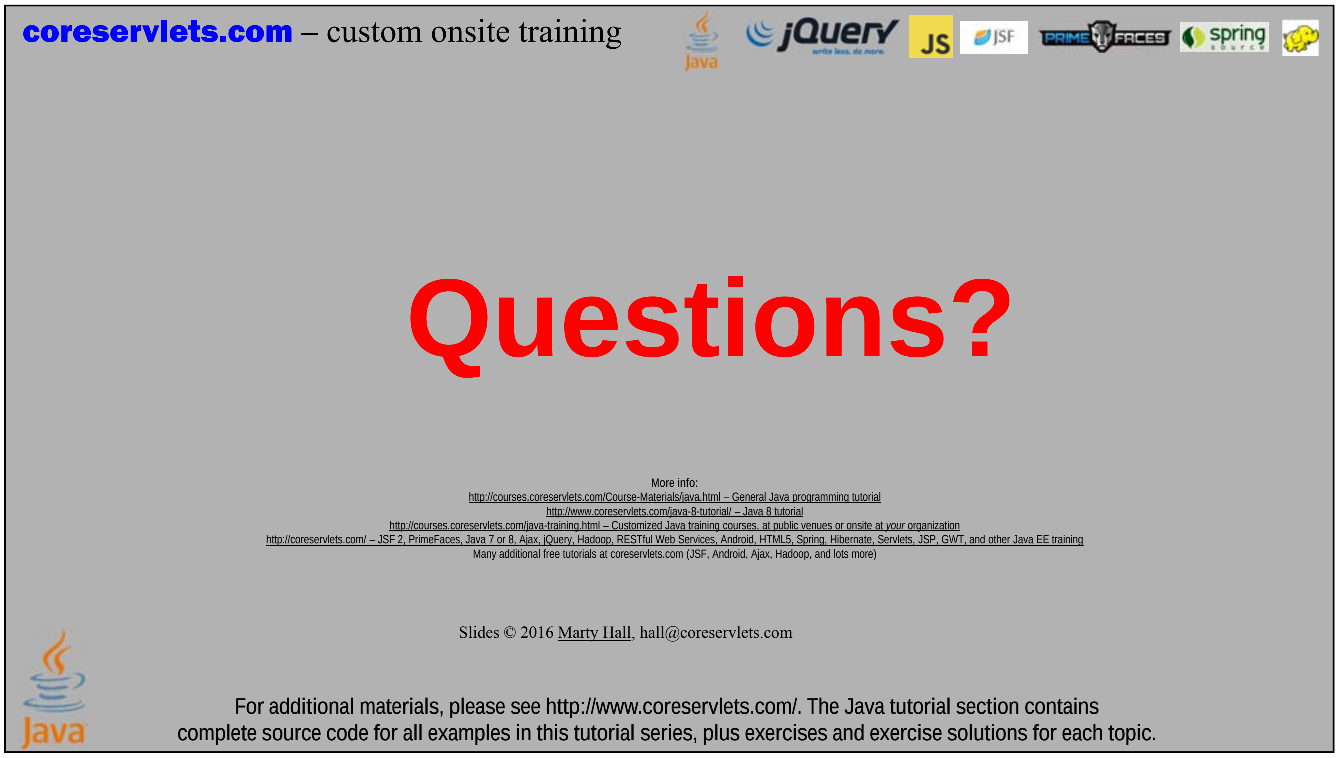
```
doSomething(new SomeOneMethodInterface() {
  @Override
  public SomeType methodOfInterface(args) {
    return(value);
  }
});
```

- **With this use of a lambda**

```
doSomething((args) -> value);
```

  - And, if method has exactly one parameter, you can omit parens around the args

- **Defining a one-method interface to use with lambdas**
  - Interface itself is almost the same as Java 7 (except for annotation we will see later)
  - Code that *uses* the interface is exactly the same as in Java 7
  - Code that *calls* method that expects the interface type can now use lambda



coreservlets.com – custom onsite training

Questions?