Lambda Expressions in Java 8: Part 1 – Basics

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

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
  - Using effectively final variables
  - @FunctionalInterface
  - Method references

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));
- **Similar constructs used in other languages**
  - Callbacks, closures, map/reduce idiom

Main 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*

Corollary Advantages: Support New Way of Thinking

- **Encourage functional programming**
  - When functional programming approach is used, many (not all!) classes of problems are easier to solve and result in code that is clearer to read and simpler to maintain
- **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 by compiler.
    - Can*not* be made parallel automatically

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

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

Lambdas: Most Basic Form

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. Both default methods and static methods are covered in later section.

Simplest Form: Syntax Summary

- **Replace this**

```
new SomeInterface() {
    @Override
    public SomeType someMethod(args) { body }
}
```

- **With this**

```
(args) -> { body }
```


Simplest Form: Example

- **Old style**

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

- **New style**

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

Sorting Strings by Length

- **Old version**

```
String[] testStrings =
  {"one", "two", "three", "four"};
...
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.length() - s2.length());
  }
});
...
```

In both cases, resultant array is {"one", "two", "four", "three"}

- **New version**

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


Sorting Strings by Last Char

- **Old version**

```
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.charAt(s1.length() - 1) –
           s2.charAt(s2.length() - 1));
  }
});
```

In both cases, resultant array is {"one", "three", "two", "four"}

- **New version**

```
Arrays.sort(testStrings,
   (String s1, String s2) -> {
      return(s1.charAt(s1.length() - 1) –
             s2.charAt(s2.length() - 1)); });
```

Where Can Lambdas Be Used?

- **Find any 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 uses interface is the same**
  - public void someMethod(**Blah** b) { … b.**foo**(…)…}
    - Code that uses the interface must still know the real method name of the interface
- **Code that calls interface can supply lambda**
  - String result = someMethod(**s -> s.toUpperCase() + "!"**)

Type Inferencing

Main Points

- **Types in argument list can usually be omitted**
  - Since Java usually already knows the expected parameter types for the *single* method of the functional interface (SAM interface)
- **Basic lambda**

  (Type1 var1, Type2 var2 …) -> { method body }
- **Lambda with type inferencing**

  (var1, var2 …) -> { method body }

Syntax Summary

- **Replace this**

```
new SomeInterface() {
    @Override
    public SomeType someMethod(T1 v1, T2 v2) {
        body
    }
}
```

- **With this**

```
(v1, v2) -> { body }
```

Example

- **Old style**

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

- **New style**

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


Sorting Strings by Length

- **Old version**

```
String[] testStrings =
  {"one", "two", "three", "four"};
...
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.length() - s2.length());
  }
});
...
```

In both cases, resultant array is {"one", "two", "four", "three"}

- **New version**

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

Sorting Strings by Last Char

- **Old version**

```
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.charAt(s1.length() - 1) –
           s2.charAt(s2.length() - 1));
  }
});
```

- **New version**

```
Arrays.sort(testStrings, (s1, s2) -> {
            return(s1.charAt(s1.length() - 1) –
                   s2.charAt(s2.length() - 1));  });
```

In both cases, resultant array is {"one", "three", "two", "four"}

Expression for Body: Implied Return Values

Main Points

- **For body, use expression instead of block**
  - Value of expression will be the return value, with no explicit "return" needed
    - If method has a void return type, then automatically no return value
    - Since lambdas are usually used only when method body is short, this approach (using expression instead of block) is very common
- **Previous version**

  (var1, var2 …) -> { return(something); }
- **Lambda with expression for body**

  (var1, var2 …) -> something

Syntax Summary

- **Replace this**

```
new SomeInterface() {
    @Override
    public SomeType someMethod(T1 v1, T2 v2) {
        return(someValue);
    }
}
```

- **With this**

```
(v1, v2) -> someValue
```

Example

- **Old style**

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

- **New style**

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


Sorting Strings by Length

- **Old version**

```
String[] testStrings =
  {"one", "two", "three", "four"};
...
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.length() - s2.length());
  }
});
...
```

In both cases, resultant array is {"one", "two", "four", "three"}

- **New version**

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

Sorting Strings by Last Char

- **Old version**

```
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.charAt(s1.length() - 1) –
           s2.charAt(s2.length() - 1));
  }
});
```

- **New version**

```
Arrays.sort(testStrings,
            (s1, s2) -> s1.charAt(s1.length() - 1) –
                        s2.charAt(s2.length() - 1));
```

In both cases, resultant array is {"one", "three", "two", "four"}

Omitting Parens

Main Points

- **If method takes single arg, parens optional**
  - No type should be used: you must let Java infer the type
    - But omitting types is normal practice anyhow
- **Previous version**

  (varName) -> someResult()
- **Lambda with parentheses omitted**

  varName -> someResult()

Syntax Summary

- **Replace this**

```
new SomeInterface() {
    @Override
    public SomeType someMethod(T1 var) {
        return(someValue);
    }
}
```

- **With this**

```
var -> someValue
```

Example (Listeners for Buttons)

- **Old style**

```
button1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
      setBackground(Color.BLUE);
    }
  });
```

New style

```
button1.addActionListener(event -> setBackground(Color.BLUE));
```


Buttons: Old Version

```
button1.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent event) {
    setBackground(Color.BLUE);
  }
});
button2.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent event) {
    setBackground(Color.GREEN);
  }
});
button3.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent event) {
    setBackground(Color.RED);
  }
});
```

Buttons: New Version

```
button1.addActionListener(event -> setBackground (Color.BLUE));
button2.addActionListener(event -> setBackground (Color.GREEN));
button3.addActionListener(event -> setBackground (Color.RED));
```

These examples do not use the ActionEvent argument (i.e., "event" above), but the lambda must still declare the variable so that the method signature of the lambda matches the method signature of the method in ActionListener (i.e., actionPerformed).

Summary: Making Lambdas More Succinct

Shortening Lambda Syntax

- **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 argument**

```
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));
                  });
```


Effectively Final Local Variables

Main Points

- **Lambdas can refer to local variables that are not declared final (but are never modified)**
  - This is known as "effectively final" – variables where it *would have been* legal to declare them final
  - You can still refer to mutable *instance* variables
    - "this" in a lambda refers to main class, not inner class that was created for the lambda. There is no OuterClass.this. Also, no new level of scoping. More on scoping later.
- **With explicit declaration**

  final String s = "...";
  doSomething(someArg -> use(s));
- **Effectively final**

  String s = "...";
  doSomething(someArg -> use(s));

Example: Button Listeners

```
public class SomeClass ... {
  private Container contentPane;

  private void someMethod() {
    button1.addActionListener
       (event -> contentPane.setBackground(Color.BLUE));
    Color b2Color = Color.GREEN;
    button2.addActionListener(event -> setBackground(b2Color));
    button3.addActionListener(event -> setBackground(Color.RED));
    ...
  }
  ...
}
```

Instance variable: same rules as with anonymous inner classes in older Java versions; they can be modified.

Local variable: need not be explicitly declared final, but cannot be modified; i.e., must be "effectively final".

The @FunctionalInterface Annotation

Review: @Override

- **What is benefit of @Override?**

```
public class MyServlet extends HttpServlet
  @Override
  public void doget(...) ... { ... }
}
```

- **Correct code will work with or without @Override, but @Override useful**
  - Catches errors at compile time
    - Real method is doGet, not doget
  - Expresses design intent
    - Tells fellow developers this is a method that came from parent class, so HttpServlet API will describe it

New: @FunctionalInterface

- **Catches errors at compile time**
  - If developer later adds a second (abstract) method, interface will not compile
- **Expresses design intent**
  - Tells fellow developers that this is interface that you expect lambdas to be used for
- **But, not technically required**
  - You can use lambdas *anywhere* 1-abstract-method interfaces (aka functional interfaces, SAM interfaces) are expected, whether or not that interface used @FunctionalInterface

Example: Numerical Integration

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

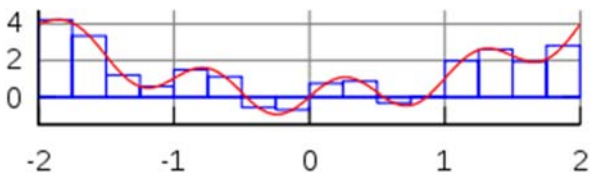


Diagram from
http://en.wikipedia.org/wiki/Numerical_integration

  - Want to use lambdas for function to be integrated. Convenient and succinct.
    - Define functional (SAM) interface with a "double eval(double x)" method to specify function to be integrated
  - Want compile-time checking that interface is in right form (exactly one abstract method), and want to alert other developers that lambdas can be used
    - Use @FunctionalInterface

Interface

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


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);
  }
}
```

Also define a simple method for printing out the expected answer based on strings based in. Full code can be downloaded from http://www.coreservlets.com/java-8-tutorial/

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 Principle

- **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
    - To catch errors (multiple methods) at compile time
    - To express design intent (developers should use lambdas)
- **Code that uses interfaces is the same in Java 8 as in Java 7**
  - I.e., 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)); } }

Method References

Main Points

- **Can use ClassName::staticMethodName or variable::instanceMethodName for lambdas**
  - E.g., Math::cos or myVar::myMethod
    - Another way of saying this is that if the function you want to describe already has a name, you don't have to write a lambda for it, but can instead just use the method name.
  - The function must match signature of method in functional (SAM) interface to which it is assigned
    - You can also use Class:instanceMethod and Class::new. These are more complicated; details online at coreservlets.com.
- **The type is found only from the context**
  - The type of a method reference depends on what it is assigned to. This is always true with lambdas, but more obvious here.
    - E.g., there *is* no predefined type for Math::cos.

Example: Numerical Integration

- **In previous example, replace these**

```
MathUtilities.integrationTest(x -> Math.sin(x), 0, Math.PI);
MathUtilities.integrationTest(x -> Math.exp(x), 2, 20);
```

- **With these**

```
MathUtilities.integrationTest(Math::sin, 0, Math.PI);
MathUtilities.integrationTest(Math::exp, 2, 20);
```

People often ask "what is the type of a method reference"? The answer is "this is not known until you try to assign it to a variable, in which case its type is whatever interface that variable expected". So, for example, Math::sin could be different types in different contexts, but all the types would be single-method interfaces whose method could accept a single double as an argument and return a double.

What is the Type of a Lambda?

- **Interfaces** (like Java 7)
  - public interface Foo { double method1(double d); }
  - public interface Bar { double method2(double d); }
  - public interface Baz { double method3(double d); }
- **Methods that use the interfaces** (like Java 7)
  - public void blah1(Foo f) { … f.method1(…)… }
  - public void blah2(Bar b) { … b.method2(…)… }
  - public void blah3(Baz b) { … b.method3(…)… }
- **Calling the methods** (use λs or method refs)
  - blah1(Math::cos) *or* blah1(d -> Math.cos(d))
  - blah2(Math::sin) *or* blah2(d -> Math.sin(d))
  - blah3(Math::log) *or* blah3(d -> Math.log(d))

Final Examples

Concurrent Image Download

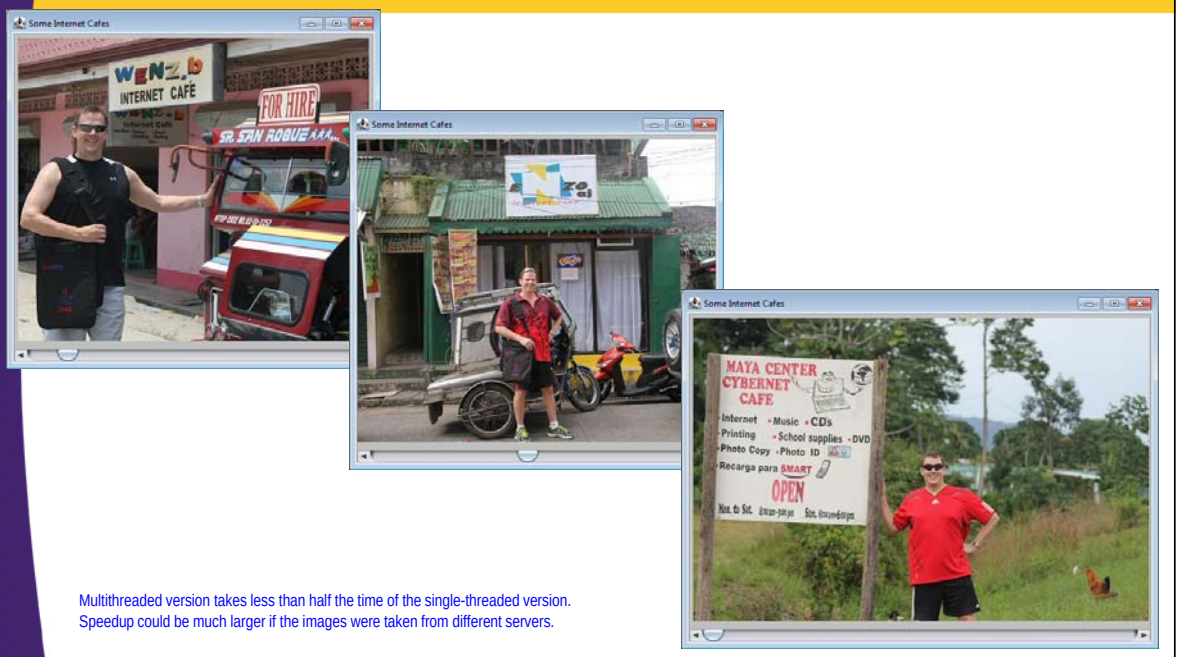
- **Idea**
  - Use standard Java threading to download a series of images of internet cafes and display them in a horizontally scrolling window
- **Java 8 twists**
  - Because ExecutorService.execute expects a Runnable, and because Runnable is a functional (SAM) interface, use lambdas to specify the body of the code that runs in background
  - Have code access local variables (which are effectively final but not explicitly declared final)

Main Code

```
...
ExecutorService taskList =
  Executors.newFixedThreadPool(poolSize);
for(int i=1; i<=numImages; i++) {
  JLabel label = new JLabel();
  URL location = new URL(String.format(imagePattern, i));
  taskList.execute(() -> {
    ImageIcon icon = new ImageIcon(location);
    label.setIcon(icon);
  });
  imagePanel.add(label);
}
...
```

Full code can be downloaded from
http://www.coreservlets.com/java-8-tutorial/

Results



Multithreaded version takes less than half the time of the single-threaded version.
Speedup could be much larger if the images were taken from different servers.

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)

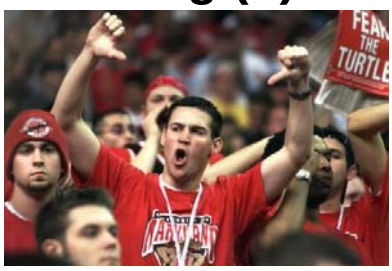
- **As variables (makes real type more obvious)**
  - AutoCloseable c = () -> cleanupForTryWithResources();
  - Thread.UncaughtExceptionHandler handler =
    (thread, exception) -> doSomethingAboutException();
  - Formattable f =
    (formatter,flags,width,precision) -> makeFormattedString();
  - ContentHandlerFactory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

Wrap-Up

Summary

- **Yay: we have lambdas**
  - Concise and succinct
  - Retrofits in existing APIs
  - Familiar to developers that know functional programming
  - Fits well with new streams API
  - Also have method refs and prebuilt functional interfaces

- **Boo: not full functional programming (?)**
  - Type 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

- **Most basic form**

  (String s1, String s2) -> { return(s1.length() - s2.length()); }
- **Type inferencing**

  (s1, s2) -> { return(s1.length() - s2.length()); }
- **Expressions for body**

  (s1, s2) -> s1.length() - s2.length()
- **Omitting parens**

  event -> doSomethingWith(event)
- **More**
  - Method references (Math::cos)
  - Mark your interfaces with @FunctionalInterface
  - Can use "effectively final" local vars

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 (JSF, Android, Ajax, Hadoop, and lots more)