



Lambda Expressions in Java 8: Part 1 – Basics

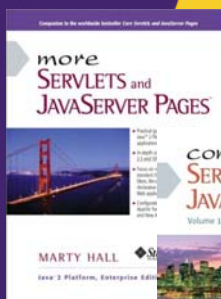
Originals of slides and source code for examples: <http://www.coreservlets.com/java-8-tutorial/>

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>



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
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**

Marty is also available for consulting and development support



Taught by lead author of *Core Servlets & JSP*, co-author of *Core JSF* (4th 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 development, Java 7 or 8 programming, 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, Hibernate/JPA, GWT, Hadoop, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details



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
 - The java.util.function package

4

© 2014 Marty Hall



Motivation and Overview



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.

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

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, useful, 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 non-final 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
- **Step toward true functional programming**
 - Real functions and mutable local vars perhaps in future

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*

8

Corollary 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

- **Support streams**

- Streams are wrappers around data sources (arrays, collections, etc.) that use lambdas, support map/reduce, use lazy evaluation, and can be made parallel automatically by compiler.

- *Cannot* 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));`

9



Lambdas: Most Basic Form



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

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.

Where Can Lambdas Be Used?

- Find any variable or parameter that expects an interface that has one method
 - (Technically one abstract method, but in Java 7 there was no distinction between a one-method interface and a one-abstract-method interface. These one-method interfaces are called “functional interfaces” or “SAM interfaces”.)
 - public interface Blah { String foo(String s); }
- 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()** + "!!")

12

Previews

- 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);

13

Previews (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 =
contentType -> **createContentHandlerForMimeType();**
 - CookiePolicy policy =
(uri, cookie) -> **decideIfCookieShouldBeAccepted();**
 - Flushable toilet = () -> **writeBufferedOutputToStream();**
 - TextListener t = **event** -> **respondToChangeInTextValue();**

14

Simplest Form: Syntax Summary

- **Replace this**

```
new SomeInterface() {  
    @Override  
    public SomeType someMethod(args) { body }  
}
```
- **With this**
(args) -> { **body** }

15

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());  
});
```

16

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()); });
```

17

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)); }));
```

18

© 2014 Marty Hall



Type Inferencing



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.

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

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 }

20

Syntax Summary

- **Replace this**
new SomeInterface() {
 @Override
 public SomeType someMethod(T1 v1, T2 v2) {
 body
 }
}
- **With this**
(v1, v2) -> { body }

21

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()); });
```

22

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()); });
```

23

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

24

© 2014 Marty Hall



Expression for Body: Implied Return Values



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.

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

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

26

Syntax Summary

- **Replace this**
new SomeInterface() {
 @Override
 public SomeType someMethod(T1 v1, T2 v2) {
 return(someValue);
 }
}
- **With this**
(v1, v2) -> someValue

27

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());
```

28

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());
```

29

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,  
    (s1, s2) -> s1.charAt(s1.length() - 1) -  
        s2.charAt(s2.length() - 1));
```

30

© 2014 Marty Hall



Omitting Parens



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.

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

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()`

32

Syntax Summary

- **Replace this**

```
new SomeInterface() {  
    @Override  
    public SomeType someMethod(T1 var) {  
        return(someValue);  
    }  
}
```
- **With this**
`var -> someValue`

33

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));
```

34

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);  
    }  
});
```

35

Buttons: New Version

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

36

© 2014 Marty Hall



Summary: Making Lambdas More Succinct



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.

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

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());
```

38

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));
```

39

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));
});
```

40

© 2014 Marty Hall



Effectively Final Local Variables



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.

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

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));
```

42

Example: Button Listeners

```
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

Local variable: need not be explicitly declared final, but must be “effectively final”.

43



The @FunctionalInterface Annotation



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

Review: @Override

- What is benefit of @Override?

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

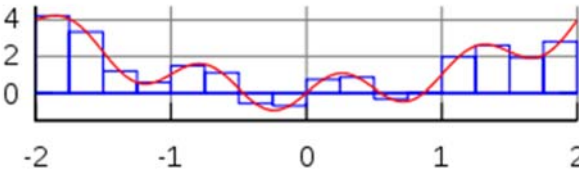
- Correct code will work with or without @Override, but @Override useful
 - Catches errors at compile time
 - Real method is doGet, not doget
 - Expresses design intent
 - Tells fellow developers this is a method that came from parent class, so HttpServlet API will describe it

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-method interfaces (aka functional interfaces, SAM interfaces) are expected, whether or not that interface used @FunctionalInterface

46

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

47

Interface

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

48

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);
}
```

49

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/>

50

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

51

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 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, ...);`

52

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

© 2014 Marty Hall



Method References



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

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))`

56

© 2014 Marty Hall



The `java.util.function` Package



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.

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

Main Points

- **Interfaces like Integrable widely used**
 - So, Java 8 should build in many common cases
- **Can be used in wide variety of contexts**
 - So need more general name than “Integrable”
- **java.util.function defines many simple functional (SAM) interfaces**
 - Named according to arguments and return values
 - E.g., replace my Integrable with builtin DoubleUnaryOperator
 - You need to look in API for the method names
 - Although lambdas don't refer to method names, your code that *uses* the lambdas will need to call the methods

Typed and Generic Interfaces

- **Types given**
 - Samples (*many* others!)
 - IntPredicate (int in, boolean out)
 - LongUnaryOperator (long in, long out)
 - DoubleBinaryOperator (two doubles in, double out)
 - Example
 - DoubleBinaryOperator f = (d1, d2) -> Math.cos(d1 + d2);
- **Genericized**
 - There are also generic interfaces (Function<T,R>, Predicate<T>) with widespread applicability
 - And concrete methods like “compose” and “negate”
 - These are covered in later tutorial section
 - Very important, but more complex than typed interfaces

Example: Numerical Integration

- **In previous example, replace this**

```
public static double integrate(Integrable function, ...) {  
    ... function.eval(...); ...  
}
```

 - With this

```
public static double integrate(DoubleUnaryOperator function, ... ) {  
    ... function.applyAsDouble(...); ...  
}
```
- **Then, omit definition of Integrable entirely**
 - Because DoubleUnaryOperator is a functional (SAM) interface containing a method with same signature as the method of the Integrable interface

General Case

- **If you are tempted to create an interface purely to be used as a target for a lambda**
 - Look through java.util.function and see if one of the functional (SAM) interfaces there can be used instead
 - DoubleUnaryOperator, IntUnaryOperator, LongUnaryOperator
 - double/int/long in, same type out
 - DoubleBinaryOperator, IntBinaryOperator, LongBinaryOperator
 - Two doubles/int/longs in, same type out
 - DoublePredicate, IntPredicate, LongPredicate
 - double/int/long in, boolean out
 - DoubleConsumer, IntConsumer, LongConsumer
 - double/int/long in, void return type
 - Genericized interfaces: Function, Predicate, Consumer, etc.
 - Covered in later tutorial section



Final Example



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

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)

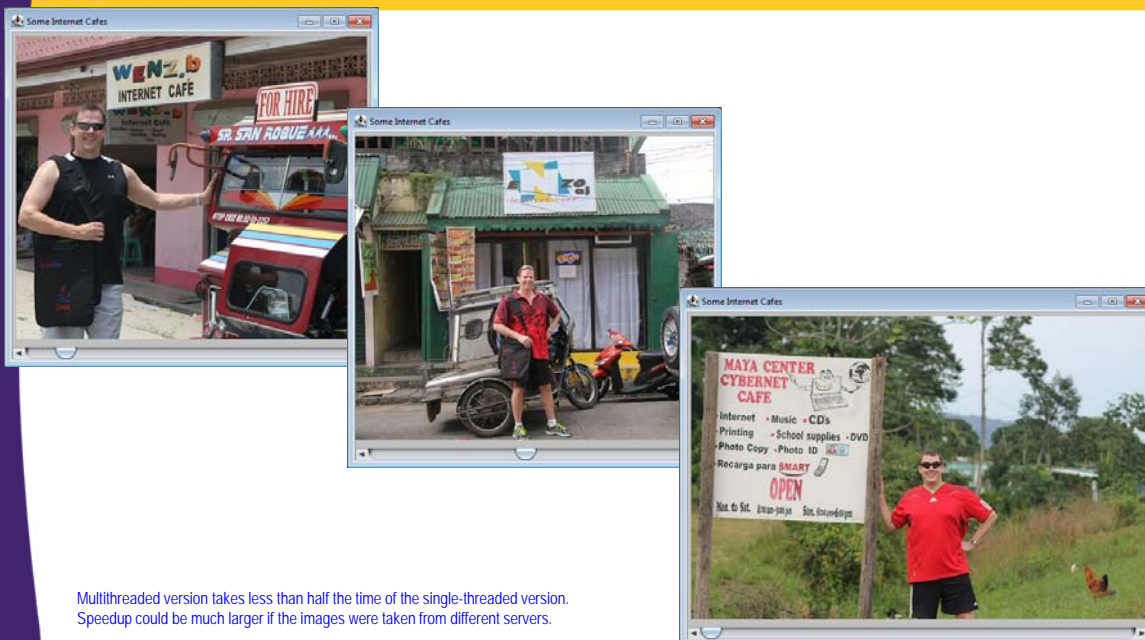
Main Code

```
...
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/>

64

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.

65



Wrap-Up

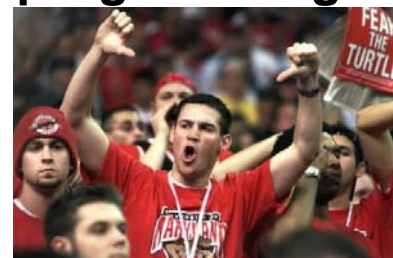


Customized Java EE Training: <http://courses.coreservlets.com/>

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

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: still not real 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



Samples Revisited

- **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);

68

Samples Revisited

- **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();

69

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
 - Many builtin functional (SAM) interfaces in java.util.function

70

© 2014 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



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

Java 7, Java 8, JSF 2.2, PrimeFaces, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.

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

71