Lambda Expressions in Java 8: Part 2 – More Details

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

coreservlets.com – custom onsite training

For customized training related to Java or JavaScript, please email hall@coreservlets.com

Marty is also available for consulting and development support

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

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

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

Contact hall@coreservlets.com for details

Topics in This Section

- The @FunctionalInterface annotation
- Method references
- Lambda scoping rules
- Effectively final local variables

The @FunctionalInterface Annotation

Review: @Override

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

```
public class MyCoolClass {
  @Override
  public String tostring() { ... }
}
```

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

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, like @Override 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

Interface Used in Numerical Integration Example

- **Last section**

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

- **Updated**

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


Interface Used in Timing Utilities Example

- **Last section**

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

- **Updated**

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

Method References

Basic Method References

- **Simplest type: static methods**
  - Replace
    ```
    (args) -> ClassName.staticMethodName(args)
    ```
  - with
    ```
    ClassName::staticMethodName
    ```
    - E.g., Math::cos, Arrays::sort, String::valueOf
  - 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 signature of the method you refer to must match signature of the method in functional (SAM) interface to which it is assigned
- **Other method references described later**
  - variable::instanceMethod (e.g., System.out::println)
  - Class::instanceMethod (e.g., String::toUpperCase)
  - ClassOrType::new (e.g., String[]::new)

Example: Numerical Integration

- **In earlier 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);
```


The Type of Method References

- **Question: what is type of Math::sin?**
  - Double? Function? Math?
- **Answer: can determine from context only**
  - The right question to ask would have been "what is the type of Math::sin *in code below*?"
    - MathUtilities.integrationTest(Math::sin, 0, Math.PI);
  - We can answer this the same way we answer any question about the type of an argument to a method: by looking at the API.
  - Conclusion: type here is Integrable
    - But in another context, Math::sin could be something else!
- **This point applies to all lambdas, not just method references**
  - The type can be determined only from context

The Type of Lambdas or Method References

- **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 references)
  - blah1(Math::cos) *or* blah1(d -> Math.cos(d))
  - blah2(Math::cos) *or* blah2(d -> Math.cos(d))
  - blah3(Math::cos) *or* blah3(d -> Math.cos(d))
    - All the above could also use Math::sin, Math::log, Math::sqrt, Math::abs, etc.

Importance of Using Method References

- **Low!**
  - If you do not understand method references, you can always use explicit lambdas
  - Replace foo(Math::cos) with foo(d -> Math.cos(d))
  - Replace bar(System.out::println) with bar(s -> System.out.println(s))
  - Replace baz(Class::twoArgMethod) with (a, b) -> Class.twoArgMethod(a, b)
- **But method references are popular**
  - More succinct
  - Familiar to developers from several other languages, where you can refer directly to existing functions. E.g., in JavaScript

```
function square(x) { return(x*x); }
var f = square;
f(10); → 100
```

The Four Kinds of Method References

Method Ref Type	Example	Equivalent Lambda
SomeClass::staticMethod	Math::cos	x -> Math.cos(x)
someObject::instanceMethod	someString::toUpperCase	() -> someString.toUpperCase()
SomeClass::instanceMethod	String::toUpperCase	s -> s.toUpperCase()
SomeClass::new	Employee::new	() -> new Employee()

var::instanceMethod vs. Class::instanceMethod

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

```
String test = "PREFIX:";
String result1 = transform(someString, test::concat);
```

- The concat method takes one arg
- This lambda takes one arg, passing s as argument to test.concat
- Equivalent lambda is s -> test.concat(s)

- **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.

```
String result2= transform(someString, String::toUpperCase);
```

- The toUpperCase method takes zero args
- This lambda takes one arg, invoking toUpperCase on that argument
- Equivalent lambda is s -> s.toUpperCase()

Method Reference Examples: Helper Interface

```
@FunctionalInterface
public interface StringFunction {
  String applyFunction(String s);
}
```


Method Reference Examples: Helper Class

```
public class Utils {
  public static String transform(String s, StringFunction f) {
    return(f.applyFunction(s));
  }

  public static String makeExciting(String s) {
    return(s + "!!");
  }

  private Utils() {}
}
```

Method Reference Examples: Testing Code

```
public static void main(String[] args) {
  String s = "Test";

  // SomeClass::staticMethod
  String result1 = Utils.transform(s, Utils::makeExciting); Test!!
  System.out.println(result1);

  // someObject::instanceMethod
  String prefix = "Blah";
  String result2 = Utils.transform(s, prefix::concat);      BlahTest
  System.out.println(result2);

  // SomeClass::instanceMethod
  String result3 = Utils.transform(s, String::toUpperCase); TEST
  System.out.println(result3);
}
```


Preview: Constructor References

- **In Java 7, difficult to randomly choose which class to create**
  - Suppose you are populating an array of random shapes, and sometimes you want a Circle, sometimes a Square, and sometimes a Rectangle
  - It requires tedious code to do this, since constructors cannot be bound to variables
- **In Java 8, this is simple**
  - Make array of constructor references and choose one at random
    - `{ Circle::new, Square::new, Rectangle::new }`
  - This will be more clear once we introduce the Supplier type, which can refer to a constructor reference

Preview: Making Random Person

```
private final static Supplier[] peopleGenerators =
  { Person::new, Writer::new, Artist::new, Consultant::new,
    EmployeeSamples::randomEmployee,
    () -> { Writer w = new Writer();
            w.setFirstName("Ernest");
            w.setLastName("Hemingway");
            w.setBookType(Writer.BookType.FICTION);
            return(w); }
  };

public static Person randomPerson() {
  Supplier<Person> generator =
    RandomUtils.randomElement(peopleGenerators);
  return(generator.get());
}
```


Preview: Array Constructor References

- **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 in the array 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

Variable Scoping in Lambdas

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 method
  - Lambdas can still refer to (and modify) instance variables from the surrounding class

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); }
```

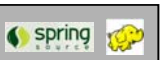
- **Legal: local name matching instance variable name**

```
private double x = 1.2;
public void bar() {  someMethod(x -> x + this.x); }
```









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.
- **With explicit declaration (explicitly final)**

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

- **Effectively final (without explicit declaration)**

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

  - Note the rule where the use of "final" is optional also applies in Java 8 to anonymous inner classes

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".

Example: Concurrent Image Download

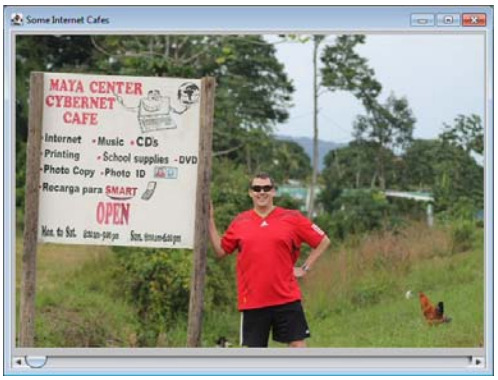
- **Idea**
  - Use standard Java threading to download a series of images of internet cafes and display them in a horizontally scrolling window
- **Java 8 twists**
  - Because ExecutorService.execute expects a Runnable, and because Runnable is a functional (SAM) interface, use lambdas to specify the 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.

Wrap-Up

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

Summary

- **@FunctionalInterface**
  - Use for all interfaces that will permanently have only a single abstract method
- **Method references**
  - `arg -> Class.method(arg)` → `Class::method`
- **Variable scoping rules**
  - Lambdas do not introduce a new scoping level
  - "this" always refers to main class
- **Effectively final local variables**
  - Lambdas can refer to, but not modify, local variables from the surrounding method
  - These variables need not be explicitly declared final as in Java 7
  - This rule (cannot modify the local variables but they do not need to be declared final) applies also to anonymous inner classes in Java 8