Lambda-Related Methods Directly in Lists and Maps

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

- **List**
  - forEach (applies to all Iterables)
  - removeIf (applies to all Collections)
  - replaceAll
  - sort
- **Map**
  - forEach
  - computeIfAbsent (and compute, computeIfPresent)
  - merge
  - replaceAll

Overview

- **Lists and other collections**
  - Have methods that are shortcuts for very similar Stream methods
  - Often modify the existing List, unlike the Stream versions
  - With very large Lists, the new methods might have small performance advantages vs. the similar Stream methods
- **Maps**
  - Have methods that significantly extend their functionality vs. Java 7
  - No equivalent Stream methods

Lists: Overview of New Java 8 Methods

Summary

- **forEach**
  - Identical to forEach for Streams, but saves you from calling "stream()" first
- **removeIf**
  - Like filter with negation of the Predicate, but removeIf modifies the original List
- **replaceAll**
  - Like map, but replaceAll modifies the original List
  - Also, with replaceAll, the Function must map values to the same type as in List
- **sort**
  - Takes Comparator just like stream.sorted and Arrays.sort

forEach (Applies to All Iterables)

- **Basic syntax**
  - `someList.forEach(someConsumer)`
    - employeeList.forEach(System.out::println)
- **Equivalent Stream code**
  - `someList.stream().forEach(someConsumer)`
    - employeeList.stream().forEach(System.out::println)
- **Advantages**
  - Slightly shorter code
  - Same performance
- **Disadvantages**
  - None

removeIf (Applies to All Collections)

- **Basic syntax**
  - `someList.removeIf(somePredicate)`
    - stringList.removeIf(s -> s.contains("q"))
- **Equivalent Stream code**
  - ```
    someList = someList.stream().filter(somePredicate.negate())
                                .collect(Collectors.toList())
    ```
    - stringList = stringList.stream().filter(s -> !s.contains("q")).collect(Collectors.toList())
      - If you want to be sure the new List is same concrete type as old one, then you should do ... collect(Collectors.toCollection(YourListType::new))
- **Advantages**
  - Shorter code if you want to modify the original List
  - Possible slight performance gain for very large Lists
- **Disadvantages**
  - Longer code if you want to result to be new List
    ```

replaceAll

- **Basic syntax**
  - `someList.replaceAll(someUnaryOperator)`
    - stringList.replaceAll(String::toUpperCase)
- **Equivalent Stream code**
  - ```
    someList = someList.stream().map(someFunction)
                                .collect(Collectors.toList())
    ```
    - stringList = stringList.stream().map(String::toUpperCase).collect(Collectors.toList())
- **Advantages**
  - Shorter code if you want to modify the original List
  - Possible slight performance gain for very large Lists
- **Disadvantages**
  - Longer code if you want to result to be new List
  - replaceAll must map to same type as entries in original List, whereas map can produce streams of totally different types

sort

- **Idea**
  - `someList.sort(someComparator)`
    - employeeList.sort(Comparator.comparing(Employee::getLastName))
- **Equivalent Stream method**
  - ```
    someList = someList.stream().sorted(someComparator)
                                .collect(Collectors.toList())
    ```
    - stringList =
      stringList.stream().sorted(Comparator.comparing(Employee::getLastName))
      .collect(Collectors.toList())
- **Advantages**
  - Shorter code if you want to modify the original List
  - Large performance gain for very large LinkedLists (not for ArrayLists)
- **Disadvantages**
  - Longer code if you want to result to be new List

Lists: Performance Comparisons

Overview

- **Questions**
  - Because removeIf, replaceAll, and sort modify the original List, will they be faster than streaming the List and accumulating the result into a new List?
- **Preview of answers**
  - removeIf: no
  - replaceAll: possibly yes for LinkedList, no for ArrayList
  - sort: possibly yes for LinkedList, no for ArrayList
- **Caution**
  - Results did not show clear trends; they should be viewed skeptically
    - Results may depend on memory usage and details of current Java release
  - What *is* clear is that for sizes below about a million entries, there was no measurable performance difference, at least with the current Java release
    - Conclusion: use the List methods for convenience (because they modify existing List), not for performance reasons

Shared Code for Examples

```
public class PerformanceTests {
  private static LinkedList<Integer> list1;
  private static ArrayList<Integer> list2;
  private static LinkedList<Integer> list3;
  private static ArrayList<Integer> list4;
  private static String message =
    "%s entries in %s with %,d elements.%n";
```


Shared Code for Examples Continued

```
  private static void initializeLists(int size) {
    list1 = new LinkedList<>();
    fillList(list1, size);
    list2 = new ArrayList<>(size);
    fillList(list2, size);
    list3 = new LinkedList<>();
    fillList(list3, size);
    list4 = new ArrayList<>();
    fillList(list4, size);
  }

  private static void fillList(List<Integer> nums, int size) {
    for(int i=0; i<size; i++) {
      nums.add(i);
    }
  }
```

removeIf: Code

```
private static void profileRemoveIf() {
   int size = 1_000_000;
   for(int i=0; i<4; i++) {
     initializeLists(size);
     Predicate<Integer> isEven = n -> n%2 == 0;
     Predicate<Integer> isOdd = n -> n%2 != 0;
     System.out.printf(message, "Removing even", "LinkedList", size);
     Op.timeOp(() -> list1.removeIf(isEven));
     System.out.printf(message, "Removing even", "ArrayList", size);
     Op.timeOp(() -> list2.removeIf(isEven));
     System.out.printf(message, "Streaming and filtering even", "LinkedList", size);
     Op.timeOp(() -> list3 = list3.stream().filter(isOdd)
                              .collect(Collectors.toCollection(LinkedList::new)));
     System.out.printf(message, "Streaming and filtering even", "ArrayList", size);
     Op.timeOp(() -> list4 = list4.stream().filter(isOdd)
                              .collect(Collectors.toCollection(ArrayList::new)));
     size = size * 2;
     System.out.println();
   }
}
```


removeIf: Representative Results

```
Removing even entries in LinkedList with 1,000,000 elements.
  Elapsed time: 0.023 seconds.
Removing even entries in ArrayList with 1,000,000 elements.
  Elapsed time: 0.027 seconds.
Streaming and filtering even entries in LinkedList with 1,000,000 elements.
  Elapsed time: 0.024 seconds.
Streaming and filtering even entries in ArrayList with 1,000,000 elements.
  Elapsed time: 0.017 seconds.

... (Similar for 2,000,000 and 4,000,000)

Removing even entries in LinkedList with 8,000,000 elements.
  Elapsed time: 0.067 seconds.
Removing even entries in ArrayList with 8,000,000 elements.
  Elapsed time: 0.053 seconds.
Streaming and filtering even entries in LinkedList with 8,000,000 elements.
  Elapsed time: 0.065 seconds.
Streaming and filtering even entries in ArrayList with 8,000,000 elements.
  Elapsed time: 0.064 seconds.
```

No consistent measurable difference for either List type on any sized List.

replaceAll: Code

```
private static void profileReplaceAll() {
  int size = 1_000_000;
  for(int i=0; i<4; i++) {
    initializeLists(size);
    UnaryOperator<Integer> doubleIt = n -> n*2;
    System.out.printf(message, "Doubling", "LinkedList", size);
    Op.timeOp(() -> list1.replaceAll(doubleIt));
    System.out.printf(message, "Doubling", "ArrayList", size);
    Op.timeOp(() -> list2.replaceAll(doubleIt));
    System.out.printf(message, "Streaming and doubling", "LinkedList", size);
    Op.timeOp(() -> list3 = list3.stream().map(doubleIt)
                             .collect(Collectors.toCollection(LinkedList::new)));
    System.out.printf(message, "Streaming and doubling", "ArrayList", size);
    Op.timeOp(() -> list4 = list4.stream().map(doubleIt)
                             .collect(Collectors.toCollection(ArrayList::new)));
    size = size * 2;
    System.out.println();
  }
}
```


replaceAll: Representative Results

```
Doubling entries in LinkedList with 1,000,000 elements.
  Elapsed time: 0.484 seconds.
Doubling entries in ArrayList with 1,000,000 elements.
  Elapsed time: 0.013 seconds.
Streaming and doubling entries in LinkedList with 1,000,000 elements.
  Elapsed time: 0.173 seconds.
Streaming and doubling entries in ArrayList with 1,000,000 elements.
  Elapsed time: 0.026 seconds.

... (Similar for 2,000,000 and 4,000,000)

Doubling entries in LinkedList with 8,000,000 elements.
  Elapsed time: 0.072 seconds.
Doubling entries in ArrayList with 8,000,000 elements.
  Elapsed time: 0.554 seconds.
Streaming and doubling entries in LinkedList with 8,000,000 elements.
  Elapsed time: 7.746 seconds.
Streaming and doubling entries in ArrayList with 8,000,000 elements.
  Elapsed time: 1.739 seconds.
```

Results were highly variable, but average results seemed worse for streaming case with LinkedList when List sizes were very large. No measurable difference for when List sizes were 2 million or less. Might depend on specific Java release.

sort: Code

```
private static void profileSort() {
  int size = 1_000_000;
  for(int i=0; i<4; i++) {
    initializeLists(size);
    Comparator<Integer> evensFirst = (i1, i2) -> (i1 % 2) - (i2 % 2);
    System.out.printf(message, "Sorting", "LinkedList", size);
    Op.timeOp(() -> list1.sort(evensFirst));
    System.out.printf(message, "Sorting", "ArrayList", size);
    Op.timeOp(() -> list2.sort(evensFirst));
    System.out.printf(message, "Streaming and sorting", "LinkedList", size);
    Op.timeOp(() -> list3 = list3.stream().sorted(evensFirst)
                             .collect(Collectors.toCollection(LinkedList::new)));
    System.out.printf(message, "Streaming and sorting", "ArrayList", size);
    Op.timeOp(() -> list4 = list4.stream().sorted(evensFirst)
                             .collect(Collectors.toCollection(ArrayList::new)));
    size = size * 2;
    System.out.println();
  }
}
```


sort: Representative Results

```
Sorting entries in LinkedList with 1,000,000 elements.
  Elapsed time: 0.117 seconds.
Sorting entries in ArrayList with 1,000,000 elements.
  Elapsed time: 0.037 seconds.
Streaming and sorting entries in LinkedList with 1,000,000 elements.
  Elapsed time: 0.064 seconds.
Streaming and sorting entries in ArrayList with 1,000,000 elements.
  Elapsed time: 0.058 seconds.

... (2,000,000 and 4,000,000 showed slightly worse for streaming with LinkedList)

Sorting entries in LinkedList with 8,000,000 elements.
  Elapsed time: 0.410 seconds.
Sorting entries in ArrayList with 8,000,000 elements.
  Elapsed time: 0.283 seconds.
Streaming and sorting entries in LinkedList with 8,000,000 elements.
  Elapsed time: 8.637 seconds.
Streaming and sorting entries in ArrayList with 8,000,000 elements.
  Elapsed time: 0.610 seconds.
```

Results were highly variable, but average results seemed worse for streaming case with LinkedList when List sizes were very large.

Maps: Overview of New Java 8 Methods

Summary

- **forEach(function)**
  – Similar to forEach for Stream and List, except function takes *two* arguments: the key and the value
- **replaceAll(function)**
  – For each Map entry, passes the key and the value to the function, takes the output, and replaces the old value with it
    - Similar to replaceAll for List, except function takes *two* arguments: key and value
- **merge(key, initialValue, function)**
  – If no value is found for the key, store the initial value
  – Otherwise pass old value and initial value to the function, and overwrite current result with that output
- **computeIfAbsent(key, function)**
  – If value is found for the key, return it
  – Otherwise pass the key to the function, store the output in Map, and return it

forEach and replaceAll

forEach and replaceAll

- **forEach(function): idea**
  - For each Map entry, passes the key and the value to the function
- **Mini example**

```
map.forEach((key, value) ->
                        System.out.printf("(%s,%s)%n", key, value));
```

- **replaceAll(function): idea**
  - For each Map entry, passes the key and the value to the function, then replaces existing value with that output
- **Mini example**

```
shapeAreas.replaceAll((shape, area) -> Math.abs(area));
```

Map Printing Utility

```
public class MapUtils {
  public static <K,V> void printMapEntries(Map<K,V> map) {
    map.forEach((key, value) -> System.out.printf("(%s,%s)%n", key, value));
  }
}
```


Example Usage

```
public class NumberMap {
  private static Map<Integer,String> numberMap = new HashMap<>();

  static {
    numberMap.put(1, "uno");
    numberMap.put(2, "dos");
    numberMap.put(3, "tres");
  }

 public static void main(String[] args) {
    MapUtils.printMapEntries(numberMap);
    numberMap.replaceAll((number, word) -> word.toUpperCase());
    MapUtils.printMapEntries(numberMap);
  }
}
```

```
(1,uno)
(2,dos)
(3,tres)
(1,UNO)
(2,DOS)
(3,TRES)
```

merge

merge(key, initialValue, function)

- **Idea**
  - Lets you update existing values
    - If no value is found for the key, store the initial value
    - Otherwise pass old value and initial value to the function, and overwrite current result with that updated result
- **Mini example (creates message or adds it on end of old one)**

```
messages.merge(key, message, (old, initial) -> old + initial);
```

  - Or, equivalently

```
messages.merge(key, message, String::concat);
```

merge: Example Usage

- **Idea**
  - Given an array of ints, produce Map that has counts of how many times each entry appeared
  - For example, if you have a very large array of grades on exams (all from 0-100), you can use this to sort them in O(N), instead of a comparison-based sort that is O(N log(N)).
- **Code snippet**

```
Map<Integer,Integer> counts = new HashMap<>();
for(int num: nums) {
  counts.merge(num, 1, (old, initial) -> old + 1);
}
```


Full Code and Results

```
public class CountEntries {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 3, 3, 3, 4, 2, 2, 1 };
    MapUtils.printMapEntries(countEntries(nums));
  }

  public static Map<Integer,Integer> countEntries(int[] nums) {
    Map<Integer,Integer> counts = new HashMap<>();
    for(int num: nums) {
      counts.merge(num, 1, (old, initial) -> old + 1);
    }
    return(counts);
  }
}
```

```
(1,2)
(2,3)
(3,4)
(4,1)
```

computeIfAbsent

computeIfAbsent(key, function)

- **Idea**
  – Lets you remember previous computations
    - If value is found for the key, return it
    - Otherwise pass the key to the function, store the output in Map, and return it
  – If this technique is applied to entire result of a method call, it is known as *memoization* (like memorization without the "r"). Memoization applied to recursive functions that have overlapping subproblems results in automatic dynamic programming and can provide huge performance gains.
- **General format for 1-argument methods**
  – First, make a Map that maps argument types to return types. Then:

```
public static ReturnType memoizedMethod(arg) {
  return(map.computeIfAbsent(arg, val -> codeForOriginalMethod(val));
}
```

Simple Example: Prime Numbers

- **Idea**
  - Use my Primes utility class to find primes of a given size. Once you find an n-digit prime, remember it and return it next time that you are asked for one of that size.
- **Original version**

```
public static BigInteger findPrime1(int numDigits) {
  return(Primes.findPrime(numDigits));
}
```

  - Finding large (e.g., 200-digit) primes is expensive, so if you do not need new results each time, this is wasteful.
- **Memoized version (after making Map<Integer,BigInteger>)**

```
public static BigInteger findPrime(int numDigits) {
  return(primes.computeIfAbsent(numDigits, n -> Primes.findPrime(n)));
}
```

  - First time you ask for prime of size n: calculates it. Second time: returns old result.

Full Code for Memoized Version

```
public class PrimeMap {
  private static Map<Integer,BigInteger> primes = new HashMap<>();

  public static void main(String[] args) {
    List<Integer> lengths = Arrays.asList(2, 10, 20, 100, 200);
    System.out.println("First pass");
    lengths.forEach(size ->
       System.out.printf("  %3s-digit prime: %s.%n", size, findPrime(size)));
    System.out.println("Second pass");
    lengths.forEach(size ->
       System.out.printf("  %3s-digit prime: %s.%n", size, findPrime(size)));
  }

  public static BigInteger findPrime(int numDigits) {
    return(primes.computeIfAbsent(numDigits, n -> Primes.findPrime(n)));
  }
}
```

Results

```
First pass
    2-digit prime: 37.
   10-digit prime: 9865938581.
   20-digit prime: 81266731015996542377.
  100-digit prime: 1303427...3109.
  200-digit prime: 8579052...2809.
Second pass
    2-digit prime: 37.
   10-digit prime: 9865938581.
   20-digit prime: 81266731015996542377.
  100-digit prime: 1303427...3109.
  200-digit prime: 8579052...2809.
```

On first pass, it took significant time to find the 100-digit and 200-digit primes. On second pass, results were almost instantaneous since the previous results were simply returned from the Map.

Example 2: Fibonacci Numbers

- **Idea**
  - Make a recursive function to calculate numbers in the sequence 0, 1, 1, 2, 3, 5, 8...
    - Fib(0) = 0
    - Fib(1) = 1
    - Fib(n) = Fib(n-1) + Fib(n-2)
- **Problem**
  - The straightforward recursive version has overlapping subproblems: when computing Fib(n-1), it computes Fib(n-2), yet it repeats that computation when finding Fib(n-2) directly. This results in exponential complexity.
- **Solutions**
  - In this case, you can build up from the bottom with an iterative version.
    - But, for many other problems, it is hard to find non-recursive solution, and that solution is far more complex than the iterative one. E.g., recursive-descent parsing, finding change with fewest coins, longest common substring, many more.
  - Better solution: use recursive version, then memoize it with computeIfAbsent

Unmemoized Version: $O(2^n)$

```
public static int fib1(int n) {
  if (n <= 1) {
    return(n);
  } else {
    return(fib1(n-1) + fib1(n-2));
  }
}
```


Memoized Version: O(n) First Time and O(1) for Later Calculations

```
private static Map<Integer,Integer> fibMap = new HashMap<>();

public static int fib(int num) {
  return
    fibMap.computeIfAbsent(num, n -> {
      if (n <= 1) {
        return(n);
      } else {
        return(fib(n-1) + fib(n-2));
      }
    });
}
```

An even better version would make the output long or even BigInteger, since Fibonacci numbers will quickly overflow ints.

Test Case

```
public static void profileFib() {
  for(i=0; i<47; i++) {
    Op.timeOp(() -> System.out.printf("fib1(%s)= %s.%n", i, fib1(i)));
    Op.timeOp(() -> System.out.printf("fib(%s) = %s.%n", i, fib(i)));
  }
}
```

```
fib1(0)= 0.
  Elapsed time: 0.001 seconds.
fib(0) = 0.
  Elapsed time: 0.001 seconds.
fib1(1)= 1.
  Elapsed time: 0.000 seconds.
fib(1) = 1.
  Elapsed time: 0.000 seconds.
...
fib1(44)= 701408733.
  Elapsed time: 3.269 seconds.
fib(44) = 701408733.
  Elapsed time: 0.000 seconds.
fib1(45)= 1134903170.
  Elapsed time: 5.293 seconds.
fib(45) = 1134903170.
  Elapsed time: 0.000 seconds.
fib1(46)= 1836311903.
  Elapsed time: 8.572 seconds.
fib(46) = 1836311903.
  Elapsed time: 0.000 seconds.
```


Wrap-Up

Summary

- **Lists**
  - forEach
    - Identical to forEach for Streams, but saves you from calling “stream()” first
  - removeIf
    - Like filter with negation of the Predicate, but removeIf modifies the original List
  - replaceAll
    - Like map, but replaceAll modifies the original List
  - sort
    - Takes Comparator just like stream.sorted and Arrays.sort, and modifies original List
- **Maps**
  - forEach(function) and replaceAll(function)
    - Similar to versions for List, except function takes *two* arguments: the key and the value
  - merge(key, initialValue, function)
    - Lets you update old values
  - computeIfAbsent(key, function)
    - Lets you make memoized functions that remember previous calculations

Questions?

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.