Streams in Java 8: Part 3

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.

Topics in This Section

- **Parallel streams**
  - Why stream approach is often better than traditional loops
  - Best practices: verifying speedup and "same" answer
  - Knowing when parallel reduce is safe
- **Infinite streams**
  - Really unbounded streams with values that are calculated on the fly
- **Grouping stream elements**
  - Fancy uses of collect

Parallel Streams

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.

Parallel Streams

- **Big idea**
  - By designating that a Stream be parallel, the operations are automatically done in parallel, without any need for explicit fork/join or threading code
- **Designating streams as parallel**
  - `anyStream.parallel()`
  - `anyList.parallelStream()`
    - Shortcut for anyList.stream().parallel()
- **Quick examples**
  - Do side effects serially
    - `someStream.forEach(someThreadSafeOp);`
  - Do side effects in parallel
    - `someStream.parallel().forEach(someThreadSafeOp);`

Parallel Streams vs. Concurrent Programming with Threads

- **Parallel streams**
  - Use fork/join framework internally (see separate lecture)
  - Have one thread per core
  - Are beneficial even when you never wait for I/O
  - Have no benefit on single-core computers
  - Can be used with minimal changes to serial code
- **Explicit threads (see separate lecture)**
  - Are often beneficial even on single-core computers
  - May have a lot more threads than there are cores
  - Require very large changes to serial code
- **Bottom line**
  - Parallel streams are often *far* easier to use than explicit threads
  - Explicit threads sometimes give you better control, and they apply in more situations

Advantage of Using Streams vs. Traditional Loops: Parallel Code

- **Computing sum with normal loop**

```
int[] nums = …;
int sum = 0;
for(int num: nums) {
    sum += num;
}
```

  - *Cannot* be easily parallelized

- **Computing sum with reduction operation**

```
int[] nums = …;
int sum = IntStream.of(nums).sum();
```

  - *Can* be easily parallelized

```
int sum2 = IntStream.of(nums).parallel().sum();
```

Best Practices: Both Fork/Join and Parallel Streams

- **Check that you get the same answer**
  - Verify that sequential and parallel versions yield the same (or close enough to the same) results. This can be harder than you think when dealing with doubles.
- **Check that the parallel version is faster**
  - Or, at the least, no slower. Test in real-life environment.
  - If running on app server, this analysis might be harder than it looks. Servers automatically handle requests concurrently, and if you have heavy server load and many of those requests use parallel streams, all the cores are likely to *already* be in use, and parallel execution might have little or no speedup.
    - Another way of saying this is that if the CPU of your server is already consistently maxed out, then parallel streams will not benefit you (and could even slow you down).

Will You Always Get Same Answer in Parallel?

- **sorted, min, max**
  - No. (Why not? And do you care?)
- **findFirst**
  - No. Use findAny if you do not care.
- **map, filter**
  - No, but you do not care about what the stream looks like in the middle; you only care about the terminal operation
- **allMatch, anyMatch, noneMatch, count**
  - Yes
- **reduce (and sum and average)**
  - It depends!

Equivalence of Parallel Reduction Operations

- **sum, average are the same if**
  - You use IntStream or LongStream
- **sum, average could be different if**
  - You use DoubleStream. Reording the additions could yield slightly different answers due to roundoff error.
    - "Almost the same" may or may not be acceptable
    - If the answers differ, it is not clear which one is "right"
- **reduce is the same if**
  - No side effects on global data are performed
  - The combining operation is associative (i.e., where reordering the operations does not matter).
    - Reordering addition or multiplication of doubles does not necessarily yield exactly the same answer. You may or may not care.

Parallel reduce: No Global Data

- **Binary operator itself should be stateless**
  - Guaranteed if an explicit lambda is used, but not guaranteed if you directly build an instance of a class that implements BinaryOperator, or if you use a method reference that refers to a statefull class
- **The operator does not modify global data**
  - The body of a lambda is allowed to mutate instance variables of the surrounding class or call setter methods that modify instance variables. If you do so, there is no guarantee that parallel reduce will be safe.

Parallel reduce: Associative Operation

- **Reordering ops should have no effect**
  - I.e., (a op b) op c == a op (b op c)
- **Examples: are these associative?**
  - Division? No.
  - Subtraction? No.
  - Addition or multiplication of ints? Yes.
  - Addition or multiplication of doubles? No.
    - Not guaranteed to get *exactly* the same result. This may or may not be tolerable in your application, so you should define an acceptable delta and test. If your answers differ by too much, you also have to decide which is the “right” answer.

Example: Parallel Sum of Square Roots

```
public class MathUtils {
  public static double fancySum1(double[] nums) {
    return DoubleStream.of(nums)
                       .map(d -> Math.sqrt(2*d))
                       .sum();
  }

  public static double fancySum2(double[] nums) {
    return DoubleStream.of(nums)
                       .parallel()
                       .map(d -> Math.sqrt(2*d))
                       .sum();
  }
```

Helper Method

```
public static double[] randomNums(int length) {
  double[] nums = new double[length];
  for(int i=0; i<length; i++) {
    nums[i] = Math.random() * 3;
  }
  return(nums);
}
```

Verifying "Same" Result: Test Code

```
public static void compareOutput() {
  double[] nums = MathUtils.randomNums(10_000_000);
  double result1 = MathUtils.fancySum1(nums);
  System.out.printf("Serial result   = %,.12f%n", result1);
  double result2 = MathUtils.fancySum2(nums);
  System.out.printf("Parallel result = %,.12f%n", result2);
}
```

Representative output

```
Serial result   = 16,328,996.081106223000
Parallel result = 16,328,996.081106225000
```

Comparing Performance: Test Code

```
public static void compareTiming() {
   for(int i=5; i<9; i++) {
      int size = (int)Math.pow(10, i);
      double[] nums = MathUtils.randomNums(size);
      Op serialSum = () -> MathUtils.fancySum1(nums);
      Op parallelSum = () -> MathUtils.fancySum2(nums);
      System.out.printf("Serial sum for length   %,d.%n", size);
      Op.timeOp(serialSum);
      System.out.printf("Parallel sum for length %,d.%n", size);
      Op.timeOp(parallelSum);
   }
}
```

Comparing Performance: Representative Output

```
Serial sum for length   100,000.
  Elapsed time: 0.012 seconds.
Parallel sum for length 100,000.
  Elapsed time: 0.008 seconds.
Serial sum for length   1,000,000.
  Elapsed time: 0.005 seconds.
Parallel sum for length 1,000,000.
  Elapsed time: 0.003 seconds.
Serial sum for length   10,000,000.
  Elapsed time: 0.047 seconds.
Parallel sum for length 10,000,000.
  Elapsed time: 0.024 seconds.
Serial sum for length   100,000,000.
  Elapsed time: 0.461 seconds.
Parallel sum for length 100,000,000.
  Elapsed time: 0.176 seconds.
```

Infinite (Unbounded On-the-Fly) Streams

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.

Infinite Streams: Big Ideas

- **Stream.generate(valueGenerator)**
  - Stream.generate lets you specify a Supplier. This Supplier is invoked each time the system needs a Stream element.
    - Powerful when Supplier maintains state, but won't work in parallel
- **Stream.iterate(initialValue, valueTransformer)**
  - Stream.iterate lets you specify a seed and a UnaryOperator f. The seed becomes the first element of the Stream, f(seed) becomes the second element, f(second) becomes third element, etc.
- **Usage**
  - The values are not calculated until they are needed
  - To avoid unterminated processing, you must eventually use a size-limiting operation like limit or findFirst (but not skip alone)
    - The point is not really that this is an "infinite" Stream, but that it is an unbounded "on the fly" Stream – one with no fixed size, where the values are calculated as you need them.

generate

- **Big ideas**
  - You supply a function (Supplier) to Stream.generate. Whenever the system needs stream elements, it invokes the function to get them.
  - You must limit the Stream size.
    - Usually with limit or findFirst (or findAny for parallel streams). skip alone is not enough, since the size is still unbounded
  - By using a real class instead of a lambda, the function can maintain state so that new values are based on any or all of the previous values
- **Quick example**

```
List<Employee> emps =
  Stream.generate(() -> randomEmployee())
        .limit(someRuntimeValue)
        .collect(Collectors.toList());
```

Stateless generate Example: Random Numbers

- **Code**

```
Supplier<Double> random = Math::random;
System.out.println("2 Random numbers:");
Stream.generate(random).limit(2).forEach(System.out::println);
System.out.println("4 Random numbers:");
Stream.generate(random).limit(4).forEach(System.out::println);
```

- **Results**

```
2 Random numbers:
0.00608980775038892
0.269606792416401 3
4 Random numbers:
0.776165188998756 7
0.818313574113532
0.0782437509160781 6
0.715478814539166 7
```

Stateful generate Example: Supplier Code

```
public class FibonacciMaker implements Supplier<Long> {
  private long previous = 0;
  private long current = 1;

  @Override
  public Long get() {
    long next = current + previous;
    previous = current;
    current = next;
    return(previous);
  }
}
```

Lambdas cannot define instance variables, so we use a regular class instead of a lambda to define the Supplier. Also, see the section on lambda-related methods in Lists and Maps for a Fibonacci-generation method.

Helper Code: Simple Methods to Get Any Amount of Fibonacci Numbers

```
public static Stream<Long> makeFibStream() {
  return(Stream.generate(new FibonacciMaker()));
}

public static Stream<Long> makeFibStream(int numFibs) {
  return(makeFibStream().limit(numFibs));
}

public static List<Long> makeFibList(int numFibs) {
  return(makeFibStream(numFibs).collect(Collectors.toList()));
}

public static Long[] makeFibArray(int numFibs) {
  return(makeFibStream(numFibs).toArray(Long[]::new));
}
```

Stateful generate Example

- **Main code**

```
System.out.println("5 Fibonacci numbers:");
FibStream.makeFibStream(5).forEach(System.out::println);
System.out.println("25 Fibonacci numbers:");
FibStream.makeFibStream(25).forEach(System.out::println);
```

- **Results**

```
5 Fibonacci numbers:
1
1
2
3
5
25 Fibonacci numbers:
1
1
...
75025
```

iterate

- **Big ideas**
  - You specify a seed value and a UnaryOperator f. The seed becomes the first element of the Stream, f(seed) becomes the second element, f(second) [i.e., f(f(seed))] becomes third element, etc.
  - You must limit the Stream size.
    - Usually with limit. skip alone is not enough, since the size is still unbounded
  - Will *not* yield the same result in parallel
- **Quick example**

```
List<Integer> powersOfTwo =
    Stream.iterate(1, n -> n * 2)
          .limit(…)
          .collect(Collectors.toList());
```

Simple Example: Twitter Messages

- **Idea**
  - Generate a series of Twitter messages
- **Approach**
  - Start with a very short String as the first message
  - Append exclamation points on the end
  - Continue to 140-character limit
- **Core code**

```
Stream.iterate("Base Msg",
                    msg -> msg + "Suffix")
      .limit(someCutoff)
```

Twitter Messages: Example

- **Code**

```
System.out.println("14 Twitter messages:");
Stream.iterate("Big News!!", msg -> msg + "!!!!!!!!!!")
      .limit(14)
      .forEach(System.out::println);
```

- **Results**

```
Big News!!
Big News!!!!!!!!!!!
Big News!!!!!!!!!!!!!!!!!!!!!
Big News!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
[Etc]
```

More Complex Example: Consecutive Large Prime Numbers

- **Idea**
  - Generate a series of very large consecutive prime numbers (e.g., 100 or 150 digits or more)
  - Large primes are used extensively in cryptography
- **Approach**
  - Start with a prime BigInteger as the seed
  - Supply a UnaryOperator that finds the first prime number higher than the given one
- **Core code**

```
Stream.iterate(Primes.findPrime(numDigits),
                       Primes::nextPrime)
      .limit(someCutoff)
```

Helper Methods: Idea

- **Idea**
  - Generate a random odd BigInteger of the requested size, check if prime, keep adding 2 until you find a match.
- **Why this is feasible**
  - The BigInteger class has a builtin probabilistic algorithm (Miller-Rabin test) for determining if a number is prime without attempting to factor it. It is ultra-fast even for 100-digit or 200-digit numbers.
  - Technically, there is a 2^{100} chance that this falsely identifies a prime, but since 2^{100} is about the number of particles in the universe, that's not a very big risk
    - Algorithm is not fooled by Carmichael numbers

Helper Methods: Code

```
public static BigInteger nextPrime(BigInteger start) {
  if (isEven(start)) {
    start = start.add(ONE);
  } else {
    start = start.add(TWO);
  }
  if (start.isProbablePrime(ERR_VAL)) {
    return(start);
  } else {
    return(nextPrime(start));
  }
}
public static BigInteger findPrime(int numDigits) {
  if (numDigits < 1) {
    numDigits = 1;
  }
  return(nextPrime(randomNum(numDigits)));
}
```

Making Stream of Primes

```
public static Stream<BigInteger> makePrimeStream(int numDigits) {
  return(Stream.iterate(Primes.findPrime(numDigits), Primes::nextPrime));
}

public static Stream<BigInteger> makePrimeStream(int numDigits, int numPrimes) {
  return(makePrimeStream(numDigits).limit(numPrimes));
}

public static List<BigInteger> makePrimeList(int numDigits, int numPrimes) {
  return(makePrimeStream(numDigits, numPrimes).collect(Collectors.toList()));
}

public static BigInteger[] makePrimeArray(int numDigits, int numPrimes) {
  return(makePrimeStream(numDigits, numPrimes).toArray(BigInteger[]::new));
}
```

Primes

- **Code**

```
System.out.println("10 100-digit primes:");
PrimeStream.makePrimeStream(100, 10).forEach(System.out::println);
```

- **Results**

```
10 100-digit primes:
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867976353
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867976647
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867976663
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867976689
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867977233
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867977859
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867977889
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867977989
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867978031
3484894489805924599033259501599559961057903572743870105972345438458556253531271262848463552867978103
```

collect: Grouping Stream Elements

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.

collect

- **Big idea**
  - Using methods in the Collectors class, you can output a Stream as many types
- **Quick examples**
  - List (shown in previous section)
    - `anyStream.collect(toList())`
  - String
    - `stringStream.collect(joining(delimiter)).toString()`
  - Set
    - `anyStream.collect(toSet())`
  - Other collection
    - `anyStream.collect(toCollection(CollectionType::new))`
  - Map
    - `strm.collect(partitioningBy(...))`, `strm.collect(groupingBy(...))`

For brevity, the examples here assume you have done "import static java.util.stream.Collectors.*", so that toList() really means Collectors.toList()

Most Common Output: Building Lists

- **Code**

```
List<Integer> ids = Arrays.asList(2, 4, 6, 8);
List<Employee> emps = ids.stream()
                         .map(EmployeeSamples::findGoogler)
                         .collect(Collectors.toList());
System.out.printf("Googlers with ids %s: %s.%n", ids, emps);
```

- **Results**

```
Googlers with ids [2, 4, 6, 8]:
   [Sergey Brin [Employee#2 $8,888,888],
    Nikesh Arora [Employee#4 $6,666,666],
    Patrick Pichette [Employee#6 $4,444,444],
    Peter Norvig [Employee#8 $900,000]].
```

Remember that you can do a static import on java.util.stream.Collectors.* so that you can use toList() instead of Collectors.toList().

Also recall that List has a builtin toString method that prints the entries comma-separated inside square brackets. Here and elsewhere, line breaks and whitespace added to printouts for readability.

Aside: The StringJoiner Class

- **Big idea**
  - Java 8 added new StringJoiner class that builds delimiter-separated Strings, with optional prefix and suffix
  - Java 8 also added static "join" method to the String class; it uses StringJoiner
- **Quick examples (result: "Java, Lisp, Ruby")**
  - Explicit StringJoiner with no prefix or suffix

```
StringJoiner joiner1 = new StringJoiner(", ");
String result1 =
    joiner1.add("Java").add("Lisp").add("Ruby").toString();
```

  - Usually easier: String.join convenience method

```
String result2 = String.join(", ", "Java", "Lisp", "Ruby");
```

Building Strings

- **Code**

```
List<Integer> ids = Arrays.asList(2, 4, 6, 8);
String lastNames =
  ids.stream().map(EmployeeSamples::findGoogler)
              .map(Employee::getLastName)
              .collect(Collectors.joining(", "))
              .toString();
System.out.printf("Last names of Googlers with ids %s: %s.%n",
                  ids, lastNames);
```

- **Results**

```
Last names of Googlers with ids [2, 4, 6, 8]:
Brin, Arora, Pichette, Norvig.
```

Building Sets

- **Code**

```
List<Employee> googlers = EmployeeSamples.getGooglers();
Set<String> firstNames =
  googlers.stream().map(Employee::getFirstName)
                   .collect(Collectors.toSet());
Stream.of("Larry", "Harry", "Peter", "Deiter", "Eric", "Barack")
      .forEach(s -> System.out.printf ("%s is a Googler? %s.%n",
                                       s, firstNames.contains(s) ? "Yes" : "No"));
```

- **Results**

```
Larry is a Googler? Yes.
Harry is a Googler? No.
Peter is a Googler? Yes.
Deiter is a Googler? No.
Eric is a Googler? Yes.
Barack is a Googler? No.
```

Building Other Collections

- **Big idea**
  - You provide a Supplier<Collection> to collect. Java takes the resultant Collection and then calls "add" on each element of the Stream.

- **Quick examples**
  - ArrayList
    - `someStream.collect(toCollection(ArrayList::new))`
  - TreeSet
    - `someStream.collect(toCollection(TreeSet::new))`
  - Stack
    - `someStream.collect(toCollection(Stack::new))`
  - Vector
    - `someStream.collect(toCollection(Vector::new))`

For brevity, the examples here assume you have done "import static java.util.stream.Collectors.*", so that toCollection(...) really means Collectors.toCollection(...).

Building Other Collections: TreeSet

- **Code**

```
TreeSet<String> firstNames2 =
  googlers.stream()
          .map(Employee::getFirstName)
          .collect(Collectors.toCollection(TreeSet::new));
Stream.of("Larry", "Harry", "Peter", "Deiter", "Eric", "Barack")
      .forEach(s -> System.out.printf("%s is a Googler? %s.%n",
                                      s, firstNames2.contains(s) ? "Yes" : "No"));
```

- **Results**

```
Larry is a Googler? Yes.
Harry is a Googler? No.
Peter is a Googler? Yes.
Deiter is a Googler? No.
Eric is a Googler? Yes.
Barack is a Googler? No.
```

partioningBy: Building Maps

- **Big idea**
  - You provide a Predicate. It builds a Map where true maps to a List of entries that passed the Predicate, and false maps to a List that failed the Predicate.
- **Quick example**

```
Map<Boolean,List<Employee>> oldTimersMap =
   employeeStream().collect(partitioningBy(e -> e.getEmployeeId() < 10));
```

  - Now, oldTimersMap.get(true) returns a List<Employee> of employees whose ID's are less than 10, and oldTimersMap.get(false) returns a List<Employee> of everyone else.

partitioningBy: Example

- **Code**

```
Map<Boolean,List<Employee>> richTable =
  googlers.stream().collect(partitioningBy(e -> e.getSalary() > 1_000_000));
System.out.printf("Googlers with salaries over $1M: %s.%n", richTable.get(true));
System.out.printf("Destitute Googlers: %s.%n", richTable.get(false));
```

- **Results**

```
Googlers with salaries over $1M: [Larry Page [Employee#1 $9,999,999],
  Sergey Brin [Employee#2 $8,888,888], Eric Schmidt [Employee#3 $7,777,777],
  Nikesh Arora [Employee#4 $6,666,666], David Drummond [Employee#5 $5,555,555],
  Patrick Pichette [Employee#6 $4,444,444], Susan Wojcicki [Employee#7 $3,333,333]].

Destitute Googlers: [Peter Norvig [Employee#8 $900,000],
  Jeffrey Dean [Employee#9 $800,000], Sanjay Ghemawat [Employee#10 $700,000],
  Gilad Bracha [Employee#11 $600,000]].
```

groupingBy: Another Way of Building Maps

- **Big idea**
  - You provide a Function. It builds a Map where each output value of the Function maps to a List of entries that gave that value.
    - E.g., if you supply Employee::getFirstName, it builds a Map where supplying a first name yields a List of employees that have that first name.
- **Quick example**

```
Map<Department,List<Employee>> deptTable =
    employeeStream()
        .collect(Collectors.groupingBy(Employee::getDepartment));
```

  - Now, deptTable.get(someDepartment) returns a List<Employee> of everyone in that department.

groupingBy: Supporting Code

- **Idea**
  - Make a class called Emp that is a simplified Employee that has a first name, last name, and office/location name, all as Strings.
- **Sample Emps**

```
private static Emp[] sampleEmps = {
  new Emp("Larry", "Page", "Mountain View"),
  new Emp("Sergey", "Brin", "Mountain View"),
  new Emp("Lindsay", "Hall", "New York"),
  new Emp("Hesky", "Fisher", "New York"),
  new Emp("Reto", "Strobl", "Zurich"),
  new Emp("Fork", "Guy", "Zurich"),
};
public static List<Emp> getSampleEmps() {
  return(Arrays.asList(sampleEmps));
}
```

groupingBy: Example

- **Code**

```
Map<String,List<Emp>> officeTable =
  EmpSamples.getSampleEmps().stream()
            .collect(Collectors.groupingBy(Emp::getOffice));
System.out.printf("Emps in Mountain View: %s.%n",
                  officeTable.get("Mountain View"));
System.out.printf("Emps in NY: %s.%n",
                  officeTable.get("New York"));
System.out.printf("Emps in Zurich: %s.%n",
                  officeTable.get("Zurich"));
```

- **Results**

```
Emps in Mountain View:
  [Larry Page [Mountain View], Sergey Brin [Mountain View]].
Emps in NY: [Lindsay Hall [New York], Hesky Fisher [New York]].
Emps in Zurich: [Reto Strobl [Zurich], Fork Guy [Zurich]].
```

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

- **Parallel streams**
  - `anyStream.parallel().normalStreamOps(…)`
    - For reduce, be sure there is no global data and that operator is associative
    - Test to verify you get same answer both ways
    - Compare timing
- **"Infinite" (really unbounded) streams**
  - `Stream.generate(someStatelessSupplier).limit(…)`
  - `Stream.generate(someStatefullSupplier).limit(…)`
  - `Stream.iterate(seedValue, operatorOnSeed).limit(…)`
- **Fancy uses of collect**
  - You can build many collection types from streams

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)

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.