Streams in Java 8: Part 1

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

more SERVLETS and JAVASERVER PAGES
MARTY HALL
core SERVLETS and JAVASERVER PAGES
2ND EDITION
Volume 1: Core Technologies
MARTY HALL · LARRY BROWN

Topics in This Section

- **Overview of streams**
- **Building streams**
- **Outputting streams into arrays or Lists**
- **Core stream methods**
  - Overview
  - forEach
  - map
  - filter
  - findFirst and findAny
- **The new Optional class**
- **Lazy evaluation and short-circuit operations**

Overview of Streams

Streams in a Nutshell – Comparison to Lists

- **Streams have more convenient methods than Lists**
  - forEach, filter, map, reduce, min, sorted, distinct, limit, etc.
- **Streams have cool properties that Lists lack**
  - Making streams more powerful, faster, and more memory efficient
  - The three coolest properties
    - Lazy evaluation
    - Automatic parallelization
    - Infinite (unbounded) streams
- **Streams do not store data**
  - They are just programmatic wrappers around existing data sources
- **Name is confusing for newbies**
  - Little relationship to IO streams
    - PrintStream, *Blah*InputStream (BufferedInputStream, ByteArrayInputStream, FileInputStream, ImageInputStream, JarInputStream, etc.), *Blah*OutputStream (BufferedOutputStream, ByteArrayOutputStream, FileOutputStream, etc.), etc.

Streams

- **Big idea**
  - Wrappers around data sources such as arrays or lists. Support many convenient and high-performance operations expressed succinctly with lambdas, executed sequentially or in parallel.
- **Quick preview**

```
Stream.of(idArray).map(EmployeeUtils::findById)
                  .filter(e -> e != null)
                  .filter(e -> e.getSalary() > 500000)
                  .findFirst()
                  .orElse(null));
```

- This appears to say "take the *n* ids, produce a Stream of *n* corresponding Employee objects, remove null entries (unknown ids), throw out all except those whose salary is above $500K, find first entry, return it if it exists, otherwise return null". But, if third id corresponds to someone with salary above $500K, it only calls findById three times, even if the id array has 100 entries. Lazy!

Another Quick Preview

- **Goal**
  - – Given very large file of words of various lengths in mixed case with possible repeats, create sorted uppercase file of n-letter words

```
List<String> words =
   Files.lines(Paths.get(inputFileName))
         .filter(s -> s.length() == n)
         .map(String::toUpperCase)
         .distinct()
         .sorted()
         .collect(Collectors.toList());
Files.write(Paths.get(outputFileName), words,
               Charset.defaultCharset());
```

This code is slightly over-simplified:

- You should catch IOException or declare that your method throws it
- You should close the Stream that results from Files.lines so that long-running code will not have memory leak. (Normal streams do not need to be closed: only those that are backed by files.)

Still, this gives the gist of the approach, and shows that as of Java 8, Java is competitive with Python for quick routines that scan and analyze data logs and other text files.

Characteristics of Streams

- **Not data structures**
  - – Streams have *no* storage. They carry values from a source through a pipeline of operations.
    - They also never modify the underlying data structure (e.g., the List or array that the Stream wraps)
- **Designed for lambdas**
  - – All Stream operations take lambdas as arguments
- **Do not support indexed access**
  - – You can ask for the first element, but not the second or third or last element. But, see next bullet.
- **Can easily be output as arrays or Lists**
  - – Simple syntax to build an array or List from a Stream

Characteristics of Streams (Continued)

- **Lazy**
  - Many Stream operations are postponed until it is known how much data is eventually needed
    - E.g., if you do a 10-second-per-item operation on a 100 element list, then select the first entry, it takes 10 seconds, not 1000 seconds.
- **Parallelizable**
  - If you designate a Stream as parallel, then operations on it will automatically be done concurrently, without having to write explicit multi-threading code
- **Can be unbounded**
  - Unlike with collections, you can designate a generator function, and clients can consume entries as long as they want, with values being generated on the fly

Getting Standard Data Structures Into and Out of Streams

Making Streams: Overview

- **Big idea**
  - Streams are not collections: they do not manage their own data. Instead, they are wrappers around existing data structures.
    - When you make or transform a Stream, it does not copy the underlying data. Instead, it just builds a pipeline of operations. How many times that pipeline will be invoked depends on what you later do with the stream (find the first element, skip some elements, see if all elements match a Predicate, etc.)
- **Three most common ways to make Stream**
  - someList.stream()
  - Stream.of(arrayOfObjects)
  - Stream.of(val1, val2, …)

Making Streams: Examples

- **From Lists**

```
List<String> words = …;
words.stream().map(…).filter(…).other(…);
List<Employee> workers = …;
workers.stream().map(…).filter(…).other(…);
```

- **From arrays**

```
Employee[] workers = …;
Stream.of(workers).map(…).filter(…).other(…);
```

- **From individual elements**

```
Employee e1 = …;
Employee e2 = …;
Stream.of(e1,e2, ...).map(…).filter(…).other(…);
```

Making Streams: Details

- **From individual values**
  - Stream.of(val1, val2, …)
- **From array**
  - Stream.of(someArray), Arrays.stream(someArray)
- **From List (and other collections)**
  - someList.stream(), someOtherCollection.stream()
- **From a “function”**
  - Stream.generate, Stream.iterate
- **From a file**
  - Files.lines(somePath)
- **From a StreamBuilder**
  - someBuilder.build()
- **From String**
  - String.chars, Stream.of(someString.split(...))
- **From another Stream**
  - distinct, filter, limit, map, sorted, skip

Caution: Use Objects, not Primitives

- **Producing IntStream by accident**
  - Alternatives
    - int[] nums = { 1, 2, 3, 4 };
    - Arrays.stream(nums)…   // IntStream
    - Integer[] nums = { 1, 2, 3, 4 };
    - Arrays.stream(nums)… or Stream.of(nums)… // Stream<Integer>
  - Differences
    - The map method of IntStream must produce Integer. Limiting.
    - But, IntStream has a useful “sum” method.
- **Making 1-item stream by accident**
  - Mistake
    - int[] nums = { 1, 2, 3, 4 };
    - Stream.of(nums)…   // 1-item Stream containing array
  - Correct
    - Integer[] nums = { 1, 2, 3, 4 };
    - Stream.of(nums)…   // 4-item Stream containing Integers

Turning Streams into Pre-Java-8 Data Structures

- **List (most common)**
  - someStream.collect(Collectors.toList())
- **Array (less common)**
  - someStream.toArray(EntryType[]::new)
    - E.g., employeeStream.toArray(Employee[]::new)
      - The argument to toArray is normally EntryType[]::new, but in general is a Supplier that takes an int (size) as an argument and returns an empty array that can be filled in. There is also a zero-argument toArray() method, but this returns Object[], not Blah[], and it is illegal to cast Object[] to Blah[], even when all entries in the array are Blahs.
- **Note**
  - You normally do this only at the end, after you have done all the cool Stream operations. For example:
    - someStream.map(…).filter(…).map(…).collect(…);

Outputting Streams: Examples

- **Outputting as Lists**
  - List<String> wordList =
    ```
    someStream.map(someLambdaThatReturnsString)
              .filter(someTestOnStrings)
              .collect(Collectors.toList());
    ```
  - List<Employee> workerList =
    ```
    someStream.map(…).collect(Collectors.toList());
    ```
- **Outputting as arrays**
  - String[] wordArray =
    ```
    someStream.map(someLambdaThatReturnsString)
              .filter(someTestOnStrings)
              .toArray(String[]::new);
    ```
  - Employee[] workerList =
    ```
    someStream.map(…).toArray(Employee[]::new);
    ```

Core Stream Methods: Overview

Stream Methods

- **Big idea**
  - You wrap a Stream around an array or List (or even a file, as seen in File IO lecture). Then, you can do operations on each element (forEach), make a new Stream by transforming each element (map), remove elements that don’t match some criterion (filter), etc.
- **Core methods (covered here)**
  - forEach, map, filter, findFirst, findAny, collect, toArray
- **Methods covered in later section**
  - reduce, collect, min, max, sum, sorted, distinct, limit, skip, noneMatch, allMatch, anyMatch, count

Core Stream Methods

- **forEach(Consumer)**
  - employees.forEach(e -> e.setSalary(e.getSalary() * 11/10))
- **map(Function)**
  - ids.map(EmployeeUtils::findEmployeeById)
- **filter(Predicate)**
  - employees.filter(e -> e.getSalary() > 500000)
- **findFirst()**
  - employees.filter(…).findFirst().get()
- **toArray(ResultType[]::new)**
  - Employee[] empArray = employees.toArray(Employee[]::new);
- **collect(Collectors.toList())**
  - List<Employee> empList = employees.collect(Collectors.toList());

Stream Examples: Setup Code

```
public class EmployeeSamples {
  private static List<Employee> GOOGLERS =
    Arrays.asList(
      new Employee("Larry", "Page", 1, 9999999),
      new Employee("Sergey", "Brin", 2, 8888888),
      new Employee("Eric", "Schmidt", 3, 7777777),
      new Employee("Nikesh", "Arora", 4, 6666666),
      new Employee("David", "Drummond", 5, 5555555),
      new Employee("Patrick", "Pichette", 6, 4444444),
      new Employee("Susan", "Wojcicki", 7, 3333333),
      new Employee("Peter", "Norvig", 8, 900000),
      new Employee("Jeffrey", "Dean", 9, 800000),
      new Employee("Sanjay", "Ghemawat", 10, 700000),
      new Employee("Gilad", "Bracha", 11, 600000) );

  public static List<Employee> getGooglers() {
    return(GOOGLERS);
  }
```

Stream Examples: Setup Code (Continued)

```
private static final List<Employee> SAMPLE_EMPLOYEES =
  Arrays.asList(
    new Employee("Harry", "Hacker", 1, 234567),
    new Employee("Polly", "Programmer", 2, 333333),
    new Employee("Cody", "Coder", 8, 199999),
    new Employee("Devon", "Developer", 11, 175000),
    new Employee("Desiree", "Designer", 14, 212000),
    new Employee("Archie", "Architect", 16, 144444),
    new Employee("Tammy", "Tester", 19, 166777),
    new Employee("Sammy", "Sales", 21, 45000),
    new Employee("Larry", "Lawyer", 22, 33000),
    new Employee("Amy", "Accountant", 25, 85000) );

public static List<Employee> getSampleEmployees() {
  return(SAMPLE_EMPLOYEES);
}
```

Using the Sample Employees

- **Right: re-stream the List each time**

```
List<Employee> googlers =
   EmployeeSamples.getGooglers();
googlers.stream().map(…).filter(…).other(…);
googlers.stream().filter(…).forEach(…);
```

- **Wrong: reuse the Stream**

```
Stream<Employee> googlerStream =
   EmployeeSamples.getGooglers().stream();
googlerStream.map(…).filter(…).other(…);
googlerStream.filter(…).forEach(…);
```

- **Explanation**
  - You can only do one chain of operations on each Stream!
    - However, it does not cost you anything to “re-stream” the List<Employee>, because “creating” a Stream just points at the existing data structure behind the scenes; it does not copy the data.

forEach –

Calling a Lambda on Each Element of a Stream

forEach

- **Big idea**
  - Easy way to loop over Stream elements
    - There are also forEach methods in Iterable, ArrayList, Map, etc.
  - You supply a lambda to forEach, and that lambda is called on each element of the Stream
    - More precisely, you supply a Consumer to forEach, and each element of the Stream is passed to that Consumer's accept method. But, few people think of it in these low-level terms.
  - There is also "peek" method, which is almost exactly the same as forEach, except it returns the original Stream at the end
- **Quick examples**
  - Print each element

```
Stream.of(someArray).forEach(System.out::println);
```

  - Clear all JTextFields

```
fieldList.stream().forEach(field -> field.setText(""));
```

forEach vs. for Loops

- **for**

```
List<Employee> employees = getEmployees();
for(Employee e: employees) {
  e.setSalary(e.getSalary() * 11/10);
}
```

- **forEach**

```
Stream<Employee> employees = getEmployees().stream();
employees.forEach(e -> e.setSalary(e.getSalary() * 11/10));
```

- **Advantages of forEach**
  - Minor: designed for lambdas
    - Marginally more succinct
  - Minor: reusable
    - You can save the lambda and use it again (see example)
  - Major: can be made parallel with minimal effort
    - someStream.parallel().forEach(someLambda);

What You Can<u>not</u> do with forEach

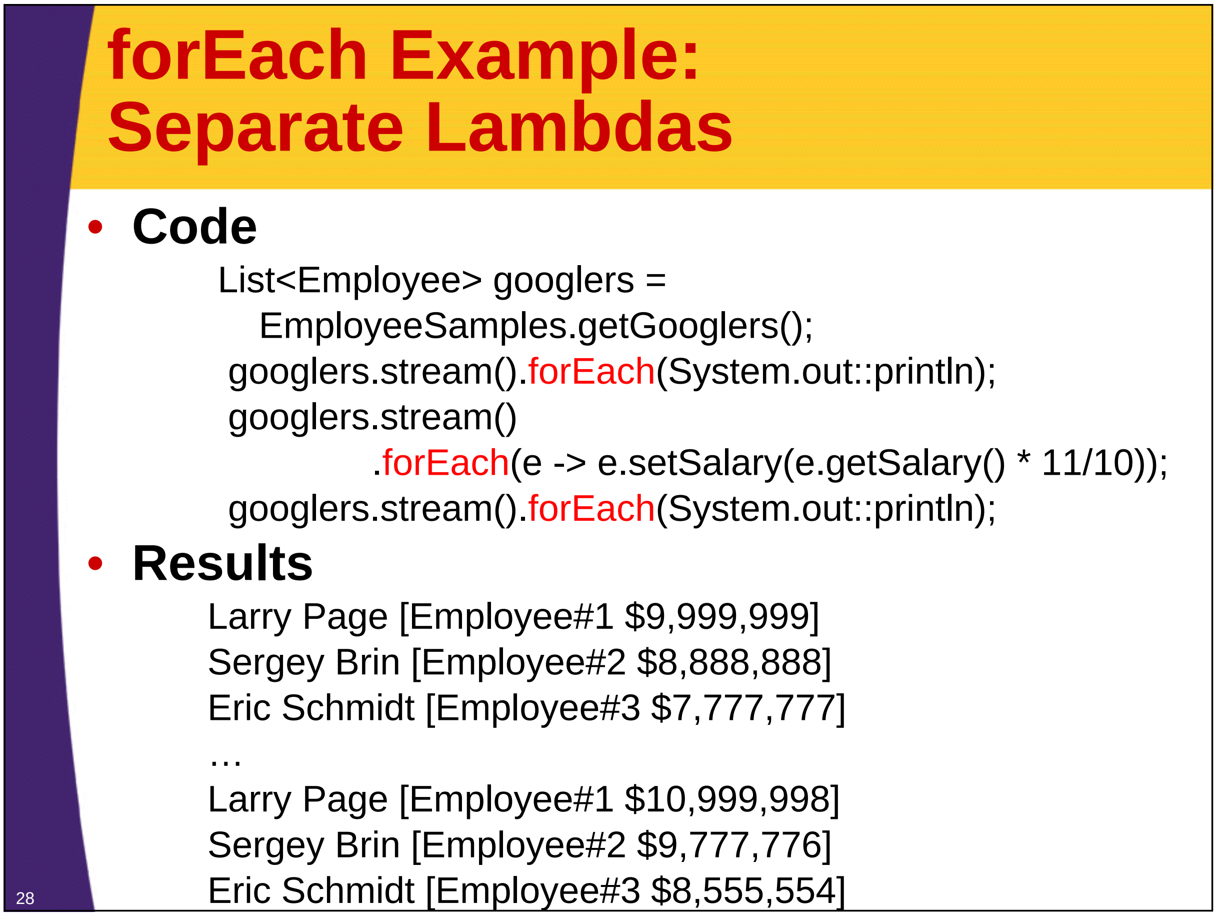
- **Loop twice**
  - forEach is a "terminal operation", which means that it consumes the elements of the Stream. So, this is illegal:

```
someStream.forEach(element -> doOneThing(element));
someStream.forEach(element -> doAnotherThing(element));
```

    - But, of course, you can combine both operations into a single lambda
    - Also, you can use "peek" instead of forEach, and then loop twice
- **Change values of surrounding local variables**
  - Illegal attempt to calculate total yearly payroll:

```
double total = 0;
employeeList.stream().forEach(e -> total += e.getSalary());
```

    - But, we will see good way of doing this with "map" and "reduce". In fact, this is so common that DoubleStream has builtin "sum" method
- **Break out of the loop early**
  - You cannot use "break" or "return" to terminate looping



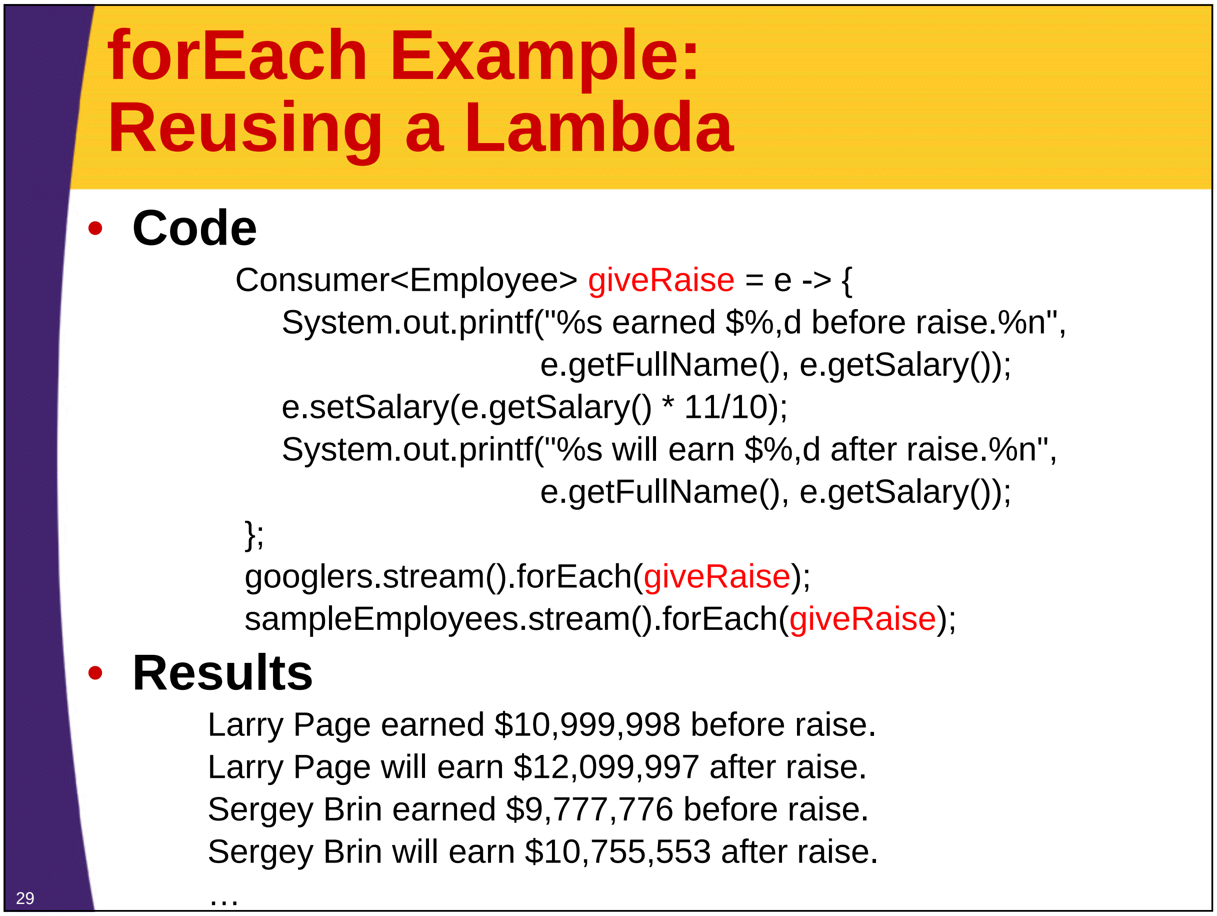
forEach Example: Separate Lambdas

- **Code**

```
List<Employee> googlers =
   EmployeeSamples.getGooglers();
googlers.stream().forEach(System.out::println);
googlers.stream()
            .forEach(e -> e.setSalary(e.getSalary() * 11/10));
googlers.stream().forEach(System.out::println);
```

- **Results**

```
Larry Page [Employee#1 $9,999,999]
Sergey Brin [Employee#2 $8,888,888]
Eric Schmidt [Employee#3 $7,777,777]
…
Larry Page [Employee#1 $10,999,998]
Sergey Brin [Employee#2 $9,777,776]
Eric Schmidt [Employee#3 $8,555,554]
```



forEach Example: Reusing a Lambda

- **Code**

```
Consumer<Employee> giveRaise = e -> {
    System.out.printf("%s earned $%,d before raise.%n",
                          e.getFullName(), e.getSalary());
    e.setSalary(e.getSalary() * 11/10);
    System.out.printf("%s will earn $%,d after raise.%n",
                          e.getFullName(), e.getSalary());
};
googlers.stream().forEach(giveRaise);
sampleEmployees.stream().forEach(giveRaise);
```

- **Results**

```
Larry Page earned $10,999,998 before raise.
Larry Page will earn $12,099,997 after raise.
Sergey Brin earned $9,777,776 before raise.
Sergey Brin will earn $10,755,553 after raise.
…
```

map –
Transforming a Stream by Passing Each Element through a Function

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

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop.

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

map

- **Big idea**
  - Produces a new Stream that is the result of applying a Function to each element of original Stream
- **Quick examples**
  - Array of squares

```
Double[] nums = { 1.0, 2.0, 3.0, 4.0, 5.0 };
Double[] squares =
    Stream.of(nums).map(n -> n * n).toArray(Double[]::new);
```

  - List of Employees with given IDs

```
Integer[] ids = { 1, 2, 4, 8 };
List<Employee> matchingEmployees =
     Stream.of(ids).map(EmployeeUtils::findById)
                   .collect(Collectors.toList());
```

map Examples: Helper Code to Find Employee by ID

- **Employee Map**

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

 static {
   GOOGLERS
     .stream()
     .forEach(e -> googleMap.put(e.getEmployeeId(), e));
 }
```

- **Employee lookup method**

```
public static Employee findGoogler(Integer employeeId) {
  return(googleMap.get(employeeId));
}
```

The method reference for the above method is EmployeeSamples::findGoogler.

map Examples: Helper Method for Printing

- **For printing a Stream**

```
private static void printStreamAsList(Stream s,
                                      String message) {
    System.out.printf("%s: %s.%n",
                      message, s.collect(Collectors.toList()));
}
```

It is also common to first do "import static java.util.stream.Collectors.*;" and then to use toList() instead of Collectors.toList().

map Example: Numbers

- **Code**

```
List<Double> nums = Arrays.asList(1.0, 2.0, 3.0, 4.0, 5.0);
printStreamAsList(nums.stream(), "Original nums");
printStreamAsList(nums.stream().map(n -> n * n),
                          "Squares");
printStreamAsList(nums.stream().map(n -> n * n)
                                           .map(Math::sqrt),
                          "Square roots of the squares");
```

- **Results**

```
Original nums: [1.0, 2.0, 3.0, 4.0, 5.0].
Squares: [1.0, 4.0, 9.0, 16.0, 25.0].
Square roots of the squares: [1.0, 2.0, 3.0, 4.0, 5.0].
```

map Example: Employees

- **Code**

```
Integer[] ids = { 1, 2, 4, 8 };
printStreamAsList(Stream.of(ids), "IDs");
printStreamAsList
     (Stream.of(ids).map(EmployeeSamples::findGoogler)
                    .map(Person::getFullName),
      "Names of Googlers with given IDs");
```

- **Results**

```
IDs: [1, 2, 4, 8].
Names of Googlers with given IDs:
   [Larry Page, Sergey Brin, Nikesh Arora, Peter Norvig].
```

filter – Keeping Only the Elements that Pass a Predicate

filter

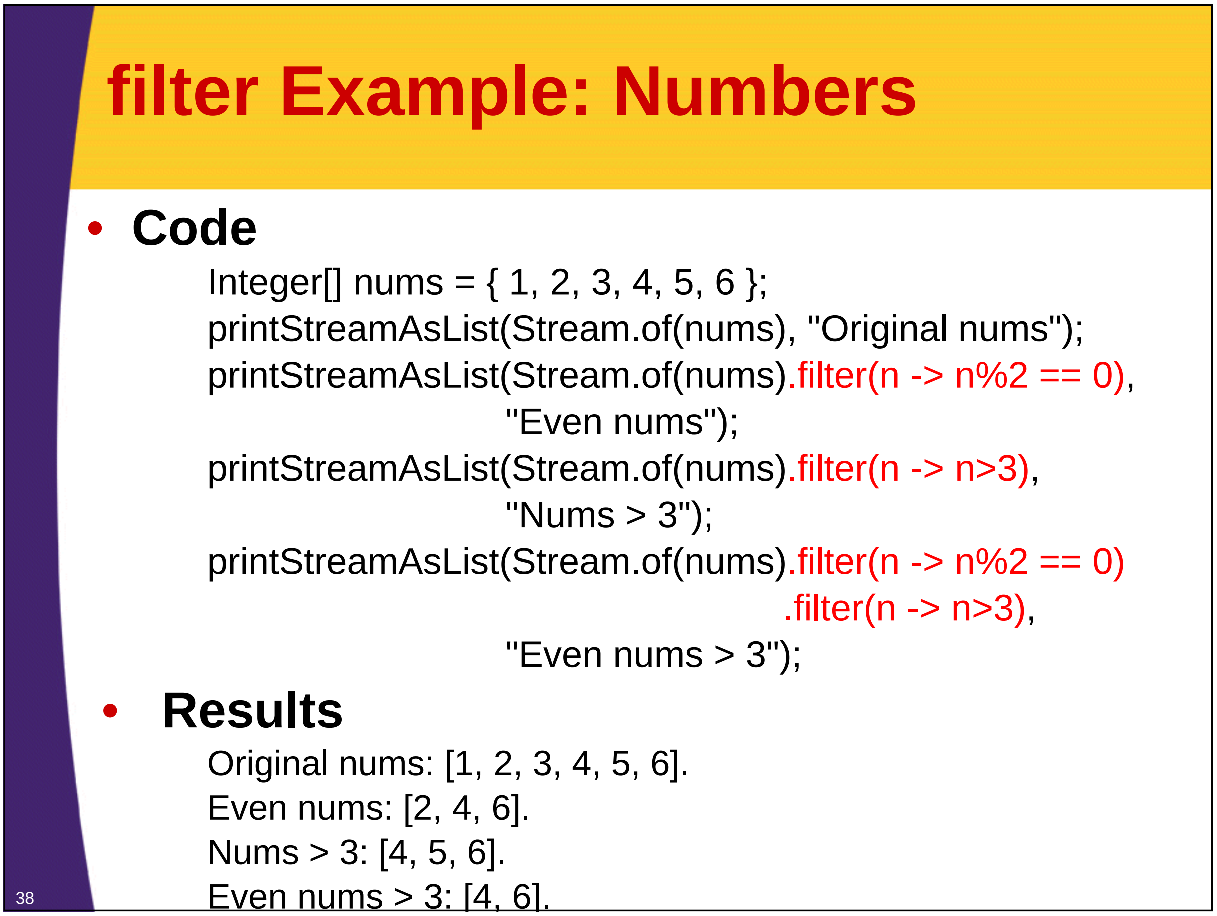
- **Big idea**
  - Produces a new Stream that contain only the elements of the original Stream that pass a given test
- **Quick examples**
  - Even numbers

```
Integer[] nums = { 1, 2, 3, 4, 5, 6 };
Integer[] evens =
   Stream.of(nums).filter(n -> n%2 == 0).toArray(Integer[]::new);
```

  - Even numbers greater than 3

```
Integer[] evens =
   Stream.of(nums).filter(n -> n%2 == 0)
                  .filter(n -> n>3)
                  .toArray(Integer[]::new);
```

This has same efficiency and memory usage as a single filter that does both tests. We will see why in section on lazy evaluation.



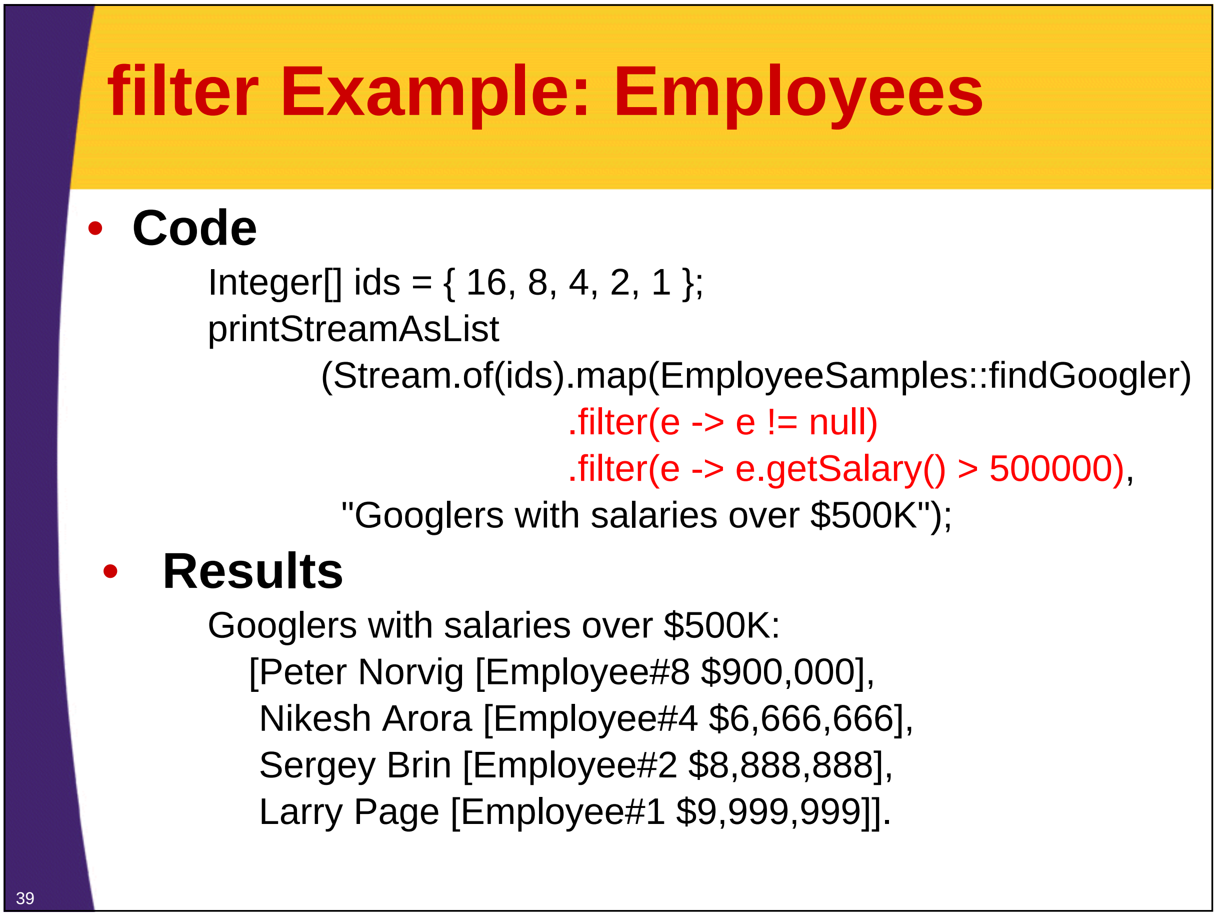
filter Example: Numbers

- **Code**

```
Integer[] nums = { 1, 2, 3, 4, 5, 6 };
printStreamAsList(Stream.of(nums), "Original nums");
printStreamAsList(Stream.of(nums).filter(n -> n%2 == 0),
                  "Even nums");
printStreamAsList(Stream.of(nums).filter(n -> n>3),
                  "Nums > 3");
printStreamAsList(Stream.of(nums).filter(n -> n%2 == 0)
                                 .filter(n -> n>3),
                  "Even nums > 3");
```

- **Results**

```
Original nums: [1, 2, 3, 4, 5, 6].
Even nums: [2, 4, 6].
Nums > 3: [4, 5, 6].
Even nums > 3: [4, 6].
```



filter Example: Employees

- **Code**

```
Integer[] ids = { 16, 8, 4, 2, 1 };
printStreamAsList
    (Stream.of(ids).map(EmployeeSamples::findGoogler)
                   .filter(e -> e != null)
                   .filter(e -> e.getSalary() > 500000),
     "Googlers with salaries over $500K");
```

- **Results**

```
Googlers with salaries over $500K:
  [Peter Norvig [Employee#8 $900,000],
   Nikesh Arora [Employee#4 $6,666,666],
   Sergey Brin [Employee#2 $8,888,888],
   Larry Page [Employee#1 $9,999,999]].
```

findFirst – Returning the First Element of a Stream while Short-Circuiting Earlier Operations

findFirst

- **Big idea**
  - Returns an Optional for the first entry in the Stream. Since Streams are often results of filtering, there might not be a first entry, so the Optional could be empty.
    - There is also a similar findAny method, which might be faster for parallel Streams (which are in later tutorial).
  - findFirst is faster than it looks when paired with map or filter. More details in section on lazy evaluation, but idea is that map or filter know to only find a single match and then stop.
- **Quick examples**
  - When you know there is at least one entry
    - someStream.map(…).findFirst().get()
  - When unsure if there are entries or not
    - someStream.filter(…).findFirst().orElse(otherValue)

Aside: the Optional Class

- **Big idea**
  - Optional either stores a T or stores nothing. Useful for methods that may or may not find a value. New in Java 8.
    - The value of findFirst of Stream is an Optional
- **Syntax**
  - Making an Optional
    - Optional<Blah> value = Optional.of(someBlah);
    - Optional<Blah> value = Optional.empty(); // Missing val
  - Most common operations on an Optional
    - value.get() – returns value if present or throws exception
    - value.orElse(other) – returns val if present or other
    - value.ifPresent(Consumer) – runs lambda if val is present
    - value.isPresent() – returns true if val is present

Quick Preview of Lazy Evaluation

- **Code** (Employee with ID 8 is first match)

```
Integer[] ids = { 16, 8, 4, …}; // 10,000 entries
System.out.printf("First Googler with salary over $500K: %s%n",
                  Stream.of(ids).map(EmployeeSamples::findGoogler)
                                .filter(e -> e != null)
                                .filter(e -> e.getSalary() > 500000)
                                .findFirst()
                                .orElse(null));
```

- **Questions**
  - How many times is:
    - findGoogler called?
    - The null check performed?
    - getSalary called?
  - What if there were 10,000,000 ids instead of 10,000 ids?

Quick Preview of Lazy Evaluation

- **Code** (Employee with ID 8 is first match)

```
Integer[] ids = { 16, 8, 4, ...}; // 10,000 entries
System.out.printf("First Googler with salary over $500K: %s%n",
                  Stream.of(ids).map(EmployeeSamples::findGoogler)
                                .filter(e -> e != null)
                                .filter(e -> e.getSalary() > 500000)
                                .findFirst()
                                .orElse(null));
```

- **Answers**
  - findGoogler: 2
  - Check for null: 2
  - getSalary: 1

Lazy Evaluation

Overview

- **Big idea**
  - Streams defer doing most operations until you actually need the results
- **Result**
  - Operations that appear to traverse Stream multiple times actually traverse it only once
  - Due to “short-circuit” methods, operations that appear to traverse entire stream can stop much earlier.
    - stream.map(someOp).filter(someTest).findFirst().get()
      - Does the map and filter operations *one element at a time*. Continues only until first match on the filter test.
    - stream.map(…).filter(…).filter(…).allMatch(someTest)
      - Does the one map, two filter, and one allMatch test *one element at a time*. The first time it gets false for the allMatch test, it stops.

Method Types: Overview

- **Intermediate methods**
  - These are methods that produce other Streams. These methods don’t get processed until there is some terminal method called.
- **Terminal methods**
  - After one of these methods is invoked, the Stream is considered consumed and no more operations can be performed on it.
    - These methods can do a side-effect (forEach) or produce a value (findFirst)
- **Short-circuit methods**
  - These methods cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.
    - Short-circuit methods can be intermediate (limit, skip) or terminal (findFirst, allMatch)
  - E.g., this example only filters until it finds a *single* match:

    Stream.of(someArray).filter(e -> someTest(e)).findFirst().get()

Method Types: Listing by Categories

- **Intermediate methods**
  - map (and related mapToInt, flatMap, etc.), filter, distinct, sorted, peek, limit, skip, parallel, sequential, unordered
- **Terminal methods**
  - forEach, forEachOrdered, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator
- **Short-circuit methods**
  - anyMatch, allMatch, noneMatch, findFirst, findAny, limit, skip

Example of Lazy Evaluation and Terminal Methods

- **Code**

```
Stream.of(idArray).map(EmployeeUtils::findById)
                  .filter(e -> e != null)
                  .filter(e -> e.getSalary() > 500000)
                  .findFirst()
                  .orElse(null));
```

- **Apparent behavior**
  - findById on all, check all for null, call getSalary on all non-null (& compare to $500K) on all remaining, find first, return it or null
- **Actual behavior**
  - findById on first, check it for null, if pass, call getSalary, if salary > $500K, return and done. Repeat for second, etc. Return null if you get to the end and never got match.

Lazy Evaluation: Showing Order of Operations

```
Function<Integer,Employee> findGoogler =
  n -> { System.out.println("Finding Googler with ID " + n);
         return(EmployeeSamples.findGoogler(n));
       };
Predicate<Employee> checkForNull =
  e -> { System.out.println("Checking for null");
         return(e != null);
       };
Predicate<Employee> checkSalary =
  e -> { System.out.println("Checking if salary > $500K");
         return(e.getSalary() > 500000);
       };
```

Same functionality as lambdas on previous slide, except with print statements added.

Lazy Evaluation: Order of Operations and Short-Circuiting

- **Code**

```
Integer[] ids = { 16, 8, 4, 2, 1 };
System.out.printf("First Googler with salary over $500K: %s%n",
                  Stream.of(ids).map(findGoogler)
                                .filter(checkForNull)
                                .filter(checkSalary)
                                .findFirst()
                                .orElse(null));
```

- **Results**

```
Finding Googler with ID 16
Checking for null
Finding Googler with ID 8
Checking for null
Checking if salary > $500K
First Googler with salary over $500K: Peter Norvig [Employee#8 $900,000]
```

If you thought of Streams as collections, you would think:

- It would first call findGoogler on all 5 ids, resulting in 5 Employees
- It would then call checkForNull on all 5 Employees
- It would then call checkSalary on all remaining Employees
- It would then get the first one (or null, if no Employees)

Instead, it builds a pipeline that, for each element in turn, calls findGoogler, then checks that same element for null, then if non-null, checks the salary of that same element, and if it exists, returns it.

Wrap-Up

Summary

- **Make a Stream**
  - Stream.of(e1, e2…), Stream.of(array), someList.stream()
- **Output from a Stream**
  - stream.collect(Collectors.toList())
  - stream.toArray(Blah[]::new)
- **forEach** [void output]
  - employeeStream.forEach(e -> e.setPay(e.getPay() * 1.1))
- **map** [outputs a Stream]
  - numStream.map(Math::sqrt)
- **filter** [outputs a Stream]
  - employeeStream.filter(e -> e.getSalary() > 500000)
- **findFirst** [outputs an Optional]
  - stream.findFirst().get(), stream.findFirst().orElse(other)

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)