File I/O in Java 8 Part 2:

Using Lambdas and Generic Types to Make File-Reading Code More Flexible, Reusable, and Testable

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.

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

- **File Reading: Second Variation**
  - Split file-reading part from Stream-processing part
- **File Reading: Third Variation**
  - Use lambdas to avoid repeating boilerplate code
- **File Reading: Fourth Variation**
  - Use generic types to flexibly return values
- **Using varargs for Predicate<T>**
  - Conclusion: fancy lambda techniques → fancy file I/O techniques

File Reading Variations

- **General principle**
  - Streams help make handling large data sets more convenient and efficient
  - Lambdas and generic types help make code more flexible and reusable
- **Variation 1 (last section)**
  - Put all code inside main; main throws Exception
    - Simple and easy, but not reusable
- **Variation 2**
  - Method 1 handles Stream; method 2 calls Files.lines and passes Stream to method 1
    - Reusable, but each version of method 2 repeats a lot of boilerplate code
- **Variation 3**
  - Define a functional interface and a static method that can use lambdas
  - Method 1 handles Stream; method 2 passes filename and lambda to static method
- **Variation 4**
  - Similar to variation 3, but uses generic types so that values can be returned

File Reading: Second Variation

Simple Script vs. Reusable Method

- **For simple script, do everything in main**

```
public static void main(String[] args) throws Exception {
  Files.lines(Paths.get("input-file"))
       .map(someFunction)
       .filter(someTest)
       .someOtherStreamOperation(...);
}
```

- **For reusable methods, break processing into two pieces**
  - First method takes a Stream<String>
    - This can be tested and reused independently of the file
  - Second method calls Files.lines and passes the result to the first method
    - But also has try/catch block to handle problems and to automatically close the file stream when done

Why Split the Processing?

- **Why use two methods?**
  - One that processes a Stream
  - One that uses Files.lines to build a Stream<String>, and passes it to first method
- **Benefits to splitting**
  - Simpler testing. You can test the first method with simple Stream created with Stream.of or someList.stream().
  - More reusable. The first method can be used for Streams created from other sources.
  - More flexible. The first method can take a Stream<T>, where T is a generic type, and thus can be used for a variety of purposes, not just String processing.
  - Better error handling. Uses try/catch blocks instead of main throwing Exception.
  - Better memory usage. Stream is closed when done.

Variation 2: General Approach

```
public static void useStream(Stream<String> lines, ...) {
  lines.filter(...).map(...)...;
}

public static void useFile(String filename, ...) {
  try(Stream<String> lines = Files.lines(Paths.get(filename))) {
    SomeClass.useStream(lines, ...);
  } catch(IOException ioe) {
    System.err.println("Error reading file: " + ioe);
  }
}
```

Example 1: Printing All Palindromes

```
public class FileUtils {
  public static void printAllPalindromes(Stream<String> words) {
    words.filter(StringUtils::isPalindrome)
         .forEach(System.out::println);
  }

  public static void printAllPalindromes(String filename) {
    try(Stream<String> words = Files.lines(Paths.get(filename))) {
      printAllPalindromes(words);
    } catch(IOException ioe) {
      System.err.println("Error reading file: " + ioe);
    }
  }
}
```


Example 1: Printing All Palindromes

```
public static void main(String[] args) {
  String filename = "enable1-word-list.txt";
  if (args.length > 0) {
    filename = args[0];
  }
  testAllPalindromes(filename);
}

public static void testAllPalindromes(String filename) {
  List<String> testWords = Arrays.asList("bog", "bob", "dam", "dad");
  System.out.printf("All palindromes in list %s:%n", testWords);
  FileUtils.printAllPalindromes(testWords.stream());
  System.out.printf("All palindromes in file %s:%n", filename);
  FileUtils.printAllPalindromes(filename);
}
```

Output

```
All palindromes in list [bog, bob, dam, dad]:
bob
dad
All palindromes in file enable1-word-list.txt:
aa
aba
...
```

Example 2: Printing N-Length Palindromes

```
public static void printPalindromes(Stream<String> words,
                                    int length) {
  words.filter(word -> word.length() == length)
       .filter(StringUtils::isPalindrome)
       .forEach(System.out::println);
}

public static void printPalindromes(String filename, int length) {
  try(Stream<String> words = Files.lines(Paths.get(filename))) {
    printPalindromes(words, length);
  } catch(IOException ioe) {
    System.err.println("Error reading file: " + ioe);
  }
}
```


Example 2: Printing N-Length Palindromes

```
public static void main(String[] args) {
  String filename = "enable1-word-list.txt";
  if (args.length > 0) {
    filename = args[0];
  }
  test3LetterPalindromes(filename);
}
public static void test3LetterPalindromes(String filename) {
  List<String> testWords =
      Arrays.asList("bog", "bob", "dam", "dad", "kook", "noon");
  System.out.printf("3-letter palindromes in list %s:%n", testWords);
  FileUtils.printPalindromes(testWords.stream(), 3);
  System.out.printf("3-letter palindromes in file %s:%n", filename);
  FileUtils.printPalindromes(filename, 3);
}
```

Output

```
3-letter palindromes in list [bog, bob, dam, dad, kook, noon]:
bob
dad
3-letter palindromes in file enable1-word-list.txt:
aba
aga
...
```

Repetitive Code: File-Processing Methods

```
public static void printAllPalindromes(String filename) {
  try(Stream<String> words = Files.lines(Paths.get(filename))) {
    printAllPalindromes(words);
  } catch(IOException ioe) {
    System.err.println("Error reading file: " + ioe);
  }
}

public static void printPalindromes(String filename, int length) {
  try(Stream<String> words = Files.lines(Paths.get(filename))) {
    printPalindromes(words, length);
  } catch(IOException ioe) {
    System.err.println("Error reading file: " + ioe);
  }
}
```


Pros/Cons of Second Variation

- **Stream-processing method: good news**
  - Can be tested with any Stream<String>, not only with file
  - Depending on operations used, could be rewritten to take a Stream<T>
- **File-processing method: good news**
  - Filename passed in, not hardcoded
  - Errors handled explicitly
  - Stream closed automatically
- **File-processing method: bad news**
  - Contains lots of tedious boilerplate code that must be repeated for each application
    - 90% of code on previous slide was repeated
  - Hint for next variation: we had same problem when using Arrays.sort
    - We used lambdas to avoid the repetition

File Reading: Third Variation

Third Variation: Overview

- **Previous variation**
  - Method that takes Stream<String> or Stream<T> and performs general Stream ops
    - Good in every way
  - Method that takes filename, makes Stream<String>, passes it to first method
    - Good in some ways: the filename is passed in, errors are handled explicitly, and the Stream is always closed
    - Bad in some ways: repeats the Path creation, Stream creation, and error handling each time
- **New variation**
  - Method that takes Stream<String> or Stream<T> and performs general Stream ops
    - Exactly the same as above
  - Method that takes filename, then calls static method with that filename and a lambda or method reference designating the above method
    - Requires us to create a new functional interface with abstract method that takes a Stream<String> and static method that does the boilerplate code

Use Lambdas to Reuse Repeated Code

- **New interface: StreamProcessor**
  - Abstract method takes a Stream<String>
  - Static method takes filename and instance of the interface (usually as a lambda), calls Files.lines, and passes result to the abstract method. Uses try/catch block and try-with-resources.
- **Stream-processing method**
  - Same as before: processes Stream<String>
- **File-processing method**
  - Calls static method with two arguments:
    - Filename
    - Lambda designating the method that should get the Stream<String> that will come from the file

Variation 3: General Approach

```
public static void useStream(Stream<String> lines) {
  lines.filter(...).map(...)...;
}

public static void useFile(String filename) {
  StreamProcessor.processFile(filename, SomeClass::useStream);
}
```

We must define this static method.

In order to pass in a method reference or explicit lambda here, the method must take a functional (1-abstract-method) interface as its second argument. We must define that interface, and its single method must take a Stream<String>.

Variation 3: StreamProcessor Interface

```
@FunctionalInterface
public interface StreamProcessor {
  void processStream(Stream<String> strings);

  public static void processFile(String filename,
                                 StreamProcessor processor) {
    try(Stream<String> lines = Files.lines(Paths.get(filename))) {
      processor.processStream(lines);
    } catch(IOException ioe) {
      System.err.println("Error reading file: " + ioe);
    }
  }
}
```

This is the single abstract method of the interface. Since it takes a Stream<String> as argument, we can supply a method reference or lambda that refers to a method that takes a Stream<String>, as with SomeClass::useStream on previous slide.

A call to this static method will be the body of the file-processing methods. It will be supplied the filename and a method reference that points to the stream-processing method.

Variation 1: Printing All Palindromes

```
public static void main(String[] args) throws Exception {
  String inputFile = "enable1-word-list.txt";
  Files.lines(Paths.get(inputFile))
       .filter(StringUtils::isPalindrome)
       .forEach(System.out::println);
}
```

Variation 2: Printing All Palindromes

```
public class FileUtils {
  public static void printAllPalindromes(Stream<String> words) {
    words.filter(StringUtils::isPalindrome)
         .forEach(System.out::println);
  }

  public static void printAllPalindromes(String filename) {
    try(Stream<String> words = Files.lines(Paths.get(filename))) {
      printAllPalindromes(words);
    } catch(IOException ioe) {
      System.err.println("Error reading file: " + ioe);
    }
  }
}
```


Variation 3: Printing All Palindromes

```
public static void printAllPalindromes(Stream<String> words) {
  words.filter(StringUtils::isPalindrome)
       .forEach(System.out::println);
}

public static void printAllPalindromes(String filename) {
  StreamProcessor.processFile(filename,
                              FileUtils::printAllPalindromes);
}
```

Printing All Palindromes (Test Code)

```
public static void testAllPalindromes(String filename) {
  List<String> testWords =
      Arrays.asList("bog", "bob", "dam", "dad");
  System.out.printf("All palindromes in list %s:%n", testWords);
  FileUtils.printAllPalindromes(testWords.stream());
  System.out.printf("All palindromes in file %s:%n", filename);
  FileUtils.printAllPalindromes(filename);
}
```

Output

```
All palindromes in list [bog, bob, dam, dad]:
bob
dad
All palindromes in file enable1-word-list.txt:
aa
aba
...
```


Printing N-Length Palindromes

```
public static void printPalindromes(Stream<String> words,
                                    int length) {
  words.filter(word -> word.length() == length)
       .filter(StringUtils::isPalindrome)
       .forEach(System.out::println);
}

public static void printPalindromes(String filename,
                                    int length) {
  StreamProcessor.processFile(filename,
                lines -> printPalindromes(lines, length));
}
```

Printing N-Length Palindromes (Test Code)

```
public static void testPalindromes(String filename, int... lengths) {
  List<String> testWords =
      Arrays.asList("rob", "bob", "reed", "deed");
  for(int length: lengths) {
    System.out.printf("%s-letter palindromes in list %s:%n",
                      length, testWords);
    FileUtils.printPalindromes(testWords.stream(), length);
    System.out.printf("%s-letter palindromes in file %s:%n",
                      length, filename);
    FileUtils.printPalindromes(filename, length);
  }
}
```

Output

```
3-letter palindromes in list [rob, bob, reed, deed]:
bob
3-letter palindromes in file enable1-word-list.txt:
aba
...
4-letter palindromes in list [rob, bob, reed, deed]:
deed
4-letter palindromes in file enable1-word-list.txt:
abba
...
```


Pros/Cons of Third Variation

- **Stream-processing method: good news (same as before)**
  - Can be tested with any Stream<String>, not only with file
  - Depending on operations used, could be rewritten to take a Stream<T>
- **File-processing method: good news**
  - Filename passed in, not hardcoded
  - Errors handled explicitly
  - Stream closed automatically
  - No repetition of the code that reads the file and handles the exception
- **File-processing method: bad news**
  - The stream-processing method had to have a void return type
    - I.e., it simply did side effects, not returned a value
  - Hint for next variation: in normal Java code, what do we do when we want to refer to a value, but we don't know what type the value is?
    - Use generic types

File Reading: Fourth Variation

Overview

- **Previous variation**
  - Method that takes Stream<String> or Stream<T> and performs general Stream ops
    - No value returned
  - Method that takes filename, then calls static method with that filename and a lambda or method reference designating the above method
    - This assumes that the above method has void return type
- **New variation**
  - Method that takes Stream<String> or Stream<T> , performs general Stream ops, and returns a value
  - Method that takes filename, then returns a value that is the result of calling a static method with that filename and a lambda or method reference designating the above method
    - This method now returns whatever the above method would return

Variation 4: General Approach

```
public static SomeType getValueFromStream(Stream<String> lines) {
  return(lines.filter(...).map(...)...);
}

public static SomeType getValueFromStream (String filename) {
  return
    StreamAnalyzer.analyzeFile(filename,
                               SomeClass::getValueFromStream);
}
```

As before, we need to define this static method.

As before, we will have to define a functional (1-abstract-method) interface to be used here. As before, the abstract method will take a Stream<String> as an argument, but this time it will have a generic return type instead of a void return type.

Variation 4: StreamAnalyzer Interface

```
@FunctionalInterface
public interface StreamAnalyzer<T> {
  T analyzeStream(Stream<String> strings);

  public static <T> T analyzeFile(String filename,
                                  StreamAnalyzer<T> analyzer) {
    try(Stream<String> lines = Files.lines(Paths.get(filename))) {
      return(analyzer.analyzeStream(lines));
    } catch(IOException ioe) {
      System.err.println("Error reading file: " + ioe);
      return(null);
    }
  }
}
```

Example: First Palindrome

```
public static String firstPalindrome(Stream<String> words) {
  return(words.filter(StringUtils::isPalindrome)
              .findFirst()
              .orElse(null));
}


public static String firstPalindrome(String filename) {
  return(StreamAnalyzer.analyzeFile(filename,
                                    FileUtils::firstPalindrome));
}
```


First Palindrome (Test Code)

```
public static void testFirstPalindrome(String filename) {
  List<String> testWords =
      Arrays.asList("bog", "bob", "dam", "dad");
  String firstPalindrome =
      FileUtils.firstPalindrome(testWords.stream());
  System.out.printf("First palindrome in list %s is %s.%n",
                    testWords, firstPalindrome);
  firstPalindrome = FileUtils.firstPalindrome(filename);
  System.out.printf("First palindrome in file %s is %s.%n",
                    filename, firstPalindrome);
}
```

Output

```
First palindrome in list [bog, bob, dam, dad] is bob.
First palindrome in file enable1-word-list.txt is aa.
```

Advanced Option: Combining Predicates

Overview: Common Pattern

- **Finding one value**

```
stream.filter(test1).filter(test2).filter(test3)...
      .findFirst().orElse(null);
```

- **Finding a List of values**

```
stream.filter(test1).filter(test2).filter(test3)...
      .collect(Collectors.toList());
```

- **Problem in both cases**
  - You do not know how many filter operations will be done
- **Solution**
  - Use varargs
    - Allow any number of Predicates
    - Combine them into a single Predicate that can be used in a single call to filter
    - No need to limit this part to Stream<String>

Predicate Combiner

```
public class FileUtils {
  @SafeVarargs
  public static <T> Predicate<T> combinedPredicate
                                      (Predicate<T>... tests) {
    Predicate<T> result = e -> true;
    for(Predicate<T> test: tests) {
      result = result.and(test);
    }
    return(result);
  }
  ...
}
```

Given any number (including 0) Predicates, this static method produces a single Predicate, that, for a given argument, returns true only when all the other Predicates would have returned true.

@SafeVarargs is difficult to understand. The issue is that it is not always safe to use varargs for generic types: the resultant array can have runtime type problems if you modify entries in it. But, if you only read the values and never modify them, varargs is perfectly safe. @SafeVarargs says "I am not doing anything dangerous, please suppress the compiler warnings".
For details, see http://docs.oracle.com/javase/8/docs/technotes/guides/language/non-reifiable-varargs.html

Directly Applying Predicate Combiner (Main Methods)

```
@SafeVarargs
public static int letterCount(Stream<String> words,
                              Predicate<String>... tests) {
  Predicate<String> combinedTest = FileUtils.combinedPredicate(tests);
  return(words.filter(combinedTest)
              .mapToInt(String::length)
              .sum());
  }

@SafeVarargs
public static Integer letterCount(String filename,
                                  Predicate<String>... tests) {
  return(StreamAnalyzer.analyzeFile(filename,
                       stream -> letterCount(stream, tests)));
}
```

Directly Applying Combiner (Test Code)

```
public static void testLetterCount(String filename) {
  List<String> testWords = Arrays.asList("hi", "hello", "hola");
  System.out.printf("In list %s:%n", testWords);
  int sum1 = FileUtils.letterCount(testWords.stream(),
                                   word -> word.contains("h"),
                                   word -> !word.contains("i"));
  printLetterCountResult(sum1, "contain h but not i");
  System.out.printf("In file %s:%n", filename);
  int sum2 = FileUtils.letterCount(filename, StringUtils::isPalindrome);
  printLetterCountResult(sum2, "are palindromes");
  int sum3 = FileUtils.letterCount(filename,
                                   word -> word.contains("q"),
                                   word -> !word.contains("qu"));
  printLetterCountResult(sum3, "contain q but not qu");
  int sum4 = FileUtils.letterCount(filename, word -> true);
  printLetterCountResult(sum4, "are in English language");
}

private static void printLetterCountResult(int sum, String message) {
  System.out.printf("  %,d total letters in words that %s.%n", sum, message);
```


Directly Applying Combiner (Results)

```
In list [hi, hello, hola]:
  9 total letters in words that contain h but not i.
In file enable1-word-list.txt:
  417 total letters in words that are palindromes.
  163 total letters in words that contain q but not qu.
  1,570,550 total letters in words that are in English language.
```

Indirectly Applying Combiner: firstMatch

```
@SafeVarargs
public static <T> T firstMatch(Stream<T> elements,
                               Predicate<T>... tests) {
  Predicate<T> combinedTest = FileUtils.combinedPredicate(tests);
  return(elements.filter(combinedTest)
                 .findFirst()
                 .orElse(null));
}

@SafeVarargs
public static String firstMatch(String filename,
                                Predicate<String>... tests) {
  return(StreamAnalyzer.analyzeFile(filename,
                stream -> firstMatch(stream, tests)));
}
```


Applying firstMatch

```
public static void testFirstMatch(String filename) {
  List<Integer> testNums = Arrays.asList(1, 10, 2, 20, 3, 30);
  Integer match1 = FileUtils.firstMatch(testNums.stream(),
                                        n -> n > 2,
                                        n -> n < 10,
                                        n -> n % 2 == 1);
  System.out.printf("First word in list %s that is greater " +
                    "than 2, less than 10, and odd is %s.%n",
                    testNums, match1);
  String match2 = FileUtils.firstMatch(filename,
                                       word -> word.contains("q"),
                                       word -> !word.contains("qu"));
  System.out.printf("First word in file %s with q but " +
                    "not u is %s.%n", filename, match2);
}
```

Output

```
First word in list [1, 10, 2, 20, 3, 30] that is greater than 2, less than 10, and odd is 3.
First word in file enable1-word-list.txt with q but not u is buqsha.
```

Indirectly Applying Combiner: allMatches

```
@SafeVarargs
public static <T> List<T> allMatches(Stream<T> elements,
                                     Predicate<T>... tests) {
  Predicate<T> combinedTest = FileUtils.combinedPredicate(tests);
  return(elements.filter(combinedTest)
                 .collect(Collectors.toList()));
  }

@SafeVarargs
public static List<String> allMatches(String filename,
                                      Predicate<String>... tests) {
  return(StreamAnalyzer.analyzeFile(filename,
                    stream -> allMatches(stream, tests)));
}
```


Applying allMatches

```
public static void testAllMatches(String filename) {
  List<Integer> testNums = Arrays.asList(2, 4, 6, 8, 10, 12);
  List<Integer> matches1 = FileUtils.allMatches(testNums.stream(),
                                                n -> n > 5,
                                                n -> n < 10);
  System.out.printf("All numbers in list %s that are " +
                    "greater than 5 and less than 10: %s.%n",
                    testNums, matches1);
  List<String> matches2 = FileUtils.allMatches(filename,
                                               word -> word.contains("q"),
                                               word -> !word.contains("qu"));
  System.out.printf("All words in file %s with q " +
                    "but not u: %s.%n", filename, matches2);
}
```

Output

```
All numbers in list [2, 4, 6, 8, 10, 12] that are greater than 5 and less than 10: [6, 8].
All words in file enable1-word-list.txt with q but not u: [buqsha, buqshas, faqir, ...].
```

Wrap-Up

Summary

- **Variation 1 (last section)**
  - Put all code inside main; main throws Exception
    - Simple and easy, but not reusable
- **Variation 2**
  - Method 1 handles Stream; method 2 calls Files.lines and passes Stream to method 1
    - Reusable, but each version of method 2 repeats a lot of boilerplate code
- **Variation 3**
  - Use lambdas to avoid the repetition
- **Variation 4**
  - Use generic types so that values can be returned
- **Varargs for combining Predicates**
  - Point: fancy Stream-processing becomes fancy file processing

coreservlets.com – custom onsite training







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 (tutorials on Java 7, 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.