File I/O in Java 8 Part 1:

Treating Files as Streams of Strings

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.

more SERVLETS and JAVASERVER PAGES
MARTY HALL
core SERVLETS and JAVASERVER PAGES
2ND EDITION
MARTY HALL · LARRY BROWN

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

- **More on try/catch blocks**
  - finally, multicatch, try-with-resources
- **Representing file paths**
  - Paths.get
- **Reading files by treating them as streams of strings**
  - Files.lines
- **Writing files**
  - Files.write
- **Exploring folders by treating them as streams of paths**
  - Files.list, Files.walk, Files.find

Quick Aside: More on try/catch Blocks

Summary

Covered earlier: basics

```
try {
  statement1;
  statement2;
  ...
} catch(Eclass1 var1) {
  ...
} catch(Eclass2 var2) {
  ...
} catch(Eclass3 var3) {
  ...
}
...
```

New: finally

```
try {...
} catch(...) {...
} finally {
   ...
}
```

New: multicatch

```
try {...
} catch(Eclass1 | Eclass e) {
   ...
} ...
```

New: try with resources

```
try (SomeAutoCloseable var = ...) {...
} catch(...) { ...
} ...
```

Finally Blocks

- **Idea**
  - The finally { … } block at the end of a try/catch is called whether or not there is an exception
- **Motivation: resetting resources, closing sockets, other cleanup**

```
HugeDataStructure blah = ...;
try {
  doSomethingWith(blah);
  ...
} catch {
  ...
} finally {
   blah = null;
}
```

Finally Blocks: Benefits

- **Question: difference between these two?**

Finally Block	Code After Entire try/catch

`try {`
`  ...`
`} catch(ExceptionType e) {`
`  ...`
`} finally {`
`  doSomeCleanup();`
`}`
	`try {`
	`  ...`
	`} catch(ExceptionType e) {`
	`  ...`
	`}`
	`doSomeCleanup();`

- **Answer: nested try/catch blocks**
  - In the example on the right above, if the catch throws an exception and the entire try/catch block is inside another try/catch block, the cleanup code might not run
    - Same issue if there is return, break, or continue
    - Many developers advocate always using finally for required cleanup, even if code does not (currently) have nested exception, return statement, etc.

Multicatch

- **Idea: can catch multiple exceptions using |**
  - In Java 7 and later, if two different catch blocks will do the same thing, you can catch more than one in the same catch clause (but also consider catching a parent type):

```
try { ... } catch(Eclass1 | Eclass2 e) { ... }
```

- **Example**

Without Multicatch	With Multicatch

String input = getSomeString();
int num;
try {
  num = Integer.parseInt(input);
} catch(NumberFormatException nfe) {
  num = someDefault;
} catch(NullPointerException npe) {
  num = someDefault;
}
	String input = getSomeString();
	int num;
	try {
	  num = Integer.parseInt(input);
	} catch(NumberFormatException | NullPointerException e) {
	  num = someDefault;
	}

try-with-resources: Overview

- **Idea**
  - In Java 7 and later, you can declare variables that implement AutoCloseable in parens after try.
    - Scope of variable is scope of try/catch block
    - The close method of each variable is called at the end, whether or not there is an exception (i.e., as if the call to close were in a finally block)
    - Can declare multiple variables, separated by semicolon
- **Example**

```
try (BufferedReader reader = …)  {
  readSomeDataWith(reader);
  ...
} catch(SomeExceptionType e) {
  ...
}
```

try-with-resources: Benefits

Without	With
`BufferedReader reader;`	`try(BufferedReader reader = ...) {`
`try {`	`  ...`
`  reader = ...;`	`} catch(SomeExceptionType e) {`
`  ...`	`  ...`
`} catch(SomeExceptionType e) {`	`}`
`  ...`	
`} finally {`	
`   reader.close();`	
`}`	

- **Advantages of approach on right**
  - Shorter and simpler
  - Can’t forget to call close
  - The reader variable is out of scope after the try/catch block finishes

Paths

Idea

- **Path is a simpler and more flexible replacement for File class**
  - And is main starting point for file I/O operations
- **Get a Path with Paths.get**

```
Path p1 = Paths.get("some-file");
Path p2 = Paths.get("/usr/local/gosling/some-file");
Path p3 =
  Paths.get("C:\\Users\\Gosling\\Documents\\some-file");
```

  - Notice the double backslashes above, because backslash already has meaning (escape next character) in Java strings
- **Paths have convenient methods**
  - toAbsolutePath, startsWith, endsWith, getFileName, getName, getNameCount, subpath, getParent, getRoot, normalize, relativize

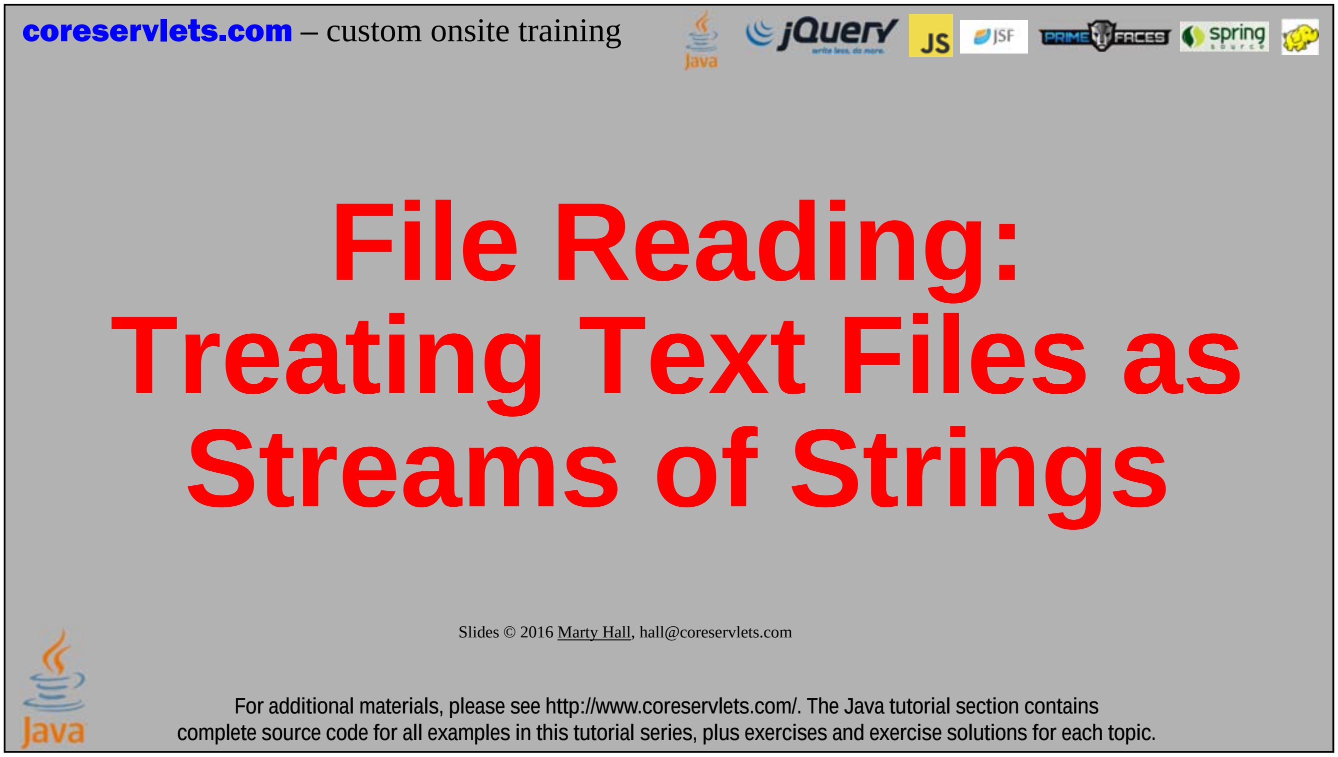
Example

```
public class PathExamples {
  public static void main(String[] args) {
    Path p1 = Paths.get("input-file.txt");
    System.out.println("Simple Path");
    System.out.printf("toString: %s%n%n", p1);
    Path p2 = p1.toAbsolutePath();
    System.out.println("Absolute Path");
    System.out.printf("toString: %s%n", p2);
    System.out.printf("getFileName: %s%n", p2.getFileName());
    System.out.printf("getName(0): %s%n", p2.getName(0));
    System.out.printf("getNameCount: %d%n", p2.getNameCount());
    System.out.printf("subpath(0,2): %s%n", p2.subpath(0,2));
    System.out.printf("getParent: %s%n", p2.getParent());
    System.out.printf("getRoot: %s%n", p2.getRoot());
  }
}
```

Example Output

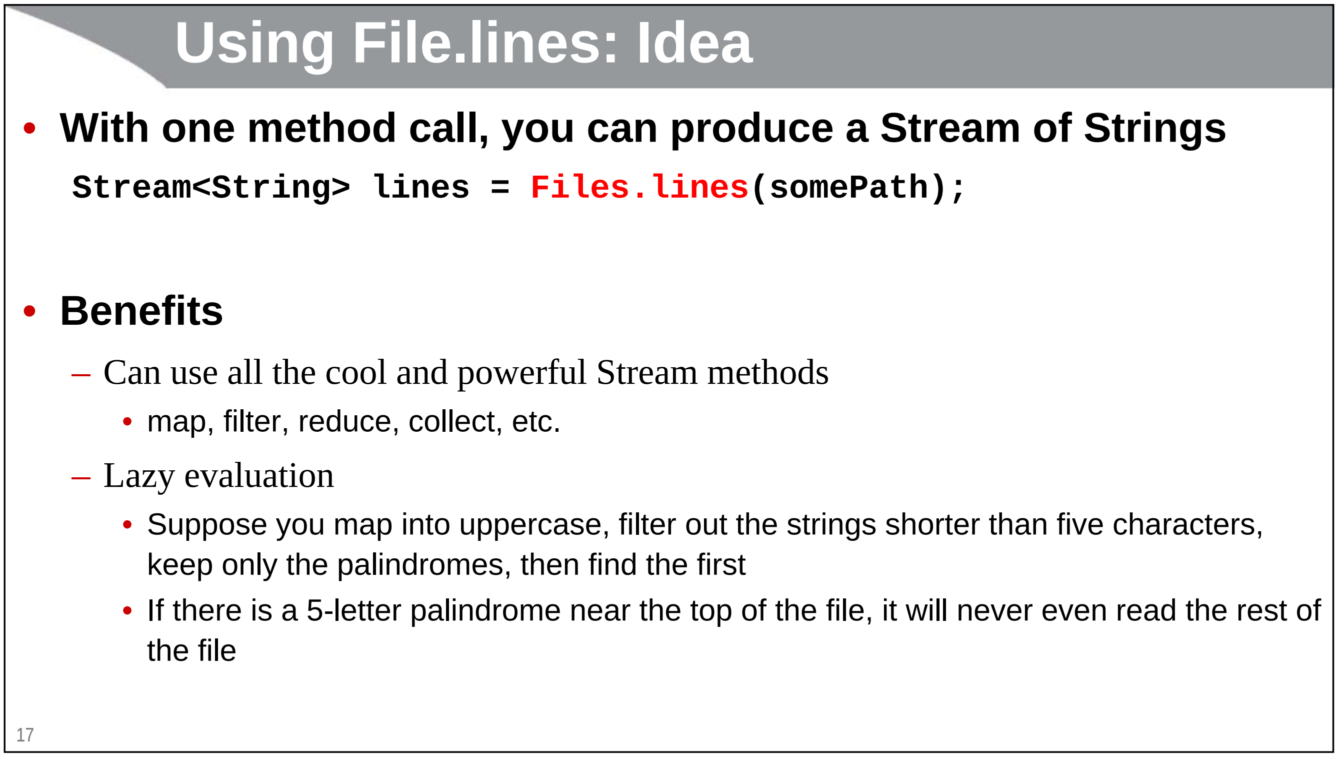
```
Simple Path
toString: input-file.txt

Absolute Path
toString: C:\eclipse-workspace\java\file-io\input-file.txt
getFileName: input-file.txt
getName(0): eclipse-workspace
getNameCount: 4
subpath(0,2): eclipse-workspace\java
getParent: C:\eclipse-workspace\java\file-io
getRoot: C:\
```



File Reading: Treating Text Files as Streams of Strings

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.



Using File.lines: Idea

- **With one method call, you can produce a Stream of Strings**

```
Stream<String> lines = Files.lines(somePath);
```

- **Benefits**
  - Can use all the cool and powerful Stream methods
    - map, filter, reduce, collect, etc.
  - Lazy evaluation
    - Suppose you map into uppercase, filter out the strings shorter than five characters, keep only the palindromes, then find the first
    - If there is a 5-letter palindrome near the top of the file, it will never even read the rest of the file

Files.lines: More Details

- **Charset option**

  `Files.lines(path)`

  – Uses UTF-8

  `Files.lines(path, someCharset)`

  – Uses specified Charset
- **Throws IOException**
  - So, you must use try/catch block or throw the exception
- **Stream should be closed**
  - Most Streams do *not* need closing, but ones connected to I/O sources (as here) do
- **Stream implements AutoCloseable**
  - You can use try-with-resources to handle IOException and automatically call close() at the end

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**
  - Put all code inside main; main throws Exception
    - Simple and easy, but not reusable
- **Variation 2 (next section)**
  - 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 (next section)**
  - 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 (next section)**
  - Similar to variation 3 but uses generic types so that values can be returned

Examples: Processing Large Word List

- **The enable1 Scrabble™ word list**
  - Public-domain file containing over 175,000 words accepted by many Scrabble clubs
    - The name comes from Enhanced North American Benchmark LExicon (ENABLE).
  - It is almost twice as large as the *Official Scrabble Player's Dictionary*™, and contains slang, offensive words, and many obscure or questionable words
  - It contains no one-letter words and no super-long words, and is not endorsed in any way by Hasbro (maker of Scrabble) or Merriam Webster (publisher of *The Official Scrabble Player's Dictionary*).
  - Details at

    http://www.puzzlers.org/dokuwiki/doku.php?id=solving:wordlists:about:enable_readme

File Reading: First Variation

Overview

- **Basic approach**

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

- **Advantage: quick and easy**
  - Many data analysis tasks involve one-up cases to read and analyze log files
- **Disadvantage: not reusable**
  - Cannot do same task to Stream<String> that came from another source
  - Cannot test without a file
  - Calling main is inconvenient from other code

Examples

- **Example 1: file of 4-letter words**
  - Assume that the enable1 word list might have a few repeats, a few words in mixed case, and a few words out of alphabetical order
  - Produce file containing all four-letter words, in upper case, without repeats, and in alphabetical order
- **Example 2: all palindromes**
  - Print out all palindromes contained in the file
- **Example 3: first 6-letter palindrome**
  - Print the first 6-letter palindrome contained in the file
- **Example 4: q's not followed by u's**
  - Count how many words have q but no qu
- **Example 5: x's and y's**
  - Count total letters in all words that have both x and y

Example 1: Create File of 4-Letter Words

```
public static void main(String[] args) throws Exception {
  String inputFile = "enable1-word-list.txt";
  String outputFile = "four-letter-words.txt";
  int length = 4;
  List<String> words =
      Files.lines(Paths.get(inputFile))
           .filter(word -> word.length() == length)
           .map(String::toUpperCase)
           .distinct()
           .sorted()
           .collect(Collectors.toList());
  Files.write(Paths.get(outputFile), words, Charset.defaultCharset());
  System.out.printf("Wrote %s words to %s.%n",
                    words.size(), outputFile);
}
```

Resultant file

```
AAHS
AALS
ABAS
ABBA
ABBE
ABED
ABET
ABLE
ABLY
...
```

Files.write takes a List<String> and produces a file that contains each of the strings on a separate line. It is discussed in the next section.

Example 2: Print 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);
}
```

Output

```
aa
aba
abba
aga
aha
ala
alula
...
```

Example 2: isPalindrome Helper Method

```
public class StringUtils {
  public static String reverseString(String s) {
    return(new StringBuilder(s).reverse().toString());
  }

  public static boolean isPalindrome(String s) {
    return(s.equalsIgnoreCase(reverseString(s)));
  }
}
```

Example 3: Print First 6-Letter Palindrome

```
public static void main(String[] args) throws Exception {
  String inputFile = "enable1-word-list.txt";
  String firstPalindrome =
      Files.lines(Paths.get(inputFile))
           .filter(word -> word.length() == 6)
           .filter(StringUtils::isPalindrome)
           .findFirst()
           .orElse(null);
  System.out.printf("First 6-letter palindrome is %s.%n",
                    firstPalindrome);
}
```

Output

```
First 6-letter palindrome is denned.
```

Example 4: # of Words with q not Followed by u

```
public static void main(String[] args) throws Exception {
  String inputFile = "enable1-word-list.txt";
  long wordCount =
      Files.lines(Paths.get(inputFile))
           .filter(word -> word.contains("q"))
           .filter(word -> !word.contains("qu"))
           .count();
  System.out.printf("%s words with q but not u.%n", wordCount);
}
```

Output

```
29 words with q but not u.
```

Example 5: Total Letters in Words with Both x & y

```
public static void main(String[] args) throws Exception {
  String inputFile = "enable1-word-list.txt";
  int letterCount =
      Files.lines(Paths.get(inputFile))
           .filter(word -> word.contains("x"))
           .filter(word -> word.contains("y"))
           .mapToInt(String::length)
           .sum();
  System.out.printf("%,d total letters in words with " +
                    "both x and y.%n", letterCount);
}
```

Output

```
8,556 total letters in words with both x and y.
```

Preview of Later Variations

- **General principle**
  - Streams help make handling large data sets more convenient and efficient
    - This was seen even in this first variation that uses Files.lines to get Stream<String>
    - Use of convenient Stream methods makes it relatively easy to do complex file reading tasks. Arguably as convenient as Python and Perl.
    - Lazy evaluation and the fact that Streams are not stored in memory all at once makes file processing efficient.
  - Lambdas and generic types help make code more flexible and reusable
    - In examples so far, code was not easily reusable
    - Variations 2 and especially 3 will show how lambdas can help
    - Variation 4 will show how generic types can help further
    - Followon examples will show how advanced lambda processing (combining predicates) can easily apply to file processing

Simple File Writing

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.

Idea

- **You can write all lines in one method call**

```
List<String> lines = …;
Files.write(somePath, lines, someCharset);
```

  - Recall that you can turn Stream into List with stream.collect(Collectors.toList()).
  - You can actually use any Iterable<String>, not just List<String>. You would think you could also use List<Object>, and the system would call toString on each Object automatically. Sadly, no. Boo.

- **You can write all bytes in one method call**

```
byte[] fileArray = …;
Files.write(somePath, fileArray);
```

The OpenOption

- **Both versions of Files.write optionally take an OpenOption as final argument**
  - `Files.write(somePath, lines, someCharset, someOption);`
  - `Files.write(somePath, fileArray, someOption);`

- **Motivation**
  - Lets you specify whether to create file if it doesn’t exist, whether to overwrite or append, and so forth
  - Default behavior is to create file if not there and to overwrite if it is there

Example 1: Write Strings to File

```
public class WriteFile1 {
  public static void main(String[] args) throws Exception {
    Charset characterSet = Charset.defaultCharset();
    Path path = Paths.get("output-file-1.txt");
    List<String> lines =
      Arrays.asList("Line One", "Line Two", "Final Line");
    Files.write(path, lines, characterSet);
  }
}
```

- Source of output-file-1.txt after execution

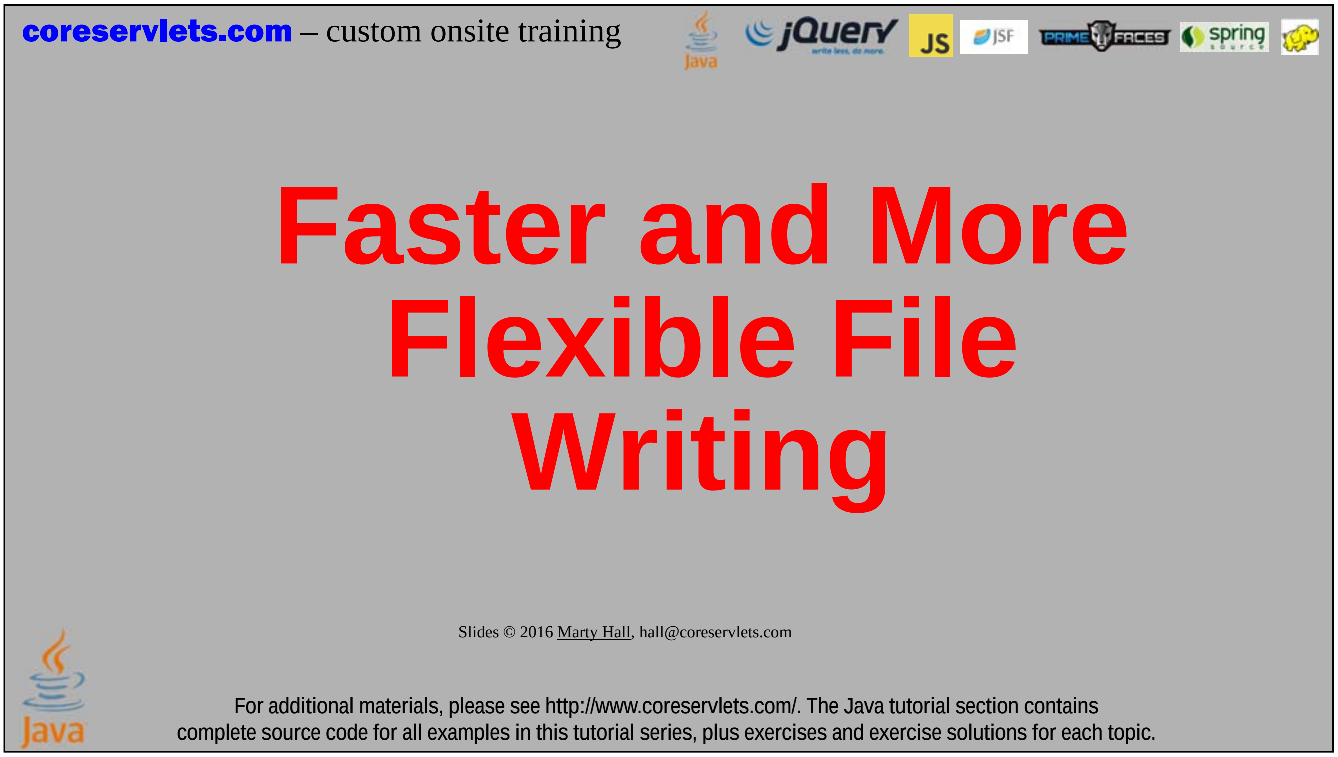
```
Line One
Line Two
Final Line
```

Example 2: File of 4-Letter Words (Shown Earlier)

```
public static void main(String[] args) throws Exception {
  String inputFile = "enable1-word-list.txt";
  String outputFile = "four-letter-words.txt";
  int length = 4;
  List<String> words =
      Files.lines(Paths.get(inputFile))
           .filter(word -> word.length() == length)
           .map(String::toUpperCase)
           .distinct()
           .sorted()
           .collect(Collectors.toList());
  Files.write(Paths.get(outputFile), words, Charset.defaultCharset());
  System.out.printf("Wrote %s words to %s.%n",
                    words.size(), outputFile);
}
```

Resultant file

```
AAHS
AALS
ABAS
ABBA
ABBE
ABED
ABET
ABLE
ABLY
...
```



coreservlets.com – custom onsite training

Faster and More Flexible File Writing

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.

Overview

- **You often need to format Strings**
  - Files.write does not let you format the Strings as you insert them into the file
- **Need higher performance for very large files**
  - You do not want to store everything in memory as List all at once
  - Buffered writing writes in blocks, and is faster for very large files
- **Shortcut method for getting BufferedWriter**

```
Writer w = Files.newBufferedWriter(somePath, someCharset);
w.write(...);
```

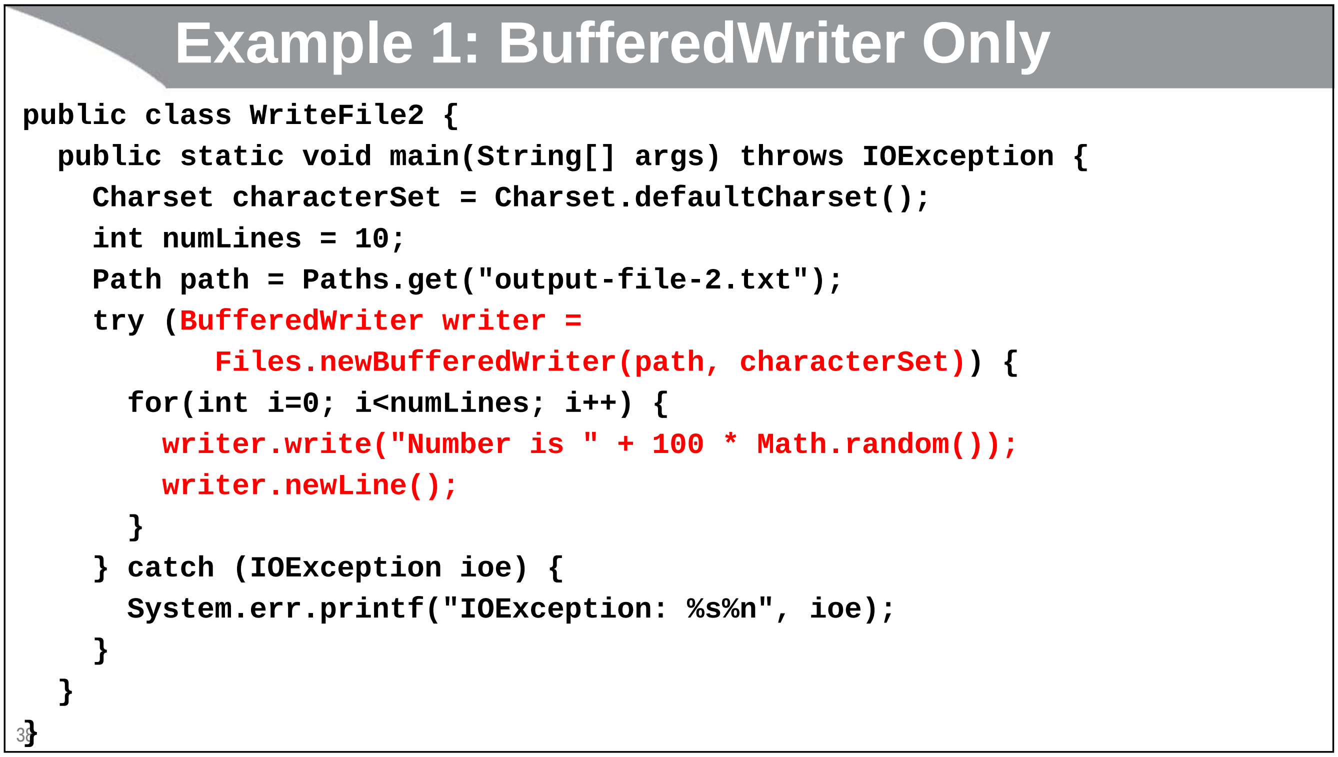
- **You usually wrap PrintWriter around the Writer**
  - Writer has only simple write method, but you can do

```
PrintWriter out = new PrintWriter(yourBufferedWriter);
```

then use the print, println, and especially printf methods of PrintWriter

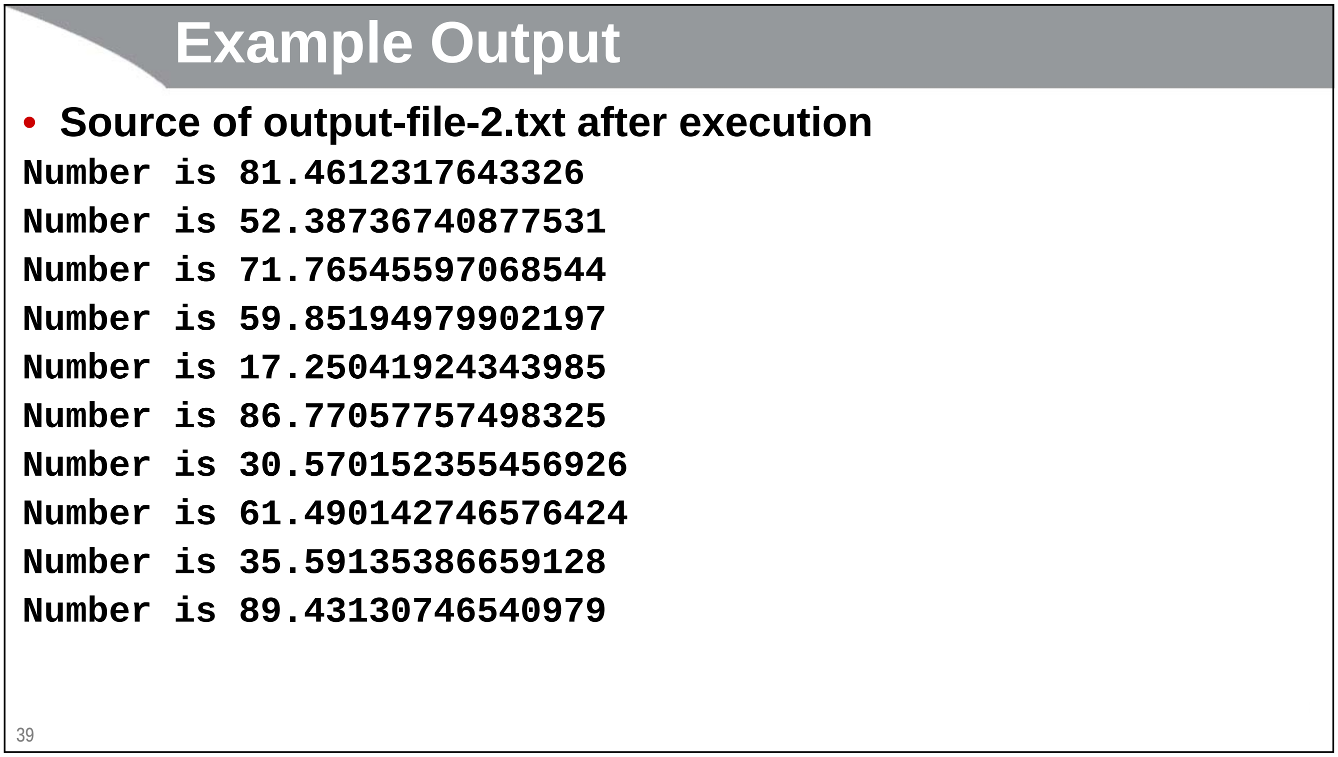
```
out.printf(...);
```

  - printf covered in lecture on Miscellaneous Utilities



Example 1: BufferedWriter Only

```
public class WriteFile2 {
  public static void main(String[] args) throws IOException {
    Charset characterSet = Charset.defaultCharset();
    int numLines = 10;
    Path path = Paths.get("output-file-2.txt");
    try (BufferedWriter writer =
           Files.newBufferedWriter(path, characterSet)) {
      for(int i=0; i<numLines; i++) {
        writer.write("Number is " + 100 * Math.random());
        writer.newLine();
      }
    } catch (IOException ioe) {
      System.err.printf("IOException: %s%n", ioe);
    }
  }
}
```



Example Output

- Source of output-file-2.txt after execution

```
Number is 81.4612317643326
Number is 52.38736740877531
Number is 71.76545597068544
Number is 59.85194979902197
Number is 17.25041924343985
Number is 86.77057757498325
Number is 30.570152355456926
Number is 61.490142746576424
Number is 35.59135386659128
Number is 89.43130746540979
```

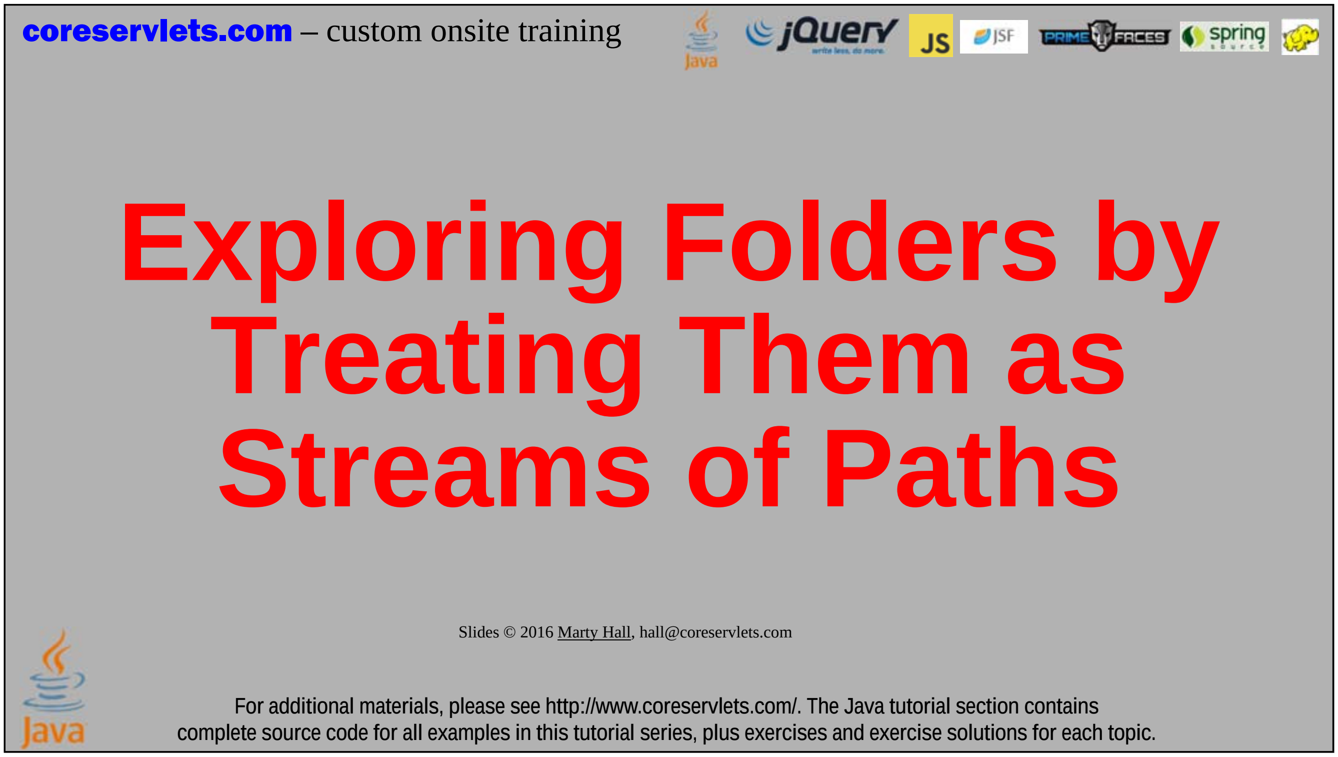
Example 2: PrintWriter

```
public class WriteFile3 {
  public static void main(String[] args) throws IOException {
    Charset characterSet = Charset.defaultCharset();
    int numLines = 10;
    Path path = Paths.get("output-file-3.txt");
    try (PrintWriter out =
           new PrintWriter(Files.newBufferedWriter(path, characterSet))) {
      for(int i=0; i<numLines; i++) {
        out.printf("Number is %5.2f%n", 100 * Math.random());
      }
    } catch (IOException ioe) {
      System.err.printf("IOException: %s%n", ioe);
    }
  }
}
```

Example Output

- Source of output-file-3.txt after execution

```
Number is 71.95
Number is 35.75
Number is 39.52
Number is 15.04
Number is  2.50
Number is 14.58
Number is 63.06
Number is 13.77
Number is 96.51
Number is  5.27
```

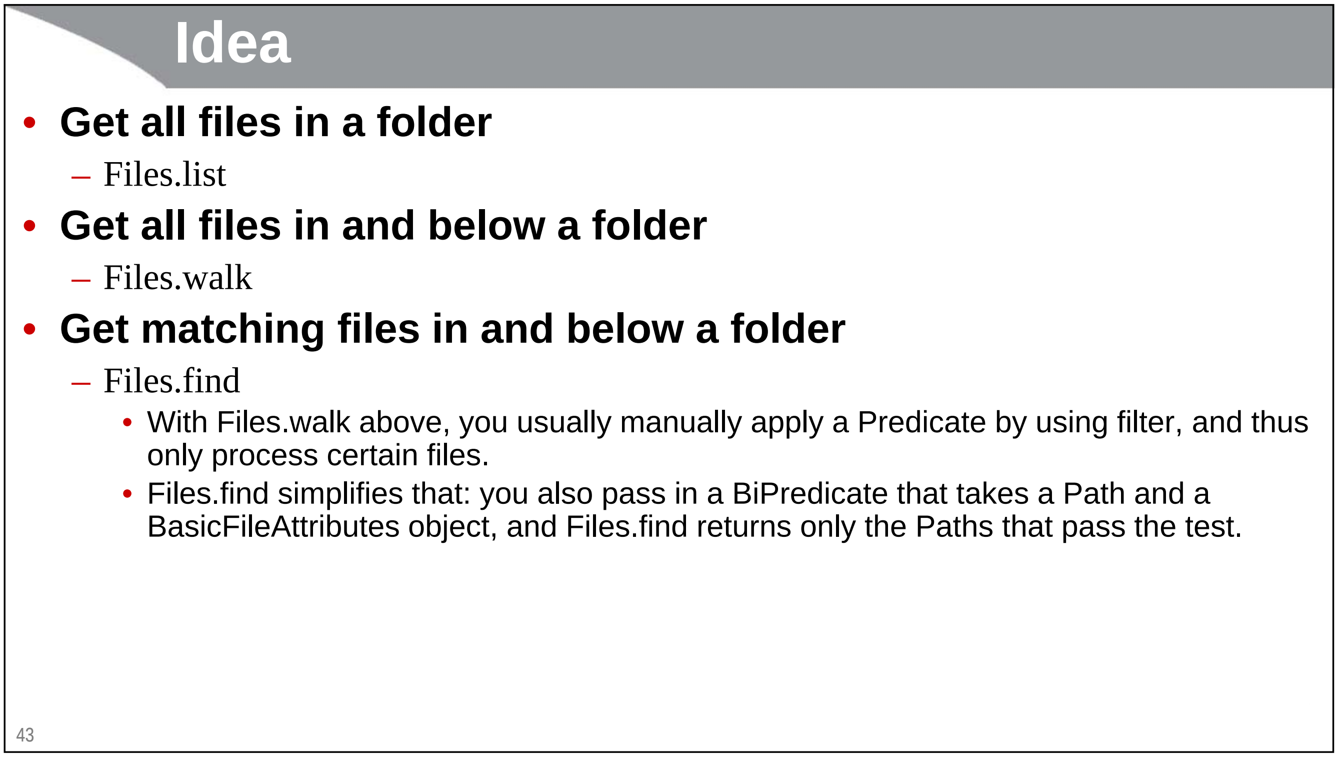


coreservlets.com – custom onsite training

Exploring Folders by Treating Them as Streams of Paths

Slides © 2016 Marty Hall, hall@coreservlets.com

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.



Idea

- **Get all files in a folder**
  - Files.list
- **Get all files in and below a folder**
  - Files.walk
- **Get matching files in and below a folder**
  - Files.find
    - With Files.walk above, you usually manually apply a Predicate by using filter, and thus only process certain files.
    - Files.find simplifies that: you also pass in a BiPredicate that takes a Path and a BasicFileAttributes object, and Files.find returns only the Paths that pass the test.

Example 1: Printing Files in Folder

```
public class FolderUtils
  public static void printAllPaths(Stream<Path> paths) {
    paths.forEach(System.out::println);
  }

  public static void printAllPathsInFolder(String folder) {
    try(Stream<Path> paths = Files.list(Paths.get(folder))) {
      printAllPaths(paths);
    } catch(IOException ioe) {
      System.err.println("IOException: " + ioe);
    }
  }
```

Example 1: Printing Files in Folder (Continued)

```
  public static void printPaths(Stream<Path> paths,
                                Predicate<Path> test) {
    paths.filter(test)
         .forEach(System.out::println);
  }

  public static void printPathsInFolder(String folder,
                                      Predicate<Path> test) {
    try(Stream<Path> paths = Files.list(Paths.get(folder))) {
      printPaths(paths, test);
    } catch(IOException ioe) {
      System.err.println("IOException: " + ioe);
    }
  }
```

Printing Files in Folder: Test Code

```
public static void listExamples() {
  System.out.println("All files in project root");
  FolderUtils.printAllPathsInFolder(".");
  System.out.println("Text files in project root");
  FolderUtils.printPathsInFolder(".",
            p -> p.toString().endsWith(".txt"));
}
```

```
All files in project root
.\.classpath
.\.project
.\.settings
.\coreservlets
.\dzone-programming-
language-list.txt
.\enable1-word-list.txt
.\four-letter-words.txt
.\input-file.txt
.\output-file-1.txt
.\output-file-2.txt
.\output-file-3.txt
.\unixdict.txt
Text files in project root
.\dzone-programming-
language-list.txt
.\enable1-word-list.txt
.\four-letter-words.txt
.\input-file.txt
.\output-file-1.txt
.\output-file-2.txt
.\output-file-3.txt
.\unixdict.txt
```

Example 2: Printing Files in Tree

```
public static void printAllPathsInTree(String rootFolder) {
   try(Stream<Path> paths = Files.walk(Paths.get(rootFolder))) {
     printAllPaths(paths);
   } catch(IOException ioe) {
     System.err.println("IOException: " + ioe);
   }
 }

 public static void printPathsInTree(String rootFolder,
                                     Predicate<Path> test) {
   try(Stream<Path> paths = Files.walk(Paths.get(rootFolder))) {
     printPaths(paths, test);
   } catch(IOException ioe) {
     System.err.println("IOException: " + ioe);
   }
 }
```

Files.walk also has options where you can limit the depth of the tree searched and where you can specify FileVisitOptions.

Printing Files in Tree: Test Code

```
public static void walkExamples() {
  System.out.println("All files under project root");
  FolderUtils.printAllPathsInTree(".");
  System.out.println("Java files under project root");
  FolderUtils.printPathsInTree(".",
                       p -> p.toString().endsWith(".java"));
}
```

```
All files under project root
.
.\.classpath
.\.project
.\.settings
.\.settings\org.eclipse.jdt.core.prefs
.\coreservlets
.\coreservlets\folders
.\coreservlets\folders\FolderExamples.class
.\coreservlets\folders\FolderExamples.java
.\coreservlets\folders\FolderUtils.class
.\coreservlets\folders\FolderUtils.java
.\coreservlets\java7
.\coreservlets\java7\FileUtils.class
.\coreservlets\java7\FileUtils.java
...
```

Example 3: Printing Matching Files in Tree

```
public static void findPathsInTree(String rootFolder,
              BiPredicate<Path,BasicFileAttributes> test) {
  try(Stream<Path> paths =
        Files.find(Paths.get(rootFolder), 10, test)) {
    printAllPaths(paths);
  } catch(IOException ioe) {
    System.err.println("IOException: " + ioe);
  }
}
```

In call above to Files.find, 10 is the maximum depth searched.

Printing Matching Files in Tree: Test Code

```
public static void findExamples() {
  System.out.println("Java files under project root");
  FolderUtils.findPathsInTree(".",
        (path,attrs) -> path.toString().endsWith(".java"));
  System.out.println("Folders under project root");
  FolderUtils.findPathsInTree(".",
        (path,attrs) -> attrs.isDirectory());
  System.out.println("Large files under project root");
  FolderUtils.findPathsInTree(".",
        (path,attrs) -> attrs.size() > 10000);
}
```

```
Java files under project root
...
.\coreservlets\folders\FolderExamples.java
.\coreservlets\folders\FolderUtils.java
...
Folders under project root
.
.\.settings
.\coreservlets
...
Large files under project root
.\enable1-word-list.txt
.\four-letter-words.txt
.\unixdict.txt
```

Wrap-Up

Summary: try/catch Blocks

- finally blocks

```
try { ...
} catch(SomeExceptionType e) { ...
} finally {
  ...
}
```

- multicatch

```
try {…
} catch(ExceptionType1 | ExceptionType2 e) {
   …
}
```

- try with resources

```
try (SomeAutoCloseable var = ...) { ...
} catch(SomeExceptionType e) { ...
}
```

Summary: File I/O in Java 8

- Use Path to refer to file location

```
Path somePath = Paths.get("/path/to/file.txt");
```

- Read all lines into a Stream

```
Stream<String> lines = Files.lines(somePath);
```

  - Can now use filter, map, distinct, sorted, findFirst, collect, etc.
  - You get benefits of lazy evaluation
  - In next section, we will make the code more reusable, flexible, and testable
- Write List or other Iterable into a file

```
Files.write(somePath, someList, someCharset);
```

- Get Writer for more flexible output

```
Files.newBufferedWriter(somePath, someCharset)
```

  - Use write method, or, more often, wrap in PrintWriter and use printf
- Explore and search folders and subfolders
  - Files.list, Files.walk, Files.find

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.