File IO in Java 8: Applying the Power of Streams

Topics in This Section

- **More on try/catch blocks**
  - finally, multicatch, try-with-resources
- **Representing file paths**
  - Paths.get
- **Reading files**
  - Files.lines
- **Writing files**
  - Files.write
- **Exploring folders**
  - Files.list, Files.walk, Files.find
- **Reading files in Java 7**
  - Files.readAllLines
- **Lower-level file IO utilities**
  - Files.newBufferedReader
  - Files.newBufferedWriter

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**

```
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(…) { …`
`} finally {`
`  doSomeCleanup();`
`}`
	`try { …`
	`} catch(…) { …`
	`}`
	`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.
    - So, usual practice for code that runs whether or not there is an exception is to simply put it below try/catch block (as in example on right above), but finally block is sometimes necessary.

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 strng = getSomeString();`
`int num;`
`try {`
`  num = Integer.parseInt(strng);`
`} catch(NumberFormatException nfe) {`
`  num = someDefault;`
`} catch(NullPointerException npe) {`
`  num = someDefault;`
`}`
	`String strng = getSomeString();`
	`int num;`
	`try {`
	`  num = Integer.parseInt(strng);`
	`} 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(…) {
  …
}
```

try-with-resources: Benefits

Without	With

BufferedReader reader;
try {
  reader = ...;
  ...
} catch (...) {
  ...
} finally {
  reader.close();
}
	try(BufferedReader reader = ...) {
	  ...
	} catch (...) {
	  ...
	}

- **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 flexible replacement for File**
  – 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 because backslash already has meaning (escape next char) 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("InputFile.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: InputFile.txt

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

Simple File Reading

Using File.lines: Idea

- **You can read all lines into Stream in one method call**
  – Stream<String> lines = Files.lines(somePath);
- **Benefits**
  – Can use all the cool and powerful Stream methods
    - map, filter, reduce, etc.
  – Lazy evaluation
    - Suppose you map into uppercase, filter out the strings shorter than four 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.

Using Files.lines: Example

- **Quick example**
  - Given 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 need to catch IOException
- You should close the Stream that results from Files.lines so that long-running code will not have memory leak.

Still, this gives the gist of the approach. Full code for this task (using try-with-resources) is given later in this tutorial section.

Java 8 vs. Java 7

- **Java 7 already let you read all lines into List**
  - List<String> lines =
    Files.readAllLines(somePath, someCharset);
- **But Stream version far better**
  - Massive memory savings
    - Does not store entire file contents in one huge list, but processes each line as you go along
  - Potentially much faster
    - You can stop partway through, and rest of file is never processed (due to lazy evaluation of Streams)
  - Many convenient filtering and transformation methods
    - You can chain these method calls together

Files.lines: More Details

- **Charset option**
  - Files.lines(path)
    - Uses UTF-8
  - Files.lines(path, someCharset)
    - Uses specified Charset
- **Throws IOException**
  - So must use try/catch block
- **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 handle IOException and automatically call close() via try-with-resources

Files.lines: Usage Template

```
public SomeType useStream(Stream<String> lines, ...) {
  return(lines.filter(…).map(…)…);
}


public SomeType useFile(String filename, ...) {
  try(Stream<String> lines =
        Files.lines(Paths.get(filename))) {
    return(useStream(lines, ...));
  } catch(IOException ioe) {
    System.err.println("IOException: " + ioe);
    return(null);
  }
}
```

Why Split the Processing?

- **Why use two methods?**
  - One that builds a Stream<String> from Files.lines.
  - One that actually processes the Stream<String>.
- **Benefits to splitting**
  - You simplify testing. You can test the second method with a basic Stream that you make with Stream.of or someList.stream().
  - It is more reusable. The second method can be used for Streams created from other sources.
  - It is more flexible. The second 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.

Examples: Processing Large Word List

- **The enable1 Scrabble™ word list**
  - Public-domain file containing over 175,000 supposed words accepted by many US 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

Printing All Palindromes: String-Specific Version

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

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

  ...
}
```

Printing All Palindromes: Generic Version

```
public class GenericFileUtils {
  public static <T> void printMatches(Stream<T> elements,
                                      Predicate<T> test) {
    elements.filter(test).forEach(System.out::println);
  }

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

Helper Methods: Detecting a Palindrome

```
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)));
  }

  ...
}
```

Printing All Palindromes: Test Code

```
public static void printPalindromeExamples(String filename) {
  System.out.println("All palindromes (String-specific version):");
  FileUtils.printPalindromes(filename);
  System.out.println("\nAll palindromes (generic version):");
  GenericFileUtils.printPalindromes(filename);
}
```

```
All palindromes (String-specific version):
aa
aba
abba
aga
aha
ala
alula
...
wow
yay

All palindromes (generic version):
aa
aba
abba
...
```

(identical results)

Used the enable1 word list as the input file.

Finding First Fancy Palindrome: String-Specific Version

```
public static String firstPalindrome(Stream<String> words,
                                     int n, String substring) {
  return(words.filter(word -> word.length() == n)
              .filter(word -> word.contains(substring))
              .filter(StringUtils::isPalindrome)
              .findFirst()
              .orElse(null));
}

public static String firstPalindrome(String filename,
                                     int n, String substring) {
  try(Stream<String> words =
        Files.lines(Paths.get(filename))) {
    return(firstPalindrome(words, n, substring));
  } catch(IOException ioe) {
    System.err.println("IOException: " + ioe);
    return(null);
  }
}
```

Finds first n-letter palindrome that contains the specified substring.

Finding First Fancy Palindrome: Generic Version

```
public static String firstPalindrome(Stream<String> words,
                                     int n, String substring) {
  return(firstMatch(words,
                    word -> word.length() == n,
                    word -> word.contains(substring),
                    StringUtils::isPalindrome));
}

public static String firstPalindrome(String filename,
                                     int n, String substring) {
  try(Stream<String> words =
        Files.lines(Paths.get(filename))) {
    return(firstPalindrome(words, n, substring));
  } catch(IOException ioe) {
    System.err.println("IOException: " + ioe);
    return(null);
  }
}
```

Finds first n-letter palindrome that contains the specified substring. Uses generic firstMatch method (next slide).

Generic Helper Methods: Finding Matches

```
@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);
}
```

Returns a Predicate that is the result of ANDing all the argument Predicates. If no Predicates are supplied, it returns a Predicate that always returns true.

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

Returns first element that matches all of the tests, null otherwise.

@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. @SafeVarargs says "I am not doing anything dangerous, please suppress the compiler warnings". For details, see http://docs.oracle.com/javase/7/docs/technotes/guides/language/non-reifiable-varargs.html

Finding First Fancy Palindrome: Test Code

```
public static void findPalindromeExamples(String filename) {
  System.out.println("Finding palindromes (String-specific version):");
  printPalindromeResult(5, "x", FileUtils.firstPalindrome(filename, 5, "x"));
  printPalindromeResult(6, "a", FileUtils.firstPalindrome(filename, 6, "h"));
  printPalindromeResult(7, "e", FileUtils.firstPalindrome(filename, 7, "f"));
  printPalindromeResult(8, "q", FileUtils.firstPalindrome(filename, 8, "q"));
  System.out.println("Finding palindromes (generic version):");
  printPalindromeResult(5, "x", GenericFileUtils.firstPalindrome(filename, 5, "x"));
  printPalindromeResult(6, "a", GenericFileUtils.firstPalindrome(filename, 6, "h"));
  printPalindromeResult(7, "e", GenericFileUtils.firstPalindrome(filename, 7, "f"));
  printPalindromeResult(8, "q", GenericFileUtils.firstPalindrome(filename, 8, "q"));
}

private static void printPalindromeResult(int n, String substring, String pal) {
  System.out.printf("First %s-letter palindrome containing '%s' is '%s'.%n",
                    n, substring, pal);
}
```

```
Finding palindromes (String-specific version):
First 5-letter palindrome containing 'x' is 'sexes'.
First 6-letter palindrome containing 'a' is 'hallah'.
First 7-letter palindrome containing 'e' is 'deified'.
First 8-letter palindrome containing 'q' is 'null'.
Finding palindromes (generic version):
First 5-letter palindrome containing 'x' is 'sexes'.
First 6-letter palindrome containing 'a' is 'hallah'.
First 7-letter palindrome containing 'e' is 'deified'.
First 8-letter palindrome containing 'q' is 'null'.
```

Finding Total Number of Chars of Words Passing Tests

```
public static int letterCount(Stream<String> words,
                              Predicate<String> test) {
  return(words.filter(test)
              .mapToInt(String::length)
              .sum());
}
```

The sum of the lengths of all words that pass the test.

```
public static int letterCount(String filename,
                              Predicate<String> test) {
  try(Stream<String> words =
        Files.lines(Paths.get(filename))) {
    return(letterCount(words, test));
  } catch(IOException ioe) {
    System.err.println("IOException: " + ioe);
    return(0);
  }
}
```

Finding Total Number of Chars: Test Code

```
public static void letterCountExamples(String filename) {
  printLetterCountResult(filename, StringUtils::isPalindrome,
                         "are palindromes");
  Predicate<String> containsQ = s -> s.contains("q");
  printLetterCountResult(filename, containsQ, "contain q");
  Predicate<String> containsQButNotU =
    containsQ.and(s -> !s.contains("u"));
  printLetterCountResult(filename, containsQButNotU,
                         "contain q but not u");
  printLetterCountResult(filename, s -> true, "are in file");
}

public static void printLetterCountResult
   (String filename, Predicate<String> testFunction, String message) {
  int sum = FileUtils.letterCount(filename, testFunction);
  System.out.printf("Sum of lengths of words that %s is %,d.%n",
                    message, sum);
}
```

```
Sum of lengths of words that are palindromes is 417.
Sum of lengths of words that contain q is 24,094.
Sum of lengths of words that contain q but not u is 119.
Sum of lengths of words that are in file is 1,570,540.
```

Simple File Writing

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 strm.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 the toString method of each Object automatically. Sadly, no.
- **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 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 IOException {
    Charset characterSet = Charset.defaultCharset();
    Path path = Paths.get("OutputFile1.txt");
    List<String> lines =
      Arrays.asList("Line One", "Line Two", "Final Line");
    Files.write(path, lines, characterSet);
  }
}
```

- **Source of OutputFile1.txt after execution**

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

Example 2: Store N-Letter Words

- **Task**
  - Given large file of words of various lengths in mixed case with possible repeats, create sorted uppercase file of n-letter words.
- **Comments**
  - File is large (1.5 million characters as shown in earlier example), so storing entire file in memory in a List as with the Java 7 Files.readAllLines described later would waste memory.
  - The set of n-letter words for any n is much smaller, so entire List can be stored in memory. Besides, since we will sort the results alphabetically, we have little choice but to store all the matches in memory.

N-Letter Words: Core Code

```
public static List<String> makeWordList(Stream<String> words,
                                        int wordLength) {
  return(words.filter(word -> word.length() == wordLength)
              .map(String::toUpperCase)
              .distinct()
              .sorted()
              .collect(Collectors.toList()));
}

public static void storeWordList(String inputFile, int wordLength,
                                 String outputFile) {
  try(Stream<String> words = Files.lines(Paths.get(inputFile))) {
    List<String> nLetterWords = makeWordList(words, wordLength);
    Path outputPath = Paths.get(outputFile);
    Files.write(outputPath, nLetterWords, Charset.defaultCharset());
    System.out.printf("Wrote %s words to %s.%n", nLetterWords.size(),
                      outputPath.toAbsolutePath());
  } catch(IOException ioe) {
    System.err.println("IOException: " + ioe);
  }
}
```

N-Letter Words: Test Code

```
public static void wordListExamples(String inputFile) {
  FileUtils.storeWordList(inputFile, 4,
                          "four-letter-words.txt");
  FileUtils.storeWordList(inputFile, 5,
                          "five-letter-words.txt");
}
```

```
Wrote 3903 words to C:\eclipse-workspace\java\file-io\four-letter-words.txt.
Wrote 8636 words to C:\eclipse-workspace\java\file-io\five-letter-words.txt.
```

Used the enable1 word list as the input file.

Exploring Folders

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 via filter, and thus and 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 (Cont.)

```
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
.\coreservlets
.\enable1-word-list.txt
.\five-letter-words.txt
.\four-letter-words.txt
.\InputFile.txt
.\OutputFile1.txt
.\OutputFile2.txt
.\OutputFile3.txt
.\unixdict.txt
Text files in project root
.\enable1-word-list.txt
.\five-letter-words.txt
.\four-letter-words.txt
.\InputFile.txt
.\OutputFile1.txt
.\OutputFile2.txt
.\OutputFile3.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
.\coreservlets
.\coreservlets\java7
.\coreservlets\java7\FileUtils.class
.\coreservlets\java7\FileUtils.java
...
Java files under project root
.\coreservlets\java7\FileUtils.java
.\coreservlets\java7\PathExamples.java
...
.\coreservlets\java8\FileReadingExamples.java
.\coreservlets\java8\FileUtils.java
.\coreservlets\java8\FileWritingExamples.java
.\coreservlets\java8\FolderExamples.java
.\coreservlets\java8\FolderUtils.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\java8\FileReadingExamples.java
.\coreservlets\java8\FileUtils.java
...
Folders under project root
.
.\coreservlets
...
Large files under project root
.\enable1-word-list.txt
.\five-letter-words.txt
.\four-letter-words.txt
.\unixdict.txt
```

Faster and More Flexible File Writing

Idea

- **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**
  – Buffered writing writes in blocks, and is faster for very large files
- **Shortcut method for getting BufferedWriter**
  – Files.newBufferedWriter(somePath, someCharset)
- **Can also use PrintWriter**
  – Writer has only simple write method, but you can do new PrintWriter(yourBufferedWriter), then use the print, println, and especially printf methods of PrintWriter
    - printf covered in lecture on More Syntax and 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("OutputFile2.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 OutputFile2.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("OutputFile3.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);
    }
  }
}
```

printf is covered in a separate tutorial section (More Syntax and Utilities). Even if you haven't seen that section however, the usage here is moderately understandable: %5.2f is a placeholder where the number get substituted in with 5 total spaces and 2 after the decimal point, and %n means a new line.

Example Output

- Source of OutputFile3.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
```

File IO in Java 7

Differences from Java 8

- **Read lines into List instead of Stream**
  - Files.readAllLines
    - Streams were not yet available in Java 7. Although returning a List is much less convenient than the current approach of returning a Stream, the Java 7 way was still substantially better than the Java 6 way.
- **Need way of reading large files**
  - Files.newBufferedReader
    - Since Lists do not support lazy evaluation like Streams do, reading large files with Files.readAllLines is inefficient because entire file contents is in memory all at once, and because you read entire file even if the data you need is near the top.

Simple File Reading in Java 7

- **You can read all lines into List in 1 method call**
  - List<String> lines =
    Files.readAllLines(somePath, someCharset);.
- **You can read all bytes in one method call**
  - byte[] fileArray = Files.readAllBytes(file);
    - Strings can easily be made from byte arrays:
      String fileData = new String(Files.readAllBytes(file));
- **Minor caveats**
  - You have to explicitly specify a Charset, even if you will use the default for the JDK
    - Charset cset1 = Charset.defaultCharset();
    - Charset cset2 = Charset.forName("UTF-8");
  - You still have to catch IOException

File Reading: Example

```
public class ReadFile1 {
  public static void main(String[] args) throws IOException {
    String file = "InputFile.txt";
    Charset characterSet = Charset.defaultCharset();
    Path path = Paths.get(file);
    List<String> lines =
      Files.readAllLines(path, characterSet);
    System.out.printf("Lines from %s: %s%n", file, lines);
  }
}
```

printf is covered in a separate tutorial section (More Syntax and Utilities). Even if you haven't seen that section however, the usage here is simple: %s is a placeholder where "file" and "lines" get substituted in, and %n means a new line. You could replace the printf with this only-slightly-clumsier println version:

System.out.println("Lines from " + file + ": " + lines);

Either way, note that you can directly print a List (unlike an array), because List has a readable toString method. The system will automatically put square brackets on the outside and separate entries with commas.

File Reading: Example Output

- **Source of InputFile.txt**

```
First line
Second line
Third line
Last line
```

- **Output of example code from previous slide**

```
Lines from InputFile.txt:
[First line, Second line, Third line, Last line]
```

Some Simple Java 7 Utilities

- **FileUtils.getLines("filename")**
  - Reading file into a List<String>
- **FileUtils.writeLines("filename", list)**
  - Writing file from a List<String>

```
public class FileUtils {
  public static List<String> getLines(String file)
        throws IOException {
    Path path = Paths.get(file);
    return(Files.readAllLines(path, Charset.defaultCharset()));
  }

  public static Path writeLines(String file, List<String> lines)
        throws IOException {
    Path path = Paths.get(file);
    return(Files.write(path, lines, Charset.defaultCharset()));
  }
}
```

Minor Variation of ReadFile1 (Using Utility Method)

```
public class ReadFile1A {
  public static void main(String[] args) throws IOException {
    String file = "InputFile.txt";
    List<String> lines = FileUtils.getLines(file);
    System.out.printf("Lines from %s: %s%n", file, lines);
  }
}
```

- **Output**
  - Same as ReadFile1. E.g.:

```
Lines from InputFile.txt:
[First line, Second line, Third line, Last line]
```

Minor Variation of WriteFile1

```
public class WriteFile1A {
  public static void main(String[] args) throws IOException {
    List<String> lines =
      Arrays.asList("Line One", "Line Two", "Final Line");
    FileUtils.writeLines("OutputFile1.txt", lines);
  }
}
```

- **Source of OutputFile1.txt after execution**

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

Lower-Level but More Flexible File Reading

- **You sometimes need only part of the file**
  - Files.readAllLines reads everything, which is wasteful
    - Stores entire file in memory
    - No way to stop if you find the info you need early in file
- **Need higher performance for very large files**
  - Buffered reading reads in blocks; faster for very large files
- **Shortcut method for getting BufferedReader**
  - Files.newBufferedReader(somePath, someCharset)
- **BufferedReader has readLine method**
  - Returns a String. Can chop the String into pieces using StringTokenizer (weak but simple) or String.split (much more powerful but requires knowledge of regular expressions).
    - Details on parsing in lectures on network programming

Files.newBufferedReader Rarely Needed in Java 8

- **When Files.newBufferedReader beneficial**
  - When you don’t want to read full line at a time as String
    - E.g., if reading single characters or arrays of characters
- **Files.lines preferable most other times**
  - Files.lines returns a Stream, and Streams use lazy evaluation
    - Instead of entire file going into memory in advance as with Files.readAllLines, each line is processed as you go along
    - You can quit partway through, so if you find the data you need early on, you never have to read the rest of the file
  - As shown earlier, Streams have many convenient filtering and transformation methods, whereas using BufferedReader requires lower-level and less convenient techniques

Example

```
public class ReadFile2 {
  public static void main(String[] args) throws IOException {
    String file = "InputFile.txt";
    Charset characterSet = Charset.defaultCharset();
    Path path = Paths.get(file);
    try(BufferedReader reader =
          Files.newBufferedReader(path, characterSet)) {
      System.out.printf("Lines from %s:%n", file);
      String line;
      while ((line = reader.readLine()) != null) {
        System.out.println(line);
      }
    } catch (IOException ioe) {
      System.err.printf("IOException: %s%n", ioe);
    }
  }
}
```

This approach using Files.newBufferedReader would be useful in Java 7 if the input file was large, so that all the lines are not stored in memory at once. In Java 8, you would just do Files.lines(path).forEach(System.out::println).

Example Output

- Source of InputFile.txt

```
First line
Second line
Third line
Last line
```

- Output of example code from previous slide

```
Lines from InputFile.txt:
First line
Second line
Third line
Last line
```

Wrap-Up

Summary: Try/Catch Blocks

- **finally blocks**

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

- **multicatch**

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

- **try with resources**

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

Summary: File IO 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, etc.
    - You get benefits of lazy evaluation
    - Can output as List with collect(Collectors.toList())
    - Put Stream processing in separate method, and consider making version of method that uses generic types
- **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 wrap in PrintWriter and use printf
- **Explore and search folders and subfolders**
  - Files.list, Files.walk, Files.find

Summary: File IO in Java 7

- **Use Path to refer to file location**
  - Path somePath = Paths.get("/path/to/file.txt");
- **Read all file lines into a List**
  - List<String> lines =
    Files.readAllLines(somePath, someCharset);
- **Write List or other Iterable into a file**
  - Files.write(somePath, someList, someCharset);
- **Minor utilities shown (not builtin)**
  - List<String> lines = FileUtils.getLines("filename");
  - FileUtils.writeLines("filename", someList);
- **Lower-level but more flexible readers & writers**
  - Files.newBufferedReader(somePath, someCharset)
    - To read, use readLine method
  - Files.newBufferedWriter(somePath, someCharset)
    - To write, use write method or wrap in PrintWriter and use printf

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)