CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Streams – 4

File I/O in Java 8

Paths & Files

- Paths : is a simpler and more flexible replacement for File class
- Get a Path with Paths.get

```
Paths.get("some-file");
```

- Paths have convenient methods
  - toAbsolutePath, startsWith, endsWith, getFileName, getName, getNameCount, subpath, getParent, getRoot, normalize, relativize

Treating Files as Streams of Strings

- 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

Exploring Folders

- **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

```
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);
    }
  }
```

File Reading v1

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

```
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: 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);
}
```

Example 2: 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);
}

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);
}
```

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);
}
```

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);
}
```

Summary

- Streams help make handling large data sets more convenient and efficient
  - E.g. 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

File Reading: v2

- For simple script, do everything in main
- For reusable methods, break processing into two pieces
- 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.

v2 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: Print 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);
    }
  }
}
```

Usage

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

Output

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

```
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);
}
```

Example 2: Print 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);
   }
 }
```

Repetitive Code

```
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 v2

- Stream-processing method:
  - 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
  - Filename passed in, not hardcoded
  - Errors handled explicitly
  - Stream closed automatically
- File-processing method
  - Contains lots of tedious boilerplate code that must be repeated for each application
    - 90% of code on previous slide was repeated

v3: Use Lambdas to Reuse Repeated Code

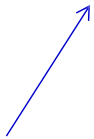
- New interface: StreamProcessor
  - Abstract method takes a Stream<String>
  - Static method takes filename and instance of the interface (usually as a lambda), calls Files.lines, and passes result to the abstract method. Uses try/catch block and try-with-resources.
- Stream-processing method
  - Same 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

v3: 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>.

v3: 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.

Examples

- 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 Length-N 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));
}
```

Pros/Cons of v3

- Stream-processing method
  - 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
  - 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
  - The stream-processing method had to have a void return type

v4: 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.

V4: 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));
}

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);
}
```

Summary

- **Version 1**
  - Put all code inside main; main throws Exception
  - Simple and easy, but not reusable
- **Version 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
- **Version 3**
  - Use lambdas to avoid the repetition
- **Version 4**
  - Use generic types so that values can be returned