Basic File IO with the NIO Library

Topics in This Section

- **More on try/catch blocks**
  - finally blocks
  - multicatch
  - try with resources
- **Path**
- **Simple file reading: all lines at once into List**
- **Simple file writing: all at once from a List**
- **Some simple file reading and writing utilities**
- **Faster and more flexible file reading**
- **Faster and more flexible file writing**

Summary for Java 6 (and Earlier) Programmers

- **Use Path instead of File**
  - Path p = Paths.get("/path/to/file.txt");
- **Can read all file lines into a List in one call**
  - List<String> lines =
    Files.readAllLines(somePath, someCharset);
- **Can write List into a file in one call**
  - Files.write(somePath, someList, someCharset);
- **Use try that automatically closes resources**
  - try(BufferedReader reader = …) { … } catch (…) {…}
- **Shortcuts to get high-performance classes**
  - Files.newBufferedReader(somePath, someCharset)
  - Files.newBufferedWriter(somePath, someCharset)

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: Need

- **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, 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

- **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 = …) {
  doSomeIOWith(reader);
  …
} catch(…) {
  …
}
```

Try with Resources: Need

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 IO operations
- **Get 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

Idea

- **You can read all lines in one method call**
  - List<String> lines =
    Files.readAllLines(somePath, someCharset);
    - Details on List are in next tutorial section, but you can pick up the basics here without much explanation.
- **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 cset1 = Charset.forName("US-ASCII");
  - You still have to catch IOException

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 (along with List) is covered in the next tutorial 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). The system will automatically put square brackets on the outside and separate entries with commas.

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]
```

Simple File Writing

Idea

- **You can write all lines in one method call**
  - List<String> lines = …;
  - Files.write(somePath, lines, someCharset);
- **You can write all bytes in one method call**
  - byte[] fileArray = …;
  - Files.write(somePath, fileArray);
- **OpenOption**
  - Both methods above optionally take an OpenOption as final argument
    - This specifies 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

```
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
```

Some Simple Utilities

Two Static Methods

- **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
```

Faster and More Flexible File Reading

Idea

- **You sometimes need only part of the file**
  - Files.readAllLines reads everything, with no way to stop reading if you find the info you need partway through
- **Need higher performance for very large files**
  - Buffered reading reads in blocks, and is 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).
    - Details on parsing in lectures on network programming

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);
    }
  }
}
```

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
```

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 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);
    }
  }
}
```

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
```

Wrap-Up

Summary: General Topics

- **finally blocks**

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

- **multicatch**

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

- **try with resources**

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

Summary: File IO Topics

- **Use Path instead of File**
  - Path p = Paths.get("/path/to/file.txt");
- **Can read all file lines into a List in one call**
  - List<String> lines =
    Files.readAllLines(somePath, someCharset);
- **Can write List into a file in one call**
  - Files.write(somePath, someList, someCharset);
- **Minor utilities (not builtin)**
  - List<String> lines = FileUtils.getLines("filename");
  - FileUtils.writeLines("filename", someList);
- **Shortcuts to get high-performance classes**
  - 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?