File I/O in Java 7:
A Very Quick Summary

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

coreservlets.com – custom onsite training

For customized training related to Java or JavaScript, please email hall@coreservlets.com
Marty is also available for consulting and development support

The instructor is author of several popular Java EE books, two of the most popular Safari videos on Java and JavaScript, and this tutorial.

Courses available at public venues, or
custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JSF 2.3, PrimeFaces, Java programming (using Java 8, for those new to Java), Java 8 (for Java 7 programmers), JavaScript, jQuery, Angular 2, Ext JS, Spring Framework, Spring MVC, Android, GWT, custom mix of topics.
  - Java 9 training coming soon.
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Spark, Hibernate/JPA, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Differences from Java 8**
- **Simple file reading with Files.readAllLines**
- **Lower-level but more powerful file reading with BufferedReader**

Overview

- **Java 8 developers should skip this tutorial entirely**
  - It is needed only for those stuck in the dark ages with Java 7
- **Tutorial is incomplete**
  - It assumes you have read the coverage of exceptions, paths, and file writing that is given in the first Java 8 File I/O section
- **Not included in the live courses**
  - The live courses use Java 8 throughout; this mini section is included only for those that wanted a quick intro to Java 7 file I/O

Differences from Java 8

- **Read lines into List instead of Stream**

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

  - 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 one method call**

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

- **You can read all bytes into array 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

Advantages of Java 8 Approach

- **Java 8**

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

- **Java 7**

```
List<String> lines = Files.readAllLines(path, charset);
```

- **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

File Reading: Example

```
public class ReadFile1 {
  public static void main(String[] args) throws Exception {
    String file = "input-file.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);
  }
}
```

File Reading: Example Output

- **Source of input-file.txt**

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

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

```
Lines from input-file.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 Exception {
    String file = "input-file.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 input-file.txt:
[First line, Second line, Third line, Last line]
```


Minor Variation of WriteFile1 (WriteFile1 Shown in File Writing Section)

```
public class WriteFile1A {
  public static void main(String[] args) throws IOException {
    List<String> lines =
      Arrays.asList("Line One", "Line Two", "Final Line");
    FileUtils.writeLines(“output-file-1.txt", lines);
  }
}
```

- **Source of output-file-1.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 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 Exception {
    String file = "input-file.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 input-file.txt**

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

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

```
Lines from input-file.txt:
First line
Second line
Third line
Last line
```






Wrap-Up

Summary

- **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

coreservlets.com – custom onsite training







Questions?

More info:
http://courses.coreservlets.com/Course-Materials/java.html – General Java programming tutorial
http://www.coreservlets.com/java-8-tutorial/ – Java 8 tutorial
http://courses.coreservlets.com/java-training.html – Customized Java training courses, at public venues or onsite at *your* organization
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training
Many additional free tutorials at coreservlets.com (tutorials on Java 7, Android, Ajax, Hadoop, and lots more)

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.