Software Systems Development and Integration CSCI 2020U

Files, Input and Output Streams

Mariana Shimabukuro

Acknowledgement: Parts of this course materials are based on material prepared by Randy Fortier.

In this module, we will learn about…

- Streams
  - InputStream
  - OutputStream
- Files
  - File
  - FileInputStream
  - FileReader
  - FileOutputStream
  - FileWriter
- Scanner

Files, Input and Output Streams

Streams

Data Blocks

- Alternative: just load data on demand
  - Too many disk accesses
  - Delays
- Blocks
  - Buffering
  - Block size
- Problem with blocks:
  - What if we don't want an entire block?

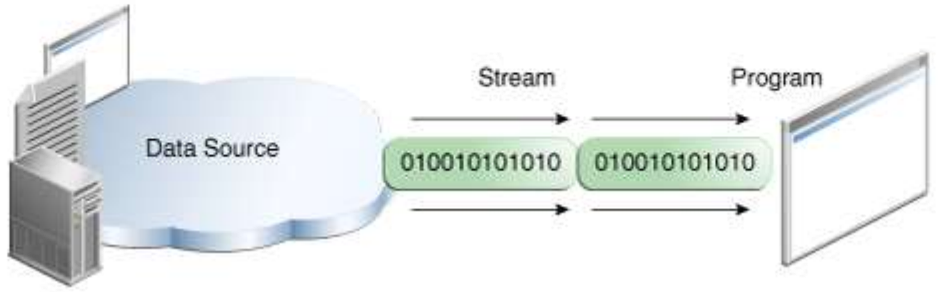
Streams

Streams are an operating system construct. A general term for data flow in Java, used to read from and write to data sources like files, network connections, or memory

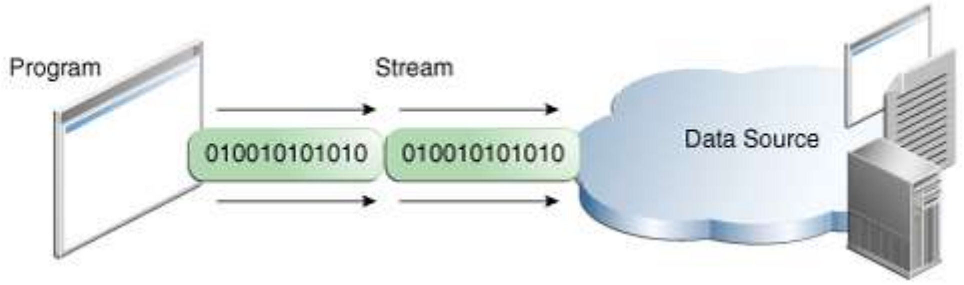
- Input stream
  - To the programmer: endless incoming data source
  - Reality: as the disk data is loaded, it is placed into the input buffer
- Output stream
  - To the programmer: endless outgoing data sink
  - Reality: the output is placed into an output buffer
- The result is much simpler file (and network) code

Input & Output Streams

- A program uses an **input stream** to read data from a source, one item at a time:



- A program uses an **output stream** to write data to a destination, one item at time:



Source: https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Input Streams in Java

- InputStream and FileInputStream:

```
final int BLOCK_SIZE = 1024;
InputStream input = new FileInputStream("myfile.txt");
byte[] buffer = new byte[BLOCK_SIZE];
int numBytesRead = 0;
while ((numBytesRead = input.read(buffer)) != -1) {
    // do something with buffer[0..numBytesRead-1]
}
```

Output Streams in Java

- OutputStream and FileOutputStream:

```
final int BLOCK_SIZE = 1024;
OutputStream output = new FileOutputStream("myotherfile.txt");
byte[] buffer = new byte[BLOCK_SIZE];
boolean keepGoing = true;
while (keepGoing) {
    // fill up buffer with data

    output.write(buffer);

    // update keepGoing if we are done writing data
}
```

Readers in Java

- FileReader: Reads characters (not bytes)
  - FileReader is used for text files, whereas FileInputStream is for binary data.
- BufferedReader:
  - Handles buffering
  - Read line-by-line
- Example:

```
FileReader fileReader = new FileReader("myotherfile.txt");
BufferedReader input = new BufferedReader(fileReader);
String line = null;
while ((line = input.readLine()) != null) {
    // do something with line
}
```

Writers in Java

- FileWriter: Writes characters (not bytes)
  - FileWriter is optimized for writing text, while FileOutputStream is for binary data.
- PrintWriter:
  - Write line-by-line
  - e.g. System.out
- Example:

```
PrintWriter output = new PrintWriter("myotherfile.txt");
boolean keepGoing = true;
String line = null;
while (keepGoing) {
    // update line with new data

    output.println(line);

    // update keepGoing, if no more data to save
}
output.close();
```

Files, Input and Output Streams

Files

Files

- File: (more about this package at https://www.w3schools.com/java/java_files.asp)
  - File::exists()
  - File::isDirectory()
  - File::mkdir(), File::mkdirs()
  - File::renameTo(File)
  - File::setLastModified(long)
  - File::setReadOnly()
  - File::File::toURL()
  - File::File::canRead()
  - File::File::canWrite()
  - File::getAbsolutePath()

File

- Example:

```
File outFile = new File("relativeFile.txt");
File inFile = new File("/path/to/file/absoluteFile.txt");
if (inFile.exists()) {
    BufferedReader input = new BufferedReader(new FileReader(inFile));
    PrintWriter output = new PrintWriter(outFile);
    String line = null;
    while ((line = input.readLine()) != null) {
        output.println(line);
    }
    input.close();
    output.close();
}
```

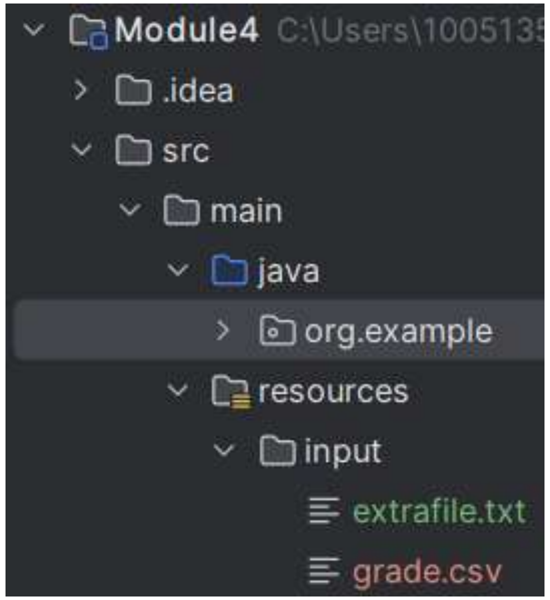
Relative vs. Absolute Path

- Relative paths are **more portable** in projects and suitable for resources within the project directory.
  - Current working directory: `C:/Projects/MyApp`
  - Relative path: `src/main/resources` refers to `C:/Projects/MyApp/src/main/resources`.

- Absolute paths ensure precise file locations but may **require adjustments** when moving the application between systems.
  - Absolute path: `C:/Projects/MyApp/src/main/resources`
  - For example, Unix systems have different models for paths.. Not portable between systems.

Files and Path in IntelliJ

- Our applications are running inside IntelliJ
  - IntelliJ will create a location where it access your "resource" files in runtime
- Relatively in the IDE, the project has its resources in a folder set by the maven project, for example:



```
<build>
     <resources>
          <resource>
               <directory>src/main/resources</directory>
          </resource>
     </resources>
</build>
```

Files and Path in IntelliJ

- Our applications are running inside IntelliJ
  - IntelliJ will create a location where it access your “resource” files in runtime
- Relatively in the IDE, the project has its resources in a folder set by the maven project, for example:

```
URL url = this.getClass().getClassLoader().getResource("/folder");
 System.out.print(url);
 File directory = null;
 try {
    directory = new File(url.toURI());
 } catch (URISyntaxException e) {
     throw new RuntimeException(e);
 }
```

Files and Path in IntelliJ

- Return a URL, this can be used as a path to find the resource itself
  - `URL url = this.getClass().getClassLoader().getResource("/folder/subfolder");`
- If you are loading a file, you can use this URL to find the path, then the file itself.
- File objects can represent both actual files, or directories
  - `directory = new File(url.toURI());`

Files and Path in IntelliJ

- Considering spaces and the path as String

```
ClassLoader classLoader = WordCounter.class.getClassLoader();

// Get the path to the resources folder as String
String resourcePath = classLoader.getResource("").getPath();
// decoder can avoid issues with spaces in path
String decodedPath = URLDecoder.decode(resourcePath, StandardCharsets.UTF_8);
// file / folder in the Resource folder
File inputFile = new File(decodedPath, “file.txt");
```

Files, Input and Output Streams

Scanner

Scanner

- Scanner:
  - Parses data values from any input stream or reader

```
File inFile = new File("/path/to/file/absoluteFile.txt");
Scanner scanner = new Scanner(inFile);
while (scanner.hasNext()) {
    String nextWord = scanner.next();
}
```

Scanner

- Values are separated by delimiters
  - By default, delimiters are whitespace characters
  - You can change them to anything you like

```
File inFile = new File("/path/to/file/absoluteFile.txt");
Scanner scanner = new Scanner(inFile);
scanner.useDelimiter("[^0-9]"); // any non-digit characters
while (scanner.hasNextInt()) {
    int nextInt = scanner.nextInt();
}
```

CSV Files

- Comma-separated values:
  - Values are separated by comma delimiters
  - Spreadsheet programs (e.g. Calc, Excel) can export it
  - Some open/API data is shared in this format
    - Toronto Parking Tickets

```
Name,Asmt1,Asmt2,Labs,Midterm,Final
Bart Simpson,6.0,4.5,6.5,20.25,29.0
Lisa Simpson,10.0,10.0,10.0,29.5,58.25
Ralph Wiggum,0.5,0.25,0.75,8.0,12.5
Homer Simpson,6.5,5.5,5.5,18.5,26.5
```

In this module, we learned about…

- Input and output streams
- Files
- Readers and writers
- Scanner