CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Java I/O – Part 1: Text Files

Department of Computer Science
University of Maryland, College Park

Text Files

- **Text Files**
  1. Data represented in human-readable form.
  2. Example: Java source code files (.java).
  3. Can be edited and manipulated using a text editor (e.g., Notepad, VS Code).
  4. Characters are stored in a specific encoding format (e.g., UTF-8, ASCII).
  5. **Text I/O** provides abstraction by encoding/decoding characters (e.g., FileReader, BufferedReader in Java).

Binary Files

- **Binary Files**
  1. Data is stored as a sequence of bytes (non-human-readable).
  2. Designed to be read and interpreted by programs, not humans.
  3. More compact than text files because no encoding/decoding is required.
  4. More efficient for storing large amounts of data (e.g., images, audio, video, compiled programs).
  5. **Note:** All files are ultimately stored in binary format at the hardware level, regardless of type (text or binary).
- **Key Difference**
  - **Text I/O** involves character encoding and decoding, which adds overhead, while **Binary I/O** directly manipulates raw bytes for faster and more efficient data handling.

TODAY WE WILL TALK ABOUT TEXT FILES

Classes for Text File I/O in Java

- Java provides **Readers and Writers** for handling **text-based** data. These classes work with **characters** instead of raw bytes and handle **encoding/decoding** automatically.
- **Key Classes (from java.io package):**

1. **Reading Text Files:**
    1. FileReader – Reads characters from a file.
    2. BufferedReader – Wraps FileReader for efficient reading (reads larger chunks at once).
2. **Writing Text Files:**
    1. FileWriter – Writes characters to a file.
    2. BufferedWriter – Wraps FileWriter to improve performance.
    3. PrintWriter – Provides convenient methods for formatted text output.

Classes for Text File I/O in Java

- **Why Use Readers/Writers?**
- ✓ Handle character encoding (e.g., UTF-8, ASCII) automatically.
  ✓ More convenient than byte-based streams for text processing.
  ✓ Buffered versions improve efficiency by reducing direct disk access.

The File Class in Java

- The File class (from java.io) **encapsulates file and directory properties** but does **not** handle reading or writing file content. It is mainly used for **file management operations**.
- **Key Features:**
- Represents a file or directory path.
- Checks for file existence and properties.
- Performs file and directory operations (create, delete, rename, etc.).
- **Note:** The File class only represents file metadata—it does **not** provide methods for reading or writing file content. For that, use FileReader, BufferedReader, FileWriter, etc

Example: FileExample.java

Method	Description
exists()	Checks if the file or directory exists.
delete()	Deletes the file or directory.
createNewFile()	Creates a new empty file (if it doesn't already exist).
isFile() / isDirectory()	Checks whether it's a file or directory.
getName()	Returns the file name.
length()	Returns the file size (in bytes).
renameTo(File dest)	Renames the file/directory.

FileReader – Basic Character-Based File Reading

- **FileReader – Basic Character-Based File Reading**
  - The FileReader class (from java.io) is used to **read characters one at a time** from a text file. It provides a simple way to process character streams, but it is not the most efficient method for large files.
- **Key Features:**
  - Reads text files character by character.
    Handles **Unicode characters** automatically.
    Works with BufferedReader for improved efficiency.

Method	Description
read()	Reads a single character and returns its ASCII value (or -1 if EOF is reached).
close()	Closes the file and releases system resources.

- **Limitations:**
  - **Not efficient for large files** – It reads **one character at a time**, leading to frequent disk access.
    No buffering – Use BufferedReader for better performance.

Example: FileReaderEx.java

BufferedReader – Efficient Text File Reading in Java

- **BufferedReader Overview**
  - BufferedReader is a Java class used for reading text from character-input streams.
  - Buffers characters to improve performance, reducing the number of read operations.
  - Efficient for reading large files or multiple lines of text.
- **Key Methods of BufferedReader**
  - **readLine()**
    - Reads an entire line from the file.
    - Returns null when the end of the file is reached.
  - **close()**
    - Closes the stream and releases system resources.
- **Why Use BufferedReader Instead of FileReader?**
  - FileReader reads one character at a time, which can be slow for large files.
  - BufferedReader reads larger chunks of data at once, making it **faster** and **more efficient**. Example: `BufferedReaderEx.java`

Scanner – Token-Based Text File Reading in Java

- **Scanner Overview**
  - Scanner is a Java utility class used for reading and parsing text input.
  - Breaks input into **tokens** (words, numbers, etc.), using whitespace as the default delimiter.
  - Useful for **structured input** where data needs to be processed in chunks.
- **Key Methods of Scanner**
  - **hasNext()** → Checks if more input is available.
  - **next()** → Reads the next token as a String.
  - **nextInt()** → Reads the next token as an int.
  - **nextDouble()** → Reads the next token as a double.
  - **close()** → Closes the scanner to release system resources.
- **Why Use Scanner for File Reading?**
  - Provides built-in parsing for **different data types** (e.g., int, double).
  - Handles **whitespace-based tokenization** automatically.
  - Easier than manually parsing text from BufferedReader.

Example: `ScannerParallelArrays`.java

FileWriter – Writing Text to Files in Java

- **FileWriter Overview**
  - FileWriter is used for writing **characters** to a file.
  - It writes data in **character form**, unlike FileOutputStream, which writes bytes.
  - Designed for **writing text** (not binary data).
  - Writes data to a **file or stream**. If the file does not exist, it is created.
- **Key Methods of FileWriter**
  - **write(int c)**
    - Writes a single character (as an integer).
    - Converts the integer value to its corresponding character and writes it to the file.
    - Returns void.
  - **close()**
    - Closes the file stream and releases system resources.
    - Always call close() to prevent resource leaks.
- **Why Use FileWriter?**
  - Ideal for simple text file writing.
  - Better than OutputStreamWriter when dealing with character data.
  - Allows writing individual characters or strings.

See: `FileWriterEx`

BufferedWriter in Java

- **Purpose:** Writes text efficiently to a character-output stream by buffering characters.
- **Advantages:**
- Reduces the number of I/O operations by writing data in chunks.
- Improves performance compared to writing character by character.
- **Key Methods:**

See: `BufferedWriterEx`

- write(String s): Writes a string to the file.
- newLine(): Writes a platform-dependent newline character.
- flush(): Forces any buffered data to be written immediately.
- close(): Closes the writer and releases system resources.

PrintWriter in Java

- **Purpose:** Used to write formatted text to files or other output streams.
- **Advantages:**
- Provides convenient methods for writing text data.
- Supports **automatic flushing** when used with System.out.
- Allows **formatted output** similar to System.out.printf().
- **Key Methods:**
- print(String s): Writes text without a newline.
- println(String s): Writes text followed by a newline.
- printf(String format, Object... args): Writes formatted text.

See: `PrintWriterEx`