



# Files / IO in Java

*Advanced Programming*



*Dr. Mojtaba Vahidi Asl*  
*Sara Shiri , Spring 1404*

# Introduction to Java I/O:

- I/O stands for Input/Output in Java.
- It involves:
  - Input: Reading data into the program.
  - Output: Writing data from the program to external sources.
- Java supports I/O operations through classes in the `java.io` package.
- Common sources/destinations for I/O include files, network connections, and the console.
- These operations are designed to be efficient and flexible for handling data.

# Types of I/O in Java:

## 1. Stream-Based I/O:

- InputStream: Reads bytes from a source.
- OutputStream: Writes bytes to a destination.
- Reader: Reads characters from a source.
- Writer: Writes characters to a destination.

## 2. Byte Streams vs Character Streams:

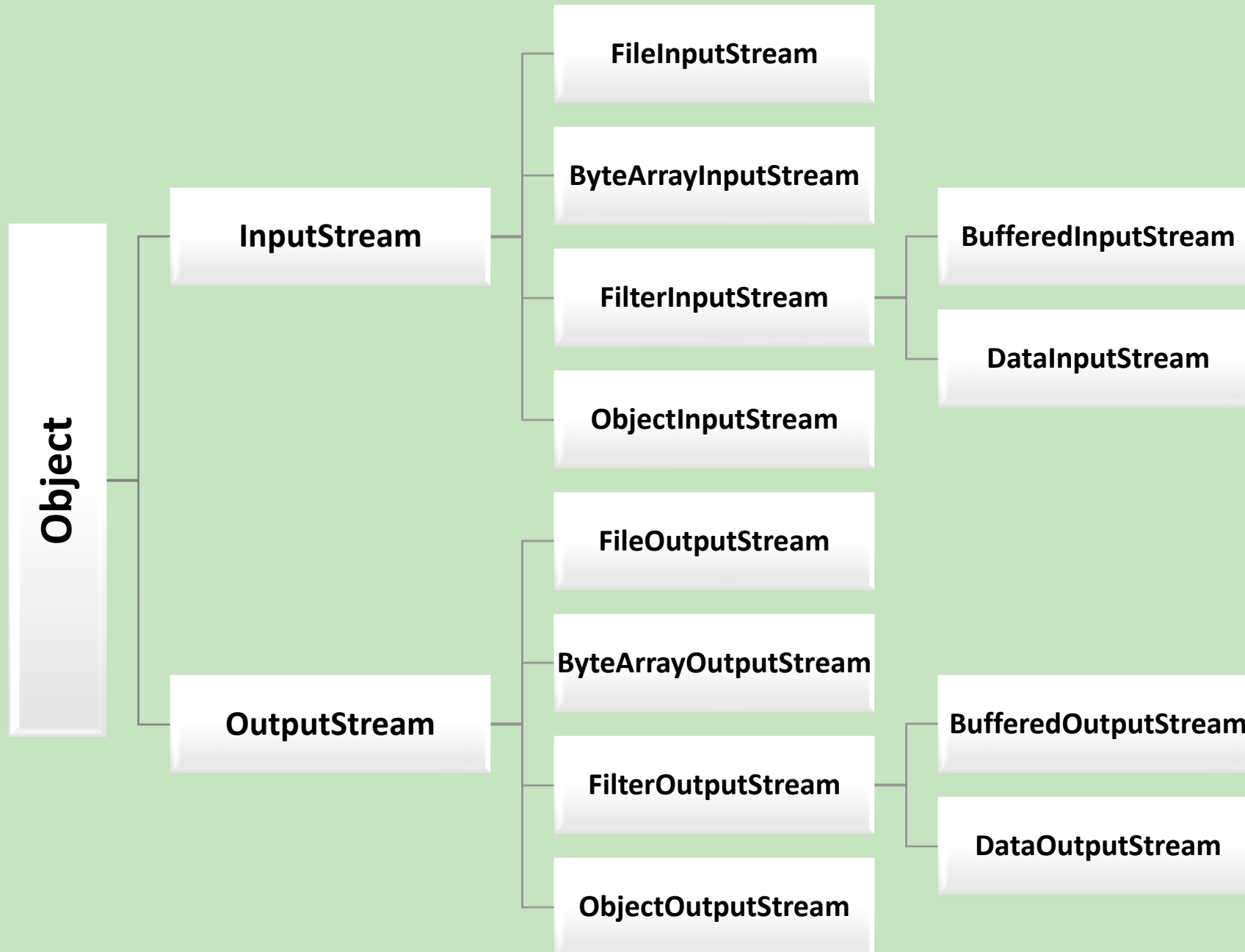
- Byte Streams: Handle raw binary data (e.g., images, files with binary data).
- Character Streams: Handle text data (e.g., files with text content).

# Common I/O Classes in Java:

1. **InputStream/OutputStream:** Base classes for byte streams.
2. **FileInputStream/FileOutputStream:** Read and write bytes to a file.
3. **Reader/Writer:** Base classes for character streams.
4. **FileReader/FileWriter:** Read and write characters to a file.
5. **BufferedReader/BufferedWriter:** More efficient way to read/write characters with a buffer.

# Key Concepts in Java I/O:


- **Stream:** A sequence of data (either input or output) that flows through the program.
- **Input Streams:** Read data from a source (e.g., file, network).
- **Output Streams:** Write data to a destination (e.g., file, network).
- **Blocking I/O:** The program waits (blocks) until the data is available (input) or fully written (output).



# File Handling in Java

- Java provides the File class in the java.io package to work with files and directories.
- Important File Class Operations:
  1. **Creating a File**
  2. **Checking if a File Exists**
  3. **Reading and Writing to a File**
  4. **Renaming and Deleting Files**
  5. **Working with Directories**

# File Handling in Java



```
File file = new File("example.txt");  
file.createNewFile();
```



# File Handling in Java




```
if (file.exists()) {  
    System.out.println("This file exists.");  
}
```

# File Handling in Java




```
FileWriter writer = new FileWriter("example.txt");  
writer.write("Hello, World!");  
writer.close();
```

# File Handling in Java




```
file.renameTo(new File("newName.txt"));  
file.delete();
```

# File Handling in Java



```
File dir = new File("exampleDir");  
dir.mkdir(); // Creates a new directory  
String[] fileList = dir.list();
```

# Buffered I/O



```
BufferedReader reader = new BufferedReader(new FileReader("example.txt"));  
String line = reader.readLine();  
reader.close();
```

- Wrap around FileReader or FileWriter for efficient reading and writing.
- Reads/writes larger chunks of data at a time to improve performance.



```
public static void main(String[] args) throws IOException {  
    File file = new File("example.txt");  
  
    if (!file.exists()) {  
        file.createNewFile();  
        System.out.println("File created: " + file.getName());  
    }  
  
    System.out.println("File path: " + file.getAbsolutePath());  
    System.out.println("Is it a file? " + file.isFile());  
  
    File renamedFile = new File("renamed_example.txt");  
    if (file.renameTo(renamedFile)) {  
        System.out.println("File renamed to: " + renamedFile.getName());  
    }  
  
    if (renamedFile.delete()) {  
        System.out.println("File deleted: " + renamedFile.getName());  
    }  
}
```

# File I/O Exceptions:


## Common Exceptions:

- **IOException:** General I/O failure (e.g., file not found, read/write failure).
  - **FileNotFoundException:** File not found or cannot be opened.
- 
- Always handle these exceptions **using try-catch** or **try-with-resources**.

# File I/O Exceptions:

## 1. Try-With-Resources (Java 7+):

- Automatically closes the resource (file) when you're done to prevent resource leaks.



```
try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {  
    String line = reader.readLine();  
    System.out.println(line);  
} catch (IOException e) {  
    e.printStackTrace();  
}
```



# File I/O Exceptions:

## 2. Using `java.nio.file` (Java 7+):

### I. Path Class:

- Represents file and directory paths.
- More modern and flexible than File class.

### II. Files Class:

- Utility class to perform common file operations easily.

### III. Walking Through Directories:

- `Files.walk()` allows recursively traversing directories.

## Usage of the Path and File Classes:



```
Path path = Paths.get("example.txt");
```



```
Files.write(Paths.get("example.txt"), "Hello, World!".getBytes());  
String content = Files.readString(Paths.get("example.txt"));
```

# Essential File Handling Tips in Java:

## 1. Always Close Resources:

- Either manually or using try-with-resources to avoid resource leaks.

## 2. Use Buffered I/O:

- For efficient reading and writing, especially for large files.

## 3. Handle Exceptions:

- Properly handle IOException and related exceptions to avoid crashes.

## 4. Use `java.nio.file` for New Code:

- Prefer the Path and Files classes in new code for more features and better flexibility.



```
File file = new File("example.txt");

if (!file.exists()) {
    file.createNewFile();
    System.out.println("File created: " + file.getName());
}

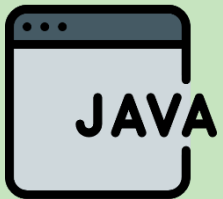
System.out.println("File path: " + file.getAbsolutePath());

if (file.delete()) {
    System.out.println("File deleted: " + file.getName());
}
```

# Resources:

- <https://www.javatpoint.com/java-io>
- [https://www.w3schools.com/java/java\\_files.asp](https://www.w3schools.com/java/java_files.asp)
- <https://www.geeksforgeeks.org/file-class-in-java/>
- [https://www.tutorialspoint.com/java/java\\_files\\_io.htm](https://www.tutorialspoint.com/java/java_files_io.htm)

**Any questions?**



**Thanks for your Attention.**