File IO

Josh Wilcox (jw14g24@soton.ac.uk)

March 4, 2025

Contents

1 Persistent Data

- Most data stored in Java objects and data structures at runtime is **temporary**
- Such data will be lost after the program has ran
- We can solve this by making the data *persistent* :
  - Writing to a local file
  - Writing it across the network
  - Connecting to a database

1.1 File Structure in Java

- Every file is considered as a sequential stream of bytes in java
  - Terminated with a special EOF marker byte
- This is different to the *complex* structure that Java objects tend to have
- So to store an object in a file, it will need to be flattend somehow into a string of bits
  - This is known as **serialisation**
  - Converting a byte stream to an object is called **deserialisation**

2 The file class

- Abstract representation of a file
- The **only** class related to the actual file itself within the entire IO package
- Creating a file object does **not** create or open a file in the underlying OS
  - It just provides a means of addressing a file in Java
  - This is **platform independent**
  - The File objects are used as pointers to input and output streams

2.1 Constructors

- The primary constructor for the File class:
  - `File(String pathname)`
- The String `pathname` is converted to an OS pathname
  - Associates this File object with a file at that path

2.2 Creating files

```
1 import java.io.File;
2 import java.io.IOException;
3
4 public class FileDemo1 {
5     public static void main(String[] args){
6         String home = System.getProperty("user.home") // Gets the home directory
          ↪  (might be D drive or usr)
7         String path = home + File.separator + "filerepository" + File.separator +
          ↪  "test.txt";
8         File f = new File(path);
```

```
        try {
            f.createNewFile();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

3 Input and Output Streams

3.1 Steps for Stream Operation

a) Get a file ready with File class

b) Set up the IO Stream

c) Read or Write

d) Close the IO Stream

3.2 Byte-based Stream

3.2.1 OutputStream

- Outputstream has the following hierarchy:
  - ByteArrayOutputStream
  - PipedOutputStream
  - FileOutputStream
  - ObjectOutputStream
  - FilterOutputStream
    - PrintStream
    - BufferedOutputStream
    - DataOutputStream

```
import java.io.File;
import java.io.IOException;

public class FileDemo1 {
    public static void main(String[] args){

        // Step 1: Get file Ready
        String home = System.getProperty("user.home") // Gets the home directory
        ↪  (might be D drive or usr)
        String path = home + File.separator + "filerepository" + File.separator +
        ↪  "test.txt";
        File f = new File(path);

        // Step 2: Set up Output Stream
        OutputStream out = new FileOutputStream(f);

```

```
        // Step 3: Write to file
        String str = "Hello World!";
        byte b[] = str.getBytes(); // Need to turn into bytes
        out.write(b);

        // Step 4: Close Output Stream
        out.close
    }
}
```

3.2.2 InputStream

- Very similar hierarchy to the OutputStream but with slightly different method names

```
import java.io.File;
import java.io.IOException;

public class FileDemo1 {
    public static void main(String[] args){

        // Step 1: Get file Ready
        String home = System.getProperty("user.home") // Gets the home directory
        ↪  (might be D drive or usr)
        String path = home + File.separator + "filerepository" + File.separator +
        ↪  "test.txt";
        File f = new File(path);

        // Step 2: Set up Input Stream
        OutputStream input = new FileInputStream(f);

        // Step 3: Read from file
        String str = "Hello World!";
        byte b[] =  new byte [(int)f.length()]; // Set size of array to the length
        ↪  of the file
        input.read(b);

        // Step 4: Close Input Stream
        input.close
        System.out.println(new String(b));
    }
}
```

3.3 Character Based Strings

- Used for reading and writing **character-based** data
- Entierely separate hierarchy of IO classes for this purpose
- Referred to as Reader and Writer

3.3.1 Writer Stream

```java
import java.io.File;
import java.io.IOException;

public class FileDemo1 {
    public static void main(String[] args){

        // Step 1: Get file Ready
        String home = System.getProperty("user.home") // Gets the home directory
        String path = home + File.separator + "filerepository" + File.separator +
        ↪  "test.txt";
        File f = new File(path);

        // Step 2: Set up Writer
        Writer out = new FileWriter(f);

        // Step 3: Write to file
        String str = "Hello World!";
        out.write(str);

        // Step 4: Close Writer Stream
        out.close;
    }
}
```

3.3.2 Writer Stream

```java
import java.io.File;
import java.io.IOException;

public class FileDemo1 {
    public static void main(String[] args){

        // Step 1: Get file Ready
        String home = System.getProperty("user.home") // Gets the home directory
        String path = home + File.separator + "filerepository" + File.separator +
        ↪  "test.txt";
        File f = new File(path);

        // Step 2: Set up Reader
        Reader reader = new FileReader(f);

        // Step 3: Read from file
        int temp = 0;
        while ((temp = reader.read()) != -1) { // If not EOF char
            System.out.println((char) temp); // Write the *character*
        }
```

```
20          out.write(str);
21
22          // Step 4: Close Writer Stream
23          out.close;
24      }
25  }
```

4 More Streams

4.1 ByteArray

- **ByteArrayOutputStream** and **ByteArrayInputStream** are used to read and write data to and from byte arrays.
- These streams are useful when you need to manipulate binary data in memory.
- They do not interact with the file system or network, making them faster for in-memory operations.

```
1   import java.io.ByteArrayOutputStream;
2   import java.io.ByteArrayInputStream;
3   import java.io.IOException;
4
5   public class ByteArrayDemo {
6       public static void main(String[] args) throws IOException {
7           // Writing to ByteArrayOutputStream
8           ByteArrayOutputStream baos = new ByteArrayOutputStream();
9           String str = "Hello ByteArray!";
10          baos.write(str.getBytes());
11
12          // Reading from ByteArrayInputStream
13          ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
14          int data;
15          while ((data = bais.read()) != -1) {
16              System.out.print((char) data);
17          }
18      }
19  }
```

4.2 Pipeline

- **PipedOutputStream** and **PipedInputStream** are used for inter-thread communication.
- They allow data to be written to a piped output stream and read from a piped input stream in a different thread.
- Useful for creating producer-consumer scenarios within a single application.

```
1   import java.io.PipedOutputStream;
2   import java.io.PipedInputStream;
3   import java.io.IOException;
4
5   public class PipedStreamDemo {
```

```
    public static void main(String[] args) throws IOException {
        PipedOutputStream pos = new PipedOutputStream();
        PipedInputStream pis = new PipedInputStream(pos);

        Thread writerThread = new Thread(() -> {
            try {
                pos.write("Hello Pipeline!".getBytes());
                pos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        });

        Thread readerThread = new Thread(() -> {
            try {
                int data;
                while ((data = pis.read()) != -1) {
                    System.out.print((char) data);
                }
                pis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        });

        writerThread.start();
        readerThread.start();
    }
}
```

4.3 Print

- **PrintStream** is used to write formatted data to an output stream.
- It provides methods to print various data types conveniently.
- Commonly used for writing to the console or log files.

```
import java.io.PrintStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class PrintStreamDemo {
    public static void main(String[] args) throws IOException {
        PrintStream ps = new PrintStream(new FileOutputStream("printstream.txt"));
        ps.println("Hello PrintStream!");
        ps.printf("Formatted number: %.2f", 123.456);
        ps.close();
    }
}
```

4.4 Filter

- **FilterOutputStream** and **FilterInputStream** are abstract classes that provide a way to wrap other streams.
- They are used to add additional functionality to existing streams, such as buffering or data conversion.
- Subclasses like **BufferedOutputStream** and **DataOutputStream** provide specific implementations.

```java
import java.io.FilterOutputStream;
import java.io.FilterInputStream;
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class FilterStreamDemo {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("filterstream.txt");
        FilterOutputStream fosFilter = new FilterOutputStream(fos);
        fosFilter.write("Hello FilterStream!".getBytes());
        fosFilter.close();

        FileInputStream fis = new FileInputStream("filterstream.txt");
        FilterInputStream fisFilter = new FilterInputStream(fis) {};
        int data;
        while ((data = fisFilter.read()) != -1) {
            System.out.print((char) data);
        }
        fisFilter.close();
    }
}
```