Class-16 : File I/O & Serialization

by Pial Kanti Samadder

Introducing the Java I/O API

The I/O stands for **Input / Output**. The Java I/O API gives all the tools your application needs to access information from the outside.

For your application, *"outside"* means two elements:

- Disks, or more generally, file system
- Network.

With the **Java I/O API**, you **can read and write files**, as well as **get and send data over a network** using different protocols.

The APIs that give you access to a database (the Java Database Connectivity API) use the Java I/O API to access databases through TCP/IP. There are many well-known APIs that are built on top of the Java I/O API.

Understanding File I/O in Java

- Reading from and writing to files for storing or retrieving data.
- **APIs Available:**
  - `java.io` – Traditional I/O for byte and character streams
  - `java.nio` – New I/O (buffer-oriented, supports non-blocking operations)
- **Key Concepts:**
  - Streams (`InputStream`, `OutputStream`, `Reader`, `Writer`)
  - Buffers and Channels (`ByteBuffer`, `FileChannel`)

File Handling Classes in Java

Java provides multiple classes for file handling, including:

- **File** – Represents a file or directory.
- **FileReader & FileWriter** – Used for character-based file operations.
- **BufferedReader & BufferedWriter** – Improve performance by buffering data.
- **FileInputStream & FileOutputStream** – Handle binary file operations.
- **RandomAccessFile** – Allows reading and writing at any file position.
- **Files** (from `java.nio.file`) – Provides modern, efficient file-handling utilities.

Java I/O Streams

Java provides two different types of streams for input and output operations:

- **Character streams**
- **Byte streams**

Character Streams

- Designed for **text data** (characters, strings, and other text-based data types).
- Automatically handle **character encoding conversion** between platform-native encoding and **Unicode**.
- Provide **platform-independent** way to read/write text.

Key Abstract Classes

- **Reader** → Superclass for all character input streams.
  - Common methods: `read()`, `read(char[] cbuf)`, `close()`.
- **Writer** → Superclass for all character output streams.
  - Common methods: `write()`, `flush()`, `close()`.

Character Streams Example

Reading file with `FileReader`

```
import java.io.FileReader;
import java.io.IOException;

public class FileReaderExample {
    public static void main(String[] args) {
        try (FileReader reader = new
FileReader("src/main/resources/input.txt")) {
            int ch;
            while ((ch = reader.read()) != -1) {
                System.out.print((char) ch);  // convert int to char
            }
            System.out.println();
        } catch (IOException e) {
            System.out.println("Error while reading file.
Message: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Writing with `FileWriter`

```
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterExample {
    public static void main(String[] args) {
        String filePath = "src/main/resources/output.txt";  //
writes file into resources folder

        try (FileWriter writer = new FileWriter(filePath)) {
            writer.write("Hello, World!\n");
            writer.write("This is written using FileWriter.\n");
            writer.write("FileWriter handles characters and stores
them in text files.");
        } catch (IOException e) {
            System.out.println("Error while writing file. Message: "
+ e.getMessage());
            e.printStackTrace();
        }

        System.out.println("Writing complete! Check output.txt
in resources folder.");
    }
}
```

Byte Streams

- Designed for **binary data** (images, audio, video, executables).
- Work with **raw 8-bit bytes**, not characters.
- Handle data in a **platform-dependent** way (no encoding conversion).

Key Abstract Classes

- **InputStream** → Superclass of all byte input streams.
  - Common methods: `read()`, `read(byte[] b)`, `close()`.
- **OutputStream** → Superclass of all byte output streams.
  - Common methods: `write(int b)`, `write(byte[] b)`, `flush()`, `close()`.

Byte Streams Example

Reading with `InputStream` **and creating copy using** `OutputStream`

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

public class ImageCopyExample {
    public static void main(String[] args) {
        String sourcePath = "src/main/resources/input.jpg";   // original image
        String destinationPath = "src/main/resources/output.jpg"; // copied image

        try (FileInputStream fis = new FileInputStream(sourcePath);
             FileOutputStream fos = new FileOutputStream(destinationPath)) {

            byte[] buffer = new byte[1024];  // read 1KB chunks
            int bytesRead;

            while ((bytesRead = fis.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead);  // write exactly what was read
            }

            System.out.println("Image copied successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Create a File in Java

When working with data in Java, we often need to **store, retrieve, or modify information on a physical storage device** such as a hard disk. For example, you may fetch data from a remote server and want to **save it for later use**. To achieve this, Java provides built-in classes and methods for **file creation and manipulation**.

The **File API** in Java allows developers to:

- Create new files and directories
- Read and update existing data
- Append additional data
- Delete files and folders

Key Classes for File Creation

- `File` **(java.io)** → Provides methods like `createNewFile()` to create a new empty file.
- `Files` **(java.nio)** → Provides methods like `createFile()` with more modern, flexible APIs.
- `FileOutputStream` **(java.io)** → Creates files while writing byte data, useful for handling binary/text data

Java NIO (New I/O)

- An alternative I/O API to traditional `java.io` and `java.net`.
- Introduced in Java 1.4 to provide **faster, more flexible I/O operations**.
- Originally meant *New I/O* (not Non-blocking I/O).

Key Features

- **Non-blocking I/O**
  - A thread can request data from a **channel** → while data is being read into a **buffer**, the thread can do other work.
  - Once data is available, the thread processes it.
  - Same applies to writing data.
- **Channels and Buffers**
  - Traditional I/O uses **streams** (byte/character streams).
- NIO uses:
  - **Channel** → connection to the data source (file, socket, etc.)
  - **Buffer** → memory container that temporarily stores the data
- Data always flows:
  - **Channel → Buffer → Program (for reading)**
  - **Program → Buffer → Channel (for writing)**

Using `Files` Class

Files.createFile() Method

- Belongs to `java.nio.file.Files` class
- Creates a **new empty file** if it doesn't already exist
- Throws **FileAlreadyExistsException** if the file exists

Syntax

```
public static Path createFile(Path path, FileAttribute<?>... attrs) throws IOException
```

Returns

- Path of the created file

Throws

- `FileAlreadyExistsException` – file already exists
- `IOException` – I/O error or parent dir missing
- `SecurityException` – no write permission

Example of creating a file using `Files.createNewFile()`

```
Path path = Paths.get("src/main/resources/newFile.txt");
try {
    Path createdFilePath = Files.createFile(path);
    System.out.println("File Created. Path: " + createdFilePath);
} catch (FileAlreadyExistsException e) {
    System.out.println("File already exists.");
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

The Java `File` Class

`File` class is the part of the *java.io* API. **The *File* class gives us the ability to work with files and directories on the file system**.

File and Directory

A file is a named location that can be used to store related information. For example, **main.java** is a Java file that contains information about the Java program.

A directory is a collection of files and subdirectories. A directory inside a directory is known as subdirectory.

Creating a *File* Object

The *File* class has 4 public constructors.

- `File(String pathname)` – Creates an instance representing the given *pathname*
- `File(String parent, String child)` – Creates an instance that represents the path formed by joining the *parent* and the *child* paths
- `File(File parent, String child)` – Creates an instance with the path formed by joining the *parent* path represented by another *File* instance and the *child* path
- *File(URI uri)* – Creates an instance that represents the given Uniform Resource Identifier

```
// creates an object of File using the path
File file = new File(String pathName);
```

Example of creating a file using `File` class

- `java.io` package is **stream oriented**.
- `File.createNewFile()` function also throws an `IOException`.

```
String path = "src/main/resources/newFile.txt";
File file = new File(path);
try {
    boolean isFileCreationSuccessful = file.createNewFile();

    if (!isFileCreationSuccessful) {
        System.out.println("File already exists.");
    } else {
        System.out.println("File Created. Path: " + path);
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

`FileOutputStream` Class

- Part of `java.io` package
- Used for **writing raw bytes** to a file or `FileDescriptor`
- If the file **doesn't exist**, it will be **created automatically**
- Throws `FileNotFoundException` if permissions or security restrictions prevent access

Important:

- Designed for **bytes** (images, binary data)
- For **text**, `FileWriter` is recommended

Constructors

Constructor	Description
`FileOutputStream(File file)`	Write to a File object
`FileOutputStream(File file, boolean append)`	Append data if `true`
`FileOutputStream(FileDescriptor fdObj)`	Write to a FileDescriptor
`FileOutputStream(String fileName)`	Write to a file by name
`FileOutputStream(String fileName, boolean append)`	Append mode by file name

Example

```
String fileNameWithPath = "src/main/resources/newFile.txt";
String content = "Hello, FileOutputStream!\n";

try (FileOutputStream fos = new FileOutputStream(fileNameWithPath, true)) {
    byte[] data = content.getBytes();
    fos.write(data);
    System.out.println("File saved: " + fileNameWithPath);
} catch (FileNotFoundException e) {
    System.out.println("File not found or permission denied: " + e);
} catch (IOException e) {
    System.out.println("I/O Error: " + e);
}
```

Reading a File in Java

- Access data stored on disk: text files, logs, config files, multimedia metadata
- Essential for **data processing, parsing, and application functionality**

Different Ways to Read a File

Method	Description
Scanner	Simple, line-by-line or token-based reading
BufferedReader	Efficient reading of large files line-by-line
FileReader	Reads characters from a file (basic character stream)
Files (java.nio)	Modern, flexible API for reading all lines or bytes

Key Notes:

- **BufferedReader** is faster than **FileReader** for large files
- **Scanner** is useful for parsing tokens or numbers
- **Files API** offers **readAllLines()**, **readAllBytes()** for concise code
- Choosing the method depends on **file size, content, and processing needs**

Writing a File in Java

Writing data to a file allows Java programs to **store information** for later use, logging, or external communication.

APIs Available:

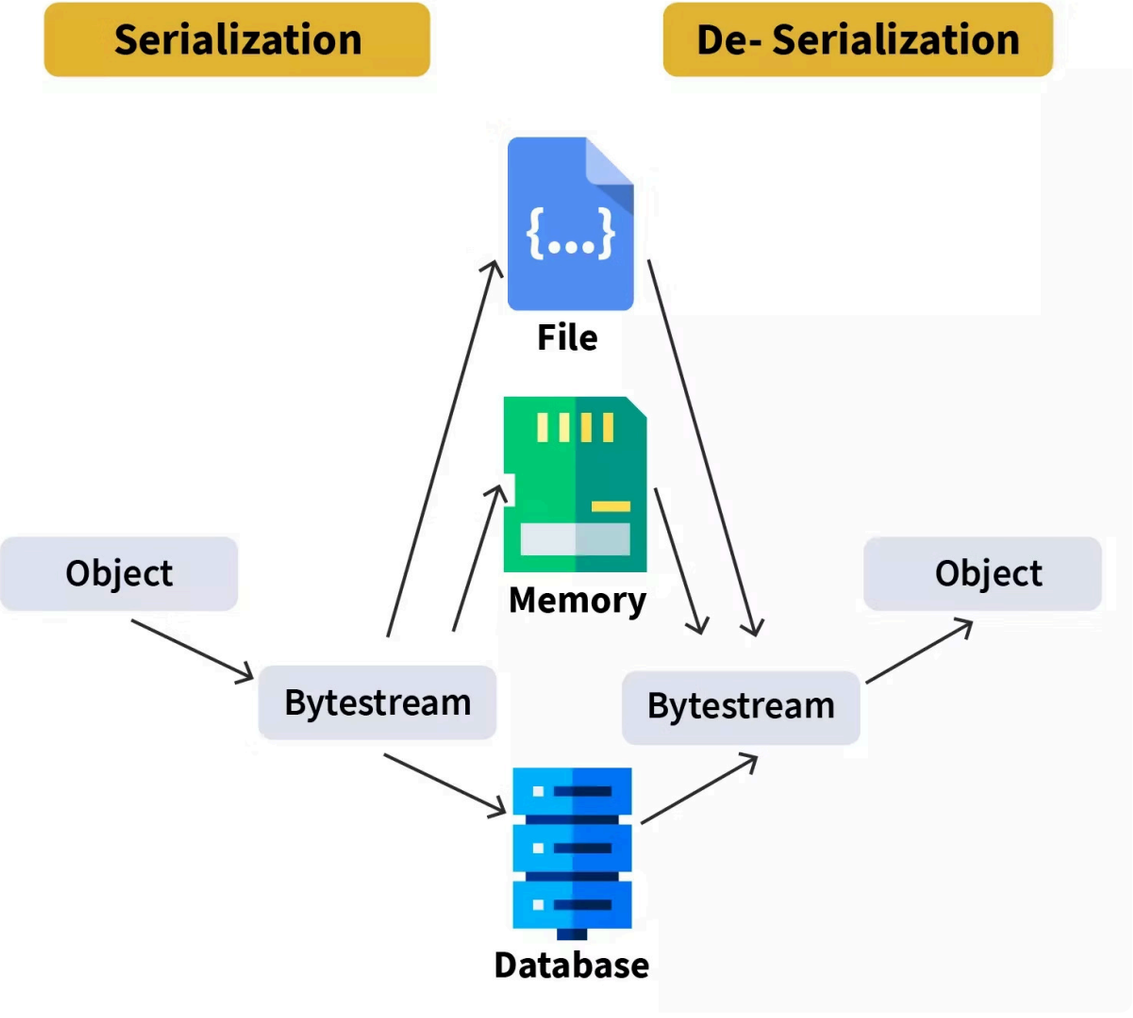
1. `java.io.FileOutputStream` – Writes raw bytes to a file.
2. `java.io.FileWriter` – Writes characters/text to a file.
3. `java.nio.file.Files` – Modern API for writing bytes or lines efficiently.

Common Approaches:

Approach	Key Points	Example
FileOutputStream	Byte-oriented, supports append mode	`new FileOutputStream("file.txt", true)`
FileWriter	Character-oriented, convenient for text	`new FileWriter("file.txt")`
Files.write()	NIO method, supports Paths and options	`Files.write(Path.of("file.txt"), data)`

Serialization and Deserialization in Java

Serialization and **deserialization** are powerful mechanisms in Java that allow objects to be converted into a format suitable for storage or transmission and then reconstructed back into their original form. Understanding these processes is essential for many applications, such as saving state, enabling inter-process communication, or even caching data



What is Serialization?

Serialization is the process of converting an **object's state** (its instance variables) into a **sequence of bytes**.

- These bytes can then be:
  - **Stored in a file** for later use
  - **Sent over a network** to another JVM
  - **Used in caching or messaging systems**
- The serialized data **represents the object's state**, not the actual running object.
- It may include some **class metadata** if needed to reconstruct the object during deserialization.
- Only objects of classes that implement `Serializable` can be serialized.

Why Use Serialization?

1. **Data Persistence:** You can store an object's state in a file or database and retrieve it later, preserving the object across application restarts.
2. **Network Communication:** Serialization allows objects to be sent over a network between different components of a distributed system.
3. **Caching:** Serialized objects can be stored in cache systems like Redis, enabling faster retrieval without reconstructing the object from scratch.

How Does Serialization Work in Java?

In Java, serialization is facilitated by the `Serializable` interface. This interface is a marker interface, meaning it doesn't contain any methods but signals to the JVM that the class is eligible for serialization. Here's a simple example:

```
import java.io.Serializable;

public class Student implements Serializable {
    private String name;
    private int age;
    private String grade;

    public Student(String name, int age, String grade) {
        this.name = name;
        this.age = age;
        this.grade = grade;
    }

    @Override
    public String toString() {
        return "Student{name='" + name + "', age=" + age + ", grade='" + grade + "'}";
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        Student student = new Student("John Doe", 20, "A");
        String fileName = "src/main/resources/student.ser";

        // ----- Serialization -----
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fileName))) {
            oos.writeObject(student);
            System.out.println("Student object serialized and saved to " + fileName);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // ----- Deserialization -----
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fileName))) {
            Student deserializedStudent = (Student) ois.readObject();
            System.out.println("Student object deserialized from file:");
            System.out.println(deserializedStudent);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

What is Deserialization?

Deserialization is the process of **converting a sequence of bytes back into a fully-formed Java object**.

These bytes typically come from:

- A **file** where the object was previously serialized
- A **network stream** from another JVM
- A **database or cache** storing serialized data

The deserialized object:

- **Restores the original state** of the object's instance variables
- Allows the application to **use it as if it had been created locally**
- Only reconstructs **non-transient fields**; transient fields remain uninitialized

Requirements:

- The class definition must be available in the JVM at the time of deserialization
- Only objects of classes that implement `Serializable` can be deserialized

Why Use Deserialization?

1. **Retrieve Stored Data:** Reconstructing objects saved to disk or a database for later use.
2. **Inter-Process Communication:** Rebuilding objects sent over a network, enabling different services or processes to communicate.
3. **Loading Cached Data:** Quickly loading precomputed or preloaded data structures from a cache.

Best Practices and Considerations

- **Use** `serialVersionUID`
  - Define a `serialVersionUID` for each serializable class.
  - Ensures **compatibility during deserialization**, even if the class structure changes.

```
private static final long serialVersionUID = 1L;
```

- **Avoid Serialization of Sensitive Data**
  - Mark sensitive fields as `transient` to prevent them from being serialized.

```
private transient String password;
```

- **Validate Input During Deserialization**
  - Never deserialize untrusted or external data blindly.
  - Unvalidated deserialization can lead to **security vulnerabilities** and attacks.
- **Performance Considerations**
  - Serialization can be **slow and memory-intensive**.
  - For performance-critical applications, consider **alternatives** like:
    - JSON (Jackson, Gson)
    - XML
    - Protocol Buffers or other binary formats