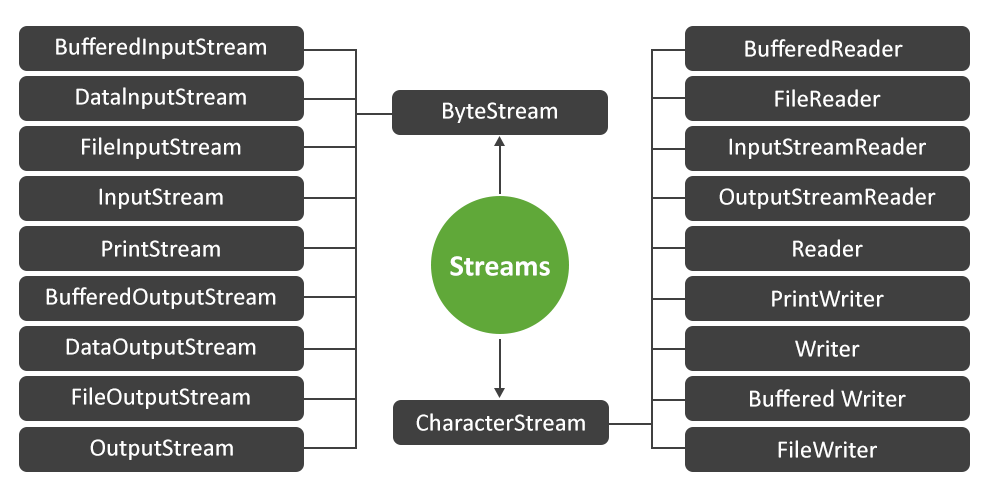
# ****File Handling and Serialization****

## ****1: Introduction to File Handling in Java****

### ****File Handling in Java****

File handling refers to the process of reading and writing data to files. Java provides a set of classes in the java.io package to perform file operations like reading, writing, deleting, and creating files.



Here’s a list of the key abstract classes in the java.io package, along with their respective concrete subclasses and brief explanations:

**1. InputStream**

The InputStream **abstract class** is the superclass for all byte input streams. It represents an input stream of bytes and provides the basic methods for reading byte data.

**Subclasses of InputStream:**

* **FileInputStream**: Reads byte data from a file. It’s the most commonly used input stream for file handling.
* **ByteArrayInputStream**: Reads bytes from a byte array. It’s useful for reading data from a byte array as a stream.
* **BufferedInputStream**: Adds buffering to an input stream for more efficient reading, especially for larger files.
* **ObjectInputStream**: Reads objects from a stream, enabling object deserialization.
* **DataInputStream**: Allows reading of primitive data types (such as int, float, etc.) from an input stream.

**2. OutputStream**

The OutputStream **abstract class** is the superclass for all byte output streams. It represents an output stream of bytes and provides basic methods for writing byte data.

**Subclasses of OutputStream:**

* **FileOutputStream**: Writes byte data to a file, typically used to write binary data to files.
* **ByteArrayOutputStream**: Writes byte data to a byte array in memory, useful for temporary data manipulation.
* **BufferedOutputStream**: Provides buffered output for efficient writing of byte data.
* **ObjectOutputStream**: Writes objects to an output stream for object serialization.
* **DataOutputStream**: Writes primitive data types (like int, float, double) to an output stream in a machine-independent way.

**3. Reader**

The Reader **abstract class** is the superclass for all character input streams. It represents an input stream of characters and provides basic methods for reading character data.

**Subclasses of Reader:**

* **FileReader**: Reads character data from a file, making it suitable for reading text files.
* **BufferedReader**: Provides efficient reading of characters, with buffering for faster reading, particularly when dealing with large text files.
* **CharArrayReader**: Reads characters from a character array, useful when working with small chunks of data.
* **InputStreamReader**: Converts byte input streams to character input streams using a specified character encoding.
* **StringReader**: Reads characters from a string, useful for treating a string as an input stream.

**4. Writer**

The Writer **abstract class** is the superclass for all character output streams. It represents an output stream of characters and provides basic methods for writing character data.

**Subclasses of Writer:**

* **FileWriter**: Writes character data to a file, ideal for writing text files.
* **BufferedWriter**: Writes characters to an output stream with buffering, improving performance for writing large text files.
* **CharArrayWriter**: Writes characters to a character array, useful for temporary storage or manipulation of text.
* **PrintWriter**: A convenience class for writing formatted text (with support for auto-flushing) to a character output stream.
* **StringWriter**: Writes characters to a StringBuffer or StringBuilder object, commonly used for string manipulation.

## ****2: File Handling Basics****

### ****The File Class****

**Purpose**: The File class provides methods to create, delete, rename, and retrieve metadata about files and directories.

### ****Sample Code for File Class****

import java.io.File;

public class FileExample {

public static void main(String[] args) {

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

// Create a new file

try {

if (file.createNewFile()) {

System.out.println("File created: " + file.getName());

} else {

System.out.println("File already exists.");

}

// Get file metadata

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

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

System.out.println("Is file: " + file.isFile());

System.out.println("Is directory: " + file.isDirectory());

} catch (Exception e) {

e.printStackTrace();

}

}

}

### ****Frequently Used Methods of**** File ****Class****

| **Method** | **Explanation** |
| --- | --- |
| createNewFile() | Creates a new file if it does not exist. |
| exists() | Checks if the file or directory exists. |
| isFile() | Returns true if the File object is a regular file. |
| isDirectory() | Returns true if the File object is a directory. |
| delete() | Deletes the file or directory. |
| getName() | Returns the name of the file or directory. |
| getAbsolutePath() | Returns the absolute path of the file. |

## ****3: Advanced File Handling Techniques****

### ****Reading and Writing Files****

#### ****Using**** FileReader ****and**** FileWriter

* **FileReader**: Reads character-based data from a file.
* **FileWriter**: Writes character-based data to a file.

### ****Sample Code for FileReader and FileWriter****

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class FileReaderWriterExample {

public static void main(String[] args) {

try (FileWriter writer = new FileWriter("output.txt");

FileReader reader = new FileReader("output.txt")) {

// Write data to the file

writer.write("Hello, World!");

// Read data from the file

int character;

while ((character = reader.read()) != -1) {

System.out.print((char) character);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

### ****Buffered Streams****

* **BufferedReader**: Reads text from a file efficiently.
* **BufferedWriter**: Writes text to a file efficiently.

### ****Sample Code for BufferedReader and BufferedWriter****

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class BufferedReaderWriterExample {

public static void main(String[] args) {

try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));

BufferedReader reader = new BufferedReader(new FileReader("output.txt"))) {

// Write to file using BufferedWriter

writer.write("Buffered File Handling in Java.");

// Read from file using BufferedReader

String line;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

## ****4: Serialization in Java****

### ****What is Serialization?****

Serialization is the process of converting an object into a byte stream, so it can be stored in a file or transmitted over a network. Deserialization is the reverse process of reconstructing the object from the byte stream.

### ****Classes Used in Serialization****

* **Serializable Interface**: Marks a class as serializable.
* **ObjectOutputStream**: Serializes objects.
* **ObjectInputStream**: Deserializes objects.

### ****Sample Code for Serialization and Deserialization****

import java.io.\*;

class Person implements Serializable {

String name;

int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

}

public class SerializationExample {

public static void main(String[] args) {

Person person = new Person("John Doe", 30);

// Serialize the object

try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("person.ser"))) {

out.writeObject(person);

System.out.println("Object serialized.");

} catch (IOException e) {

e.printStackTrace();

}

// Deserialize the object

try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("person.ser"))) {

Person deserializedPerson = (Person) in.readObject();

System.out.println("Object deserialized: " + deserializedPerson.name + ", " + deserializedPerson.age);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

### ****Frequently Used Methods in Serialization****

| **Method** | **Explanation** |
| --- | --- |
| writeObject(Object obj) | Serializes the object to the output stream. |
| readObject() | Deserializes the object from the input stream. |

## ****5: File Handling and Serialization in Practice****

### ****Project 1: File-based Object Storage System****

**Project Code**

import java.io.\*;

import java.util.\*;

class Book implements Serializable {

String title;

String author;

public Book(String title, String author) {

this.title = title;

this.author = author;

}

}

public class Library {

private static final String FILE\_NAME = "library.ser";

public static void saveLibrary(List<Book> books) {

try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(FILE\_NAME))) {

out.writeObject(books);

System.out.println("Library saved!");

} catch (IOException e) {

e.printStackTrace();

}

}

public static List<Book> loadLibrary() {

List<Book> books = null;

try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(FILE\_NAME))) {

books = (List<Book>) in.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

return books;

}

public static void main(String[] args) {

List<Book> books = new ArrayList<>();

books.add(new Book("Java Programming", "John Doe"));

books.add(new Book("Effective Java", "Joshua Bloch"));

// Save and load books

saveLibrary(books);

List<Book> loadedBooks = loadLibrary();

loadedBooks.forEach(book -> System.out.println(book.title + " by " + book.author));

}

}

## ****6: Java NIO (New I/O)****

### ****Overview of NIO****

Java NIO (New Input/Output) is an advanced I/O library that allows for non-blocking file operations, memory-mapped files, and better file handling performance.

### ****Key Classes in NIO****

* **Path**: Represents file or directory paths.
* **Files**: Provides utility methods for working with files.
* **FileChannel**: Provides a channel for reading/writing files.

### ****Sample Code for Path and Files****

import java.nio.file.\*;

import java.io.IOException;

public class NIOExample {

public static void main(String[] args) {

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

// Write data using NIO

try {

Files.write(path, "Hello NIO!".getBytes(), StandardOpenOption.CREATE);

} catch (IOException e) {

e.printStackTrace();

}

// Read data using NIO

try {

Files.lines(path).forEach(System.out::println);

} catch (IOException e) {

e.printStackTrace();

}

}

}