**File handling**

File handling in Java is essential for performing operations like **reading**, **writing**, and **manipulating** files and directories.

Here’s a basic overview of file handling in Java:

**File Class**

The **java.io.File** class is the foundation of file handling in Java.

It provides an abstract representation of file and directory pathnames.

**Creating a File Object**

import java.io.File;

public class Main {

public static void main(String[] args) {

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

}

}

**Checking File Properties**

import java.io.File;

public class Main {

public static void main(String[] args) {

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

if (file.exists()) {

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

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

System.out.println("Writable: " + file.canWrite());

System.out.println("Readable: " + file.canRead());

System.out.println("File size in bytes: " + file.length());

}

else {

System.out.println("The file does not exist.");

}

}

}

**Creating a Directory**

import java.io.File;

public class Main {

public static void main(String[] args) {

File directory = new File("mydir");

if (directory.mkdir()) {

System.out.println("Directory created successfully");

} else {

System.out.println("Failed to create directory");

}

}

}

**Reading Files**

**Using FileReader and BufferedReader**

In Java, FileReader and BufferedReader are used to read data from files, but they serve slightly different purposes and have distinct features.

**FileReader**

**FileReader** is a character stream class that reads data from a file.

It is suitable for reading text files.

* **Key Features:**
  + Reads characters from a file.
  + Does not perform buffering; reads directly from the file.
  + Suitable for smaller files as it reads one character at a time or into a small buffer.
* **Common Methods:**
  + read() - Reads a single character.
  + read(char[] cbuf) - Reads characters into a provided array.
  + close() - Closes the stream.

**try resources**

When you use try resources there is no need to close the resource in the finally block.

Java compiler writes the finally block in this case.

**try(create the object) {**

**}**

**Example:**

import java.io.FileReader;

import java.io.IOException;

public class FileReaderExample {

public static void main(String[] args) {

try (FileReader reader = new FileReader("example.txt")) {

int character;

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

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

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**BufferedReader**

BufferedReader is a wrapper class for Reader classes (including FileReader).

It buffers the input to improve performance and provides methods for reading text efficiently.

* **Key Features:**
  + Buffers input for efficiency, reducing I/O operations.
  + Offers methods to read lines of text, making it more convenient for text processing.
  + Suitable for larger files or line-by-line reading.
* **Common Methods:**
  + read() - Reads a single character.
  + readLine() - Reads a line of text.
  + close() - Closes the stream.

**Example:**

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class BufferedReaderExample {

public static void main(String[] args) {

try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {

String line;

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

System.out.println(line);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Comparison**

| **Feature** | **FileReader** | **BufferedReader** |
| --- | --- | --- |
| **Purpose** | Reads characters. | Buffers input for efficient text reading. |
| **Buffering** | No buffering. | Uses an internal buffer. |
| **Performance** | Slower for large files. | Faster due to reduced I/O operations. |
| **Convenience** | Reads character-by-character or arrays of characters. | Provides readLine() for line-by-line reading. |
| **Usage** | Suitable for small text files. | Suitable for large text files and line-based reading. |

**When to Use**

* Use **FileReader** for simple, small-scale character reading.
* Use **BufferedReader** for performance-critical applications or when you need line-by-line reading.

**One more example**

import java.io.FileReader;

import java.io.BufferedReader;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {

String line;

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

System.out.println(line);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Using Scanner**

In Java, the Scanner class is a utility that provides methods for parsing primitive types (like int, double) and strings using regular expressions.

It's commonly used for reading user input, **reading data from files**, or parsing strings.

**Features of Scanner**

1. **Input Sources:**
   * Can read from standard input (System.in), files, or strings.
2. **Parsing Capabilities:**
   * Can parse primitive data types (int, double, float, etc.) and String.
3. **Delimiters:**
   * By default, it uses whitespace as the delimiter.
   * You can customize the delimiter using the **useDelimiter()** method.
4. **Convenience Methods:**
   * Provides methods like nextInt(), nextDouble(), nextLine(), etc., to easily parse inputs.

**Key Methods**

| **Method** | **Description** |
| --- | --- |
| next() | Reads the next token as a String. |
| nextLine() | Reads an entire line of input as a String. |
| nextInt() | Reads the next token as an int. |
| nextDouble() | Reads the next token as a double. |
| hasNext() | Returns true if there is another token available. |
| hasNextInt() | Returns true if the next token is an int. |
| useDelimiter(String) | Sets the delimiter pattern. |

**Examples**

**1. Reading Input from the Console**

import java.util.Scanner;

public class ScannerExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Enter your name:");

String name = scanner.nextLine(); // Reads a full line

System.out.println("Enter your age:");

int age = scanner.nextInt(); // Reads an integer

System.out.println("Hello, " + name + ". You are " + age + " years old.");

scanner.close(); // Close the scanner to release resources

}

}

**2. Reading from a File**

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class FileScannerExample {

public static void main(String[] args) {

try {

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

Scanner scanner = new Scanner(file);

while (scanner.hasNextLine()) { // Check if there is another line

String line = scanner.nextLine(); // Read the line

System.out.println(line);

}

scanner.close();

} catch (FileNotFoundException e) {

System.err.println("File not found!");

}

}

}

**3. Using a Custom Delimiter**

import java.util.Scanner;

public class CustomDelimiterExample {

public static void main(String[] args) {

String data = "apple,orange,banana,grape";

Scanner scanner = new Scanner(data);

scanner.useDelimiter(","); // Set delimiter to comma

while (scanner.hasNext()) {

System.out.println(scanner.next()); // Prints each fruit

}

scanner.close();

}

}

**4. Handling Mixed Input**

import java.util.Scanner;

public class MixedInputExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Enter your ID (integer):");

int id = scanner.nextInt();

System.out.println("Enter your GPA (decimal):");

double gpa = scanner.nextDouble();

scanner.nextLine(); // Consume the leftover newline

System.out.println("Enter your full name:");

String name = scanner.nextLine();

System.out.println("ID: " + id + ", GPA: " + gpa + ", Name: " + name);

scanner.close();

}

}

**Advantages of Scanner**

1. Easy to use for reading various types of input.
2. Flexible input sources (console, file, string).
3. Customizable delimiters.

**Limitations of Scanner**

1. Slower compared to other input classes like BufferedReader for large inputs.
2. May require additional handling for edge cases like newline characters left in the input stream.

**When to Use**

* Use Scanner for simple, user-interactive programs or for parsing structured input from files or strings.
* For performance-critical applications, consider BufferedReader or other classes.

**One more Example**

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

try (Scanner scanner = new Scanner(new File("example.txt"))) {

while (scanner.hasNextLine()) {

System.out.println(scanner.nextLine());

}

} catch (FileNotFoundException e) {

e.printStackTrace();

}

}

}

**Writing Files**

**Using FileWriter and BufferedWriter**

In Java, FileWriter and BufferedWriter are used to write text data to files. They are part of the java.io package and serve slightly different purposes.

**FileWriter**

FileWriter is a character-based class used to write characters to a file.

* **Key Features:**
  + Writes data to a file directly.
  + Supports appending to a file.
  + Not buffered; writing large amounts of data may result in slower performance.
* **Common Methods:**
  + write(int c) - Writes a single character.
  + write(char[] cbuf) - Writes an array of characters.
  + write(String str) - Writes a string.
  + flush() - Flushes the stream.
  + close() - Closes the stream.

**Example:**

import java.io.FileWriter;

import java.io.IOException;

public class FileWriterExample {

public static void main(String[] args) {

try (FileWriter writer = new FileWriter("example.txt")) {

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

writer.write("This is an example using FileWriter.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**BufferedWriter**

BufferedWriter is a wrapper for Writer classes (including FileWriter) that adds buffering to improve performance.

* **Key Features:**
  + Buffers the output for efficient writing.
  + Reduces the number of I/O operations by writing chunks of data at a time.
  + Provides a newLine() method to write platform-independent line separators.
* **Common Methods:**
  + write(String str) - Writes a string.
  + write(char[] cbuf) - Writes an array of characters.
  + newLine() - Writes a newline.
  + flush() - Flushes the buffer to the file.
  + close() - Closes the stream.

**Example:**

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class BufferedWriterExample {

public static void main(String[] args) {

try (BufferedWriter writer = new BufferedWriter(new FileWriter("example.txt"))) {

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

writer.newLine(); // Adds a new line

writer.write("This is an example using BufferedWriter.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Comparison of FileWriter and BufferedWriter**

| **Feature** | **FileWriter** | **BufferedWriter** |
| --- | --- | --- |
| **Buffering** | No buffering. | Uses an internal buffer for efficiency. |
| **Performance** | Slower for large data. | Faster due to reduced I/O operations. |
| **Convenience** | Basic writing methods. | Includes additional methods like newLine(). |
| **Usage** | Suitable for small text data. | Suitable for large text data or frequent writes. |

**When to Use**

* Use **FileWriter** for simple, direct file writing tasks.
* Use **BufferedWriter** when writing large amounts of data or when frequent writes are needed for better performance.

**Appending to a File**

Both FileWriter and BufferedWriter can append to an existing file by using the FileWriter constructor with the append flag set to true.

**Example:**

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class AppendToFileExample {

public static void main(String[] args) {

try (BufferedWriter writer = new BufferedWriter(new FileWriter("example.txt", true))) {

writer.newLine();

writer.write("Appending a new line to the file.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Using FileWriter with BufferedWriter**

It is common to combine both for ease of use and performance.

**Example:**

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class CombinedExample {

public static void main(String[] args) {

try (BufferedWriter writer = new BufferedWriter(new FileWriter("example.txt"))) {

writer.write("This is written using BufferedWriter and FileWriter together.");

writer.newLine();

writer.write("BufferedWriter makes it efficient.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Flushing and Closing**

* Always **flush** the writer to ensure all buffered data is written to the file.
* **Close** the writer to release resources and avoid resource leaks. Using the try-with-resources statement, as shown in examples, ensures proper closing automatically.

**One more Example**

import java.io.FileWriter;

import java.io.BufferedWriter;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

try (BufferedWriter bw = new BufferedWriter(new FileWriter("example.txt", true))) {

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

bw.newLine();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Using PrintWriter**

import java.io.PrintWriter;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

try (PrintWriter pw = new PrintWriter("example.txt")) {

pw.println("Hello, World!");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**File Operations**

**Renaming a File**

import java.io.File;

public class Main {

public static void main(String[] args) {

File oldFile = new File("oldname.txt");

File newFile = new File("newname.txt");

if (oldFile.renameTo(newFile)) {

System.out.println("File renamed successfully");

} else {

System.out.println("Failed to rename file");

}

}

}

**Deleting a File**

import java.io.File;

public class Main {

public static void main(String[] args) {

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

if (file.delete()) {

System.out.println("File deleted successfully");

} else {

System.out.println("Failed to delete file");

}

}

}

This is a basic introduction to file handling in Java. There are more advanced topics like working with file streams, using the Files class from java.nio.file package, and handling different file encodings.

**Serialization and Deserialization**

Serialization and deserialization are processes used to convert an object's state to and from a byte stream, respectively.

They are essential for tasks like saving objects to files or transmitting them over a network.

**Serialization**

**Serialization** is the process of converting an object's state into a byte stream so that it can be saved to a file, sent over a network, or stored in a database.

This process allows objects to be easily saved and reconstructed later.

**Key Points:**

* **Purpose**: To save the state of an object so it can be reconstructed later.
* **Formats**: Objects can be serialized into various formats, such as binary, JSON, or XML.
* **Java Mechanism**: In Java, this is typically done using the Serializable interface. The **ObjectOutputStream** class is used to serialize objects.

**Example:**

import java.io.Serializable;

public class Person implements Serializable {

private static final long serialVersionUID = 1L; // Unique ID for serialization

private String name;

private int age;

// Default constructor

public Person() {}

// Parameterized constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter for name

public String getName() {

return name;

}

// Setter for name

public void setName(String name) {

this.name = name;

}

// Getter for age

public int getAge() {

return age;

}

// Setter for age

public void setAge(int age) {

this.age = age;

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + "}";

}

}

import java.io.FileOutputStream;

import java.io.ObjectOutputStream;

import java.io.IOException;

public class SerializeExample {

public static void main(String[] args) {

Person person = new Person("Alice", 30);

try (FileOutputStream fileOut = new FileOutputStream("person.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

out.writeObject(person);

System.out.println("Serialized data is saved in person.ser");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Deserialization**

**Deserialization** is the reverse process of serialization.

It involves converting a byte stream back into an object.

This allows you to reconstruct the original object from the byte stream that was saved or transmitted.

**Key Points:**

* **Purpose**: To reconstruct an object from its serialized byte stream.
* **Java Mechanism**: In Java, deserialization is typically performed using the ObjectInputStream class.

**Example:**

import java.io.FileInputStream;

import java.io.ObjectInputStream;

import java.io.IOException;

public class DeserializeExample {

public static void main(String[] args) {

Person person = null;

try (FileInputStream fileIn = new FileInputStream("person.ser");

ObjectInputStream in = new ObjectInputStream(fileIn)) {

person = (Person) in.readObject();

System.out.println("Deserialized Person: " + person);

}

catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

**Why Use Serialization and Deserialization?**

* **Persistence**: Save the state of an object to a file or database and reload it later.
* **Communication**: Send objects over a network in a serialized format.
* **Cloning**: Create a deep copy of an object by serializing and then deserializing it.

**Important Considerations**

1. **serialVersionUID**: A unique identifier for each class used during the deserialization process to ensure that a loaded class corresponds exactly to the serialized object.
2. **Transience**: Fields marked with transient are not serialized. This is useful for fields that are not part of the object's persistent state, such as temporary data or sensitive information.
3. **Compatibility**: When evolving classes (e.g., adding new fields), you need to ensure that the serialized data remains compatible with the new version of the class.

Serialization and deserialization are powerful tools for object persistence and communication, and understanding them is key for many applications in Java.

**transient**

**Compress and decompress the files**

To compress and decompress files in Java, you can use the java.util.zip package, which provides classes for working with ZIP files and GZIP files.

Below, I'll provide examples for both compressing and decompressing files using GZIP.

**Compressing a File with GZIP**

**1. Compressing a File**

Here’s how you can compress a file using GZIP:

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.zip.GZIPOutputStream;

public class CompressFileExample {

public static void main(String[] args) {

String sourceFile = "example.txt"; // File to be compressed

String compressedFile = "example.txt.gz"; // Compressed file

try (FileInputStream fis = new FileInputStream(sourceFile);

FileOutputStream fos = new FileOutputStream(compressedFile);

GZIPOutputStream gos = new GZIPOutputStream(fos)) {

byte[] buffer = new byte[1024];

int len;

while ((len = fis.read(buffer)) > 0) {

gos.write(buffer, 0, len);

}

System.out.println("File successfully compressed to " + compressedFile);

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Explanation:**

* **FileInputStream**: Reads the original file.
* **FileOutputStream**: Writes to the compressed file.
* **GZIPOutputStream**: Compresses the data.
* **Buffer**: Reads and writes data in chunks to efficiently handle large files.

**Decompressing a File with GZIP**

**2. Decompressing a File**

Here’s how you can decompress a GZIP-compressed file:

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.zip.GZIPInputStream;

public class DecompressFileExample {

public static void main(String[] args) {

String compressedFile = "example.txt.gz"; // Compressed file

String decompressedFile = "example.txt"; // Decompressed file

try (FileInputStream fis = new FileInputStream(compressedFile);

GZIPInputStream gis = new GZIPInputStream(fis);

FileOutputStream fos = new FileOutputStream(decompressedFile)) {

byte[] buffer = new byte[1024];

int len;

while ((len = gis.read(buffer)) > 0) {

fos.write(buffer, 0, len);

}

System.out.println("File successfully decompressed to " + decompressedFile);

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Explanation:**

* **FileInputStream**: Reads the compressed file.
* **GZIPInputStream**: Decompresses the data.
* **FileOutputStream**: Writes the decompressed data to a new file.
* **Buffer**: Handles reading and writing data in chunks.

**Summary**

* **Compression**: Reduces the file size using GZIPOutputStream.
* **Decompression**: Restores the file to its original state using GZIPInputStream.

These examples cover the basic process of compressing and decompressing files in Java. If you need more advanced compression features or support for different formats, you might consider libraries such as Apache Commons Compress.

**Examples**

**Example 01**

Let's create a more comprehensive program to perform CRUD (Create, Read, Update, Delete) operations on a collection of user-defined objects using file handling.

We'll use the Person class from previous examples, and manage a list of Person objects stored in a file.

**Person Class**

First, let's define the Person class with serialization capabilities:

import java.io.Serializable;

public class Person implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private int age;

public Person() {}

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + "}";

}

}

**CRUD Operations Program**

Here’s a Java program that demonstrates CRUD operations on Person objects stored in a file. This example uses serialization to manage the list of Person objects.

import java.io.\*;

import java.util.ArrayList;

import java.util.List;

public class PersonCRUD {

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

// Load people from the file

public static List<Person> loadPeople() {

List<Person> people = new ArrayList<>();

File file = new File(FILE\_NAME);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

people = (List<Person>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

return people;

}

// Save people to the file

public static void savePeople(List<Person> people) {

try (FileOutputStream fos = new FileOutputStream(FILE\_NAME);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(people);

} catch (IOException e) {

e.printStackTrace();

}

}

// Add a new person

public static void addPerson(Person person) {

List<Person> people = loadPeople();

people.add(person);

savePeople(people);

System.out.println("Person added: " + person);

}

// Read all people

public static void readAllPeople() {

List<Person> people = loadPeople();

for (Person person : people) {

System.out.println(person);

}

}

// Update a person's details by name

public static void updatePerson(String name, Person newDetails) {

List<Person> people = loadPeople();

for (int i = 0; i < people.size(); i++) {

Person person = people.get(i);

if (person.getName().equals(name)) {

people.set(i, newDetails);

savePeople(people);

System.out.println("Person updated: " + newDetails);

return;

}

}

System.out.println("Person not found: " + name);

}

// Delete a person by name

public static void deletePerson(String name) {

List<Person> people = loadPeople();

boolean removed = people.removeIf(person -> person.getName().equals(name));

if (removed) {

savePeople(people);

System.out.println("Person deleted: " + name);

} else {

System.out.println("Person not found: " + name);

}

}

public static void main(String[] args) {

// Example usage

addPerson(new Person("Alice", 30));

addPerson(new Person("Bob", 25));

System.out.println("All people:");

readAllPeople();

updatePerson("Alice", new Person("Alice", 31));

System.out.println("All people after update:");

readAllPeople();

deletePerson("Bob");

System.out.println("All people after deletion:");

readAllPeople();

}

}

**Explanation**

1. **loadPeople**: Reads a list of Person objects from the file. If the file does not exist, it returns an empty list.
2. **savePeople**: Writes a list of Person objects to the file.
3. **addPerson**: Adds a new Person to the list and saves the updated list to the file.
4. **readAllPeople**: Reads and prints all Person objects from the file.
5. **updatePerson**: Updates the details of a Person with the specified name. If found, the person’s details are updated and saved.
6. **deletePerson**: Removes a Person with the specified name from the list. If found, the person is removed and the updated list is saved.

The program demonstrates how to perform CRUD operations using file handling and serialization in Java.

**Example 02:**

Let's create a more comprehensive example with two related classes: Student and Course.

We’ll perform CRUD operations on a list of Student objects, where each student is enrolled in one or more courses.

**Class Definitions**

1. **Course Class**: Represents a course with a name and a course code.
2. **Student Class**: Represents a student with a name, ID, and a list of enrolled courses.

**Course Class**

import java.io.Serializable;

public class Course implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private String code;

public Course() {}

public Course(String name, String code) {

this.name = name;

this.code = code;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

@Override

public String toString() {

return "Course{name='" + name + "', code='" + code + "'}";

}

}

**Student Class**

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

public class Student implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private String studentId;

private List<Course> courses;

public Student() {

courses = new ArrayList<>();

}

public Student(String name, String studentId) {

this.name = name;

this.studentId = studentId;

this.courses = new ArrayList<>();

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getStudentId() {

return studentId;

}

public void setStudentId(String studentId) {

this.studentId = studentId;

}

public List<Course> getCourses() {

return courses;

}

public void addCourse(Course course) {

courses.add(course);

}

public void removeCourse(String courseCode) {

courses.removeIf(course -> course.getCode().equals(courseCode));

}

@Override

public String toString() {

return "Student{name='" + name + "', studentId='" + studentId + "', courses=" + courses + "}";

}

}

**CRUD Operations Program**

Here’s a Java program to handle CRUD operations for Student objects, using serialization to manage the list of students in a file.

import java.io.\*;

import java.util.ArrayList;

import java.util.List;

public class StudentCRUD {

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

// Load students from the file

public static List<Student> loadStudents() {

List<Student> students = new ArrayList<>();

File file = new File(FILE\_NAME);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

students = (List<Student>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

return students;

}

// Save students to the file

public static void saveStudents(List<Student> students) {

try (FileOutputStream fos = new FileOutputStream(FILE\_NAME);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(students);

} catch (IOException e) {

e.printStackTrace();

}

}

// Add a new student

public static void addStudent(Student student) {

List<Student> students = loadStudents();

students.add(student);

saveStudents(students);

System.out.println("Student added: " + student);

}

// Read all students

public static void readAllStudents() {

List<Student> students = loadStudents();

for (Student student : students) {

System.out.println(student);

}

}

// Update a student's details by ID

public static void updateStudent(String studentId, Student newDetails) {

List<Student> students = loadStudents();

for (int i = 0; i < students.size(); i++) {

Student student = students.get(i);

if (student.getStudentId().equals(studentId)) {

students.set(i, newDetails);

saveStudents(students);

System.out.println("Student updated: " + newDetails);

return;

}

}

System.out.println("Student not found: " + studentId);

}

// Delete a student by ID

public static void deleteStudent(String studentId) {

List<Student> students = loadStudents();

boolean removed = students.removeIf(student -> student.getStudentId().equals(studentId));

if (removed) {

saveStudents(students);

System.out.println("Student deleted: " + studentId);

} else {

System.out.println("Student not found: " + studentId);

}

}

public static void main(String[] args) {

// Example usage

Course course1 = new Course("Math", "M101");

Course course2 = new Course("Science", "S101");

Student student1 = new Student("John Doe", "S123");

student1.addCourse(course1);

student1.addCourse(course2);

Student student2 = new Student("Jane Smith", "S456");

addStudent(student1);

addStudent(student2);

System.out.println("All students:");

readAllStudents();

// Update student details

student1.setName("Johnathan Doe");

updateStudent("S123", student1);

System.out.println("All students after update:");

readAllStudents();

// Delete a student

deleteStudent("S456");

System.out.println("All students after deletion:");

readAllStudents();

}

}

**Explanation**

1. **Course Class**: Represents courses with name and code.
2. **Student Class**: Represents students with name, ID, and a list of enrolled courses.
   * **addCourse and removeCourse Methods**: Manage the student's course list.
3. **StudentCRUD Class**:
   * **loadStudents and saveStudents Methods**: Handle reading and writing the list of students from/to a file.
   * **addStudent, readAllStudents, updateStudent, and deleteStudent Methods**: Perform CRUD operations on the list of students.

This program demonstrates how to manage a collection of Student objects, with each student potentially enrolled in multiple Course objects. The use of serialization makes it possible to save and load the list of students between program runs.

**Example 03:**

Let's create a more complex example involving three classes with CRUD operations, sorting, and searching functionalities. We’ll use the following classes:

1. **Book**: Represents a book with a title, author, and ISBN.
2. **Author**: Represents an author with a name and a list of books written.
3. **Library**: Manages a collection of books and authors, including CRUD operations, sorting, and searching.

**Class Definitions**

**1. Book Class**

import java.io.Serializable;

public class Book implements Serializable {

private static final long serialVersionUID = 1L;

private String title;

private String author;

private String isbn;

public Book() {}

public Book(String title, String author, String isbn) {

this.title = title;

this.author = author;

this.isbn = isbn;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

public String getIsbn() {

return isbn;

}

public void setIsbn(String isbn) {

this.isbn = isbn;

}

@Override

public String toString() {

return "Book{title='" + title + "', author='" + author + "', isbn='" + isbn + "'}";

}

}

**2. Author Class**

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

public class Author implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private List<Book> books;

public Author() {

books = new ArrayList<>();

}

public Author(String name) {

this.name = name;

this.books = new ArrayList<>();

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public List<Book> getBooks() {

return books;

}

public void addBook(Book book) {

books.add(book);

}

@Override

public String toString() {

return "Author{name='" + name + "', books=" + books + "}";

}

}

**3. Library Class**

import java.io.\*;

import java.util.\*;

public class Library {

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

private List<Book> books;

private List<Author> authors;

public Library() {

books = new ArrayList<>();

authors = new ArrayList<>();

}

// Load data from file

@SuppressWarnings("unchecked")

public void loadLibrary() {

File file = new File(FILE\_NAME);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

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

authors = (List<Author>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

// Save data to file

public void saveLibrary() {

try (FileOutputStream fos = new FileOutputStream(FILE\_NAME);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(books);

oos.writeObject(authors);

} catch (IOException e) {

e.printStackTrace();

}

}

// Add a new book

public void addBook(Book book) {

books.add(book);

saveLibrary();

System.out.println("Book added: " + book);

}

// Add a new author

public void addAuthor(Author author) {

authors.add(author);

saveLibrary();

System.out.println("Author added: " + author);

}

// Search books by title

public List<Book> searchBooksByTitle(String title) {

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

for (Book book : books) {

if (book.getTitle().toLowerCase().contains(title.toLowerCase())) {

results.add(book);

}

}

return results;

}

// Search books by author

public List<Book> searchBooksByAuthor(String authorName) {

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

for (Book book : books) {

if (book.getAuthor().toLowerCase().contains(authorName.toLowerCase())) {

results.add(book);

}

}

return results;

}

// Sort books by title

public void sortBooksByTitle() {

books.sort(Comparator.comparing(Book::getTitle));

System.out.println("Books sorted by title.");

}

// Sort authors by name

public void sortAuthorsByName() {

authors.sort(Comparator.comparing(Author::getName));

System.out.println("Authors sorted by name.");

}

// Print all books

public void printAllBooks() {

for (Book book : books) {

System.out.println(book);

}

}

// Print all authors

public void printAllAuthors() {

for (Author author : authors) {

System.out.println(author);

}

}

public static void main(String[] args) {

Library library = new Library();

library.loadLibrary();

// Add some authors

Author author1 = new Author("J.K. Rowling");

Author author2 = new Author("George Orwell");

library.addAuthor(author1);

library.addAuthor(author2);

// Add some books

Book book1 = new Book("Harry Potter and the Philosopher's Stone", "J.K. Rowling", "9780747532699");

Book book2 = new Book("1984", "George Orwell", "9780451524935");

library.addBook(book1);

library.addBook(book2);

System.out.println("All books:");

library.printAllBooks();

System.out.println("All authors:");

library.printAllAuthors();

// Search for books

System.out.println("Searching for books by title 'Harry':");

List<Book> foundBooks = library.searchBooksByTitle("Harry");

for (Book book : foundBooks) {

System.out.println(book);

}

// Sort and print

library.sortBooksByTitle();

System.out.println("All books after sorting by title:");

library.printAllBooks();

library.sortAuthorsByName();

System.out.println("All authors after sorting by name:");

library.printAllAuthors();

// Save changes

library.saveLibrary();

}

}

**Explanation**

1. **Book Class**: Represents a book with title, author, and ISBN.
2. **Author Class**: Represents an author with a name and a list of books they have written.
3. **Library Class**:
   * **loadLibrary and saveLibrary**: Load and save books and authors to/from a file.
   * **addBook and addAuthor**: Add books and authors to the library and save the updated lists.
   * **searchBooksByTitle and searchBooksByAuthor**: Search books by title or author.
   * **sortBooksByTitle and sortAuthorsByName**: Sort books by title and authors by name.
   * **printAllBooks and printAllAuthors**: Print all books and authors.

This example demonstrates how to manage a collection of Book and Author objects with CRUD operations, sorting, and searching functionalities. Serialization is used to persist the state of the Library between runs.

**Example 04**

To simulate e-commerce activities using file handling in Java, we can design a simple system with the following components:

1. **Product Class**: Represents a product available in the store.
2. **Customer Class**: Represents a customer with personal details and an order history.
3. **Order Class**: Represents an order placed by a customer, including a list of products.
4. **ECommerceSystem Class**: Manages the products, customers, and orders, including CRUD operations, sorting, and searching functionalities.

**Class Definitions**

**1. Product Class**

import java.io.Serializable;

public class Product implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private double price;

private String productId;

public Product() {}

public Product(String name, double price, String productId) {

this.name = name;

this.price = price;

this.productId = productId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

public String getProductId() {

return productId;

}

public void setProductId(String productId) {

this.productId = productId;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + ", productId='" + productId + "'}";

}

}

**2. Customer Class**

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

public class Customer implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private String email;

private List<Order> orders;

public Customer() {

orders = new ArrayList<>();

}

public Customer(String name, String email) {

this.name = name;

this.email = email;

this.orders = new ArrayList<>();

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public List<Order> getOrders() {

return orders;

}

public void addOrder(Order order) {

orders.add(order);

}

@Override

public String toString() {

return "Customer{name='" + name + "', email='" + email + "', orders=" + orders + "}";

}

}

**3. Order Class**

import java.io.Serializable;

import java.util.List;

public class Order implements Serializable {

private static final long serialVersionUID = 1L;

private List<Product> products;

private double totalPrice;

public Order() {}

public Order(List<Product> products) {

this.products = products;

this.totalPrice = calculateTotalPrice();

}

private double calculateTotalPrice() {

return products.stream().mapToDouble(Product::getPrice).sum();

}

public List<Product> getProducts() {

return products;

}

public void setProducts(List<Product> products) {

this.products = products;

this.totalPrice = calculateTotalPrice();

}

public double getTotalPrice() {

return totalPrice;

}

@Override

public String toString() {

return "Order{products=" + products + ", totalPrice=" + totalPrice + "}";

}

}

**4. ECommerceSystem Class**

import java.io.\*;

import java.util.\*;

public class ECommerceSystem {

private static final String PRODUCT\_FILE = "products.ser";

private static final String CUSTOMER\_FILE = "customers.ser";

private static final String ORDER\_FILE = "orders.ser";

private List<Product> products;

private List<Customer> customers;

private List<Order> orders;

public ECommerceSystem() {

products = new ArrayList<>();

customers = new ArrayList<>();

orders = new ArrayList<>();

}

// Load data from files

@SuppressWarnings("unchecked")

public void loadData() {

loadProducts();

loadCustomers();

loadOrders();

}

private void loadProducts() {

File file = new File(PRODUCT\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

products = (List<Product>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

private void loadCustomers() {

File file = new File(CUSTOMER\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

customers = (List<Customer>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

private void loadOrders() {

File file = new File(ORDER\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

orders = (List<Order>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

// Save data to files

public void saveData() {

saveProducts();

saveCustomers();

saveOrders();

}

private void saveProducts() {

try (FileOutputStream fos = new FileOutputStream(PRODUCT\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(products);

} catch (IOException e) {

e.printStackTrace();

}

}

private void saveCustomers() {

try (FileOutputStream fos = new FileOutputStream(CUSTOMER\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(customers);

} catch (IOException e) {

e.printStackTrace();

}

}

private void saveOrders() {

try (FileOutputStream fos = new FileOutputStream(ORDER\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(orders);

} catch (IOException e) {

e.printStackTrace();

}

}

// Add a new product

public void addProduct(Product product) {

products.add(product);

saveProducts();

System.out.println("Product added: " + product);

}

// Add a new customer

public void addCustomer(Customer customer) {

customers.add(customer);

saveCustomers();

System.out.println("Customer added: " + customer);

}

// Add a new order

public void addOrder(Order order) {

orders.add(order);

saveOrders();

System.out.println("Order added: " + order);

}

// Search products by name

public List<Product> searchProductsByName(String name) {

List<Product> results = new ArrayList<>();

for (Product product : products) {

if (product.getName().toLowerCase().contains(name.toLowerCase())) {

results.add(product);

}

}

return results;

}

// Search customers by name

public List<Customer> searchCustomersByName(String name) {

List<Customer> results = new ArrayList<>();

for (Customer customer : customers) {

if (customer.getName().toLowerCase().contains(name.toLowerCase())) {

results.add(customer);

}

}

return results;

}

// Sort products by price

public void sortProductsByPrice() {

products.sort(Comparator.comparingDouble(Product::getPrice));

System.out.println("Products sorted by price.");

}

// Print all products

public void printAllProducts() {

for (Product product : products) {

System.out.println(product);

}

}

// Print all customers

public void printAllCustomers() {

for (Customer customer : customers) {

System.out.println(customer);

}

}

// Main method to demonstrate functionalities

public static void main(String[] args) {

ECommerceSystem system = new ECommerceSystem();

system.loadData();

// Add products

Product product1 = new Product("Laptop", 999.99, "P001");

Product product2 = new Product("Smartphone", 499.99, "P002");

system.addProduct(product1);

system.addProduct(product2);

// Add customers

Customer customer1 = new Customer("Alice Johnson", "alice.johnson@example.com");

Customer customer2 = new Customer("Bob Smith", "bob.smith@example.com");

system.addCustomer(customer1);

system.addCustomer(customer2);

// Create an order

Order order1 = new Order(Arrays.asList(product1, product2));

customer1.addOrder(order1);

system.addOrder(order1);

System.out.println("All products:");

system.printAllProducts();

System.out.println("All customers:");

system.printAllCustomers();

// Search for products

System.out.println("Searching for products with name 'Laptop':");

List<Product> foundProducts = system.searchProductsByName("Laptop");

for (Product product : foundProducts) {

System.out.println(product);

}

// Sort and print products

system.sortProductsByPrice();

System.out.println("All products after sorting by price:");

system.printAllProducts();

// Save changes

system.saveData();

}

}

**Explanation**

1. **Product Class**: Represents a product with a name, price, and product ID.
2. **Customer Class**: Represents a customer with a name, email, and a list of orders.
3. **Order Class**: Represents an order with a list of products and a total price.
4. **ECommerceSystem Class**:
   * **Loading and Saving Data**: Handles reading and writing data to files for products, customers, and orders.
   * **CRUD Operations**: Add products, customers, and orders.
   * **Search and Sort**: Search products and customers by name, and sort products by price.
   * **Print All**: Print all products and customers.

**How to Run**

1. **Compile**: javac Product.java Customer.java Order.java ECommerceSystem.java
2. **Run**: java ECommerceSystem

This example demonstrates a basic e-commerce system where you can manage products, customers, and orders. It includes functionality for adding, searching, sorting, and displaying data, all while using file handling to persist the state between runs.

**Example 06**

An HR Management System (HRMS) typically involves managing various aspects such as employees, departments, salaries, and more.

For this example, we'll create a simplified HRMS with the following classes:

1. **Employee**: Represents an employee.
2. **Department**: Represents a department.
3. **Salary**: Represents an employee's salary.
4. **Position**: Represents an employee's position.
5. **Project**: Represents a project an employee is working on.
6. **Attendance**: Represents employee attendance records.
7. **HRMS**: Manages employees, departments, salaries, positions, projects, and attendance records.

**Class Definitions**

**1. Employee Class**

import java.io.Serializable;

public class Employee implements Serializable {

private static final long serialVersionUID = 1L;

private String employeeId;

private String name;

private Department department;

private Position position;

private Salary salary;

public Employee() {}

public Employee(String employeeId, String name, Department department, Position position, Salary salary) {

this.employeeId = employeeId;

this.name = name;

this.department = department;

this.position = position;

this.salary = salary;

}

public String getEmployeeId() {

return employeeId;

}

public void setEmployeeId(String employeeId) {

this.employeeId = employeeId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Department getDepartment() {

return department;

}

public void setDepartment(Department department) {

this.department = department;

}

public Position getPosition() {

return position;

}

public void setPosition(Position position) {

this.position = position;

}

public Salary getSalary() {

return salary;

}

public void setSalary(Salary salary) {

this.salary = salary;

}

@Override

public String toString() {

return "Employee{employeeId='" + employeeId + "', name='" + name + "', department=" + department +

", position=" + position + ", salary=" + salary + "}";

}

}

**2. Department Class**

import java.io.Serializable;

public class Department implements Serializable {

private static final long serialVersionUID = 1L;

private String departmentId;

private String name;

public Department() {}

public Department(String departmentId, String name) {

this.departmentId = departmentId;

this.name = name;

}

public String getDepartmentId() {

return departmentId;

}

public void setDepartmentId(String departmentId) {

this.departmentId = departmentId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public String toString() {

return "Department{departmentId='" + departmentId + "', name='" + name + "'}";

}

}

**3. Salary Class**

import java.io.Serializable;

public class Salary implements Serializable {

private static final long serialVersionUID = 1L;

private double baseSalary;

private double bonus;

public Salary() {}

public Salary(double baseSalary, double bonus) {

this.baseSalary = baseSalary;

this.bonus = bonus;

}

public double getBaseSalary() {

return baseSalary;

}

public void setBaseSalary(double baseSalary) {

this.baseSalary = baseSalary;

}

public double getBonus() {

return bonus;

}

public void setBonus(double bonus) {

this.bonus = bonus;

}

public double getTotalSalary() {

return baseSalary + bonus;

}

@Override

public String toString() {

return "Salary{baseSalary=" + baseSalary + ", bonus=" + bonus + ", totalSalary=" + getTotalSalary() + "}";

}

}

**4. Position Class**

import java.io.Serializable;

public class Position implements Serializable {

private static final long serialVersionUID = 1L;

private String positionId;

private String title;

public Position() {}

public Position(String positionId, String title) {

this.positionId = positionId;

this.title = title;

}

public String getPositionId() {

return positionId;

}

public void setPositionId(String positionId) {

this.positionId = positionId;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

@Override

public String toString() {

return "Position{positionId='" + positionId + "', title='" + title + "'}";

}

}

**5. Project Class**

import java.io.Serializable;

public class Project implements Serializable {

private static final long serialVersionUID = 1L;

private String projectId;

private String projectName;

public Project() {}

public Project(String projectId, String projectName) {

this.projectId = projectId;

this.projectName = projectName;

}

public String getProjectId() {

return projectId;

}

public void setProjectId(String projectId) {

this.projectId = projectId;

}

public String getProjectName() {

return projectName;

}

public void setProjectName(String projectName) {

this.projectName = projectName;

}

@Override

public String toString() {

return "Project{projectId='" + projectId + "', projectName='" + projectName + "'}";

}

}

**6. Attendance Class**

import java.io.Serializable;

import java.time.LocalDate;

public class Attendance implements Serializable {

private static final long serialVersionUID = 1L;

private LocalDate date;

private boolean present;

public Attendance() {}

public Attendance(LocalDate date, boolean present) {

this.date = date;

this.present = present;

}

public LocalDate getDate() {

return date;

}

public void setDate(LocalDate date) {

this.date = date;

}

public boolean isPresent() {

return present;

}

public void setPresent(boolean present) {

this.present = present;

}

@Override

public String toString() {

return "Attendance{date=" + date + ", present=" + present + "}";

}

}

**7. HRMS Class**

import java.io.\*;

import java.util.\*;

import java.time.LocalDate;

public class HRMS {

private static final String EMPLOYEE\_FILE = "employees.ser";

private static final String DEPARTMENT\_FILE = "departments.ser";

private static final String SALARY\_FILE = "salaries.ser";

private static final String POSITION\_FILE = "positions.ser";

private static final String PROJECT\_FILE = "projects.ser";

private static final String ATTENDANCE\_FILE = "attendance.ser";

private List<Employee> employees;

private List<Department> departments;

private List<Salary> salaries;

private List<Position> positions;

private List<Project> projects;

private List<Attendance> attendanceRecords;

public HRMS() {

employees = new ArrayList<>();

departments = new ArrayList<>();

salaries = new ArrayList<>();

positions = new ArrayList<>();

projects = new ArrayList<>();

attendanceRecords = new ArrayList<>();

}

// Load data from files

@SuppressWarnings("unchecked")

public void loadData() {

loadEmployees();

loadDepartments();

loadSalaries();

loadPositions();

loadProjects();

loadAttendance();

}

private void loadEmployees() {

File file = new File(EMPLOYEE\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

employees = (List<Employee>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

private void loadDepartments() {

File file = new File(DEPARTMENT\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

departments = (List<Department>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

private void loadSalaries() {

File file = new File(SALARY\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

salaries = (List<Salary>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

private void loadPositions() {

File file = new File(POSITION\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

positions = (List<Position>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

private void loadProjects() {

File file = new File(PROJECT\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

projects = (List<Project>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

private void loadAttendance() {

File file = new File(ATTENDANCE\_FILE);

if (file.exists()) {

try (FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis)) {

attendanceRecords = (List<Attendance>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

// Save data to files

public void saveData() {

saveEmployees();

saveDepartments();

saveSalaries();

savePositions();

saveProjects();

saveAttendance();

}

private void saveEmployees() {

try (FileOutputStream fos = new FileOutputStream(EMPLOYEE\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(employees);

} catch (IOException e) {

e.printStackTrace();

}

}

private void saveDepartments() {

try (FileOutputStream fos = new FileOutputStream(DEPARTMENT\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(departments);

} catch (IOException e) {

e.printStackTrace();

}

}

private void saveSalaries() {

try (FileOutputStream fos = new FileOutputStream(SALARY\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(salaries);

} catch (IOException e) {

e.printStackTrace();

}

}

private void savePositions() {

try (FileOutputStream fos = new FileOutputStream(POSITION\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(positions);

} catch (IOException e) {

e.printStackTrace();

}

}

private void saveProjects() {

try (FileOutputStream fos = new FileOutputStream(PROJECT\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(projects);

} catch (IOException e) {

e.printStackTrace();

}

}

private void saveAttendance() {

try (FileOutputStream fos = new FileOutputStream(ATTENDANCE\_FILE);

ObjectOutputStream oos = new ObjectOutputStream(fos)) {

oos.writeObject(attendanceRecords);

} catch (IOException e) {

e.printStackTrace();

}

}

// Add a new employee

public void addEmployee(Employee employee) {

employees.add(employee);

saveEmployees();

System.out.println("Employee added: " + employee);

}

// Add a new department

public void addDepartment(Department department) {

departments.add(department);

saveDepartments();

System.out.println("Department added: " + department);

}

// Add a new salary

public void addSalary(Salary salary) {

salaries.add(salary);

saveSalaries();

System.out.println("Salary added: " + salary);

}

// Add a new position

public void addPosition(Position position) {

positions.add(position);

savePositions();

System.out.println("Position added: " + position);

}

// Add a new project

public void addProject(Project project) {

projects.add(project);

saveProjects();

System.out.println("Project added: " + project);

}

// Add an attendance record

public void addAttendance(Attendance attendance) {

attendanceRecords.add(attendance);

saveAttendance();

System.out.println("Attendance record added: " + attendance);

}

// Search employees by name

public List<Employee> searchEmployeesByName(String name) {

List<Employee> results = new ArrayList<>();

for (Employee employee : employees) {

if (employee.getName().toLowerCase().contains(name.toLowerCase())) {

results.add(employee);

}

}

return results;

}

// Search departments by name

public List<Department> searchDepartmentsByName(String name) {

List<Department> results = new ArrayList<>();

for (Department department : departments) {

if (department.getName().toLowerCase().contains(name.toLowerCase())) {

results.add(department);

}

}

return results;

}

// Sort employees by salary

public void sortEmployeesBySalary() {

employees.sort(Comparator.comparingDouble(e -> e.getSalary().getTotalSalary()));

System.out.println("Employees sorted by salary.");

}

// Print all employees

public void printAllEmployees() {

for (Employee employee : employees) {

System.out.println(employee);

}

}

// Print all departments

public void printAllDepartments() {

for (Department department : departments) {

System.out.println(department);

}

}

// Main method to demonstrate functionalities

public static void main(String[] args) {

HRMS hrms = new HRMS();

hrms.loadData();

// Add departments

Department department1 = new Department("D001", "IT");

Department department2 = new Department("D002", "HR");

hrms.addDepartment(department1);

hrms.addDepartment(department2);

// Add positions

Position position1 = new Position("P001", "Software Engineer");

Position position2 = new Position("P002", "HR Manager");

hrms.addPosition(position1);

hrms.addPosition(position2);

// Add salaries

Salary salary1 = new Salary(60000, 5000);

Salary salary2 = new Salary(70000, 6000);

hrms.addSalary(salary1);

hrms.addSalary(salary2);

// Add employees

Employee employee1 = new Employee("E001", "Alice", department1, position1, salary1);

Employee employee2 = new Employee("E002", "Bob", department2, position2, salary2);

hrms.addEmployee(employee1);

hrms.addEmployee(employee2);

// Add projects

Project project1 = new Project("PRJ001", "Project Alpha");

Project project2 = new Project("PRJ002", "Project Beta");

hrms.addProject(project1);

hrms.addProject(project2);

// Add attendance records

Attendance attendance1 = new Attendance(LocalDate.now(), true);

Attendance attendance2 = new Attendance(LocalDate.now().minusDays(1), false);

hrms.addAttendance(attendance1);

hrms.addAttendance(attendance2);

System.out.println("All employees:");

hrms.printAllEmployees();

System.out.println("All departments:");

hrms.printAllDepartments();

// Search for employees

System.out.println("Searching for employees with name 'Alice':");

List<Employee> foundEmployees = hrms.searchEmployeesByName("Alice");

for (Employee employee : foundEmployees) {

System.out.println(employee);

}

// Sort and print employees

hrms.sortEmployeesBySalary();

System.out.println("All employees after sorting by salary:");

hrms.printAllEmployees();

// Save changes

hrms.saveData();

}

}

**Explanation**

1. **Employee Class**: Represents an employee with details like ID, name, department, position, and salary.
2. **Department Class**: Represents a department with ID and name.
3. **Salary Class**: Represents an employee's salary with base salary and bonus.
4. **Position Class**: Represents an employee's position with ID and title.
5. **Project Class**: Represents a project with ID and name.
6. **Attendance Class**: Represents attendance records with date and presence status.
7. **HRMS Class**:
   * **Loading and Saving Data**: Handles reading and writing data to files for employees, departments, salaries, positions, projects, and attendance records.
   * **CRUD Operations**: Add employees, departments, salaries, positions, projects, and attendance records.
   * **Search and Sort**: Search employees and departments by name, and sort employees by salary.
   * **Print All**: Print all employees and departments.

**How to Run**

1. **Compile**: javac Employee.java Department.java Salary.java Position.java Project.java Attendance.java HRMS.java
2. **Run**: java HRMS

This example provides a comprehensive overview of an HRMS with file handling. It includes CRUD operations, searching, sorting, and printing functionalities, and demonstrates how to manage various HR-related data using Java.