PROGRAMMING IN JAVA

Lecture 12

File Input/Output and Data Persistence

Byte and Character Streams, Buffered I/O, NIO, `Path`, `Files`, Serialization, Properties, and JAR Organization

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

After completing this lecture, students should be able to:

1. explain the purpose of file input/output in Java;
2. distinguish byte streams from character streams;
3. read and write text files;
4. use buffered readers and writers;
5. use try-with-resources for safe resource handling;
6. work with the legacy `File` class;
7. use modern NIO classes such as `Path` and `Files`;
8. create, copy, move, and delete files;
9. create and inspect directories;
10. process structured text such as CSV-like data;
11. explain the concept of object serialization;
12. save and restore simple serializable objects;
13. work with Java properties files;
14. organize file-processing code into reusable methods;
15. explain the role of JAR files in packaging Java applications.

2. Why File I/O Is Important

Variables and objects normally exist only while a program is running.

When the program stops, ordinary in-memory data is lost.

Many applications need data to persist after execution.

Examples include:

- student records;
- application settings;
- log files;
- reports;
- exported data;
- cached information;
- configuration values.

File input/output provides one of the simplest forms of persistence.

3. What Is a Stream?

A **stream** is a flow of data between a source and a destination.

For input:

Illustration

```
File ---> Java program
```

For output:

Illustration

```
Java program ---> File
```

Java provides two major stream families:

- byte streams;
- character streams.

4. Byte Streams

Byte streams work with raw bytes.

Important base classes are:

- `InputStream`
- `OutputStream`

Common subclasses include:

- `FileInputStream`
- `FileOutputStream`
- `BufferedInputStream`
- `BufferedOutputStream`

Byte streams are suitable for binary data such as:

- images;
- audio;
- compressed data;
- arbitrary binary files.

5. Character Streams

Character streams work with textual data.

Important base classes are:

- `Reader`
- `Writer`

Common subclasses include:

- `FileReader`
- `FileWriter`
- `BufferedReader`
- `BufferedWriter`

Character streams are usually preferable for text.

6. Byte Streams vs Character Streams

Feature	Byte Streams	Character Streams
Basic unit	byte	character
Base classes	`InputStream`, `OutputStream`	`Reader`, `Writer`
Best for	binary data	text data
Examples	images, binary files	text, CSV, reports

7. Writing a Text File

A simple text file can be written using `FileWriter`.

Example 1. Writing Text with `FileWriter`

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

public class Main {

    public static void main(String[] args) {

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

            writer.write("Aruzhan\n");
            writer.write("Dias\n");
            writer.write("Ali\n");

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

        } catch (IOException e) {

            System.out.println(
                "Write error: " +
                e.getMessage()
            );
        }
    }
}
```

Expected output

```
File written successfully.
```

The generated file contains:

Illustration

```
Aruzhan
Dias
Ali
```

8. Appending to a File

By default, `FileWriter` overwrites an existing file.

To append:

```
new FileWriter("students.txt", true)
```

Example:

```
try (FileWriter writer =
        new FileWriter(
            "students.txt",
            true)) {

    writer.write("Dana\n");
}
```

9. Reading a Text File

A text file can be read using `FileReader` and `BufferedReader`.

Example 2. Reading Line by Line

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

public class Main {

    public static void main(String[] args) {

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

            String line;

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

                System.out.println(line);
            }

        } catch (IOException e) {

            System.out.println(
                "Read error: " +
                e.getMessage()
            );
        }
    }
}
```

If the file contains three names, the output is:

Illustration

```
Aruzhan
Dias
Ali
```

10. Why Buffering Is Useful

Reading or writing one small unit directly to disk can be inefficient.

Buffered streams and readers reduce the number of low-level I/O operations by using an intermediate memory buffer.

Common buffered classes include:

- `BufferedReader`
- `BufferedWriter`
- `BufferedInputStream`
- `BufferedOutputStream`

Example 3. Buffered Writer

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

public class Main {

    public static void main(String[] args) {

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

            writer.write(
                "Java Programming Report"
            );

            writer.newLine();

            writer.write(
                "Students: 30"
            );

            writer.newLine();

            writer.write(
                "Status: Completed"
            );

        } catch (IOException e) {
```

Example 3 (continued)

```
31
32              e.printStackTrace();
33          }
34      }
35  }
```

11. Try-With-Resources

Files and streams should be closed after use.

Modern Java uses try-with-resources:

```
try (Resource resource =
        createResource()) {

    // use resource
}
```

The resource is closed automatically.

This is safer and cleaner than manually closing it in a `finally` block.

12. The Legacy `File` Class

The class `java.io.File` represents a file-system path.

It does not itself read or write file contents.

Example:

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

Useful methods include:

- `exists()`
- `isFile()`
- `isDirectory()`
- `length()`
- `getAbsolutePath()`
- `mkdir()`
- `mkdirs()`
- `delete()`
- `listFiles()`

Example 4. Inspecting a File

```
import java.io.File;

public class Main {

    public static void main(String[] args) {

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

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

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

        System.out.println(
            "Size: " +
            file.length()
        );
    }
}
```

13. Modern Java NIO

Modern Java applications commonly use classes from:

Illustration

`java.nio.file`

Important classes include:

- `Path`
- `Paths` in older style APIs
- `Files`
- `StandardCopyOption`

A path can be created with:

```
Path path =
        Path.of("students.txt");
```

14. Reading Entire Text with `Files.readString()`

Example 5. Reading a Whole File

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Main {

    public static void main(String[] args) {

        Path path =
                Path.of("students.txt");

        try {

            String content =
                    Files.readString(path);

            System.out.println(content);

        } catch (IOException e) {

            System.out.println(
                "Could not read file."
            );
        }
    }
}
```

15. Writing Text with `Files.writeString()`

Example 6. Writing with NIO

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Main {

    public static void main(String[] args) {

        Path path =
                Path.of("message.txt");

        try {

            Files.writeString(
                path,
                "Hello from Java NIO!"
            );

            System.out.println(
                "Saved."
            );

        } catch (IOException e) {

            System.out.println(
                e.getMessage()
            );
        }
    }
}
```

Expected output

```
Saved.
```

16. Reading All Lines

`Files.readAllLines()` returns a list of strings.

Example 7. Reading Lines into a List

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        try {

            List<String> lines =
                    Files.readAllLines(
                        Path.of(
                            "students.txt"
                        )
                    );

            for (String line : lines) {
                System.out.println(line);
            }

        } catch (IOException e) {

            System.out.println(
                "File error."
            );
        }
    }
}
```

17. Creating Directories

Directories can be created with:

```
Files.createDirectory(path);
```

or:

```
Files.createDirectories(path);
```

The second method can create multiple missing parent directories.

Example 8. Creating a Directory

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Main {

    public static void main(String[] args) {

        Path directory =
                Path.of(
                    "data",
                    "students"
                );

        try {

            Files.createDirectories(
                directory
            );

            System.out.println(
                "Directory ready."
            );

        } catch (IOException e) {

            System.out.println(
                e.getMessage()
            );
        }
    }
}
```

18. Copying Files

Java NIO can copy files directly.

```
Files.copy(source, target);
```

To replace an existing target:

```
Files.copy(
    source,
    target,
    StandardCopyOption.REPLACE_EXISTING
);
```

Example 9. Copying a File

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;

public class Main {

    public static void main(String[] args) {

        Path source =
                Path.of("students.txt");

        Path target =
                Path.of(
                    "students_backup.txt"
                );

        try {

            Files.copy(
                source,
                target,
                StandardCopyOption
                    .REPLACE_EXISTING
            );

            System.out.println(
                "Backup created."
            );

        } catch (IOException e) {

            System.out.println(
                "Copy failed."
            );
        }
    }
```

Example 9 (continued)

```
38 }
```

19. Moving and Renaming Files

A file can be moved or renamed using:

```
Files.move(source, target);
```

This can be used to:

- rename a file;
- move a file to another directory.

20. Deleting Files

Useful methods:

```
Files.delete(path);
```

and:

```
Files.deleteIfExists(path);
```

The second form is convenient when it is acceptable that the file may already be missing.

21. Listing Directory Contents

The NIO API can list directory contents.

Example 10. Listing Files in a Directory

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Main {

    public static void main(String[] args) {

        Path directory =
                Path.of(".");

        try (var stream =
                Files.list(directory)) {

            stream.forEach(
                System.out::println
            );

        } catch (IOException e) {

            System.out.println(
                "Cannot list directory."
            );
        }
    }
}
```

This example uses `var` for a local variable whose type is inferred by the compiler.

22. Processing Numeric Data from a File

A common task is:

1. read text lines;
2. convert them to numbers;
3. process the numeric values.

Suppose `numbers.txt` contains:

Illustration

```
10
20
30
40
```

Java program:

Example 11. Sum and Average from a File

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        try {

            List<String> lines =
                Files.readAllLines(
                    Path.of(
                        "numbers.txt"
                    )
                );

            double sum = 0.0;

            for (String line : lines) {

                sum +=
                    Double.parseDouble(
                        line
                    );
            }

            double average =
                    sum / lines.size();

            System.out.println(
                "Sum = " + sum
            );

            System.out.println(
```

Example 11 (continued)

```
                "Average = " + average
            );

        } catch (IOException |
                 NumberFormatException e) {

            System.out.println(
                "Data processing error."
            );
        }
    }
}
```

Expected output

```
Sum = 100.0
Average = 25.0
```

23. Structured Text Files

Applications often store structured records in plain text.

A simple CSV-like format:

Illustration

```
1001,Aruzhan,3.9
1002,Dias,3.5
1003,Ali,3.7
```

Each line is one record.

Fields are separated by commas.

Example 12. Reading Simple CSV-Like Data

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class Main {

```

Example 12 (continued)

```
8       public static void main(String[] args) {
9
10          try {
11
12              List<String> lines =
13                  Files.readAllLines(
14                      Path.of(
15                          "students.csv"
16                      )
17                  );
18
19              for (String line : lines) {
20
21                  String[] parts =
22                          line.split(",");
23
24                  int id =
25                      Integer.parseInt(
26                          parts[0]
27                      );
28
29                  String name =
30                          parts[1];
31
32                  double gpa =
33                      Double.parseDouble(
34                          parts[2]
35                      );
36
37                  System.out.println(
38                      id + " " +
39                      name + " " +
40                      gpa
41                  );
42              }
43
44          } catch (Exception e) {
45
46              System.out.println(
47                  "Invalid file data."
48              );
49          }
50      }
51  }
```

For production software, real CSV files may require a dedicated CSV library, especially when

fields can contain commas, quotes, or newlines.

24. Writing Object Data as Text

An object can be converted into a text representation.

Example:

```
class Student {

    private int id;
    private String name;
    private double gpa;

    public String toCsv() {
        return id + "," +
               name + "," +
               gpa;
    }
}
```

The resulting string can be written to a file.

25. Serialization

Serialization is the process of converting an object state into a form that can be stored or transmitted.

Traditional Java object serialization uses:

- `Serializable`
- `ObjectOutputStream`
- `ObjectInputStream`

A class may implement:

```
implements Serializable
```

The `Serializable` interface is a marker interface.

Example 13. Serializing an Object

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Main {

    static class Student
            implements Serializable {

        private int id;
        private String name;

        public Student(
                int id,
                String name) {

            this.id = id;
            this.name = name;
        }
    }

    public static void main(String[] args) {

        Student student =
                new Student(
                    1001,
                    "Aruzhan"
                );

        try (ObjectOutputStream out =
                new ObjectOutputStream(
                    new FileOutputStream(
                        "student.bin"))) {

            out.writeObject(student);

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

        } catch (IOException e) {

            System.out.println(
```

Example 13 (continued)

```
                "Serialization failed."
            );
        }
    }
}
```

Expected output

```
Object saved.
```

Example 14. Deserializing an Object

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.Serializable;

public class Main {

    static class Student
            implements Serializable {

        private int id;
        private String name;

        @Override
        public String toString() {
            return id + " " + name;
        }
    }

    public static void main(String[] args) {

        try (ObjectInputStream in =
                new ObjectInputStream(
                    new FileInputStream(
                        "student.bin"))) {

            Student student =
                    (Student)
                    in.readObject();

            System.out.println(student);

        } catch (Exception e) {
```

Example 14 (continued)

```
33
34              System.out.println(
35                  "Could not restore object."
36              );
37          }
38      }
39  }
```

26. Serialization Caution

Traditional Java serialization is useful for teaching object persistence concepts and for some controlled environments.

However, it should not be treated as the universal persistence format for modern applications.

For long-term or interoperable storage, applications often prefer formats such as:

- JSON;
- XML;
- CSV;
- relational databases.

These are easier to inspect and exchange between different systems.

27. The `transient` Keyword

A field declared `transient` is skipped by traditional Java serialization.

Example:

```
private transient String password;
```

This may be useful for data that should not be serialized.

It is not, by itself, a complete security mechanism.

28. Properties Files

Java provides the `Properties` class for simple key-value configuration.

Example file:

Illustration

```
app.name=Student System
app.version=1.0
max.students=30
```

Example 15. Writing Properties

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

public class Main {

    public static void main(String[] args) {

        Properties properties =
                new Properties();

        properties.setProperty(
                "app.name",
                "Student System"
        );

        properties.setProperty(
                "max.students",
                "30"
        );

        try (FileWriter writer =
                new FileWriter(
                    "config.properties")) {

            properties.store(
                writer,
                "Application settings"
            );

        } catch (IOException e) {

```

Example 15 (continued)

```
            System.out.println(
                "Could not save settings."
            );
        }
    }
}
```

Example 16. Reading Properties

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

public class Main {

    public static void main(String[] args) {

        Properties properties =
                new Properties();

        try (FileReader reader =
                new FileReader(
                    "config.properties")) {

            properties.load(reader);

            System.out.println(
                properties.getProperty(
                    "app.name"
                )
            );

            System.out.println(
                properties.getProperty(
                    "max.students"
                )
            );

        } catch (IOException e) {

            System.out.println(
                "Could not load settings."
            );
        }
    }
```

Example 16 (continued)

```
37 }
```

Possible output:

Illustration

```
Student System
30
```

29. Path Operations

`Path` provides useful path operations.

Example:

```
Path path =
        Path.of(
            "data",
            "students.txt"
        );
```

Useful methods include:

- `getFileName()`
- `getParent()`
- `toAbsolutePath()`
- `normalize()`
- `resolve()`

Example 17. Resolving Paths

```
1  import java.nio.file.Path;
2
3  public class Main {
4
5      public static void main(String[] args) {
6
7          Path base =
8                  Path.of("data");
9
10         Path file =
11                 base.resolve(
12                     "students.txt"
13                 );
14
15         System.out.println(file);
16
17         System.out.println(
18             file.toAbsolutePath()
19         );
20     }
21 }
```

30. Recursive Directory Traversal

Java can traverse directory trees.

For example:

```
Files.walk(path)
```

returns a stream of paths.

This is useful for:

- file search;
- backup tools;
- directory statistics;
- recursive processing.

Example 18. Counting Text Files

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Main {

    public static void main(String[] args) {

        Path root = Path.of(".");

        try (var paths =
                Files.walk(root)) {

            long count =
                paths
                    .filter(
                        Files::isRegularFile
                    )
                    .filter(
                        p -> p.toString()
                              .endsWith(
                                  ".txt")
                    )
                    .count();

            System.out.println(
```

Example 18 (continued)

```
                    "Text files: " +
                    count
                );

            } catch (IOException e) {

                System.out.println(
                    "Traversal failed."
                );
            }
        }
    }
```

31. File Processing and Exceptions

File operations can fail for many reasons:

- file not found;
- insufficient permissions;
- invalid path;
- storage unavailable;
- malformed data;
- file already exists;
- directory not empty.

Therefore robust file-processing code should:

1. use try-with-resources;
2. handle meaningful exception types;
3. validate external data;
4. give useful messages;
5. avoid silently ignoring errors.

32. JAR Files

A **JAR** file is a Java Archive.

It can package:

- compiled `.class` files;
- resources;
- metadata;
- configuration files.

A JAR file normally has the extension:

Illustration

```
.jar
```

Example command:

Illustration

```
jar --create --file app.jar *.class
```

An executable JAR can contain a manifest that specifies the main class.

It can then be run with:

Illustration

```
java -jar app.jar
```

33. Typical Project Resource Organization

A small project may look like:

Illustration

```
project/
  src/
    app/
      Main.java

  data/
    students.csv
```

```
config/
  app.properties

output/
  report.txt
```

Separating source code, data, configuration, and generated output improves organization.

Example 19. File-Based Student Repository

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

public class Main {

    static class Student {

        private int id;
        private String name;
        private double gpa;

        public Student(
                int id,
                String name,
                double gpa) {

            this.id = id;
            this.name = name;
            this.gpa = gpa;
        }

        public String toCsv() {

            return id + "," +
                   name + "," +
                   gpa;
        }

        public static Student fromCsv(
                String line) {

            String[] parts =
                    line.split(",");
```

Example 19 (continued)

```
37
38              return new Student(
39                  Integer.parseInt(
40                      parts[0]
41                  ),
42                  parts[1],
43                  Double.parseDouble(
44                      parts[2]
45                  )
46              );
47          }
48
49          @Override
50          public String toString() {
51              return id + " " +
52                     name + " " +
53                     gpa;
54          }
55      }
56
57      static class StudentRepository {
58
59          private Path file;
60
61          public StudentRepository(
62                  Path file) {
63
64              this.file = file;
65          }
66
67          public void save(
68                  List<Student> students)
69                  throws IOException {
70
71              List<String> lines =
72                      new ArrayList<>();
73
74              for (Student s : students) {
75                  lines.add(s.toCsv());
76              }
77
78              Files.write(file, lines);
79          }
80
81          public List<Student> load()
```

Example 19 (continued)

```
82                 throws IOException {
83
84             List<Student> students =
85                     new ArrayList<>();
86
87             if (!Files.exists(file)) {
88                 return students;
89             }
90
91             for (String line :
92                     Files.readAllLines(
93                         file)) {
94
95                 students.add(
96                     Student.fromCsv(
97                         line
98                     )
99                 );
100            }
101
102            return students;
103        }
104    }
105
106    public static void main(String[] args)
107            throws IOException {
108
109        StudentRepository repository =
110                new StudentRepository(
111                    Path.of(
112                        "students.csv"
113                    )
114                );
115
116        List<Student> students =
117                List.of(
118                    new Student(
119                        1,
120                        "Aruzhan",
121                        3.9
122                    ),
123                    new Student(
124                        2,
125                        "Dias",
126                        3.5
```

Example 19 (continued)

```
                    )
                );

        repository.save(students);

        List<Student> loaded =
                repository.load();

        for (Student s : loaded) {
            System.out.println(s);
        }
    }
}
```

Expected output

```
1 Aruzhan 3.9
2 Dias 3.5
```

This example combines:

- classes and objects;
- collections;
- file I/O;
- string parsing;
- exceptions;
- persistence.

34. Common Mistakes

34.1 Mistake 1: Forgetting to Close Resources

Use try-with-resources.

34.2 Mistake 2: Using Byte Streams for Text Without Considering Encoding

For text, character-oriented APIs are usually clearer.

34.3 Mistake 3: Assuming External File Data Is Always Valid

Always validate parsed values.

34.4 Mistake 4: Hard-Coding Absolute Paths

Avoid machine-specific paths such as:

Illustration

```
C:\Users\Name\Desktop\data.txt
```

Prefer relative paths or configurable locations when appropriate.

34.5 Mistake 5: Assuming a File Exists

Check existence or handle the relevant exception.

34.6 Mistake 6: Treating Java Serialization as a Universal Storage Format

Use it carefully and prefer interoperable formats when needed.

35. Design Guidelines

When working with files:

1. use try-with-resources for closeable resources;
2. prefer `Path` and `Files` for modern file operations;
3. separate file-processing logic from business logic;
4. validate external data before using it;
5. avoid hard-coded absolute paths;
6. use meaningful file and directory names;
7. create backups before destructive operations when appropriate;

8. choose text or binary storage according to the use case;
9. use properties for simple configuration;
10. document the expected file format.

36. Summary

In this lecture, we learned that:

- file I/O provides basic persistence;
- byte streams process raw binary data;
- character streams process text;
- buffered I/O improves efficiency and convenience;
- try-with-resources closes resources automatically;
- `File` represents legacy file-system paths;
- `Path` and `Files` provide modern NIO operations;
- files can be created, read, written, copied, moved, and deleted;
- directories can be created and traversed;
- structured text can store application records;
- Java serialization can persist object state in controlled contexts;
- properties files store simple configuration values;
- JAR files package compiled Java applications and resources.

37. Review Questions

1. What is file I/O?
2. What is a stream?
3. What is the difference between input and output?
4. What is a byte stream?
5. What is a character stream?
6. When should byte streams be used?
7. When should character streams be used?
8. What is `FileInputStream`?
9. What is `FileOutputStream`?
10. What is `FileReader`?
11. What is `FileWriter`?
12. What is `BufferedReader`?
13. What is `BufferedWriter`?
14. Why is buffering useful?
15. What is try-with-resources?
16. Which interface must a try-with-resources resource implement?
17. What is the purpose of the `File` class?
18. What is NIO?
19. What is `Path`?
20. What is `Files`?
21. How can an entire file be read as a string?
22. How can all file lines be read into a list?
23. How can a directory be created?
24. How can a file be copied?
25. How can a file be moved?
26. How can a file be deleted?
27. What is CSV-like structured text?
28. What is serialization?
29. What does `Serializable` represent?
30. What is deserialization?
31. What does `transient` mean?
32. What is a properties file?
33. What is a JAR file?
34. How can an executable JAR be run?
35. Why should file-processing logic be separated from business logic?

38. Programming Exercises

Beginner Level

1. Create a text file and write your name into it.
2. Write five student names to a file.
3. Append one additional name to an existing file.
4. Read a text file line by line.
5. Read an entire file using `Files.readString()`.
6. Write text using `Files.writeString()`.
7. Display whether a file exists.
8. Display the absolute path of a file.
9. Create a directory.
10. Copy one text file to another.

Intermediate Level

1. Count the number of lines in a file.
2. Count words in a text file.
3. Count characters in a file.
4. Read numbers from a file and calculate their sum.
5. Read numbers and calculate their average.
6. Parse a simple CSV-like student file.
7. Write student objects into a CSV-like file.
8. Create and load a properties file.
9. List all files inside a directory.
10. Copy all `.txt` files from one directory to another.

Advanced Level

1. Develop a persistent Student Repository using CSV-like storage.
2. Add search, update, and delete operations to the file-based repository.
3. Create a backup utility that copies all files from one folder to another.
4. Recursively count all files in a directory tree.
5. Recursively calculate the total size of files in a directory tree.
6. Search all text files for a user-specified word.
7. Serialize and deserialize a list of simple objects.
8. Create a configurable application using a properties file.
9. Create a log file that records application actions with timestamps.
10. Package a small Java application into a JAR file and run it.

39. Homework

Task: Develop a File-Based Student Management System

Create the following classes:

Illustration

```
Student
StudentRepository
StudentService
FileUtil
Main
```

Requirements:

1. Student data must be stored in a text or CSV-like file.
2. Each student must contain:
   - ID;
   - name;
   - age;
   - GPA.
3. The program must support:
   - adding students;
   - displaying all students;
   - searching by ID;
   - updating student information;
   - deleting a student;
   - calculating average GPA;
   - saving all changes;
   - reloading data when the program starts.
4. Use `Path` and `Files`.
5. Use try-with-resources where appropriate.
6. Validate all external file data.
7. Create a backup copy before destructive updates.
8. Store at least two application settings in a properties file.
9. Create one application log file.

Submission requirements:

- all Java source files;

- sample data file;
- sample properties file;
- sample log file;
- console output;
- explanation of the chosen file format;
- explanation of exception handling;
- explanation of how data persists between program runs.

40. Key Terms

File I/O
Reading data from files and writing data to files.

Stream
A flow of data between a source and a destination.

Byte stream
Stream that processes raw bytes.

Character stream
Stream that processes textual characters.

Buffer
Temporary memory area used to improve I/O efficiency.

Reader
Base abstraction for character input.

Writer
Base abstraction for character output.

`Path`
Modern Java representation of a file-system path.

`Files`
Utility class for file and directory operations.

NIO
Java New I/O API.

Serialization
Converting object state into storable or transferable form.

Deserialization
Restoring object state from serialized data.

`Serializable`
Marker interface used by traditional Java object serialization.

Properties file
Simple key-value configuration file.

JAR
Java Archive used to package classes and resources.