

UNIT 8

I/O and Streams

(2 Hrs.) — *Object Oriented Programming in Java*
Course Code: CACS204 | Year/Semester: II/III

Syllabus Coverage: java.io package; files and directories; byte streams and character streams; console input/output; reading and writing files; the Serialization interface; serialization and deserialization.

UNIT ROADMAP

Understand what I/O means and why Java provides streams.
Work with files and directories using File and related APIs.
Distinguish byte streams from character streams.
Read from and write to the console and files.
Understand object serialization and deserialization.

1. Introduction to I/O

I/O stands for Input/Output. Input is data entering a program, while output is data produced by a program. In Java, I/O operations are mainly organized through classes and interfaces in the java.io package.

Term	Meaning	Examples
Input	Data supplied to a program	Keyboard input, file data
Output	Data produced by a program	Screen output, file output
Stream	A flow of data between a source and a destination	InputStream, Reader
Source	Where data comes from	Keyboard, file
Destination	Where data goes	Screen, file

Example: Input and Output

Keyboard → Java program is an input operation.
Java program → monitor is an output operation.
File → Java program is file input.
Java program → file is file output.

2. java.io Package

The java.io package contains classes and interfaces for handling input and output, files, streams, readers, writers, and object serialization.

Class / Interface	Purpose
InputStream	Abstract base class for byte input
OutputStream	Abstract base class for byte output
FileInputStream	Reads bytes from a file
FileOutputStream	Writes bytes to a file
Reader	Abstract base class for character input
Writer	Abstract base class for character output
FileReader	Reads characters from a file
FileWriter	Writes characters to a file
BufferedReader	Reads text efficiently using buffering
BufferedWriter	Writes text efficiently using buffering
PrintWriter	Convenient text output
File	Represents file and directory path information
ObjectInputStream	Reads serialized objects
ObjectOutputStream	Writes serialized objects

A stream does not necessarily mean a Java collection or a continuous network connection. In I/O, it represents an ordered flow of data that can be read or written.

3. Files and Directories

Java provides the File class in java.io to represent the pathname of a file or directory. It can be used to test existence, inspect properties, create directories, and perform basic file operations.

```
import java.io.File;

File f = new File("notes.txt");

System.out.println(f.exists());
System.out.println(f.getName());
System.out.println(f.getAbsolutePath());
```

Method	Purpose
exists()	Checks whether the file or directory exists
isFile()	Checks whether the path represents a file
isDirectory()	Checks whether the path represents a directory
getName()	Returns the name

Method	Purpose
getPath()	Returns the pathname
getAbsolutePath()	Returns the absolute pathname
length()	Returns file size in bytes
mkdir()	Creates a directory
mkdirs()	Creates required parent directories as well
delete()	Deletes a file or empty directory
list()	Returns names in a directory

Creating and Checking a Directory

```
File dir = new File("data");
if (!dir.exists()) { dir.mkdir(); }
System.out.println(dir.isDirectory());
```

4. Streams

A stream provides a standard way to move data. Java broadly divides streams into byte streams and character streams.

Feature	Byte Stream	Character Stream
Unit of data	8-bit byte	Character
Base classes	InputStream / OutputStream	Reader / Writer
Typical use	Binary data	Text data
Examples	Images, audio, binary files	Text files, characters
Common file classes	FileInputStream / FileOutputStream	FileReader / FileWriter

5. Byte Streams

Byte streams process raw 8-bit data. The main abstract classes are `InputStream` for reading and `OutputStream` for writing. Byte streams are appropriate when the exact bytes must be preserved, such as for binary files.

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

FileInputStream in = new FileInputStream("input.bin");
FileOutputStream out = new FileOutputStream("output.bin");

int data;
while ((data = in.read()) != -1) {
```

```
        out.write(data);
    }

    in.close();
    out.close();
}
```

How the Example Works

FileInputStream opens input.bin for byte input.

read() returns one byte at a time as an int, or -1 at end of stream.

write(data) writes the byte to the output stream.

close() releases the associated resources.

6. Character Streams

Character streams are designed for text and character data. Their base classes are Reader and Writer. FileReader and FileWriter are convenient classes for reading and writing text files.

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

FileReader reader = new FileReader("input.txt");
FileWriter writer = new FileWriter("output.txt");

int ch;
while ((ch = reader.read()) != -1) {
    writer.write(ch);
}

reader.close();
writer.close();
```

For larger text files, buffered classes such as BufferedReader and BufferedWriter can reduce the number of direct I/O operations and provide convenient methods.

7. Reading Console Input

Console input can be read using several approaches. A common modern approach in introductory Java is Scanner, which is in java.util rather than java.io. When specifically working with java.io, InputStreamReader can convert byte input from System.in into character input, and BufferedReader can then read lines.

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

BufferedReader br =
    new BufferedReader(new InputStreamReader(System.in));

System.out.print("Enter your name: ");
String name = br.readLine();

System.out.println("Hello " + name);
```

Scanner Alternative

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
System.out.print("Enter age: ");
int age = sc.nextInt();
```

8. Writing Console Output

Java provides `System.out` for standard console output. `print()` writes without automatically moving to the next line, while `println()` writes and then moves to the next line. `printf()` allows formatted output.

```
System.out.print("Hello ");
System.out.println("Java");
System.out.printf("Age = %d%n", 20);
```

Method	Behavior
<code>print()</code>	Displays the supplied value
<code>println()</code>	Displays the value and moves to a new line
<code>printf()</code>	Displays formatted output

9. Reading and Writing Files

Files can be read and written using byte streams or character streams depending on the type of data. For text, `FileReader/FileWriter` or buffered `Reader/Writer` classes are commonly used.

```
import java.io.*;

public class FileCopy {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(
            new FileReader("input.txt"));
        BufferedWriter bw = new BufferedWriter(
            new FileWriter("output.txt"));

        String line;
        while ((line = br.readLine()) != null) {
            bw.write(line);
            bw.newLine();
        }

        br.close();
        bw.close();
    }
}
```

Buffered Text File Copy

FileReader connects the program to the input text file.

BufferedReader reads complete lines conveniently.

FileWriter connects the program to the output text file.

BufferedWriter writes text efficiently.

The loop continues until readLine() returns null.

10. Exception Handling in I/O

Many I/O operations can cause checked exceptions, especially IOException. Programs must handle the exception with try-catch or declare it using throws.

```
try {
    FileReader fr = new FileReader("data.txt");
    // read data...
    fr.close();
} catch (IOException e) {
    System.out.println("I/O error: " + e.getMessage());
}
```

The try-with-resources statement is a safer modern pattern because resources that implement AutoCloseable are closed automatically.

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

    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

11. Serialization

Serialization is the process of converting an object's state into a byte stream so that the state can be stored or transmitted. In Java, a class normally enables default serialization by implementing the java.io.Serializable marker interface.

```
import java.io.Serializable;

class Student implements Serializable {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Serializable is called a marker interface because it does not require the class to implement methods. It indicates that objects of the class may participate in Java's serialization mechanism.

12. Serialization Process

An object can be serialized with `ObjectOutputStream`. The `writeObject()` method writes the object to the underlying output stream.

```
import java.io.*;

Student s = new Student(101, "Ram");

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

    out.writeObject(s);
}
```

Component	Role
Serializable	Marks the class as serializable
FileOutputStream	Provides byte output to the file
ObjectOutputStream	Converts/writes the object representation
writeObject()	Writes the object

13. Deserialization

Deserialization is the reverse process: the serialized byte stream is read and used to reconstruct the object. `ObjectInputStream` provides `readObject()`.

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

    Student s = (Student) in.readObject();
    System.out.println(s.id);
    System.out.println(s.name);
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

Serialization vs Deserialization

Serialization: Object → byte stream → storage/transmission.

Deserialization: byte stream → object.

Serialization commonly uses `ObjectOutputStream`.

Deserialization commonly uses `ObjectInputStream`.

14. Complete Serialization Example

```

import java.io.*;

class Student implements Serializable {
    private static final long serialVersionUID = 1L;

    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

public class Demo {
    public static void main(String[] args) {
        Student s = new Student(1, "Sita");

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

            out.writeObject(s);

        } catch (IOException e) {
            e.printStackTrace();
        }

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

            Student x = (Student) in.readObject();
            System.out.println(x.id + " " + x.name);

        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

15. Important Exam Points

Question Area	Key Point
java.io	Package containing major Java I/O classes and interfaces
Stream	Flow of data between source and destination
Byte stream	Handles raw byte-oriented data
Character stream	Handles character/text-oriented data
InputStream	Base class for byte input
OutputStream	Base class for byte output
Reader	Base class for character input

Question Area	Key Point
Writer	Base class for character output
File	Represents file/directory path information
Serializable	Marker interface used for object serialization
ObjectOutputStream	Writes objects
ObjectInputStream	Reads objects
Serialization	Object state converted to a byte stream
Deserialization	Byte stream converted back to an object

KEY TAKEAWAYS

Java I/O uses streams to move data between sources and destinations.

Byte streams use InputStream/OutputStream and are suitable for raw binary data.

Character streams use Reader/Writer and are suitable for text.

File represents file and directory path information in java.io.

System.in is standard input and System.out is standard output.

IOException is a common checked exception for I/O operations.

Serializable is a marker interface.

ObjectOutputStream performs object output through writeObject().

ObjectInputStream performs object input through readObject().

Serialization converts object state to a byte stream; deserialization reconstructs it.

16. Quick Revision Questions

- What is I/O? What is meant by a stream?
- What is the purpose of the java.io package?
- Differentiate byte streams and character streams.
- What is the difference between InputStream and Reader?
- What is the use of the File class?
- How can console input be read using BufferedReader?
- What are print(), println(), and printf()?
- What is IOException?
- What is serialization?
- Why is Serializable called a marker interface?
- Differentiate serialization and deserialization.
- What are ObjectOutputStream and ObjectInputStream?

— End of Unit 8 —