

JAVA PROGRAMMING

Chapter 6

STREAMS AND FILE I/O

Detailed Study Notes for Engineering Students

With 15 Multiple Choice Questions (with Answer Key) and 10 Broad Questions

Contents of this Chapter

No.	Section	Topics covered
6.1	Stream Classes	Concept of streams; standard streams; classification of streams; the java.io package; byte stream and character stream class hierarchies; the File class; reading keyboard input; I/O exceptions and closing streams
6.2	Character Stream, Byte Stream	InputStream and OutputStream; FileInputStream and FileOutputStream; buffered and data streams; Reader and Writer; FileReader and FileWriter; BufferedReader and BufferedWriter; InputStreamReader; PrintWriter; comparison of byte and character streams
6.3	Serialization	Serialization and deserialization; the Serializable interface; ObjectOutputStream and ObjectInputStream; transient and static fields; serialVersionUID; inheritance rules; Externalizable; exceptions
	Chapter Practice	15 MCQs with answer key; 10 broad questions

6.1 Stream Classes

Almost every useful program has to communicate with the outside world: it reads data from the keyboard or a file, and it displays results on the screen or stores them in a file. Java handles all such communication through **streams**. A stream is an **ordered sequence of data** that flows between a program and a source or destination of data. The classes that create and manage streams are collected in the package `java.io`, which must be imported with `import java.io.*`; in programs that use them.

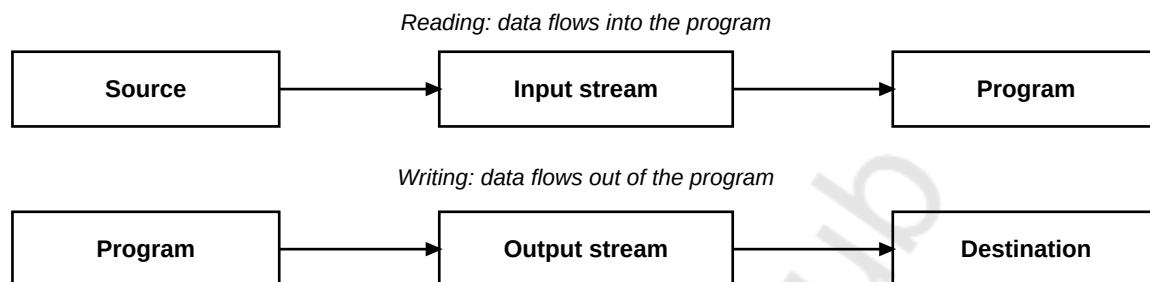


Figure 6.1 Flow of data through input and output streams

6.1.1 Concept of Streams

- A stream is an **abstraction** of an input or output device. The program uses the same methods whether the other end is a disk file, the keyboard, the screen, a network connection or a block of memory.
- An **input stream** carries data *into* a program from a source; an **output stream** carries data *out of* a program to a destination.
- Streams are **sequential**: data is read or written one item after another, in order.
- Streams are **one-directional**: a single stream is either for input or for output, not both.
- Reading from a stream waits (blocks) until data is available, and data is moved by the Java Virtual Machine with the help of the operating system.

Sources and destinations of data

Source of input	Destination of output
Keyboard (System.in)	Screen (System.out, System.err)
Disk file	Disk file
Network socket	Network socket
Array of bytes in memory	Array of bytes in memory
Another program (through a pipe)	Another program (through a pipe)

6.1.2 Standard Streams

Java automatically creates three streams, which are public static members of the class `System`:

Stream	Type	Purpose
<code>System.in</code>	<code>InputStream</code>	Standard input - normally the keyboard
<code>System.out</code>	<code>PrintStream</code>	Standard output - normally the screen

Stream	Type	Purpose
System.err	PrintStream	Standard error output - normally the screen; used for error messages

6.1.3 Classification of Streams

Streams can be classified in two ways. According to the **type of data** handled, they are **byte streams** and **character streams**. According to the **direction** of the flow, they are **input streams** and **output streams**. Combining the two gives four abstract base classes from which all other stream classes are derived.

	Input (reading)	Output (writing)
Byte stream (8-bit data)	InputStream	OutputStream
Character stream (16-bit Unicode)	Reader	Writer

- **Byte streams** read and write raw 8-bit bytes. They are used for binary data such as images, audio, video and executable files.
- **Character streams** read and write 16-bit Unicode characters and automatically convert between the external character encoding and Java's internal Unicode. They are used for text.

Node streams and filter (processing) streams

- **Node streams** connect directly to a data source or destination, for example `FileInputStream` and `FileReader`.
- **Filter (processing) streams** do not connect to a source themselves; they **wrap another stream** and add features such as buffering or conversion of data types, for example `BufferedInputStream` and `DataInputStream`. Wrapping streams one inside another is called **chaining**.

```
FileInputStream fis = new FileInputStream("data.bin");           // node stream
BufferedInputStream bis = new BufferedInputStream(fis);         // filter stream
DataInputStream dis = new DataInputStream(bis);                 // another filter
```

6.1.4 The java.io Package

Class / interface	Purpose
InputStream, OutputStream	Abstract base classes of all byte streams
Reader, Writer	Abstract base classes of all character streams
File	Represents the name and properties of a file or directory (not its contents)
FileInputStream, FileOutputStream	Read and write bytes of a file
FileReader, FileWriter	Read and write characters of a text file
BufferedReader, BufferedWriter	Buffered character I/O; read and write whole lines
DataInputStream, DataOutputStream	Read and write primitive data types in binary form
ObjectInputStream, ObjectOutputStream	Read and write whole objects (serialization)
PrintStream, PrintWriter	Formatted output with print(), println(), printf()
RandomAccessFile	Read and write a file at any position (random access)
Serializable	Marker interface for objects that can be serialized

6.1.5 Byte Stream Classes

All byte stream classes are descended from InputStream or OutputStream.

InputStream	OutputStream
+-+ FileInputStream	+-+ FileOutputStream
+-+ ByteArrayInputStream	+-+ ByteArrayOutputStream
+-+ ObjectInputStream	+-+ ObjectOutputStream
+-+ PipedInputStream	+-+ PipedOutputStream
+-+ SequenceInputStream	+-+ FilterOutputStream
+-+ FilterInputStream	+-+ BufferedOutputStream
+-+ BufferedInputStream	+-+ DataOutputStream
+-+ DataInputStream	+-+ PrintStream
+-+ PushbackInputStream	

Class	Description
FileInputStream / FileOutputStream	Read bytes from / write bytes to a file
ByteArrayInputStream / ByteArrayOutputStream	Use an array of bytes in memory as the source / destination
BufferedInputStream / BufferedOutputStream	Add a memory buffer so that data is moved in large blocks; faster I/O
DataInputStream / DataOutputStream	Read / write int, double, boolean, char and String values in a machine-independent binary form
ObjectInputStream / ObjectOutputStream	Read / write complete objects (see 6.3)
PrintStream	Convenient printing of values; System.out is a PrintStream
SequenceInputStream	Joins two or more input streams so that they are read one after the other

Class	Description
PipedInputStream / PipedOutputStream	Pass data between two threads

6.1.6 Character Stream Classes

All character stream classes are descended from Reader or Writer.

Reader +-- InputStreamReader +-- FileReader +-- BufferedReader +-- CharArrayReader +-- StringReader +-- PipedReader +-- FilterReader	Writer +-- OutputStreamWriter +-- FileWriter +-- BufferedWriter +-- CharArrayWriter +-- StringWriter +-- PrintWriter +-- PipedWriter +-- FilterWriter
---	---

Class	Description
FileReader / FileWriter	Read characters from / write characters to a text file
BufferedReader / BufferedWriter	Buffer characters; BufferedReader provides readLine() and BufferedWriter provides newLine()
InputStreamReader / OutputStreamWriter	Bridge from a byte stream to a character stream (and back), with a chosen character encoding
PrintWriter	Prints values as text using print(), println() and printf()
CharArrayReader / CharArrayWriter	Use a char array in memory as source / destination
StringReader / StringWriter	Use a String (or a StringBuffer) as source / destination

6.1.7 The File Class

The File class does not read or write the contents of a file. It represents the **pathname** of a file or directory and provides methods to find its properties and to create, rename or delete it.

Constructors

File(String pathname) File(String directory, String filename) File(File directory, String filename)

Commonly used methods

Method	Purpose
<code>boolean exists()</code>	Tests whether the file or directory exists
<code>boolean isFile() / isDirectory()</code>	Tests whether the path is a normal file / a directory
<code>String getName()</code>	Returns the name of the file or directory
<code>String getPath() / getAbsolutePath()</code>	Returns the path as given / the full path
<code>String getParent()</code>	Returns the parent directory
<code>long length()</code>	Returns the size of the file in bytes
<code>boolean canRead() / canWrite()</code>	Tests read / write permission
<code>boolean createNewFile()</code>	Creates a new empty file; returns false if it already exists
<code>boolean delete()</code>	Deletes the file or an empty directory
<code>boolean mkdir() / mkdirs()</code>	Creates a directory / a directory with all missing parents
<code>boolean renameTo(File dest)</code>	Renames the file
<code>String[] list() / File[] listFiles()</code>	Lists the contents of a directory (returns null if the path is not a directory)

Example 1 - Information about a file

```
import java.io.File;

public class FileInfo {
    public static void main(String[] args) {
        File f = new File("data.txt");
        System.out.println("Name      : " + f.getName());
        System.out.println("Path      : " + f.getAbsolutePath());
        System.out.println("Exists   : " + f.exists());
        System.out.println("Is file  : " + f.isFile());
        System.out.println("Size     : " + f.length() + " bytes");
        System.out.println("Readable : " + f.canRead());
        System.out.println("Writable : " + f.canWrite());
    }
}
```

Example 2 - Listing the contents of a directory

```
import java.io.File;

public class ListDir {
    public static void main(String[] args) {
        File dir = new File(".");           // current directory
        String[] names = dir.list();
        if (names != null) {
            for (String n : names) {
                File f = new File(dir, n);
                System.out.println((f.isDirectory() ? "[DIR] " : "[FILE] ") + n);
            }
        }
    }
}
```

6.1.8 Reading Keyboard Input

The keyboard is connected to the stream `System.in`, which is a byte stream. To read a line of text conveniently, it is chained with an `InputStreamReader` (bytes to characters) and a `BufferedReader` (provides `readLine()`):

```
import java.io.*;

public class ConsoleRead {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter your name: ");
        String name = br.readLine();
        System.out.println("Hello, " + name);
    }
}
```

The `Scanner` class of `java.util` is another popular way to read keyboard input.

6.1.9 I/O Exceptions and Closing Streams

Input and output operations can fail, for example when a file does not exist or the disk is full. Therefore most stream methods throw **checked exceptions** that must be either caught with try-catch or declared with throws.

Exception	Meaning
<code>IOException</code>	General I/O failure; the parent of most I/O exceptions
<code>FileNotFoundException</code>	The file cannot be found or opened (a subclass of <code>IOException</code>)
<code>EOFException</code>	End of the stream was reached unexpectedly while reading
<code>NotSerializableException</code>	An object that is not <code>Serializable</code> was written (see 6.3)

A stream should always be **closed** with `close()` after use, so that the buffered data is written and the system resources are released. The **try-with-resources** statement (Java 7 onwards) closes the streams declared in the parentheses automatically, even if an exception occurs:

```
try (FileInputStream fis = new FileInputStream("data.txt")) {
    // use the stream
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
}
// fis is closed automatically here
```

Quick revision - 6.1 Stream Classes

- A stream is a sequence of data flowing between a program and a source or destination; classes are in `java.io`.
- Four base classes: `InputStream`, `OutputStream` (bytes) and `Reader`, `Writer` (characters).
- `System.in`, `System.out` and `System.err` are the standard streams.
- Filter streams wrap other streams (chaining) to add buffering or data-type conversion.
- `File` describes a path and its properties; it does not read or write contents.
- Close streams after use, preferably with try-with-resources.

6.2 Character Stream, Byte Stream

This section explains how to read and write data using the two families of streams. **Byte streams** work with raw bytes and can handle any kind of file. **Character streams** work with text and take care of the conversion between bytes and Unicode characters.

6.2.1 Byte Streams

Methods of InputStream

Method	Description
<code>int read()</code>	Reads one byte (0 to 255) and returns it as an int; returns -1 at the end of the stream
<code>int read(byte[] b)</code>	Reads up to <code>b.length</code> bytes into the array; returns the number of bytes read or -1 at the end
<code>int read(byte[] b, int off, int len)</code>	Reads up to <code>len</code> bytes into the array starting at index <code>off</code>
<code>int available()</code>	Returns the number of bytes that can be read without blocking
<code>long skip(long n)</code>	Skips over <code>n</code> bytes of input
<code>void mark(int limit) / reset()</code>	Marks the current position and later returns to it (if supported)
<code>void close()</code>	Closes the stream and releases its resources

Methods of OutputStream

Method	Description
<code>void write(int b)</code>	Writes one byte (the low-order 8 bits of <code>b</code>)
<code>void write(byte[] b)</code>	Writes all bytes of the array
<code>void write(byte[] b, int off, int len)</code>	Writes <code>len</code> bytes of the array starting at index <code>off</code>
<code>void flush()</code>	Forces any buffered output bytes to be written
<code>void close()</code>	Flushes and closes the stream

FileInputStream and FileOutputStream

Constructor	Meaning
<code>FileInputStream(String name)</code>	Opens the named file for reading; throws <code>FileNotFoundException</code> if it does not exist
<code>FileInputStream(File f)</code>	Opens the file represented by a <code>File</code> object
<code>FileOutputStream(String name)</code>	Opens the file for writing; creates it if absent and overwrites it if present
<code>FileOutputStream(String name, boolean append)</code>	If <code>append</code> is true, new data is added at the end of the file

Example 3 - Writing bytes to a file

```
import java.io.*;

public class ByteWrite {
    public static void main(String[] args) {
        String text = "Java Streams";
        try (FileOutputStream fos = new FileOutputStream("out.txt")) {
            fos.write(text.getBytes());
            System.out.println("Data written");
        } catch (IOException e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Example 4 - Reading bytes from a file

```
import java.io.*;

public class ByteRead {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("out.txt")) {
            int b;
            while ((b = fis.read()) != -1) {
                System.out.print((char) b);
            }
        } catch (IOException e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Output: Java Streams

Example 5 - Copying a file using a byte buffer (works for text, images and any other file)

```
import java.io.*;

public class FileCopy {
    public static void main(String[] args) {
        try (FileInputStream in = new FileInputStream("source.jpg");
            FileOutputStream out = new FileOutputStream("copy.jpg")) {
            byte[] buffer = new byte[1024];
            int n;
            while ((n = in.read(buffer)) != -1) {
                out.write(buffer, 0, n);
            }
            System.out.println("File copied");
        } catch (IOException e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Reading a block of bytes at a time is much faster than reading one byte per call. Wrapping the streams in `BufferedInputStream` and `BufferedOutputStream` gives the same advantage automatically:

```
BufferedInputStream in = new BufferedInputStream(new FileInputStream("a.dat"));
BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("b.dat"));
```

DataInputStream and DataOutputStream

These filter streams read and write the primitive data types in binary form. Data must be read in **exactly the same order** in which it was written.

DataOutputStream	DataInputStream
writeInt(int)	int readInt()
writeDouble(double)	double readDouble()
writeBoolean(boolean)	boolean readBoolean()
writeChar(int)	char readChar()
writeUTF(String)	String readUTF()

Example 6 - Writing and reading primitive data

```
import java.io.*;

public class DataStreamDemo {
    public static void main(String[] args) throws IOException {
        DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin"));
        dos.writeInt(101);
        dos.writeDouble(78.5);
        dos.writeUTF("Ravi");
        dos.close();

        DataInputStream dis = new DataInputStream(new FileInputStream("data.bin"));
        int id = dis.readInt();           // same order as written
        double marks = dis.readDouble();
        String name = dis.readUTF();
        dis.close();

        System.out.println(id + " " + marks + " " + name);
    }
}
```

Output: 101 78.5 Ravi

6.2.2 Character Streams

Methods of Reader

Method	Description
int read()	Reads one character and returns it as an int; returns -1 at the end of the stream
int read(char[] cbuf)	Reads characters into an array; returns the number read or -1
boolean ready()	Tells whether the stream is ready to be read
void close()	Closes the stream

Methods of Writer

Method	Description
void write(int c)	Writes a single character
void write(String s)	Writes a string
void write(char[] cbuf)	Writes an array of characters
Writer append(CharSequence cs)	Appends a character sequence to the stream
void flush()/close()	Forces buffered output to be written / closes the stream

FileReader and FileWriter

- new FileReader("notes.txt") opens a text file for reading; it throws FileNotFoundException if the file is missing.
- new FileWriter("notes.txt") creates or **overwrites** the file.
- new FileWriter("notes.txt", true) opens the file in **append** mode, so existing text is kept.

Example 7 - Writing text to a file

```
import java.io.*;

public class CharWrite {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("notes.txt")) {
            fw.write("Streams in Java\n");
            fw.write("Character streams handle text.\n");
            System.out.println("Text written");
        } catch (IOException e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Example 8 - Reading text from a file

```
import java.io.*;

public class CharRead {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("notes.txt")) {
            int ch;
            while ((ch = fr.read()) != -1) {
                System.out.print((char) ch);
            }
        } catch (IOException e) {
            System.out.println("Error: " + e);
        }
    }
}
```

BufferedReader and BufferedWriter

A **BufferedReader** collects characters in a buffer and provides String readLine(), which returns one line of text without the line terminator and returns **null** at the end of the stream. A **BufferedWriter** buffers output and provides newLine() to write a platform-independent line separator. They reduce the number of actual disk accesses and so make text I/O faster.

Example 9 - Writing lines and reading them back with line numbers

```
import java.io.*;

public class LineIO {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("lines.txt"))) {
            bw.write("First line");
            bw.newLine();
            bw.write("Second line");
            bw.newLine();
        } catch (IOException e) {
            System.out.println("Error: " + e);
        }
    }
}
```

```

        try (BufferedReader br = new BufferedReader(new FileReader("lines.txt"))) {
            String line;
            int count = 0;
            while ((line = br.readLine()) != null) {
                count++;
                System.out.println(count + ": " + line);
            }
        } catch (IOException e) {
            System.out.println("Error: " + e);
        }
    }
}

```

Output:

```

1: First line
2: Second line

```

InputStreamReader, OutputStreamWriter and PrintWriter

- **InputStreamReader** converts a byte input stream into a character stream. A character encoding may be given, for example `new InputStreamReader(System.in, "UTF-8")`. **OutputStreamWriter** does the reverse for output.
- **PrintWriter** writes values as text using `print()`, `println()` and `printf()`, exactly as `System.out` does, but to any destination.

```

PrintWriter pw = new PrintWriter(new FileWriter("report.txt"));
pw.println("Student report");
pw.printf("Marks = %d\n", 85);
pw.close();

```

Example 10 - Counting lines, words and characters of a text file

```

import java.io.*;

public class WordCount {
    public static void main(String[] args) {
        int lines = 0, words = 0, chars = 0;
        try (BufferedReader br = new BufferedReader(new FileReader("notes.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                lines++;
                chars += line.length();
                String t = line.trim();
                if (!t.isEmpty()) {
                    words += t.split("\\s+").length;
                }
            }
        } catch (IOException e) {
            System.out.println("Error: " + e);
            return;
        }
        System.out.println("Lines      = " + lines);
        System.out.println("Words      = " + words);
        System.out.println("Characters = " + chars);
    }
}

```

For the two lines written in Example 7 the program prints Lines = 2, Words = 7 and Characters = 45 (the line terminators are not counted by `readLine()`).

6.2.3 Byte Streams versus Character Streams

Point	Byte streams	Character streams
Base classes	InputStream, OutputStream	Reader, Writer
Unit of data	8-bit byte	16-bit Unicode character
Suitable for	Binary files: images, audio, video, class files	Text files
Main file classes	FileInputStream, FileOutputStream	FileReader, FileWriter
Conversion of encoding	None; bytes are passed as they are	Automatic conversion between bytes and Unicode
Line-oriented methods	Not provided (except in print streams)	readLine(), newLine(), println()
Typical filter classes	BufferedInputStream, DataInputStream	BufferedReader, PrintWriter
Bridge between them	InputStreamReader (byte to character) and OutputStreamWriter (character to byte)	

Which stream should be used?

- Use **character streams** for text that people read - source code, notes, CSV and log files.
- Use **byte streams** for everything else - images, sound, PDF files, compressed files - and for copying any file exactly.
- Use **buffered** versions for better speed, and **data** streams when primitive values must be stored in binary form.

Common mistakes in file I/O

- Forgetting to close or flush a stream, so that part of the output never reaches the file.
- Opening a file with new `FileWriter("a.txt")` and unintentionally erasing its old contents (use append mode).
- Reading data with `DataInputStream` in a different order from the order in which it was written.
- Using a relative path and not knowing the current directory of the program.

Quick revision - 6.2 Character Stream, Byte Stream

- `read()` returns -1 at the end of a stream; `readLine()` returns null.
- `FileOutputStream(name, true)` and `FileWriter(name, true)` append to a file.
- Buffered streams improve speed; `BufferedReader` gives `readLine()`.
- `DataOutputStream` and `DataInputStream` store primitives in binary form in the same order.
- `InputStreamReader` bridges a byte stream to a character stream.

6.3 Serialization

Normally the data of a program disappears when the program ends. **Serialization** is the process of converting the **state of an object** into a **stream of bytes** so that it can be stored in a file or sent across a network. The reverse process, which rebuilds the object from the stream of bytes, is called **deserialization**. The object that is rebuilt has the same state as the object that was saved.

Term	Meaning
Serialization	Writing an object (its state) to an output stream as bytes
Deserialization	Reading the bytes from an input stream and recreating the object
Persistence	Keeping an object available after the program that created it has ended

Uses of serialization

- **Persistence** - saving objects such as student records or game state to a file.
- **Communication** - sending objects between computers in network applications such as Remote Method Invocation (RMI).
- **Deep copying** - creating an exact, independent copy of an object and everything it refers to.

6.3.1 The Serializable Interface

Only the objects of classes that implement the interface `java.io.Serializable` can be serialized. This interface is a **marker interface**: it contains no methods and only tells the Java Virtual Machine that the objects of the class may be written to a stream.

```
public class Student implements Serializable {  
    ...  
}
```

- If a subclass inherits from a serializable class, the subclass is automatically serializable.
- All the **objects referred to** by the fields (for example an Address object inside a Student) must also be serializable; otherwise `NotSerializableException` is thrown.
- Most standard classes such as `String`, the wrapper classes, arrays and the collection classes (`ArrayList`, `HashMap`) are serializable.

6.3.2 ObjectOutputStream and ObjectInputStream

Class	Important methods
<code>ObjectOutputStream</code>	<code>ObjectOutputStream(OutputStream out)</code> - constructor wraps another output stream; <code>void writeObject(Object obj)</code> - writes an object; also <code>writeInt()</code> , <code>writeUTF()</code> etc.
<code>ObjectInputStream</code>	<code>ObjectInputStream(InputStream in)</code> - constructor wraps another input stream; <code>Object readObject()</code> - reads an object (must be cast to the proper type; throws <code>ClassNotFoundException</code>)

Steps to serialize an object

1. Make the class implement `Serializable`.

2. Create a `FileOutputStream` for the destination file.
3. Wrap it in an `ObjectOutputStream`.
4. Call `writeObject()` with the object, then close the stream.

Steps to deserialize an object

1. Create a `FileInputStream` for the file that holds the saved object.
2. Wrap it in an `ObjectInputStream`.
3. Call `readObject()` and cast the returned `Object` to the correct class.
4. Close the stream.

Example 11 - Serializing and deserializing a Student object

File `Student.java`:

```
import java.io.Serializable;

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

    int rollNo;
    String name;
    double marks;
    transient String password;           // transient: will NOT be saved

    Student(int rollNo, String name, double marks, String password) {
        this.rollNo = rollNo;
        this.name = name;
        this.marks = marks;
        this.password = password;
    }
}
```

File `WriteStudent.java` (serialization):

```
import java.io.*;

public class WriteStudent {
    public static void main(String[] args) {
        Student s = new Student(101, "Ravi", 78.5, "secret123");
        try (ObjectOutputStream oos =
            new ObjectOutputStream(new FileOutputStream("student.ser"))) {
            oos.writeObject(s);
            System.out.println("Object serialized");
        } catch (IOException e) {
            System.out.println("Error: " + e);
        }
    }
}
```

File `ReadStudent.java` (deserialization):

```
import java.io.*;

public class ReadStudent {
    public static void main(String[] args) {
        try (ObjectInputStream ois =
            new ObjectInputStream(new FileInputStream("student.ser"))) {
            Student s = (Student) ois.readObject();
            System.out.println(s.rollNo + " " + s.name + " " + s.marks
                + " " + s.password);
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Error: " + e);
        }
    }
}
```

```

    }
}
}

```

Output: First run: **Object serialized**. Second run: **101 Ravi 78.5 null**. The password is null because a transient field is not saved and is given its default value when the object is rebuilt.

Serializing a collection - an ArrayList of serializable objects can be written with a single call:

```

ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Streams");
oos.writeObject(list); // while writing

ArrayList<String> copy = (ArrayList<String>) ois.readObject(); // while reading

```

6.3.3 The transient and static Keywords

- A field declared **transient** is skipped during serialization. After deserialization it contains the default value of its type: 0 for numbers, false for boolean and null for references. Use it for sensitive data such as passwords, or for values that can be recomputed.
- **static** fields belong to the class, not to an individual object, so they are **not serialized**.

Member of the object	Saved by serialization?
Non-static, non-transient instance variables (primitive or serializable objects)	Yes
transient instance variables	No - default value after deserialization
static variables	No
Methods and constructors	No - only the state (data) is stored
Fields of a non-serializable superclass	No - the superclass no-argument constructor runs instead

6.3.4 serialVersionUID

Every serializable class has a version number called `serialVersionUID`. It is written to the stream along with the object. During deserialization the Java Virtual Machine compares the number stored in the stream with the number of the class that is loaded. If they differ, `InvalidClassException` is thrown, because the class has changed since the object was saved. If the programmer does not declare it, the system computes it from the class details, so even a small change to the class changes the number. It is therefore good practice to declare it explicitly:

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

6.3.5 Inheritance and Object Graphs

- If a superclass implements `Serializable`, all its subclasses are serializable and the superclass fields are saved.
- If the superclass is **not** serializable, its fields are not saved. During deserialization its **no-argument constructor** is executed to initialise them, so it must exist and be accessible, otherwise an exception occurs.
- When an object refers to other objects, the whole **object graph** is saved. If two fields refer to the same object, that object is written only once and the sharing is preserved after deserialization.

6.3.6 Externalizable and Custom Serialization

The interface `java.io.Externalizable` gives the programmer complete control over what is written. The class must implement the two methods `writeExternal(ObjectOutput out)` and `readExternal(ObjectInput in)`, and it must have a **public no-argument constructor**. A `Serializable` class can also customise the process by defining the private methods `writeObject()` and `readObject()`.

Point	Serializable	Externalizable
Type of interface	Marker interface (no methods)	Has methods <code>writeExternal()</code> and <code>readExternal()</code>
Control	Automatic; the JVM decides what is saved	Programmer decides exactly what is saved
Constructor needed	Not required for the class itself	Public no-argument constructor is required
Speed and size	Slower, larger output	Can be faster and smaller when written carefully
Use	Common, simple cases	Special cases needing custom format

6.3.7 Exceptions in Serialization

Exception	Cause
<code>NotSerializableException</code>	The object (or an object it refers to) does not implement <code>Serializable</code>
<code>InvalidClassException</code>	The <code>serialVersionUID</code> differs, or the class cannot be rebuilt (for example the superclass has no no-argument constructor)
<code>ClassNotFoundException</code>	The class of the object read from the stream cannot be found
<code>IOException</code>	A general input or output error occurred
<code>StreamCorruptedException</code>	The data in the stream is not valid serialized data

Advantages	Disadvantages
Saves a whole object, with the objects it refers to, using very little code	The saved format is specific to Java; other languages cannot easily read it
Objects can be sent over a network	Changes in the class can make old saved files unreadable
Preserves the structure and sharing of objects	Slower and larger than simple text formats

Security note

- Never deserialize data that comes from an untrusted source: a crafted stream can cause harm when it is rebuilt. Applications should validate input, for example with an `ObjectInputFilter`, or use a safer data format such as JSON for untrusted data.

Quick revision - 6.3 Serialization

- Serialization = object to byte stream; deserialization = byte stream back to object.
- The class must implement `Serializable` (a marker interface); use `writeObject()` and `readObject()`.
- transient and static fields are not saved.
- serialVersionUID mismatch causes `InvalidClassException`.
- All objects referred to must be serializable, otherwise `NotSerializableException`.
- `readObject()` returns `Object` and must be cast; it throws `ClassNotFoundException`.

Poly Notes Hub

Chapter Practice

A. Multiple Choice Questions (15)

Choose the correct option for each question. The answer key is given at the end of this section.

Q1. Which package contains the stream classes of Java?

- (a) java.lang
- (b) java.util
- (c) java.io
- (d) java.net

Q2. Which is the abstract base class of all byte input streams?

- (a) InputStream
- (b) Reader
- (c) FileReader
- (d) Scanner

Q3. Which of the following is a character stream class?

- (a) FileInputStream
- (b) DataOutputStream
- (c) BufferedInputStream
- (d) FileReader

Q4. What does the read() method of a stream return at the end of the stream?

- (a) 0
- (b) -1
- (c) null
- (d) 255

Q5. Which class provides the method readLine() for reading a line of text?

- (a) FileInputStream
- (b) DataInputStream
- (c) BufferedReader
- (d) PrintStream

Q6. System.out is an object of which class?

- (a) PrintStream
- (b) OutputStream
- (c) PrintWriter
- (d) Writer

Q7. Which class converts a byte input stream into a character input stream?

- (a) BufferedReader
- (b) PrintWriter
- (c) DataInputStream
- (d) InputStreamReader

- Q8.** Which statement opens the file "log.txt" so that new text is added to its end?
- (a) new FileWriter("log.txt")
 - (b) new FileWriter("log.txt", true)
 - (c) new FileWriter("log.txt", false)
 - (d) new FileWriter(true)
- Q9.** The try-with-resources statement automatically:
- (a) catches all exceptions
 - (b) ignores errors
 - (c) closes the declared resources
 - (d) creates missing files
- Q10.** Which method of the File class tests whether a file or directory exists?
- (a) exists()
 - (b) find()
 - (c) present()
 - (d) hasFile()
- Q11.** Converting the state of an object into a byte stream is called:
- (a) Compilation
 - (b) Parsing
 - (c) Deserialization
 - (d) Serialization
- Q12.** Which interface must a class implement so that its objects can be serialized?
- (a) Cloneable
 - (b) Serializable
 - (c) Runnable
 - (d) Comparable
- Q13.** Which keyword prevents an instance variable from being serialized?
- (a) final
 - (b) volatile
 - (c) transient
 - (d) private
- Q14.** Which method of ObjectOutputStream writes an object to the stream?
- (a) writeObject()
 - (b) saveObject()
 - (c) putObject()
 - (d) serialize()

Q15. Which exception is thrown when an object whose class does not implement Serializable is written?

- (a) ClassCastException
- (b) FileNotFoundException
- (c) InvalidClassException
- (d) NotSerializableException

Answer Key

Q. No.	Correct answer	Reason
1	(c) java.io	The stream classes are in the java.io package.
2	(a) InputStream	InputStream is the root of the byte input stream hierarchy.
3	(d) FileReader	FileReader is a subclass of Reader; the others are byte streams.
4	(b) -1	read() returns -1 when no more data is available.
5	(c) BufferedReader	BufferedReader.readLine() returns a line, or null at the end.
6	(a) PrintStream	System.out and System.err are PrintStream objects.
7	(d) InputStreamReader	InputStreamReader is the bridge from bytes to characters.
8	(b) new FileWriter("log.txt", true)	The second argument true selects append mode.
9	(c) closes the declared resources	Resources declared in the parentheses are closed automatically.
10	(a) exists()	File.exists() returns true if the path exists.
11	(d) Serialization	Serialization writes the object state as bytes.
12	(b) Serializable	Serializable is the marker interface for serialization.
13	(c) transient	transient fields are skipped and get default values on deserialization.
14	(a) writeObject()	writeObject(Object) serializes the object.
15	(d) NotSerializableException	NotSerializableException signals a non-serializable object.

B. Broad Questions (10)

- Q1.** What is a stream? Explain the classification of streams in Java and describe the standard streams System.in, System.out and System.err.
- Q2.** Explain the hierarchy of the byte stream classes. Describe the purpose of FileInputStream, BufferedInputStream, DataInputStream, ByteArrayInputStream and PrintStream.
- Q3.** Explain the File class with its constructors and any eight methods. Write a program that displays the name, path, size and permissions of a file and lists the contents of a directory.
- Q4.** Describe the important methods of InputStream and OutputStream. Write a program to copy the contents of one file to another using byte streams.
- Q5.** Differentiate between byte streams and character streams. Explain the Reader and Writer classes and the role of InputStreamReader and OutputStreamWriter.
- Q6.** Write a program that writes several lines of text to a file using FileWriter (or BufferedWriter) and then reads the file back using BufferedReader, displaying a line number before each line.
- Q7.** Explain DataInputStream and DataOutputStream. Write a program that stores an integer, a double and a string in a file and reads them back.
- Q8.** What is serialization? Explain the steps of serialization and deserialization and write a program that serializes and deserializes an object of a Student class.
- Q9.** Explain the use of the transient and static keywords, serialVersionUID and the effect of inheritance in serialization. Differentiate between Serializable and Externalizable.
- Q10.** Explain how exceptions are handled in file I/O and the use of the try-with-resources statement. Write a program that counts the number of lines, words and characters in a text file.