Chapter 8: Input and Output

Object Oriented Programming with Java

Streams | Text and binary files | Character encoding | File and Scanner | Reader / Writer | Buffering | Copying files | Random access | Serialization | NIO: buffers, channels, selectors

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_08_Input_and_Output.html

Learning Outcomes

Explain streams

Describe input/output streams and `System.in`, `System.out`, `System.err`; read the keyboard with `Scanner`.

Tell text from binary

Distinguish text and binary files, the ASCII, ISO-8859-9 and UTF-8 encodings, and work out how many bytes a number takes in a file.

Read and write files

Process text files with `File`, `Scanner`, `BufferedReader`, `PrintWriter`, `FileWriter`; close resources with `try`-with-resources.

Pick the right stream

Tell byte streams (`InputStream` / `OutputStream`) from character streams (`Reader` / `Writer`), and buffered from wrapper streams.

Copy and seek

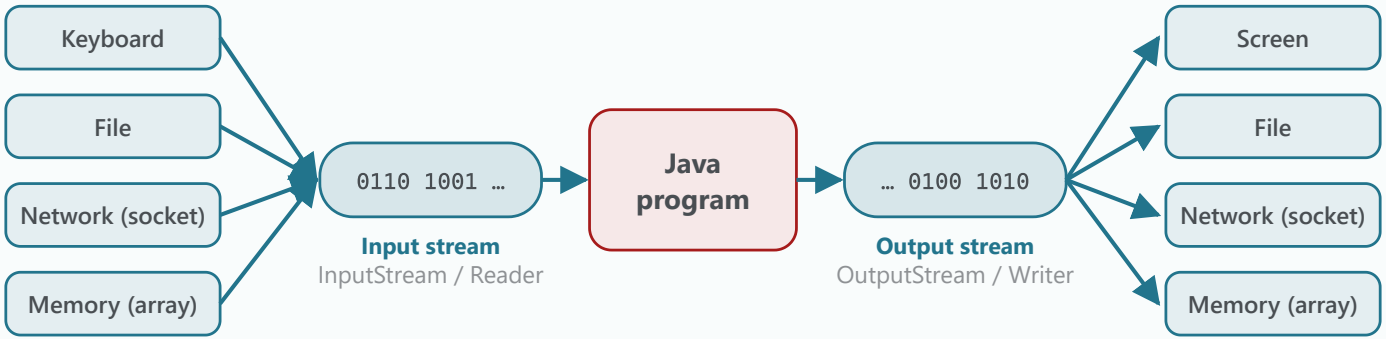
Copy byte by byte, by block and by line; jump to any position with `RandomAccessFile`.

Store objects; use NIO

Serialize objects (`Serializable`, `transient`); explain what NIO buffers, channels and selectors are for.

Streams

- A **stream** carries data from a source to a destination **in order**, piece by piece. The data may be characters, numbers or raw **bytes**.
- **Input stream**: data flows **into** the program. Example: `System.in` (keyboard).
- **Output stream**: data flows **out of** the program. Example: `System.out` (screen).
- Think of a water pipe: water leaves the far end **in the order** it went in, and the program doesn't care whether a tap or a tank sits at the other end.

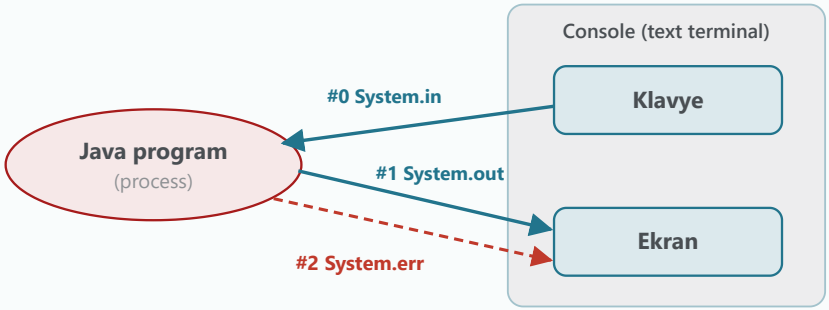


Keyboard, file or network: the program uses the same `read()` / `write()` methods. That shared interface is the strength of Java's I/O library.

Keyboard and Screen: Standard Streams

- **Transient data:** keyboard and screen data is not stored; it is gone when the program ends.
- **Live flow:** input arrives as the user types; output appears as the program writes.
- Every Java program starts with three ready-made streams:

Stream	Java object	Type
standard input	`System.in`	`InputStream`
standard output	`System.out`	`PrintStream`
standard error	`System.err`	`PrintStream`



`System.err` also writes to the screen, but on a separate channel: even when output is redirected to a file (`java App > out.txt`), error messages stay on screen.

First Example: Reading and Writing

```java
import java.util.Scanner;

public class StreamDemo {
  public static void main(String[] args) {
    // Read from the input stream
    Scanner scanner = new Scanner(System.in);
    int userInput = scanner.nextInt();
    // Write to the output stream
    System.out.println("Hello, World!");
    System.out.println("You entered: " + userInput);
    System.err.println("This line went to stderr.");
  }
}
```

CONSOLE (INPUT: 42)

```
42
Hello, World!
You entered: 42
This line went to stderr.
```

- `new Scanner(System.in)` creates a **scanner** that reads the keyboard stream. `Scanner` lives in `java.util`, so the `import` is required.
- `nextInt()` **blocks** until the user types a number and presses Enter.
- `System.out.println` writes a line to the output stream.
- `System.err.println` writes to the error stream; most IDEs show it in **red**.
- If the user types `abc` instead of a number, an `InputMismatchException` is thrown.

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
▶      Scanner keyboard = new Scanner(System.in);
6      System.out.print("Name: ");
7      String name = keyboard.nextLine();
8      System.out.print("Age: ");
9      int age = keyboard.nextInt();
10     int nextYear = age + 1;
11     System.out.println("Hello " + name + "!");
12     System.out.println("Next year you'll be " + nextYear +
   ".");
13     keyboard.close();
14   }
15 }
```

System.in buffer



next

The `Scanner` is attached to `System.in`. The keyboard buffer is empty.

Step 1 / 9

OUTPUT

VARIABLES

keyboard =	Scanner(System.in)

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
5      Scanner keyboard = new Scanner(System.in);
▶      System.out.print("Name: ");
7      String name = keyboard.nextLine();
8      System.out.print("Age: ");
9      int age = keyboard.nextInt();
10     int nextYear = age + 1;
11     System.out.println("Hello " + name + "!");
12     System.out.println("Next year you'll be " + nextYear +
   ".");
13     keyboard.close();
14   }
15 }
```

System.in buffer



next

`print` (not println) shows the prompt, so the user answers **on the same line**.

Step 2 / 9

OUTPUT

```
Name:
```

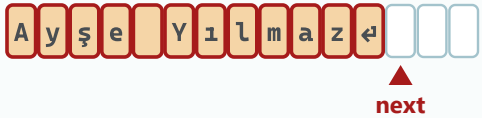
VARIABLES

keyboard =	Scanner(System.in)

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
5      Scanner keyboard = new Scanner(System.in);
6      System.out.print("Name: ");
▶      String name = keyboard.nextLine();
8      System.out.print("Age: ");
9      int age = keyboard.nextInt();
10     int nextYear = age + 1;
11     System.out.println("Hello " + name + "!");
12     System.out.println("Next year you'll be " + nextYear +
   ".");
13     keyboard.close();
14   }
15 }
```

System.in buffer



The user types `Ayşe Yılmaz` and presses Enter. `nextLine()` takes the **whole line**, spaces included, and drops the newline (⏎).

Step 3 / 9

OUTPUT

```
Name: Ayşe Yılmaz
```

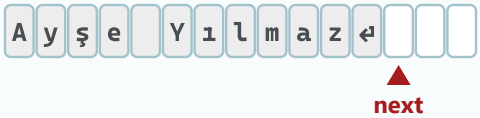
VARIABLES

`name = "Ayşe Yılmaz"`

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
5      Scanner keyboard = new Scanner(System.in);
6      System.out.print("Name: ");
7      String name = keyboard.nextLine();
▶      System.out.print("Age: ");
9      int age = keyboard.nextInt();
10     int nextYear = age + 1;
11     System.out.println("Hello " + name + "!");
12     System.out.println("Next year you'll be " + nextYear +
   ".");
13     keyboard.close();
14   }
15 }
```

System.in buffer



The second prompt is printed.

Step 4 / 9

OUTPUT

```
Name: Ayşe Yılmaz
Age:
```

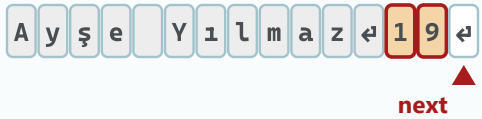
VARIABLES

`name = "Ayşe Yılmaz"`

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
5      Scanner keyboard = new Scanner(System.in);
6      System.out.print("Name: ");
7      String name = keyboard.nextLine();
8      System.out.print("Age: ");
▶      int age = keyboard.nextInt();
10     int nextYear = age + 1;
11     System.out.println("Hello " + name + "!");
12     System.out.println("Next year you'll be " + nextYear +
   ".");
13     keyboard.close();
14   }
15 }
```

System.in buffer



`nextInt()` reads the digits and converts them to the `int` 19. The newline (⏎) **stays** in the buffer.

Step 5 / 9

OUTPUT

```
Name: Ayşe Yılmaz
Age: 19
```

VARIABLES

`name = "Ayşe Yılmaz"`

`age = 19`

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
5      Scanner keyboard = new Scanner(System.in);
6      System.out.print("Name: ");
7      String name = keyboard.nextLine();
8      System.out.print("Age: ");
9      int age = keyboard.nextInt();
▶      int nextYear = age + 1;
11     System.out.println("Hello " + name + "!");
12     System.out.println("Next year you'll be " + nextYear +
   ".");
13     keyboard.close();
14   }
15 }
```

System.in buffer



The value is now an ordinary `int` : 19 + 1 = 20.

Step 6 / 9

OUTPUT

```
Name: Ayşe Yılmaz
Age: 19
```

VARIABLES

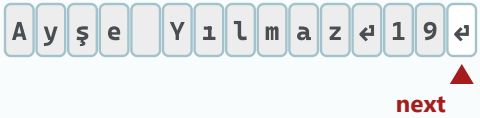
`name = "Ayşe Yılmaz"`

`age = 19` `nextYear = 20`

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
5      Scanner keyboard = new Scanner(System.in);
6      System.out.print("Name: ");
7      String name = keyboard.nextLine();
8      System.out.print("Age: ");
9      int age = keyboard.nextInt();
10     int nextYear = age + 1;
▶      System.out.println("Hello " + name + "!");
12     System.out.println("Next year you'll be " + nextYear +
   ".");
13     keyboard.close();
14   }
15 }
```

System.in buffer



Lines are written to the output stream.

Step 7 / 9

OUTPUT

```
Name: Ayşe Yılmaz
Age: 19
Hello Ayşe Yılmaz!
```

VARIABLES

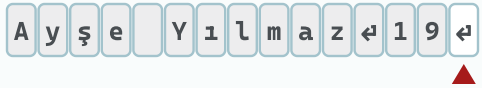
`name = "Ayşe Yılmaz"`

`age = 19` `nextYear = 20`

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
5      Scanner keyboard = new Scanner(System.in);
6      System.out.print("Name: ");
7      String name = keyboard.nextLine();
8      System.out.print("Age: ");
9      int age = keyboard.nextInt();
10     int nextYear = age + 1;
11     System.out.println("Hello " + name + "!");
▶      System.out.println("Next year you'll be " + nextYear +
   ".");
13     keyboard.close();
14   }
15 }
```

System.in buffer



String concatenation: `"Next year you'll be " + 20 + "."`.

Step 8 / 9

OUTPUT

```
Name: Ayşe Yılmaz
Age: 19
Hello Ayşe Yılmaz!
Next year you'll be 20.
```

VARIABLES

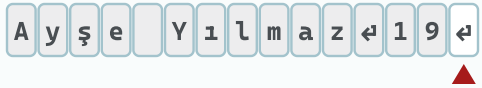
`name = "Ayşe Yılmaz"`

`age = 19` `nextYear = 20`

Reading the Keyboard with Scanner, Step by Step

```
1  import java.util.Scanner;
2
3  public class ConsoleInput {
4    public static void main(String[] args) {
5      Scanner keyboard = new Scanner(System.in);
6      System.out.print("Name: ");
7      String name = keyboard.nextLine();
8      System.out.print("Age: ");
9      int age = keyboard.nextInt();
10     int nextYear = age + 1;
11     System.out.println("Hello " + name + "!");
12     System.out.println("Next year you'll be " + nextYear +
   ".");
▶      keyboard.close();
14   }
15 }
```

System.in buffer



`close()` closes `System.in` along with the scanner; the program can no longer read the keyboard.

Step 9 / 9

OUTPUT

```
Name: Ayşe Yılmaz
Age: 19
Hello Ayşe Yılmaz!
Next year you'll be 20.
```

VARIABLES

`name = "Ayşe Yılmaz"`

`age = 19` `nextYear = 20`

Files

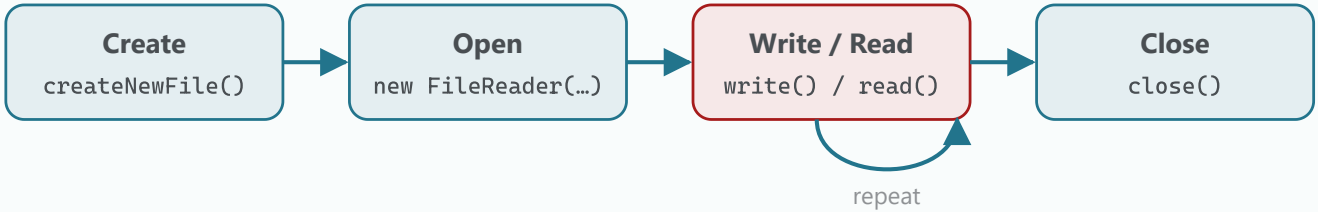
- A file stores data **persistently** on disk: it survives the program exiting and the computer restarting (variables in RAM do not).
- Every file is ultimately **0s and 1s**, a sequence of bytes. What differs is how those bytes are **interpreted**.
- **Text files**: bytes are turned into letters by a character encoding; people can read them.
- **Binary files**: bytes are numbers, pixels, audio samples, machine instructions; only a program that knows the format can make sense of them.

file.txt in a hex editor:

```
0000  4D 65 72 68 61 62 61 2C  Merhaba,
0008  20 64 C3 BC 6E 79 61 21   d..nya!
```

Offset on the left, each byte in hex in the middle, the ASCII view on the right. `4D` = 77 = `M`. The 15 characters (Turkish for “Hello, world!”) take 16 bytes: **ü** is two bytes in UTF-8 (`C3 BC`).

Working with a File



Step	What happens	In Java
Create	A new, empty file appears on disk.	`file.createNewFile()`; opening for writing also creates it
Open	A stream is attached to an existing file.	`new FileInputStream(name)`, `new FileReader(name)`, `new Scanner(file)`
Write	Data is transferred to the file.	`write()`, `print()`, `println()`
Read	Data is brought from the file into the program.	`read()`, `readLine()`, `nextLine()`
Close	Buffered data is flushed to disk and the OS resource is released.	`close()` or `try`-with-resources

An unclosed file causes both **data loss** (buffered bytes never reach the disk) and a resource leak: the OS limits how many files can be open at once.

A First File Program

```java
import java.io.*;

public class FirstFile {
  public static void main(String[] args) throws IOException {
    // Write to a text file
    FileWriter writer = new FileWriter("file.txt");
    writer.write("Merhaba, dünya!");
    writer.close();
    // Write one byte to a binary file
    FileOutputStream out = new FileOutputStream("data.dat");
    out.write(127);
    out.close();
    // Read from the binary file
    FileInputStream stream = new FileInputStream("data.dat");
    int b = stream.read();
    stream.close();
    System.out.println("Byte read: " + b);
    File text = new File("file.txt");
    System.out.println("file.txt: " + text.length() + " bytes");
  }
}
```

OUTPUT

```
Byte read: 127
file.txt: 16 bytes
```

FILE.TXT

```
Merhaba, dünya!
```

- `FileWriter` writes **text**.
- `FileOutputStream` writes **raw bytes**; `write(127)` is the single byte `7F`.
- `read()` returns the byte as an `int` from 0 to 255.

Text Files

- Text files contain **printable characters**: letters, digits, punctuation, spaces, line breaks.
- Any **text editor** (Notepad, VS Code) can open and edit them. Examples: source code (`.java` , `.c`), `.txt` , `.csv` , `.html` .
- A **character set / encoding** is the table that says which number (which bytes) stores each character. Writer and reader must use **the same encoding**.

ASCII

7 bits, 128 characters: English letters, digits, punctuation, control characters.

ISO-8859-1

8 bits, 256 characters: Western Europe (Latin-1). Has é, ü, ö, ç; **no ş, ğ, ı**.

ISO-8859-9

8 bits, Latin-5: the Turkish variant. Six Latin-1 letters are replaced by ğ, Ğ, ı, İ, ş, Ş.

UTF-8

The universal Unicode encoding: every language, 1–4 bytes. The default on the web and in Java 18+.

Binary Files

- Bytes are stored not as characters but in a **format** the program defines: numbers, pixels, audio samples, machine code.
- Meaningful to a computer (the right program); people **cannot read them directly**.
- Examples: executables (`.exe` , Java's `.class` files), images (`.png` , `.jpg`), music (`.mp3`), video (`.mp4`), data files (`.dat`).
- Advantage: **smaller** and **faster** to process; no number-to-text conversion.

FirstFile.class in a hex editor (first 32 bytes):

```
0000  CA FE BA BE 00 00 00 41  .......A
0008  00 5C 0A 00 02 00 03 07  .\......
0010  00 04 0C 00 05 00 06 01  ........
0018  00 10 6A 61 76 61 2F 6C  ..java/l
```

The same file opened in Notepad:

```
Êþº¾···A·\···············java/lang/Object···<init>·
··()V·····java/io/FileWriter·····file.txt·········
·····(
```

Every `.class` file starts with the bytes `CA FE BA BE` : the JVM recognizes the file type by this “magic number”.

Extended ASCII and Turkish Characters

- **Extended ASCII** uses 8 bits (1 byte) to represent 2^8 = **256** characters.
- **0–127:** standard ASCII, the same in every table. **128–255:** special characters, symbols and national letters.
- There is no single “extended ASCII”: the upper half depends on the **code page**. Turkish uses **ISO-8859-9** (Latin-5) and **Windows-1254**.

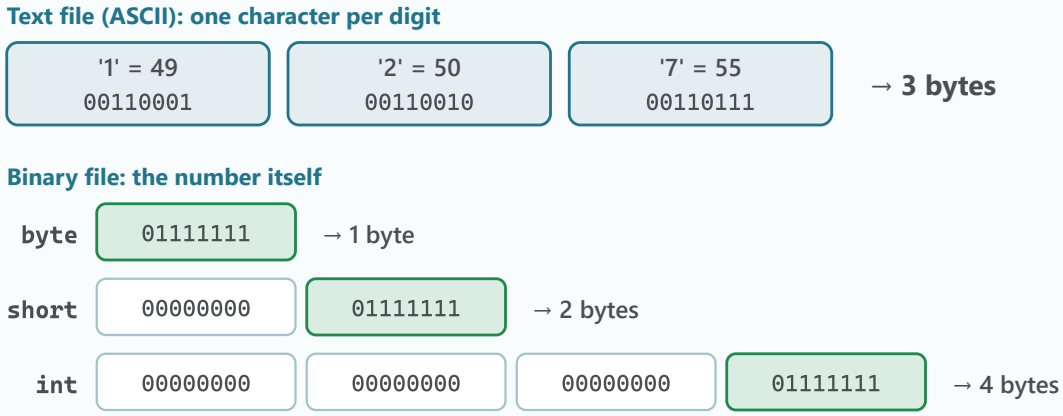
Byte	D0	DD	DE	F0	FD	FE	FC	E7
ISO-8859-1 (Western Europe)	Đ	Ý	Þ	ð	ý	þ	ü	ç
ISO-8859-9 (Turkish)	**Ğ**	**İ**	**Ş**	**ğ**	**ı**	**ş**	ü	ç



The same byte (FE) is **ş** in one table and **þ** in the other. Opening a file with the wrong code page gives garbled text such as “þimdi”. The fix: Unicode / UTF-8, which covers every language.

Writing a Number (127) to a File

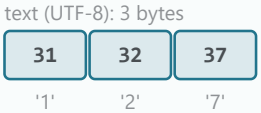
- How a number is stored depends on its data type, its purpose and the file format.
- **As text:** the number is first turned into characters, one ASCII byte per digit: the code of `'1'` is 49, not 1.
- **As binary:** the binary value (0111 1111) is written directly, using as many bytes as the type needs. More efficient for numeric data.

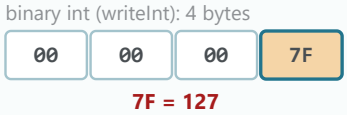


Text or Binary? A Comparison

How 127 is written	Size
ASCII / UTF-8 text file	3 bytes
UTF-16 text file (no BOM)	6 bytes
Binary, `byte`	1 byte
Binary, `short`	2 bytes
Binary, `int`	4 bytes

- **Text:** human-readable, opens in any editor, portable across platforms. Logs, settings, CSV tables.
- **Binary:** stores numbers smaller and faster, but you must know the writer's format. Images, audio, database files.
- An `int` is always 4 bytes, whether 7 or 2 000 000 000. In text the size depends on the digit count (7 → 1 byte, 2 000 000 000 → 10 bytes).





Use cases: ASCII / text for data people will read; binary files for storing numeric data efficiently.

Java tools: `DataOutputStream` writes primitives in **binary** (`writeByte` , `writeShort` , `writeInt` ...), `DataInputStream` reads them back in the same order; `OutputStreamWriter` turns characters into bytes with a given encoding.

Checking in Code: 127

```
public class Number127 {
  static void writeText(String name, Charset cs) throws IOException {
    Writer w = new OutputStreamWriter(new FileOutputStream(name), cs);
    w.write(String.valueOf(127));            // "127": 3 chars
    w.close();
  }
  static void writeBinary(String name, int size) throws IOException {
    FileOutputStream fos = new FileOutputStream(name);
    DataOutputStream d = new DataOutputStream(fos);
    if (size == 1) d.writeByte(127);        // 01111111
    else if (size == 2) d.writeShort(127);  // 00000000 01111111
    else d.writeInt(127);                    // 4 bytes
    d.close();
  }
  public static void main(String[] args) throws IOException {
    writeText("text.txt", StandardCharsets.US_ASCII);
    writeText("text16.txt", StandardCharsets.UTF_16BE);
    writeBinary("b.dat", 1); writeBinary("s.dat", 2); writeBinary("i.dat", 4);
    String[] names = {"text.txt", "text16.txt",
             "b.dat", "s.dat", "i.dat"};
    for (String name : names)
      System.out.println(name + ": " + new File(name).length());
  }
}
```

OUTPUT (BYTES)

```
text.txt: 3
text16.txt: 6
b.dat: 1
s.dat: 2
i.dat: 4
```

The java.io.File Class

- A `File` object represents a **path** on disk: a file or a directory. It does not read or write the file's **contents**; streams do that.
- It queries the file system and answers:
- Does the path exist? Is it a file or a directory?
- Is it readable, writable, executable? What is its size and last-modified time?
- It can **create** a file or directory, **delete** one, and **list** a directory's contents.
- `new File("sample.txt")` creates nothing on disk; it only holds the path.

Method	Returns
`exists()`	whether the path exists (`boolean`)
`isFile()` / `isDirectory()`	file / directory?
`canRead()`, `canWrite()`, `canExecute()`	permissions
`createNewFile()`	created? (`false` if it exists)
`mkdir()` / `mkdirs()`	directory (with parents)
`delete()`	deleted? (not a non-empty dir)
`list()` / `listFiles()`	`String[]` / `File[]`
`length()`	size (bytes, `long`)
`lastModified()`	milliseconds since 1970
`getName()`, `getAbsolutePath()`	name, full path

Files and Directories with File

```java
public class FileInfo {
  public static void main(String[] args) throws IOException {
    File dir = new File("sample");
    File file = new File("sample/file.txt");
    System.out.println("exists: " + file.exists());
    System.out.println("mkdir: " + dir.mkdir());
    System.out.println("created: " + file.createNewFile());
    System.out.println("isFile: " + file.isFile());
    System.out.println("isDir: " + file.isDirectory());
    System.out.println("readable: " + file.canRead());
    System.out.println("writable: " + file.canWrite());
    System.out.println("size: " + file.length());
    String[] items = dir.list();
    System.out.println("in dir: " + items.length + " item(s)");
    long time = file.lastModified();
    System.out.println("time > 0: " + (time > 0));
    System.out.println("deleted: " + file.delete());
    System.out.println("exists: " + file.exists());
    dir.delete();
  }
}
```

OUTPUT

```
exists: false
mkdir: true
created: true
isFile: true
isDir: false
readable: true
writable: true
size: 0
in dir: 1 item(s)
time > 0: true
deleted: true
exists: false
```

- `/` works as a path separator on every system; `File.separator` is also available.
- A relative path is resolved against the program's working directory.

Reading a File with Scanner

```
public class ScannerFile {
  public static void main(String[] args) {
    Scanner scanner = null;
    try {
      scanner = new Scanner(new File("sample.txt"));
      // Read the file and print it
      while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        System.out.println(line);
      }
    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e.getMessage());
    } finally {
      if (scanner != null)
        scanner.close();
    }
  }
}
```

OUTPUT

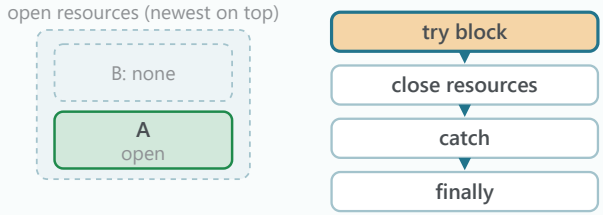
```
Java
Input Output
Streams
```

- A missing file throws a checked exception: `catch` is required.
- `hasNextLine()` : is there a line? `nextLine()` : take it.
- If opening fails, `scanner` stays `null` : hence the `finally` check.

`Scanner` reads files and other inputs such as the keyboard or a `String` . Imports: `java.io.File` , `java.io.FileNotFoundException` , `java.util.Scanner` . `Scanner` is `AutoCloseable` : with try-with-resources, no `finally` or `null` check is needed.

try-with-resources: Closing Order

```
1  class Res implements AutoCloseable {
2    String name;
3    Res(String name) { this.name = name;
   System.out.println(name + " opened"); }
4    public void close() { System.out.println(name + "
   closed"); }
5  }
6  public class CloseOrder {
7    public static void main(String[] args) {
▶      try (Res a = new Res("A"); Res b = new Res("B")) {
9        System.out.println("block running");
10       throw new IllegalStateException("error!");
11     } catch (IllegalStateException e) {
12       System.out.println("caught: " + e.getMessage());
13     } finally { System.out.println("finally"); }
14   }
15 }
```



The first resource in the parentheses is created; its constructor prints a message. `a` is now open.

Step 1 / 8

VARIABLES

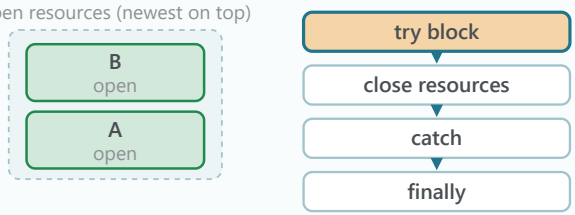
`a = Res("A")`

OUTPUT

```
A opened
```

try-with-resources: Closing Order

```
1  class Res implements AutoCloseable {
2    String name;
3    Res(String name) { this.name = name;
   System.out.println(name + " opened"); }
4    public void close() { System.out.println(name + "
   closed"); }
5  }
6  public class CloseOrder {
7    public static void main(String[] args) {
▶      try (Res a = new Res("A"); Res b = new Res("B")) {
9        System.out.println("block running");
10       throw new IllegalStateException("error!");
11     } catch (IllegalStateException e) {
12       System.out.println("caught: " + e.getMessage());
13     } finally { System.out.println("finally"); }
14   }
15 }
```



The second resource, `b`, is opened. Resources are created **left to right**.

Step 2 / 8

VARIABLES

`a = Res("A")` `b = Res("B")`

OUTPUT

```
A opened
B opened
```

try-with-resources: Closing Order

```
1  class Res implements AutoCloseable {
2    String name;
3    Res(String name) { this.name = name;
   System.out.println(name + " opened"); }
4    public void close() { System.out.println(name + "
   closed"); }
5  }
6  public class CloseOrder {
7    public static void main(String[] args) {
8      try (Res a = new Res("A"); Res b = new Res("B")) {
▶        System.out.println("block running");
10       throw new IllegalStateException("error!");
11     } catch (IllegalStateException e) {
12       System.out.println("caught: " + e.getMessage());
13     } finally { System.out.println("finally"); }
14   }
15 }
```



The block body runs; both resources are open.

Step 3 / 8

VARIABLES

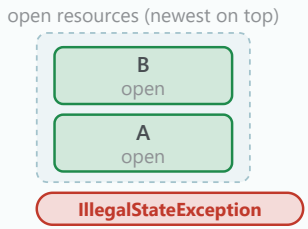
`a = Res("A")` `b = Res("B")`

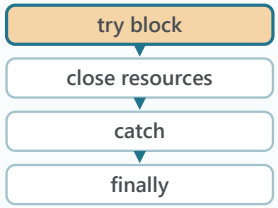
OUTPUT

```
A opened
B opened
block running
```

try-with-resources: Closing Order

```
1  class Res implements AutoCloseable {
2    String name;
3    Res(String name) { this.name = name;
   System.out.println(name + " opened"); }
4    public void close() { System.out.println(name + "
   closed"); }
5  }
6  public class CloseOrder {
7    public static void main(String[] args) {
8      try (Res a = new Res("A"); Res b = new Res("B")) {
9        System.out.println("block running");
▶        throw new IllegalStateException("error!");
11     } catch (IllegalStateException e) {
12       System.out.println("caught: " + e.getMessage());
13     } finally { System.out.println("finally"); }
14   }
15 }
```





An exception is thrown. Before control reaches `catch`, the resources are closed **automatically**.

Step 4 / 8

VARIABLES

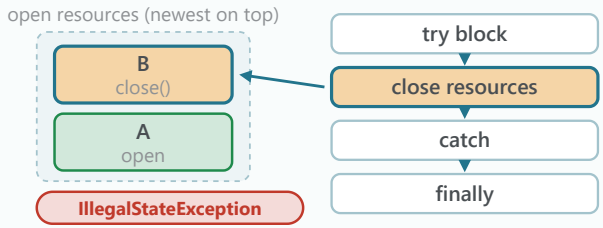
`a = Res("A")` `b = Res("B")`

OUTPUT

```
A opened
B opened
block running
```

try-with-resources: Closing Order

```
1  class Res implements AutoCloseable {
2    String name;
3    Res(String name) { this.name = name;
   System.out.println(name + " opened"); }
▶    public void close() { System.out.println(name + "
   closed"); }
5  }
6  public class CloseOrder {
7    public static void main(String[] args) {
8      try (Res a = new Res("A"); Res b = new Res("B")) {
9        System.out.println("block running");
10       throw new IllegalStateException("error!");
11     } catch (IllegalStateException e) {
12       System.out.println("caught: " + e.getMessage());
13     } finally { System.out.println("finally"); }
14   }
15 }
```



Closing runs in **reverse order**: first `close()` on `b`, the last one opened.

Step 5 / 8

VARIABLES

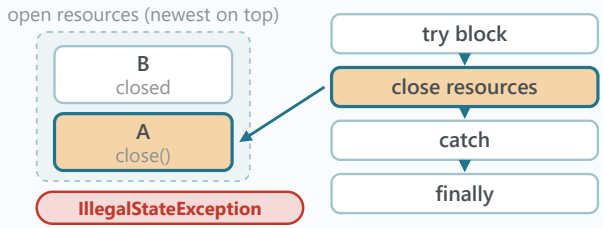
`this.name = "B"`

OUTPUT

```
A opened
B opened
block running
B closed
```

try-with-resources: Closing Order

```
1  class Res implements AutoCloseable {
2    String name;
3    Res(String name) { this.name = name;
   System.out.println(name + " opened"); }
▶    public void close() { System.out.println(name + "
   closed"); }
5  }
6  public class CloseOrder {
7    public static void main(String[] args) {
8      try (Res a = new Res("A"); Res b = new Res("B")) {
9        System.out.println("block running");
10       throw new IllegalStateException("error!");
11     } catch (IllegalStateException e) {
12       System.out.println("caught: " + e.getMessage());
13     } finally { System.out.println("finally"); }
14   }
15 }
```



Then `a.close()` runs. The exception is still in flight, not yet caught.

Step 6 / 8

VARIABLES

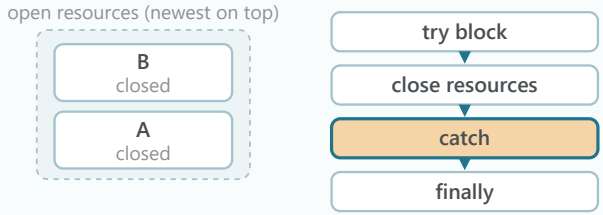
`this.name = "A"`

OUTPUT

```
A opened
B opened
block running
B closed
A closed
```

try-with-resources: Closing Order

```
1  class Res implements AutoCloseable {
2    String name;
3    Res(String name) { this.name = name;
   System.out.println(name + " opened"); }
4    public void close() { System.out.println(name + "
   closed"); }
5  }
6  public class CloseOrder {
7    public static void main(String[] args) {
8      try (Res a = new Res("A"); Res b = new Res("B")) {
9        System.out.println("block running");
10       throw new IllegalStateException("error!");
▶      } catch (IllegalStateException e) {
▶        System.out.println("caught: " + e.getMessage());
13     } finally { System.out.println("finally"); }
14   }
15 }
```



Only **after** the resources are closed does `catch` catch the exception. By now `a` and `b` are out of scope.

Step 7 / 8

VARIABLES

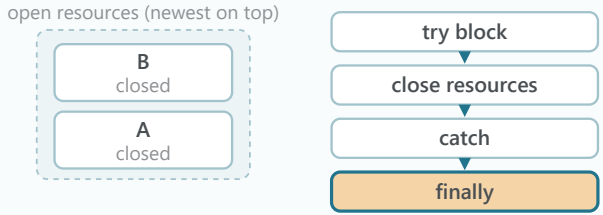
`e = IllegalStateException("error!")`

OUTPUT

```
A opened
B opened
block running
B closed
A closed
caught: error!
```

try-with-resources: Closing Order

```
1  class Res implements AutoCloseable {
2    String name;
3    Res(String name) { this.name = name;
   System.out.println(name + " opened"); }
4    public void close() { System.out.println(name + "
   closed"); }
5  }
6  public class CloseOrder {
7    public static void main(String[] args) {
8      try (Res a = new Res("A"); Res b = new Res("B")) {
9        System.out.println("block running");
10       throw new IllegalStateException("error!");
11     } catch (IllegalStateException e) {
12       System.out.println("caught: " + e.getMessage());
▶      } finally { System.out.println("finally"); }
14   }
15 }
```



`finally` runs last. Order: open A, open B, body, close B, close A, `catch`, `finally`.

Step 8 / 8

VARIABLES

—

OUTPUT

```
A opened
B opened
block running
B closed
A closed
caught: error!
finally
```

The Java I/O Library

- The `java.io` package is a powerful, flexible toolkit for input and output, used for everything from files to network programming.

Input / output
Receive and send data from sources such as files, the network and memory.

Binary / text
Supports both byte-based (binary) and character-based (text) I/O.

Sequential / random
Data in a file can be read in order or by jumping to any position.

Kind	Examples	Purpose
Classes	`File`	file and directory (path) operations
	`InputStream`, `OutputStream`	basic byte I/O
	`Reader`, `Writer`	text (character) I/O
Interfaces	`Closeable`, `AutoCloseable`	release the resource: `close()`
	`Flushable`	empty the buffer: `flush()`
Exceptions	`IOException`	superclass of I/O errors
	`FileNotFoundException`, `EOFException`	no such file / unexpected end of file

Byte-Oriented and Character-Oriented I/O

Byte streams

- `InputStream` and `OutputStream` classes
- Data is handled as **bytes** (8 bits)
- Works with **binary** data: images, audio, video, `.class`
- `read()` returns a byte from 0 to 255

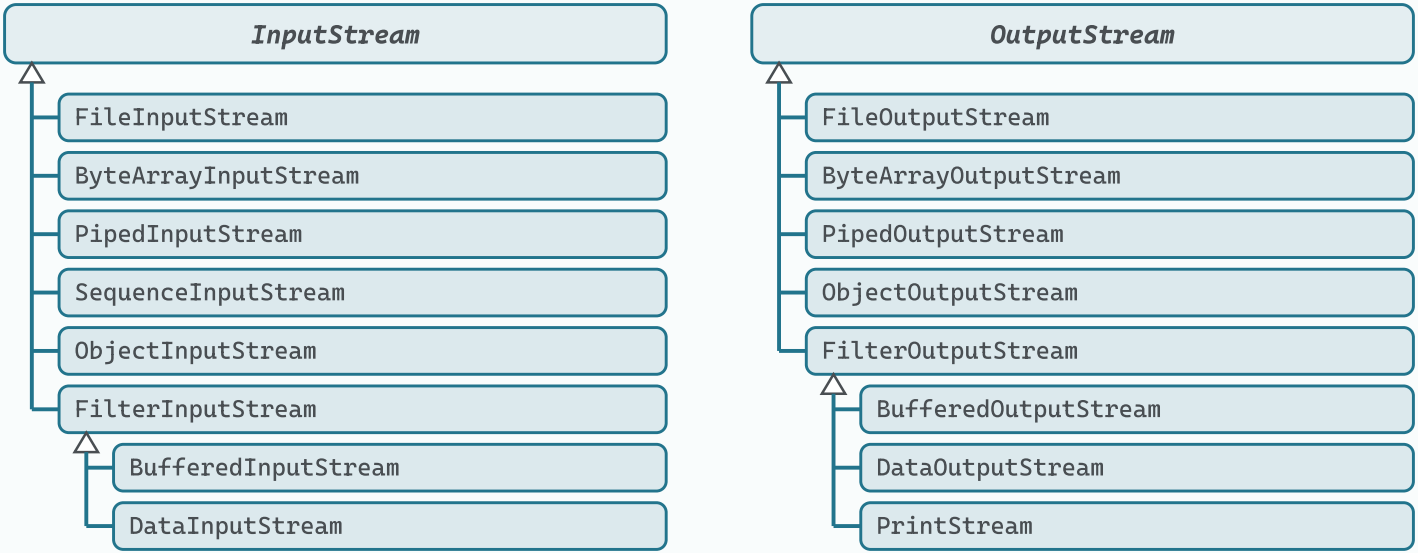
Character streams

- `Reader` and `Writer` classes
- Data is handled as **char** (16-bit Unicode)
- More efficient for **text** data
- Decodes the encoding (UTF-8...): supports any language or alphabet

	Byte stream	Character stream
Unit read	`byte`	`char`
File classes	`FileInputStream`, `FileOutputStream`	`FileReader`, `FileWriter`
Buffered	`BufferedInputStream`, `BufferedOutputStream`	`BufferedReader`, `BufferedWriter`
Use	any file, copying	text files only

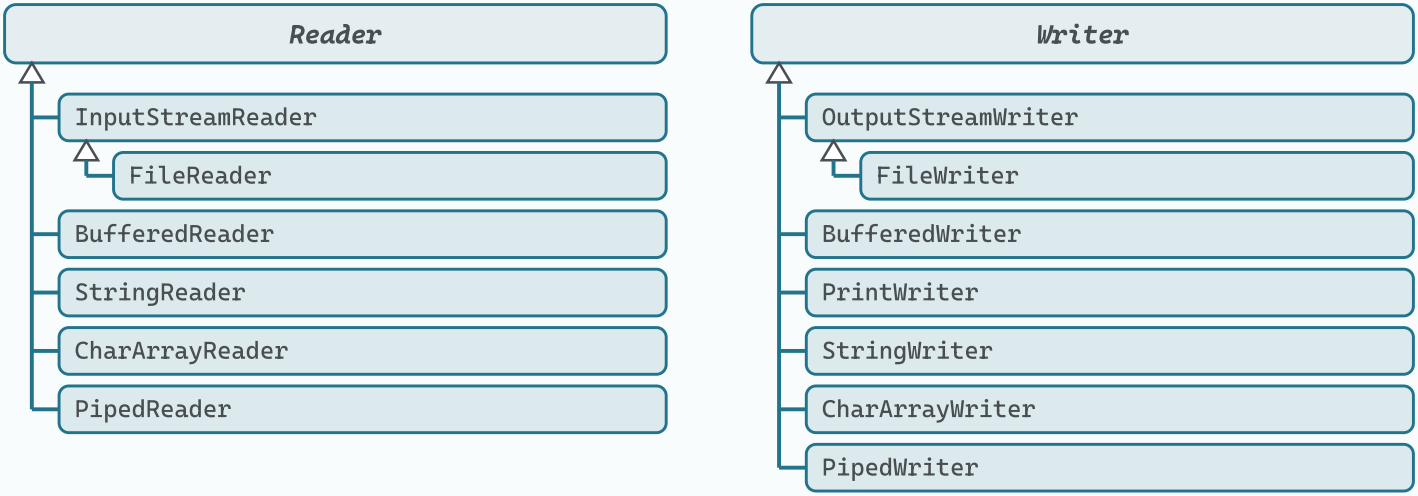
Bridge classes: `InputStreamReader` turns a byte stream into a character stream, `OutputStreamWriter` turns characters into a byte stream (this is where the encoding is applied).

Byte Stream Classes



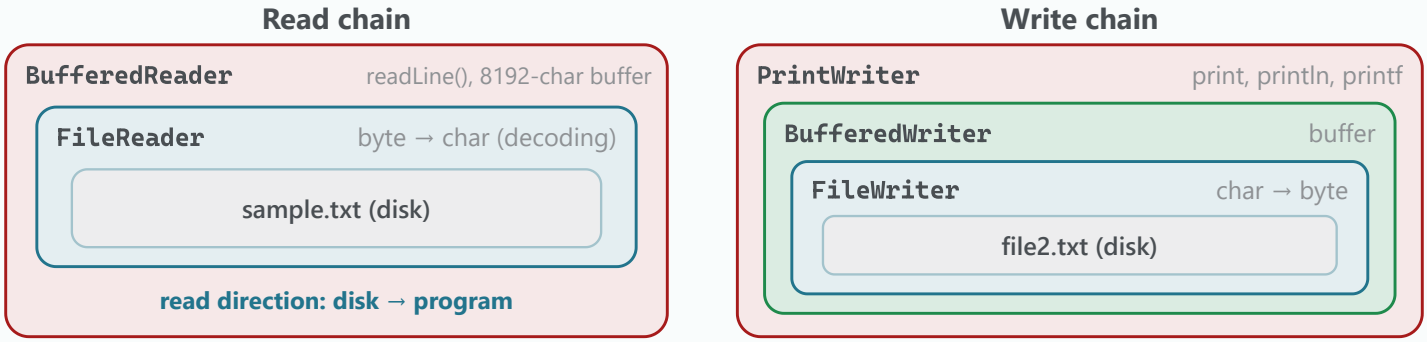
Italic boxes are **abstract** classes; hollow-triangle arrows show inheritance. `System.out` is a `PrintStream`. `Filter…` subclasses **wrap** another stream to add a capability (buffering, reading primitives).

Character Stream Classes



`FileReader` is really an `InputStreamReader`: it decodes the bytes from a `FileInputStream` into characters with the chosen encoding. Likewise, `FileWriter` extends `OutputStreamWriter`.

Wrapper (Decorator) Streams




```
BufferedReader reader = new BufferedReader(new FileReader("sample.txt"));
PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter("file2.txt")));
```

- Each stream class does **one job** well; you combine capabilities by **wrapping** streams inside one another (the Decorator pattern).
- Think of a hose with a filter and then a spray nozzle on the end: the water (data) passes through each in turn.
- Closing the outermost stream (`close()`) closes all the inner ones.

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
▶      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
9        while ((line = reader.readLine()) != null) {
10         System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

▶ Java↵
Input Output↵
Streams↵
(end of file)

`File` only holds the path; the file is not open yet.

Step 1 / 10

OUTPUT

VARIABLES

`file = "sample.txt"`

CALL STACK

`main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
9        while ((line = reader.readLine()) != null) {
10         System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
Streams↵
(end of file)
```

`FileReader` opens the file and `BufferedReader` wraps it. The read position is at the start.

Step 2 / 10

OUTPUT

VARIABLES

`reader = BufferedReader`

CALL STACK

`main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
▶9       while ((line = reader.readLine()) != null) {
10         System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
Streams↵
(end of file)
```

`readLine()` reads a line and **drops** the ⏎.
Not `null` → enter the loop.

Step 3 / 10

OUTPUT

VARIABLES

`line = "Java"`

CALL STACK

`reader.readLine()`

`main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
9        while ((line = reader.readLine()) != null) {
▶          System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
Streams↵
(end of file)
```

The line is printed.

Step 4 / 10

OUTPUT

```
Java
```

VARIABLES

`line = "Java"`

CALL STACK

`main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
▶9       while ((line = reader.readLine()) != null) {
10         System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
▶ Streams↵
(end of file)
```

The next line is read: `line` = `"Input Output"`.

Step 5 / 10

OUTPUT

```
Java
```

VARIABLES

`line = "Input Output"`

CALL STACK

`reader.readLine()`

`main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
9        while ((line = reader.readLine()) != null) {
▶          System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
▶ Streams↵
(end of file)
```

The line is printed.

Step 6 / 10

OUTPUT

```
Java
Input Output
```

VARIABLES

`line = "Input Output"`

CALL STACK

`main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
▶        while ((line = reader.readLine()) != null) {
10         System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
Streams↵
(end of file)
```

The next line is read: `line` = `"Streams"`.

Step 7 / 10

OUTPUT

```
Java
Input Output
```

VARIABLES

`line = "Streams"`

CALL STACK

`reader.readLine()`

`main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
9        while ((line = reader.readLine()) != null) {
▶          System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
Streams↵
(end of file)
```

The line is printed.

Step 8 / 10

OUTPUT

```
Java
Input Output
Streams
```

VARIABLES

`line = "Streams"`

CALL STACK

`main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
▶        while ((line = reader.readLine()) != null) {
10         System.out.println(line);
11       }
12     } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
Streams↵
(end of file)
```

End of file: `readLine()` returns `null` → the loop ends.

Step 9 / 10

OUTPUT

```
Java
Input Output
Streams
```

VARIABLES

`line = null`

CALL STACK

- `reader.readLine()`
- `main()`

BufferedReader Line by Line: Step by Step

```
1  import java.io.*;
2
3  public class ReadLines {
4    public static void main(String[] args) {
5      File file = new File("sample.txt");
6      try (BufferedReader reader =
7          new BufferedReader(new FileReader(file))) {
8        String line;
9        while ((line = reader.readLine()) != null) {
10         System.out.println(line);
11       }
▶      } catch (IOException e) {
13       System.err.println("Read error: " + e.getMessage());
14       e.printStackTrace();
15     }
16   }
17 }
```

sample.txt

```
Java↵
Input Output↵
Streams↵
(end of file)
```

The `try` ends: `reader` is **closed automatically**; `catch` does not run.

Step 10 / 10

OUTPUT

```
Java
Input Output
Streams
```

VARIABLES

—

CALL STACK

`main()`

java.io.PrintWriter

```java
public class PrintWriterDemo {
  public static void main(String[] args) {
    PrintWriter writer = null;
    try {
      writer = new PrintWriter("newFile.txt");
      writer.println("Hello, world!");
      writer.println("This is a sample file.");
      writer.printf(Locale.ROOT, "Pi is about %.2f%n", Math.PI);
      writer.println(42);
    } catch (IOException e) {
      System.err.println("File error: " + e.getMessage());
      e.printStackTrace();
    } finally {
      if (writer != null)
        writer.close();
    }
  }
}
```

- Writes text files **easily**: `print`, `println`, `printf`.
- Writes other types by **converting them to text**.
- Creates the file if missing; otherwise **truncates** it.
- Don't forget `close`: data may be left in the buffer.

NEWFILE.TXT

```
Hello, world!
This is a sample file.
Pi is about 3.14
42
```

java.io.FileWriter

- Writes **character-based** data to files; it is a character stream (`Writer`).
- It has only `write(String)` , `write(char[])` , `write(int c)` : **formatting is limited**, and line breaks (`\n`) must be written explicitly.
- A **lower-level**, flexible approach; usually wrapped in a `BufferedWriter` or `PrintWriter` .
- `new FileWriter(name, true)` **appends** to the end of the file; `false` or no flag truncates it.

	`FileWriter`	`BufferedWriter`	`PrintWriter`
Write methods	`write`	`write` , `newLine()`	`print` , `println` , `printf`
Writing a number	convert to `String` first	convert to `String` first	directly: `println(42)`
Buffer	small internal buffer	8192-char buffer	depends on the wrapped stream
Error reporting	`IOException`	`IOException`	`checkError()`

FileWriter Example: Write and Append

```
public class FileWriterDemo {
  public static void main(String[] args) {
    try (FileWriter writer = new FileWriter("newFile.txt")) {
      writer.write("Hello, world!\n");
      writer.write("This is a FileWriter example.\n");
      System.out.println("File created and data written.");
    } catch (IOException e) {
      System.err.println("File error: " + e.getMessage());
      e.printStackTrace();
    }
    // true: append to the end of the file
    try (FileWriter app = new FileWriter("newFile.txt", true)) {
      app.write("Appended line.\n");
    } catch (IOException e) {
      System.err.println("File error: " + e.getMessage());
    }
  }
}
```

OUTPUT

```
File created and data written.
```

NEWFILE.TXT

```
Hello, world!
This is a FileWriter example.
Appended line.
```

Specifying the Character Encoding

```
public class EncodingRead {
  public static void main(String[] args) throws IOException {
    File file = new File("turkish.txt");
    try (BufferedWriter writer = new BufferedWriter(
        new FileWriter(file, StandardCharsets.UTF_8))) {    // Java 11+
      writer.write("Işık, şimşek, çiğ, öğrenci, Ünye");
    }
    try (BufferedReader right = new BufferedReader(
        new FileReader(file, StandardCharsets.UTF_8))) {
      System.out.println(right.readLine());
    }
    try (BufferedReader wrong = new BufferedReader(
        new FileReader(file, StandardCharsets.ISO_8859_1))) {
      System.out.println(wrong.readLine());
    }
  }
}
```

OUTPUT

```
Işık, şimşek, çiğ, öğrenci, Ünye
IÅÄ±k, Åimåek, Ã§iÄ, Ã¶Ärenci, Ãnye
```

- Without an explicit encoding the **default** is used: UTF-8 in Java 18+, previously the OS setting (Windows-1254). Portable code names the encoding **explicitly**; in Java 10 and earlier, via the `InputStreamReader` bridge.
- A character with no mapping becomes `?` (`3F`): `ş` is `3F` in ASCII, `C5 9F` in UTF-8.

Buffering

- The basic methods: `read()` **reads** a byte or a byte array, `write()` **writes** a byte or a byte array.
- A separate disk access per byte **slows the program badly**: the disk is thousands of times slower than memory.
- The fix: collect data in a **buffer** in memory and read from or write to the disk **in bulk**. Fewer physical disk accesses.

Analogy: instead of carrying 10 items home from the store one trip at a time, put them in a basket and carry them **in one go**.

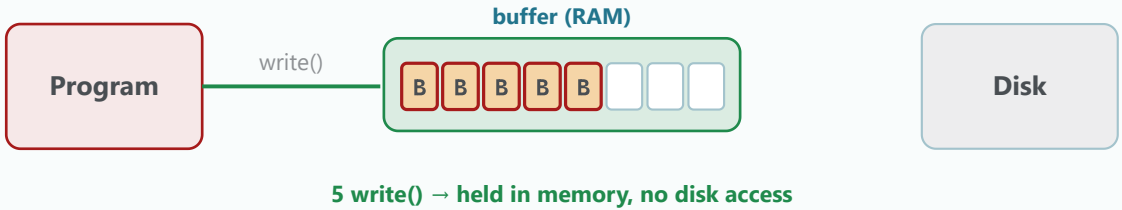


Unbuffered: every `write()` call goes straight to the disk.

Buffering

- The basic methods: `read()` **reads** a byte or a byte array, `write()` **writes** a byte or a byte array.
- A separate disk access per byte **slows the program badly**: the disk is thousands of times slower than memory.
- The fix: collect data in a **buffer** in memory and read from or write to the disk **in bulk**. Fewer physical disk accesses.

Analogy: instead of carrying 10 items home from the store one trip at a time, put them in a basket and carry them **in one go**.

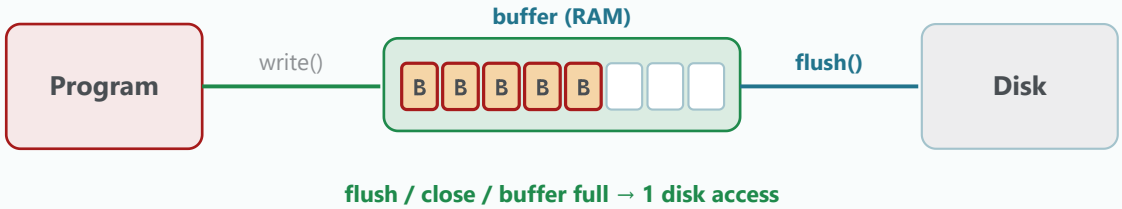


Buffered: `write()` calls first put the bytes in a buffer in memory.

Buffering

- The basic methods: `read()` **reads** a byte or a byte array, `write()` **writes** a byte or a byte array.
- A separate disk access per byte **slows the program badly**: the disk is thousands of times slower than memory.
- The fix: collect data in a **buffer** in memory and read from or write to the disk **in bulk**. Fewer physical disk accesses.

Analogy: instead of carrying 10 items home from the store one trip at a time, put them in a basket and carry them **in one go**.



When the buffer fills up, or on `flush()` or `close()`, the whole buffer is written to disk at once.

Buffered Stream Classes

Class	Wraps	What it does
`BufferedInputStream`	`InputStream`	speeds up reading: on `read()` it pulls 8192 bytes from disk into a memory buffer at once
`BufferedReader`	`Reader`	the same, for characters; plus `readLine()`
`BufferedOutputStream`	`OutputStream`	speeds up writing: collects `write()` data in the buffer
`BufferedWriter`	`Writer`	the same, for characters; plus `newLine()`

```
public class FlushDemo {
  public static void main(String[] args) throws IOException {
    File log = new File("log.txt");
    BufferedWriter bw = new BufferedWriter(new FileWriter(log));
    bw.write("Record 1");
    System.out.println("before: " + log.length());
    bw.flush();
    System.out.println("after: " + log.length());
    bw.close();
  }
}
```

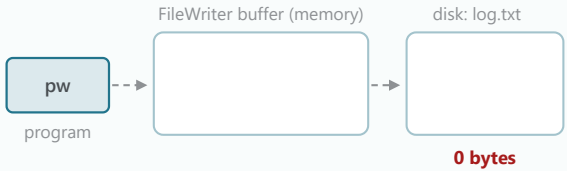
OUTPUT

```
before: 0
after: 8
```

`flush()` empties the buffer to disk immediately; `close()` flushes first. With an unclosed buffered writer, the last data is **lost**.

Buffer and flush: Step by Step

```
▶ File f = new File("log.txt");
▶ PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
4  pw.print("entry1;");
5  System.out.println("1: " + f.length());
6  pw.print("entry2;");
7  System.out.println("2: " + f.length());
8  pw.flush();
9  System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



`FileWriter` creates the (empty) file. Writes go to the **buffer** in memory first.

Step 1 / 11

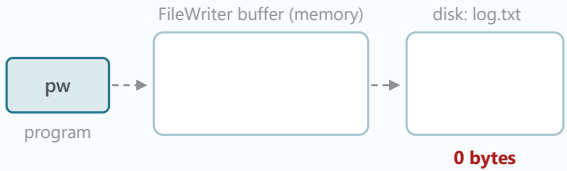
VARIABLES

`f = log.txt` `pw = PrintWriter`

OUTPUT

Buffer and flush: Step by Step

```
 1 File f = new File("log.txt");
 2 PrintWriter pw = new PrintWriter(new FileWriter(f));
▶ System.out.println("0: " + f.length());
 4 pw.print("entry1;");
 5 System.out.println("1: " + f.length());
 6 pw.print("entry2;");
 7 System.out.println("2: " + f.length());
 8 pw.flush();
 9 System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



Nothing written yet: 0 bytes.

Step 2 / 11

VARIABLES

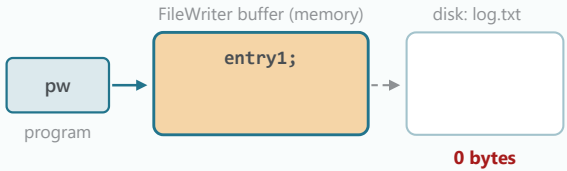
`f.length() = 0`

OUTPUT

```
0: 0
```

Buffer and flush: Step by Step

```
1  File f = new File("log.txt");
2  PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
▶  pw.print("entry1;");
5  System.out.println("1: " + f.length());
6  pw.print("entry2;");
7  System.out.println("2: " + f.length());
8  pw.flush();
9  System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



`print` puts 7 characters in the **buffer**; the disk is untouched.

Step 3 / 11

VARIABLES

`buffer = "entry1`

OUTPUT

```
0: 0
```

Buffer and flush: Step by Step

```
1  File f = new File("log.txt");
2  PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
4  pw.print("entry1;");
▶  System.out.println("1: " + f.length());
6  pw.print("entry2;");
7  System.out.println("2: " + f.length());
8  pw.flush();
9  System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



The file is still 0 bytes: the data waits in memory.

Step 4 / 11

VARIABLES

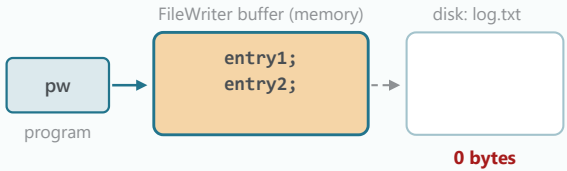
`f.length() = 0`

OUTPUT

```
0: 0
1: 0
```

Buffer and flush: Step by Step

```
1  File f = new File("log.txt");
2  PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
4  pw.print("entry1;");
5  System.out.println("1: " + f.length());
▶  pw.print("entry2;");
▶  System.out.println("2: " + f.length());
8  pw.flush();
9  System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



The second entry joins the buffer; the file is still 0 bytes.

Step 5 / 11

VARIABLES

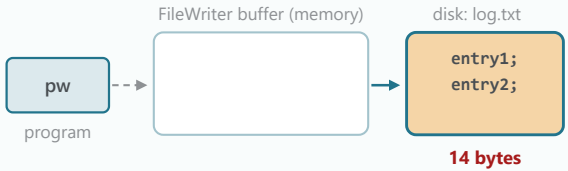
`buffer = "entry1`

OUTPUT

```
0: 0
1: 0
2: 0
```

Buffer and flush: Step by Step

```
1  File f = new File("log.txt");
2  PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
4  pw.print("entry1;");
5  System.out.println("1: " + f.length());
6  pw.print("entry2;");
7  System.out.println("2: " + f.length());
▶  pw.flush();
9  System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



`flush()` empties the buffer to disk: 14 bytes written.

Step 6 / 11

VARIABLES

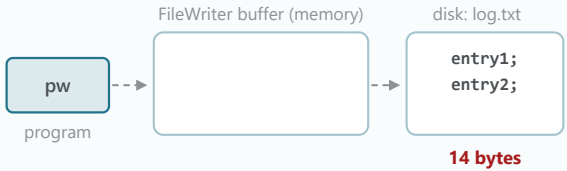
`buffer = ""`

OUTPUT

```
0: 0
1: 0
2: 0
```

Buffer and flush: Step by Step

```
1  File f = new File("log.txt");
2  PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
4  pw.print("entry1;");
5  System.out.println("1: " + f.length());
6  pw.print("entry2;");
7  System.out.println("2: " + f.length());
8  pw.flush();
▶  System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



The length is now 14.

Step 7 / 11

VARIABLES

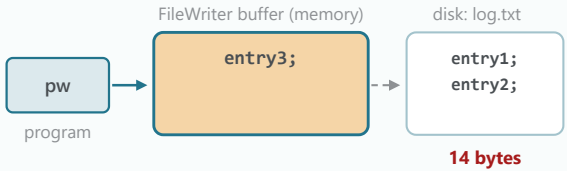
`f.length() = 14`

OUTPUT

```
0: 0
1: 0
2: 0
3: 14
```

Buffer and flush: Step by Step

```
1  File f = new File("log.txt");
2  PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
4  pw.print("entry1;");
5  System.out.println("1: " + f.length());
6  pw.print("entry2;");
7  System.out.println("2: " + f.length());
8  pw.flush();
9  System.out.println("3: " + f.length());
▶ pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



The third entry goes into the buffer.

Step 8 / 11

VARIABLES

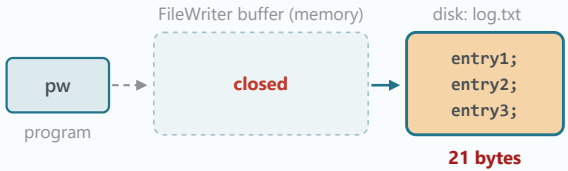
`buffer = "entry3`

OUTPUT

```
0: 0
1: 0
2: 0
3: 14
```

Buffer and flush: Step by Step

```
1  File f = new File("log.txt");
2  PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
4  pw.print("entry1;");
5  System.out.println("1: " + f.length());
6  pw.print("entry2;");
7  System.out.println("2: " + f.length());
8  pw.flush();
9  System.out.println("3: " + f.length());
10 pw.print("entry3;");
▶  pw.close();
12 System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



`close()` flushes first, then closes the file.

Step 9 / 11

VARIABLES

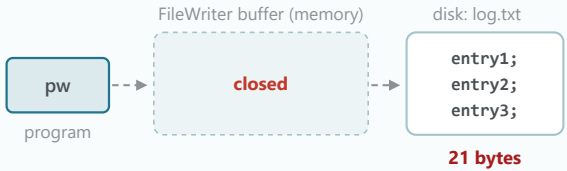
`pw = closed`

OUTPUT

```
0: 0
1: 0
2: 0
3: 14
```

Buffer and flush: Step by Step

```
 1 File f = new File("log.txt");
 2 PrintWriter pw = new PrintWriter(new FileWriter(f));
 3 System.out.println("0: " + f.length());
 4 pw.print("entry1;");
 5 System.out.println("1: " + f.length());
 6 pw.print("entry2;");
 7 System.out.println("2: " + f.length());
 8 pw.flush();
 9 System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
▶ System.out.println("4: " + f.length());
13 pw.print("entry4;");   // closed: silently ignored
14 System.out.println("error? " + pw.checkError());
```



The file is 21 bytes. Without `close`, the last entry could have been lost.

Step 10 / 11

VARIABLES

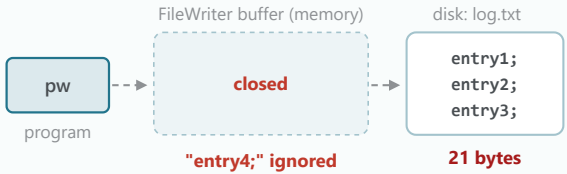
`f.length() = 21`

OUTPUT

```
0: 0
1: 0
2: 0
3: 14
4: 21
```

Buffer and flush: Step by Step

```
1  File f = new File("log.txt");
2  PrintWriter pw = new PrintWriter(new FileWriter(f));
3  System.out.println("0: " + f.length());
4  pw.print("entry1;");
5  System.out.println("1: " + f.length());
6  pw.print("entry2;");
7  System.out.println("2: " + f.length());
8  pw.flush();
9  System.out.println("3: " + f.length());
10 pw.print("entry3;");
11 pw.close();
12 System.out.println("4: " + f.length());
▶  pw.print("entry4;");   // closed: silently ignored
▶  System.out.println("error? " + pw.checkError());
```



`PrintWriter` never throws; writing to a closed stream fails **silently**. Only `checkError()` reveals the error.

Step 11 / 11

VARIABLES

`pw.checkError() = true`

OUTPUT

```
0: 0
1: 0
2: 0
3: 14
4: 21
error? true
```

BufferedInputStream and BufferedReader

```
try {
  FileInputStream fis = new FileInputStream("file.txt");
  BufferedInputStream bis = new BufferedInputStream(fis);
  int b;
  while ((b = bis.read()) != -1) {
    System.out.print((char) b);
  }
  bis.close();     // also closes fis
} catch (IOException e) {
  System.err.println("File error: " + e.getMessage());
}
```

OUTPUT: BYTE BY BYTE

```
Java I/O
Buffer
```

```
try {
  FileReader fr = new FileReader("file.txt");
  BufferedReader br = new BufferedReader(fr);
  String line;
  while ((line = br.readLine()) != null) {
    System.out.println(line);
  }
  br.close();     // also closes fr
} catch (IOException e) {
  System.err.println("File error: " + e.getMessage());
}
```

OUTPUT: LINE BY LINE

```
Java I/O
Buffer
```

How Much Does Buffering Help?

```java
public class BufferTiming {
  static long time(InputStream in) throws IOException {
    long start = System.nanoTime();
    while (in.read() != -1) { }          // 1 byte at a time
    in.close();
    return (System.nanoTime() - start) / 1_000_000;
  }
  public static void main(String[] args) throws IOException {
    File f = new File("big.dat");
    try (FileOutputStream fos = new FileOutputStream(f)) {
      fos.write(new byte[2 * 1024 * 1024]);   // 2 MB
    }
    long t1 = time(new FileInputStream(f));
    long t2 = time(new BufferedInputStream(new FileInputStream(f)));
    System.out.println("Unbuffered: " + t1 + " ms");
    System.out.println("Buffered  : " + t2 + " ms");
    f.delete();
  }
}
```

OUTPUT (SAMPLE RUN)

```
Unbuffered: 3151 ms
Buffered  : 94 ms
```

- The same 2 MB file, read byte by byte twice.
- Unbuffered: 2 097 152 `read()` calls = that many **system calls**.
- Buffered: one system call per 8 KB, 256 in all. **Tens of times** faster.
- Times vary by machine; the ratio stays similar.

java.io.FileInputStream

- Extends `InputStream`; reads raw **bytes** straight from a file. Low-level: for non-text data such as images and audio.
- `read()`: one byte (0–255), **`-1`** at end of file. `read(byte[] b)`: reads up to `b.length` bytes and returns **how many** it read.
- Awkward for text (multi-byte characters such as ü can be split); character classes such as `FileReader` are a better fit.

```
public class ChunkRead {
  public static void main(String[] args) throws IOException {
    try (FileInputStream fis = new FileInputStream("file.txt")) {
      byte[] buf = new byte[4];
      int n;
      while ((n = fis.read(buf)) != -1) {
        System.out.println(n + ": " + new String(buf, 0, n));
      }
    }
  }
}
```

OUTPUT

```
4: Hell
4: o, J
4: ava!
```

file.txt: “Hello, Java!”, 12 bytes, read in chunks of 4. `new String(buf, 0, n)` converts only the part that was read.

java.io.ByteArrayInputStream

- Extends `InputStream`; reads from a **byte array** in memory. No physical device is involved, so it is **fast**.
- Read-only; its counterpart for writing to memory is `ByteArrayOutputStream`. With large data, memory use can become a problem.

```
public class ByteArrayRead {
  public static void main(String[] args) throws IOException {
    byte[] arr = {74, 97, 118, 97};           // "Java"
    ByteArrayInputStream bais = new ByteArrayInputStream(arr);
    int b;
    while ((b = bais.read()) != -1) {
      System.out.print(b + "=" + (char) b + " ");
    }
    System.out.println();
    // Writing to memory: ByteArrayOutputStream
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    baos.write(arr);
    baos.write('!');
    byte[] result = baos.toByteArray();
    System.out.println(result.length + " bytes: " + baos);
  }
}
```

OUTPUT

```
74=J 97=a 118=v 97=a
5 bytes: Java!
```

StringBufferInputStream: Deprecated

- `java.io.StringBufferInputStream` has been **deprecated** since Java 1.1; avoid it.
- It does **not convert** characters to bytes correctly (only the low 8 bits). Use `ByteArrayInputStream` + `getBytes(UTF_8)` or a `StringReader` instead.

```
public class OldStream {
  @SuppressWarnings("deprecation")
  public static void main(String[] args) throws IOException {
    InputStream old = new StringBufferInputStream("$€");
    int b;
    while ((b = old.read()) != -1) {
      System.out.print(b + " ");
    }
    System.out.println();
    byte[] right = "$€".getBytes("UTF-8");
    InputStream fresh = new ByteArrayInputStream(right);
    while ((b = fresh.read()) != -1) {
      System.out.print(b + " ");
    }
    System.out.println();
  }
}
```

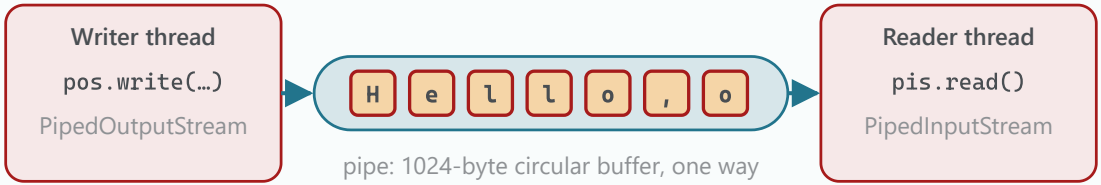
OUTPUT

```
95 172
197 159 226 130 172
```

`ş` (351) → 95 = `_`, `€` → 172. Correct UTF-8: 197 159 226 130 172.

PipedInputStream and PipedOutputStream

- Derived from `InputStream` / `OutputStream`; they let two **threads** pass data safely.
- One thread **writes** with a `PipedOutputStream`, the other **reads** it with a `PipedInputStream`. The ends are connected with `new PipedOutputStream(pis)`.
- Reads and writes are synchronized: the reader **waits** when the pipe is empty, the writer when it is full.
- Data flows **one way** only; two-way traffic needs two pipes.
- Using both ends in the same thread can cause a deadlock.



Pipe Example

```
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
```

```
// Reader thread
Thread reader = new Thread(() -> {
  try {
    int b;
    while ((b = pis.read()) != -1)
      System.out.print((char) b);
    System.out.println();
    pis.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
});
```

```
// Writer thread
Thread writer = new Thread(() -> {
  try {
    pos.write("Hello, pipe!".getBytes());
    pos.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
});
reader.start();
writer.start();
writer.join();
reader.join();
```

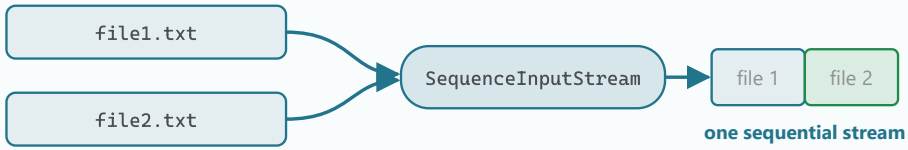
OUTPUT

```
Hello, pipe!
```

- `start()` starts a thread; `join()` waits for it to finish.
- When the writer calls `pos.close()`, the reader's `read()` returns −1 and the loop ends.

java.io.SequenceInputStream

- Joins several `InputStream`s into one **sequential** input stream: when the first ends (−1), it moves on to the second.
- Merges data from different sources into one stream. If a stream fails, an `IOException` is thrown and the next one is not read.




```
try {
  FileInputStream is1 = new FileInputStream("file1.txt");
  FileInputStream is2 = new FileInputStream("file2.txt");
  SequenceInputStream sis = new SequenceInputStream(is1, is2);
  int b;
  while ((b = sis.read()) != -1)
    System.out.print((char) b);
  sis.close();     // also closes is1, is2
} catch (IOException e) {
  System.err.println("Read error: " + e.getMessage());
}
```

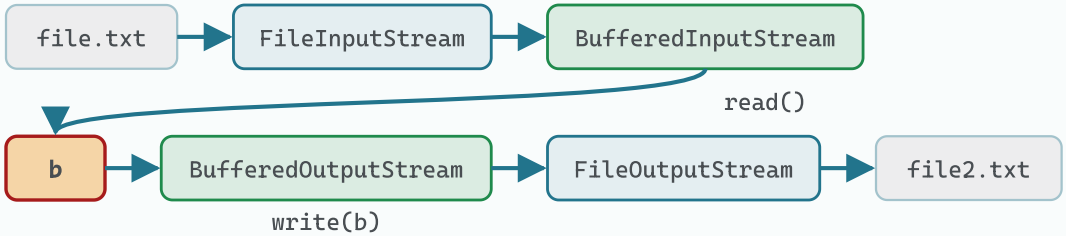
OUTPUT

```
First file.
Second file.
```

`sis.close()` closes the inner streams too.

Copying a File Byte by Byte

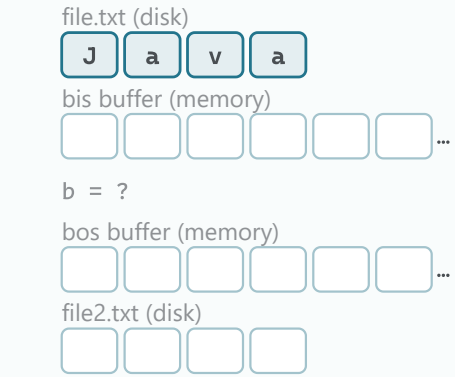
- Writes every byte read from the source file to the target: copies **any kind** of file (text, image, `.exe`) intact.
- `BufferedInputStream` and `BufferedOutputStream` speed up I/O by reading and writing through a buffer in bulk.
- `bis.read()` reads one byte and returns −1 at end of file. `bos.write(b)` writes that byte to the output file.
- Keep the value in an `int`, so that −1 can be told apart from the byte value 255.



The copy loop is a single line: `while ((b = bis.read()) != -1) bos.write(b);`

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
▶      File src = new File("file.txt");
▶      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
14       bos.write(b);     // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



The two `File` objects only hold paths.

Step 1 / 15

OUTPUT

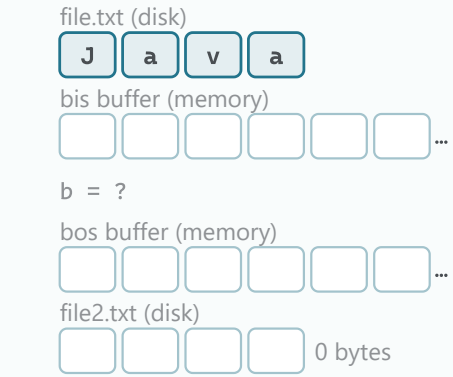
VARIABLES

`src` = **"file.txt"**

`dst` = **"file2.txt"**

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
▶      FileInputStream fis = new FileInputStream(src);
▶      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



`FileOutputStream` **creates** the target (truncating it if it exists): 0 bytes.

Step 2 / 15

OUTPUT

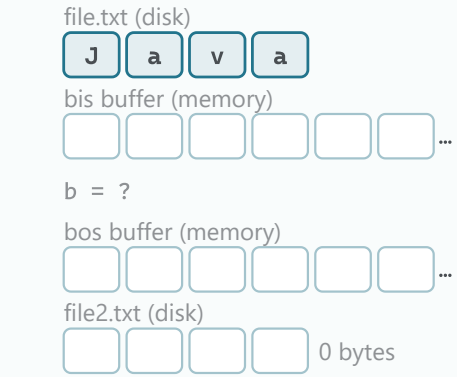
VARIABLES

`src = "file.txt"`

`dst = "file2.txt"`

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
▶      BufferedInputStream bis = new BufferedInputStream(fis);
▶      BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



Both streams are wrapped in buffered streams; the buffers (8192 bytes) are empty.

Step 3 / 15

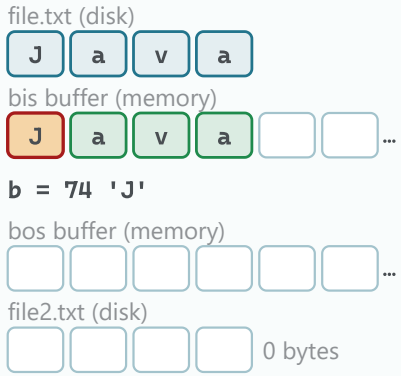
OUTPUT

VARIABLES

b = ?

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
▶      while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



First `read()`: the buffer is empty, so all 4 bytes come from disk **in one go**; the first byte, 74 ('J'), is returned.

Step 4 / 15

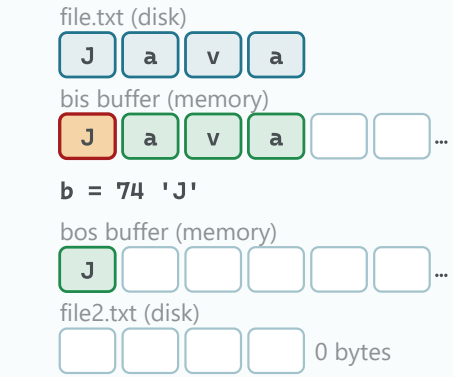
OUTPUT

VARIABLES

b = 74

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
▶        bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



`write` puts the byte in the output **buffer**; `file2.txt` is still 0 bytes.

Step 5 / 15

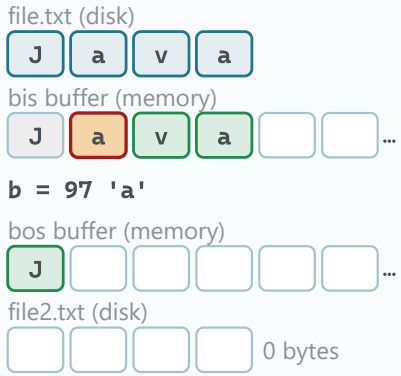
OUTPUT

VARIABLES

b = **74**

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
▶      while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



No disk access this time: the byte comes from the buffer (97 = 'a').

Step 6 / 15

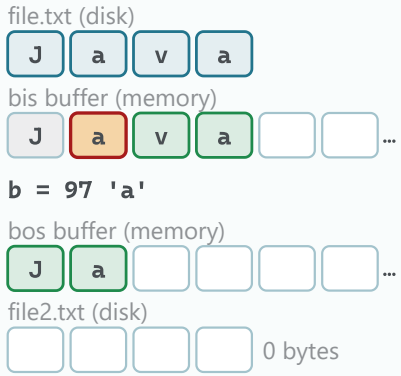
OUTPUT

VARIABLES

b = 97

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
▶        bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



The output buffer now holds 2 bytes.

Step 7 / 15

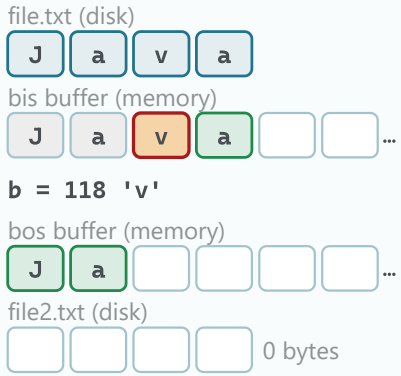
OUTPUT

VARIABLES

b = **97**

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
▶      while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



No disk access this time: the byte comes from the buffer (118 = 'v').

Step 8 / 15

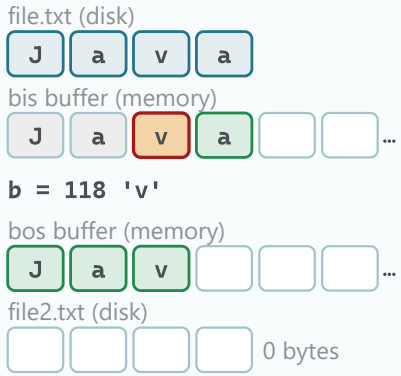
OUTPUT

VARIABLES

b = 118

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
▶        bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



The output buffer now holds 3 bytes.

Step 9 / 15

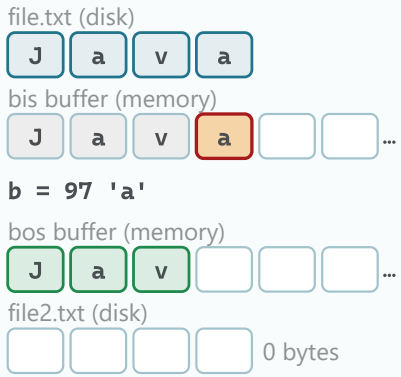
OUTPUT

VARIABLES

b = **118**

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
▶      while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



No disk access this time: the byte comes from the buffer (97 = 'a').

Step 10 / 15

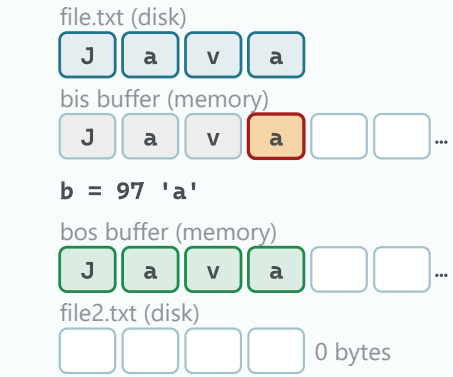
OUTPUT

VARIABLES

b = 97

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
▶        bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



The output buffer now holds 4 bytes.

Step 11 / 15

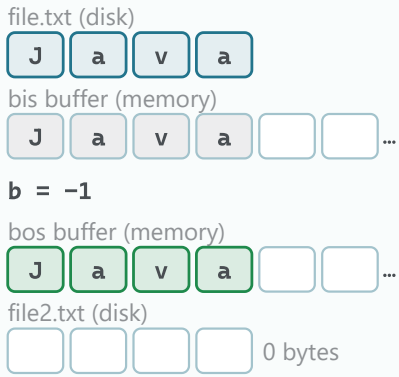
OUTPUT

VARIABLES

b = 97

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
▶      while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



The buffer is drained; a second disk read reports end of file: `read()` returns **−1** and the loop ends.

Step 12 / 15

OUTPUT

VARIABLES

b = −1

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
▶      bos.close();
16     bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```

file.txt (disk)

J	a	v	a

bis buffer (memory)

J	a	v	a			…

b = −1

bos buffer (memory)

						…

file2.txt (disk)

J	a	v	a

`close()` flushes first: 4 bytes go to disk **in one write**, then `fos` closes.

Step 13 / 15

OUTPUT

VARIABLES

b = **−1**

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
▶      bis.close();
17     System.out.println(dst.length() + " bytes copied");
18   }
19 }
```

file.txt (disk)

J	a	v	a

bis buffer (memory)

J	a	v	a			…

b = −1

bos buffer (memory)

						…

file2.txt (disk)

J	a	v	a

The input streams are closed too.

Step 14 / 15

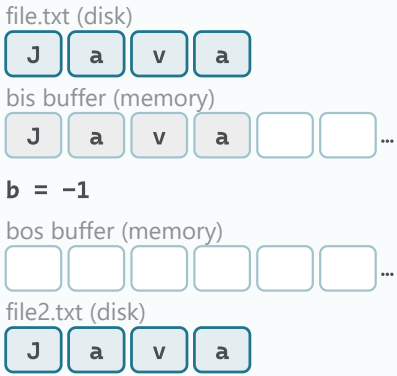
OUTPUT

VARIABLES

b = **−1**

Byte-by-Byte Copy: Step by Step

```
1  import java.io.*;
2
3  public class ByteCopy {
4    public static void main(String[] args) throws IOException {
5      File src = new File("file.txt");
6      File dst = new File("file2.txt");
7      FileInputStream fis = new FileInputStream(src);
8      FileOutputStream fos = new FileOutputStream(dst);
9      BufferedInputStream bis = new BufferedInputStream(fis);
10     BufferedOutputStream bos = new BufferedOutputStream(fos);
11     int b;
12     // Read a byte; -1 at end of file.
13     while ((b = bis.read()) != -1)
14       bos.write(b);    // write it out
15     bos.close();
16     bis.close();
▶      System.out.println(dst.length() + " bytes copied");
18   }
19 }
```



2 disk reads (the second only reports end of file) and 1 write; unbuffered, it would take 5 reads and 4 writes.

Step 15 / 15

OUTPUT

```
4 bytes copied
```

VARIABLES

b = **−1**

Copying Block by Block

```
public class BlockCopy {
  public static void main(String[] args) throws IOException {
    // Data is read in 16K chunks
    byte[] buf = new byte[1024 * 16];
    File src = new File("file.txt");
    File dst = new File("file2.txt");
    FileInputStream fis = new FileInputStream(src);
    FileOutputStream fos = new FileOutputStream(dst);
    BufferedInputStream bis = new BufferedInputStream(fis);
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    int size;
    // Read up to 16K; -1 at end of file.
    while ((size = bis.read(buf)) > -1)
      bos.write(buf, 0, size);   // write the chunk
    bos.close();
    bis.close();
    System.out.println(dst.length() + " bytes copied");
  }
}
```

OUTPUT (40,000 BYTES)

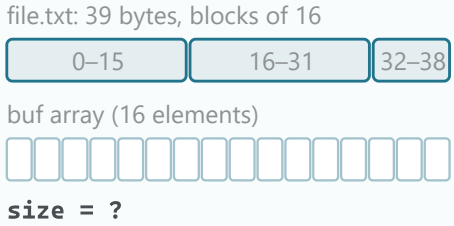
```
40000 bytes copied
```

- Reading in bulk **improves** performance.
- `read(buf)` returns the **number** of bytes read.
- `write(buf, 0, size)` writes only the first `size` bytes.

`new byte[1024 * 16]`: data is processed in 16 KB chunks, so memory use stays constant however large the file is.

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
▶      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```



A 16-byte array (8–16 KB in real programs).

Step 1 / 13

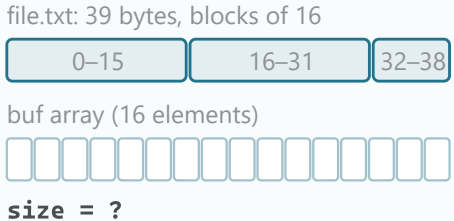
OUTPUT

VARIABLES

buf = **byte[16]**

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
▶      try (InputStream in = new FileInputStream("file.txt");
▶        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```



Both resources are opened together; both close when the `try` ends.

Step 2 / 13

OUTPUT

VARIABLES

`buf = byte[16]`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
▶        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```

file.txt: 39 bytes, blocks of 16



buf array (16 elements)



size = 16

`read(buf)` fills the array and returns 16.

Step 3 / 13

OUTPUT

VARIABLES

`size = 16`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
▶          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```

file.txt: 39 bytes, blocks of 16



buf array (16 elements)



size = 16

`write(buf, 0, 16)` : only the 16 bytes read are written.

Step 4 / 13

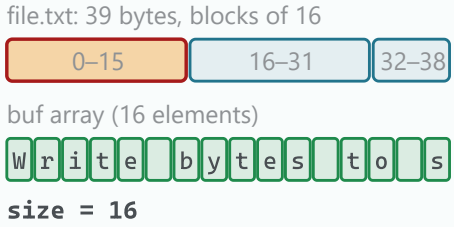
OUTPUT

VARIABLES

`size = 16`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
▶          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```



The count is printed.

Step 5 / 13

OUTPUT

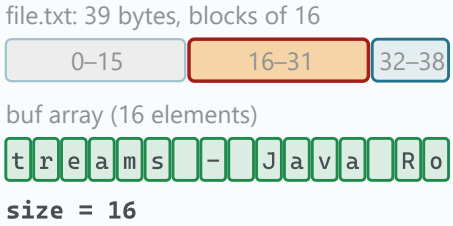
```
size = 16
```

VARIABLES

`size = 16`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
▶        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```



The next 16 bytes **overwrite** the same array.

Step 6 / 13

OUTPUT

```
size = 16
```

VARIABLES

`size = 16`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
▶          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```

file.txt: 39 bytes, blocks of 16



buf array (16 elements)



size = 16

`write(buf, 0, 16)` : only the 16 bytes read are written.

Step 7 / 13

OUTPUT

```
size = 16
```

VARIABLES

`size = 16`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
▶          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```

file.txt: 39 bytes, blocks of 16



buf array (16 elements)



size = 16

The count is printed.

Step 8 / 13

OUTPUT

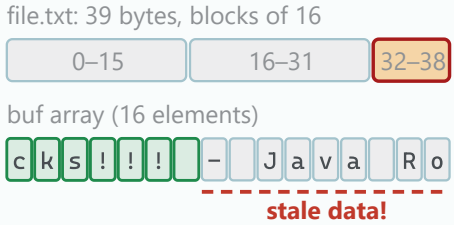
```
size = 16
size = 16
```

VARIABLES

size = **16**

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
▶        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```



`size = 7`

Only 7 bytes are left: the first 7 elements are refreshed, the other 9 are **left over** from the previous block!

Step 9 / 13

OUTPUT

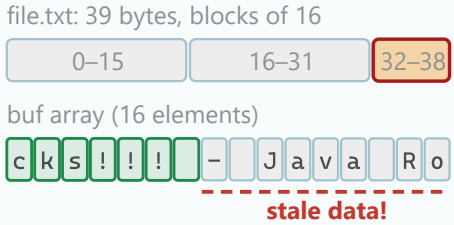
```
size = 16
size = 16
```

VARIABLES

`size = 7`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
▶          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```



size = 7

`write(buf, 0, 7)` : only the 7 bytes read are written. With `write(buf)` , the 9 stale bytes would go too, and the copy would be 48 bytes.

Step 10 / 13

OUTPUT

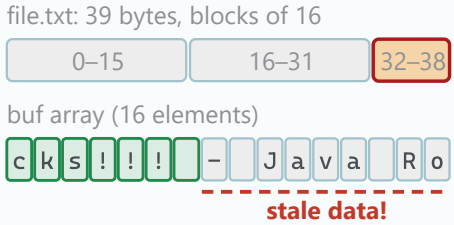
```
size = 16
size = 16
```

VARIABLES

`size = 7`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
▶          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```



size = 7

The count is printed.

Step 11 / 13

OUTPUT

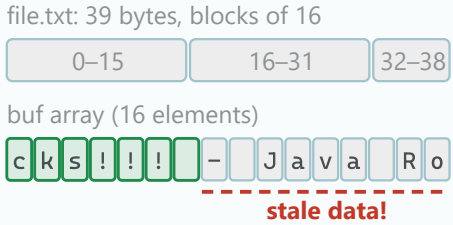
```
size = 16
size = 16
size = 7
```

VARIABLES

size = **7**

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
▶        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
13     System.out.println("Done: " + n + " bytes");
14   }
15 }
```



`size = -1`

End of file: −1 is returned, `> -1` is false, the loop ends.

Step 12 / 13

OUTPUT

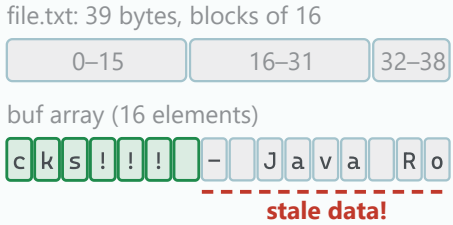
```
size = 16
size = 16
size = 7
```

VARIABLES

`size = -1`

Copying Block by Block, Step by Step

```
1  public class BlockTrace {
2    public static void main(String[] args) throws IOException {
3      byte[] buf = new byte[16];
4      try (InputStream in = new FileInputStream("file.txt");
5        OutputStream out = new FileOutputStream("file2.txt")) {
6        int size;
7        while ((size = in.read(buf)) > -1) {
8          out.write(buf, 0, size);
9          System.out.println("size = " + size);
10       }
11     }
12     long n = new File("file2.txt").length();
▶      System.out.println("Done: " + n + " bytes");
14   }
15 }
```



`size = -1`

The resources are closed; the copy matches the source: 16 + 16 + 7 = 39 bytes.

Step 13 / 13

OUTPUT

```
size = 16
size = 16
size = 7
Done: 39 bytes
```

VARIABLES

`size = -1` `n = 39`

Copying a File Line by Line

```
public class LineCopy {
  public static void main(String[] args) throws IOException {
    File src = new File("file.txt");
    File dst = new File("file2.txt");
    FileReader fr = new FileReader(src);
    BufferedReader br = new BufferedReader(fr);
    FileWriter fw = new FileWriter(dst);
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter pw = new PrintWriter(bw);
    String line;
    // read a line until end of file
    while ((line = br.readLine()) != null)
      pw.println(line);    // write the line
    pw.close();
    br.close();
    System.out.println("Copied: " + dst.getName());
  }
}
```

OUTPUT

```
Copied: file2.txt
```

- Uses character streams: **text** files only.
- `readLine()` drops the line terminator; `println` adds the system's own (Windows: `\r\n`, Linux: `\n`). The copy may not be byte-identical.
- `pw.close()` closes the whole chain (`bw`, `fw`).

Downloading a Web Page

```
public class WebDownload {
  public static void main(String[] args) throws IOException {
    URL url = URI.create("https://sercankulcu.github.io/").toURL();
    BufferedInputStream bis = new BufferedInputStream(url.openStream());
    FileOutputStream fos = new FileOutputStream(new File("copy.html"));
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    for (int c = bis.read(); c != -1; c = bis.read()) {
      bos.write(c);
    }
    bis.close();
    bos.close();
    System.out.println("copy.html: " + new File("copy.html").length() + " bytes");
  }
}
```

OUTPUT (SIZE DEPENDS ON THE PAGE)

```
copy.html: 14283 bytes
```

- The source is the **network**, not a file: `url.openStream()` returns an `InputStream`, and the copy loop is the same.
- `new URL(String)` is deprecated since Java 20; use `URI.create(…).toURL()`.
- The `for` loop: read; if not −1, write; read again.

Shortcuts: transferTo and Files.copy

```
public class QuickCopy {
  public static void main(String[] args) throws IOException {
    // Java 9+: transferTo
    try (InputStream in = new FileInputStream("file.txt");
      OutputStream out = new FileOutputStream("file2.txt")) {
      long n = in.transferTo(out);
      System.out.println(n + " bytes transferred");
    }
    // Java 7+: Files.copy
    Files.copy(Path.of("file.txt"), Path.of("file3.txt"),
         StandardCopyOption.REPLACE_EXISTING);
    System.out.println(Files.size(Path.of("file3.txt")) + " bytes copied");
  }
}
```

OUTPUT

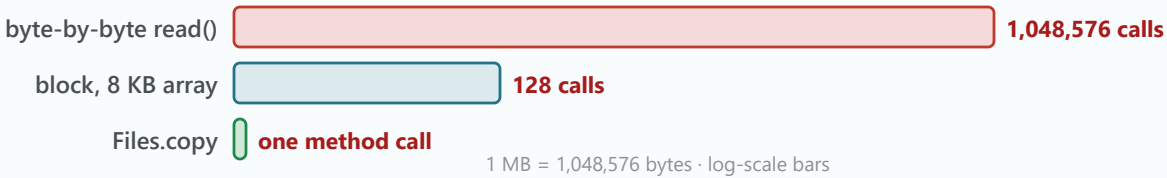
```
4 bytes transferred
4 bytes copied
```

Library methods copy in blocks internally. If the target exists, `Files.copy` throws `FileAlreadyExistsException` unless you pass `REPLACE_EXISTING`.

COPYING FILES

Ways to Copy

Method	Unit	Best for
Byte by byte (`read()` / `write(int)`)	1 byte	learning; very slow unbuffered
Block by block (`read(byte[])`)	8–16 KB	any file, fast
Line by line (`readLine` / `println`)	1 line	text only; when processing lines
`in.transferTo(out)` (Java 9+)	block	one-line copy between two streams
`Files.copy(src, dst)` (Java 7+)	block	shortest file-to-file route



Whatever the method: work **buffered**, **close** your streams (preferably with try-with-resources) and handle `IOException` .

File Access: Sequential and Random

Sequential access

- Each read or write continues from the **next** position; going back needs special support (e.g. `mark` / `reset`, within a limited buffer); you cannot jump around freely.
- Used when the file's structure is unknown, or when making a copy.
- All streams (`FileInputStream`, `BufferedReader`...) are sequential.

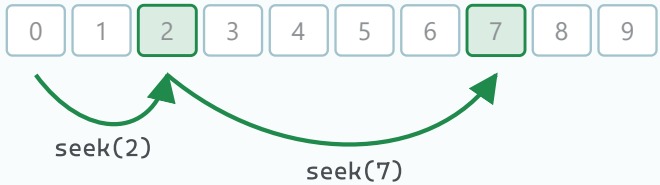
Random access

- Move to **any position** in the file and read or write there.
- Works when the **size and position** of records are known: record k = k × record size.
- Handy for reading one record or **updating it in place**.

Sequential: like a tape, start to end



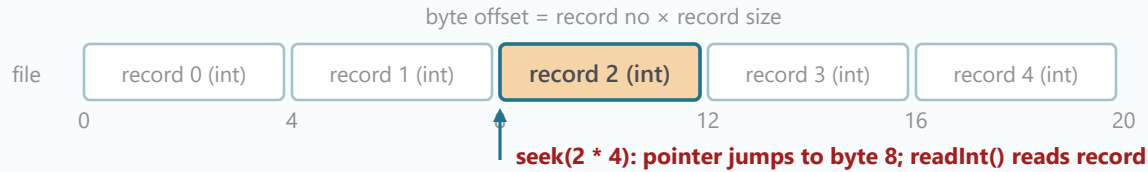
Random: like a vinyl record, jump anywhere



RANDOM ACCESS

java.io.RandomAccessFile

Member	Purpose
`RandomAccessFile(String name, String mode)`	mode `"r"`: read, `"rw"`: read and write
`seek(long pos)`	moves the file pointer to the given position
`getFilePointer()`	pointer's offset from the start (bytes)
`length()`	file length (bytes)
`read()` / `write(int b)`	reads (returned as int) / writes one byte
`readInt()`, `writeInt(int)`, `readDouble()`...	read/write primitives in binary (int: 4 bytes)
`close()`	releases system resources



Unlike streams, one object both reads and writes; it is not part of the `InputStream` / `OutputStream` hierarchy.

Sequential and Random Access: Code

```java
public class RandomAccess {
  public static void main(String[] args) throws IOException {
    // Sequential access
    FileInputStream seq = new FileInputStream("data");
    int first = seq.read();
    seq.close();
    System.out.println("First byte: " + first);
    // Random access
    RandomAccessFile raf = new RandomAccessFile("data", "rw");
    // Go to 100 bytes from the start
    raf.seek(100);
    // Read a 4-byte int from position 100
    int value = raf.readInt();
    System.out.println("int at 100: " + value);
    raf.close();
  }
}
```

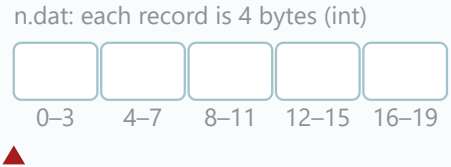
OUTPUT

```
First byte: 1
int at 100: 256
```

`seek(100)` moves the pointer to position 100, the **101st** byte (positions start at 0). The 4 bytes there are `00 00 01 00`, so `readInt()` returns 256. Here `data` is a 124-byte data file.

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
▶      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                // record 4
14     f.writeInt(50);               // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```



`"rw"`: the file is created if missing. The pointer is at 0.

Step 1 / 13

OUTPUT

VARIABLES

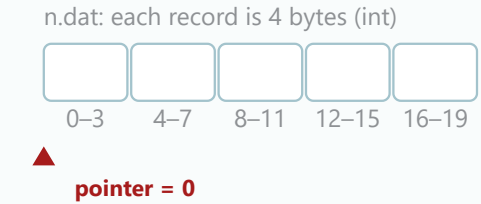
—

CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                // record 4
14     f.writeInt(50);               // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```



Five grades to write.

Step 2 / 13

OUTPUT

VARIABLES

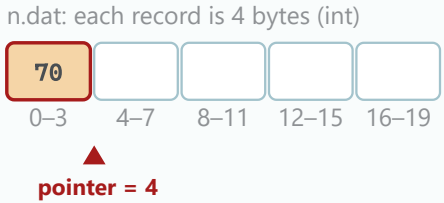
`grades = [70, 85, 90, 45, 60]`

CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
▶      for (int n : grades)
▶        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                  // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                  // record 4
14     f.writeInt(50);                 // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```



Each `writeInt` writes 4 bytes and advances the pointer by 4.

Step 3 / 13

OUTPUT

VARIABLES

`n = 70`

CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
▶      for (int n : grades)
▶        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                  // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                  // record 4
14     f.writeInt(50);                 // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



Loop done: 5 × 4 = 20 bytes; the pointer is at the end (20).

Step 4 / 13

OUTPUT

VARIABLES

`n = 60`

CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
▶      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                 // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                 // record 4
14     f.writeInt(50);                // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



`length()` = 20.

Step 5 / 13

OUTPUT

```
Length: 20
```

VARIABLES

—

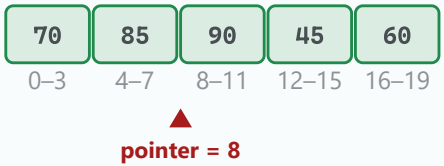
CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
▶      f.seek(2 * 4);                 // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                 // record 4
14     f.writeInt(50);                // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



The 3rd record (index 2) starts at byte 8: we **jump** straight there without reading the others.

Step 6 / 13

OUTPUT

```
Length: 20
```

VARIABLES

—

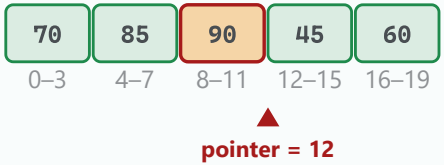
CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                // go to record 3
▶      System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                // record 4
14     f.writeInt(50);               // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



`readInt()` reads bytes 8–11: 90. The pointer moves to 12.

Step 7 / 13

OUTPUT

```
Length: 20
Grade 3: 90
```

VARIABLES

—

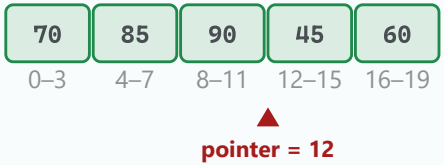
CALL STACK

- `f.readInt()`
- `main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
▶      System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                // record 4
14     f.writeInt(50);               // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



`getFilePointer()` = 12.

Step 8 / 13

OUTPUT

```
Length: 20
Grade 3: 90
Position: 12
```

VARIABLES

—

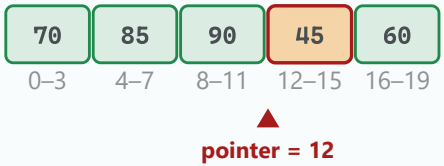
CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
▶      f.seek(3 * 4);                // record 4
14     f.writeInt(50);               // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



The 4th record also starts at 12; the pointer is already there.

Step 9 / 13

OUTPUT

```
Length: 20
Grade 3: 90
Position: 12
```

VARIABLES

—

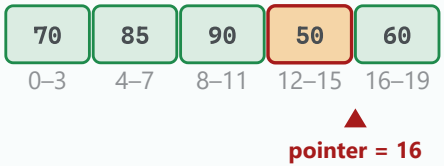
CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                // record 4
▶      f.writeInt(50);               // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



50 replaces 45 **in place**; the rest of the file is untouched. A sequential stream can't do this.

Step 10 / 13

OUTPUT

```
Length: 20
Grade 3: 90
Position: 12
```

VARIABLES

—

CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                // record 4
14     f.writeInt(50);               // 50 replaces 45
▶      f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



Back to the start.

Step 11 / 13

OUTPUT

```
Length: 20
Grade 3: 90
Position: 12
```

VARIABLES

`total = 0`

CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                // record 4
14     f.writeInt(50);               // 50 replaces 45
15     f.seek(0);
16     int total = 0;
▶      while (f.getFilePointer() < f.length())
▶        total += f.readInt();
19     System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



Read until the pointer reaches the length: 70 + 85 + 90 + 50 + 60 = 355.

Step 12 / 13

OUTPUT

```
Length: 20
Grade 3: 90
Position: 12
```

VARIABLES

`total = 355`

CALL STACK

`main()`

Random Access to Records, Step by Step

```
1  import java.io.*;
2
3  public class Records {
4    public static void main(String[] args) throws IOException {
5      RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
6      int[] grades = {70, 85, 90, 45, 60};
7      for (int n : grades)
8        f.writeInt(n);              // 4 bytes each
9      System.out.println("Length: " + f.length());
10     f.seek(2 * 4);                  // go to record 3
11     System.out.println("Grade 3: " + f.readInt());
12     System.out.println("Position: " + f.getFilePointer());
13     f.seek(3 * 4);                  // record 4
14     f.writeInt(50);                 // 50 replaces 45
15     f.seek(0);
16     int total = 0;
17     while (f.getFilePointer() < f.length())
18       total += f.readInt();
▶      System.out.println("Total: " + total);
20     f.close();
21   }
22 }
```

n.dat: each record is 4 bytes (int)



The result is printed; then the file is closed.

Step 13 / 13

OUTPUT

```
Length: 20
Grade 3: 90
Position: 12
Total: 355
```

VARIABLES

`total = 355`

CALL STACK

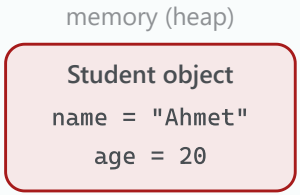
`main()`

Serialization

- **Serialization** turns an object's state (its field values) into a **byte sequence**, so the object can be written to a file or sent over the network. The reverse is **deserialization**.
- For its objects to be serializable, a class must **implement** `java.io.Serializable` (a marker interface with no methods).
- Declaring a version ID, `serialVersionUID`, is **recommended**; otherwise one is computed from the class structure, and old files become unreadable once the class changes.
- `static` fields (they belong to the class) and fields marked `transient` are **not serialized**; use this to leave out fields such as passwords.

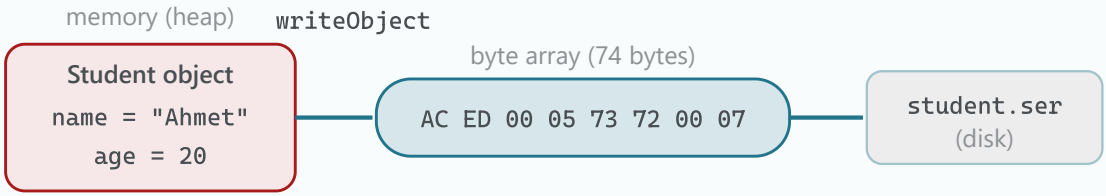
Uses: game saves, caches, sending objects to another JVM over the network. For long-term or cross-language storage, text formats such as JSON are usually preferred.

The Serialization Flow



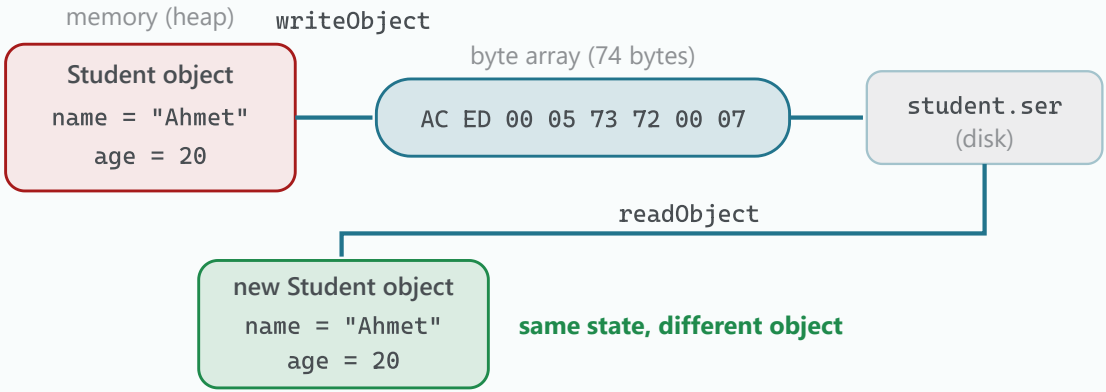
A `Student` object lives in memory.

The Serialization Flow



`ObjectOutputStream.writeObject` turns the object into bytes; the file starts with the magic number `AC ED`.

The Serialization Flow



`ObjectInputStream.readObject` builds a **new** object from the bytes: same values, different object.

A Serializable Class

```
import java.io.Serializable;

class Student implements Serializable {
  private static final long serialVersionUID = 1L;
  private String name;
  private int age;

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

  public String toString() {
    return "Student [name=" + name + ", age=" + age + "]";
  }
}
```

- `implements Serializable`: a **marker** interface with no methods.
- `serialVersionUID`: the class's version number. If writer and reader versions differ: `InvalidClassException`.
- Fields must be serializable too; otherwise `NotSerializableException`.
- `readObject` returns `Object`: a `(Student)` cast is needed.

SERIALIZATION

Writing and Reading Back an Object

```
// Serialize
try (ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("student.ser"))) {
  Student student = new Student("Ahmet", 20);
  out.writeObject(student);
  System.out.println("Serialization succeeded.");
} catch (IOException e) {
  e.printStackTrace();
}
System.out.println("student.ser: " + new File("student.ser").length() + " bytes");
// Deserialize
try (ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("student.ser"))) {
  Student student = (Student) in.readObject();
  System.out.println("Student: " + student);
} catch (IOException | ClassNotFoundException e) {
  e.printStackTrace();
}
```

OUTPUT

```
Serialization succeeded.
student.ser: 74 bytes
Student: Student [name=Ahmet, age=20]
```

transient and static Fields

```
class Account implements Serializable {
  private static final long serialVersionUID = 1L;
  static int count;
  private String name;
  private transient String password;   // not written

  Account(String name, String password) {
    this.name = name;
    this.password = password;
  }

  public String toString() {
    return "Account [name=" + name + ", password=" + password + "]";
  }
}
```

- `transient`: the field is **skipped** during serialization; when read back it gets its default value (`null`, 0, `false`).
- For fields that shouldn't or can't be stored: passwords, temporary caches, open files or connections.
- `static` fields belong to the class, not the object: they are never written to the file.

transient and static: Experiment

```
Account k = new Account("ayse", "secret12");
Account.count = 5;
ObjectOutputStream c = new ObjectOutputStream(
    new FileOutputStream("k.ser"));
c.writeObject(k);
c.close();
Account.count = 99;        // static: per class
ObjectInputStream g = new ObjectInputStream(
    new FileInputStream("k.ser"));
Account copy = (Account) g.readObject();
g.close();
System.out.println(copy);
System.out.println("count = " + Account.count);
System.out.println("same object? " + (copy == k));
```

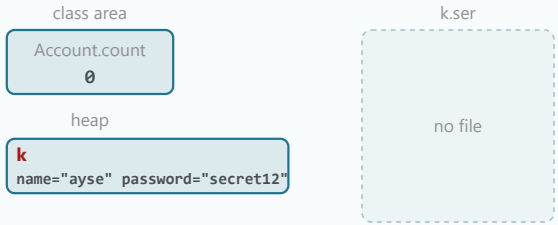
OUTPUT

```
Account [name=ayse,
password=null]
count = 99
same object? false
```

- `password` → `null`: not written.
- `count` → 99: the last value in memory.
- `==` is false: the copy is a **new** object.

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
5  c.writeObject(k);
6  c.close();
7  Account.count = 99;         // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



A new `Account` object on the heap: `name` and `password` are set. The `static` `count` lives in the class area: 0.

Step 1 / 11

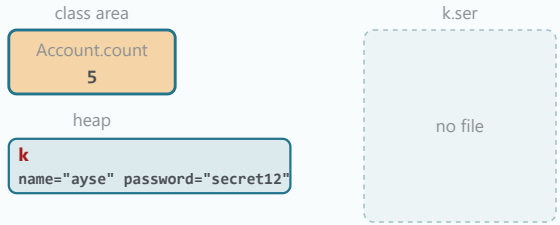
VARIABLES

`k = Account@1`

OUTPUT

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
▶  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
5  c.writeObject(k);
6  c.close();
7  Account.count = 99;         // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



`count` belongs to the **class**, not the object; there is only one copy.

Step 2 / 11

VARIABLES

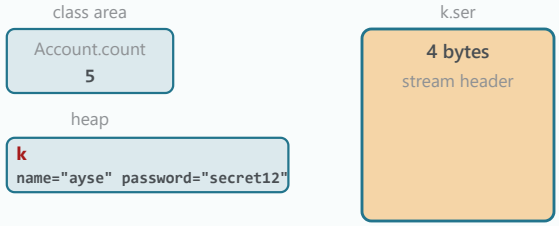
`k = Account@1`

`Account.count = 5`

OUTPUT

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
▶  ObjectOutputStream c = new ObjectOutputStream(
▶      new FileOutputStream("k.ser"));
5  c.writeObject(k);
6  c.close();
7  Account.count = 99;        // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



The stream opens and immediately writes a 4-byte **stream header**.

Step 3 / 11

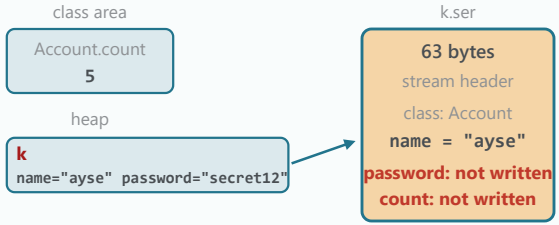
VARIABLES

`c = ObjectOutputStream`

OUTPUT

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
▶  c.writeObject(k);
6  c.close();
7  Account.count = 99;         // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



`writeObject` writes the class description and the fields: 63 bytes. `transient password` and `static count` are **not written**.

Step 4 / 11

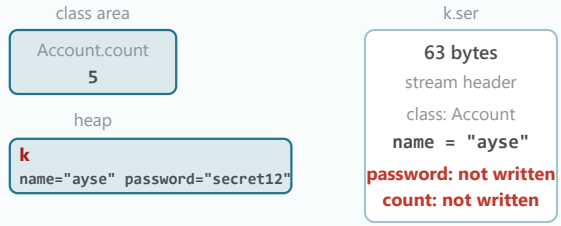
VARIABLES

`c = ObjectOutputStream`

OUTPUT

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
5  c.writeObject(k);
▶  c.close();
7  Account.count = 99;          // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



The stream is closed; the file stays at 63 bytes.

Step 5 / 11

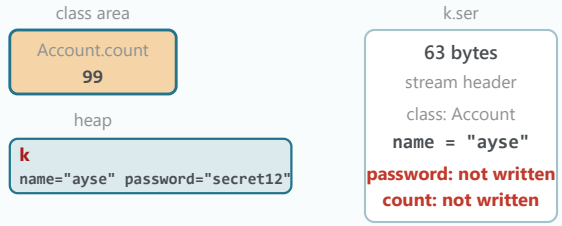
VARIABLES

`c = closed`

OUTPUT

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
5  c.writeObject(k);
6  c.close();
▶  Account.count = 99;          // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



The class's `count` becomes 99. The file has no `count`, so reading can't change it.

Step 6 / 11

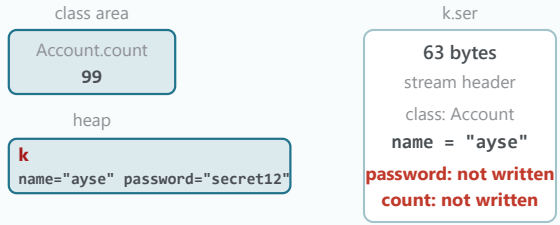
VARIABLES

`Account.count = 99`

OUTPUT

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
5  c.writeObject(k);
6  c.close();
7  Account.count = 99;        // static: per class
▶  ObjectInputStream g = new ObjectInputStream(
▶      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



The input stream opens and first checks the stream header.

Step 7 / 11

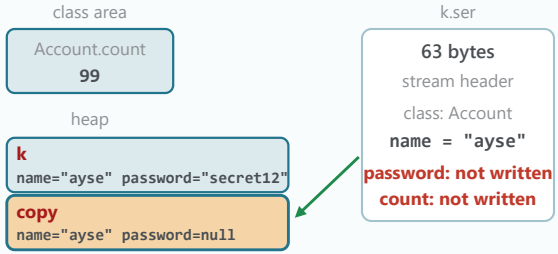
VARIABLES

`g = ObjectInputStream`

OUTPUT

Serialization: Writing and Reading Back, Step by Step

```
 1 Account k = new Account("ayse", "secret12");
 2 Account.count = 5;
 3 ObjectOutputStream c = new ObjectOutputStream(
 4     new FileOutputStream("k.ser"));
 5 c.writeObject(k);
 6 c.close();
 7 Account.count = 99;        // static: per class
 8 ObjectInputStream g = new ObjectInputStream(
 9     new FileInputStream("k.ser"));
▶ Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



A **new** object is built on the heap; the constructor is **not called**. `name` comes from the file; `password` gets the default, `null`.

Step 8 / 11

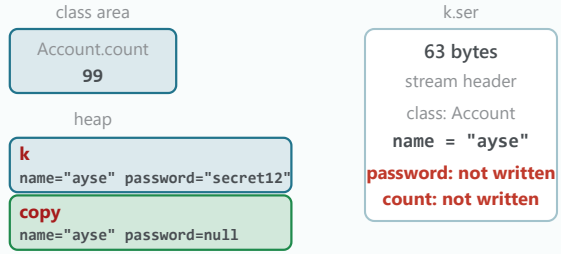
VARIABLES

`copy = Account@2`

OUTPUT

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
5  c.writeObject(k);
6  c.close();
7  Account.count = 99;          // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
▶ System.out.println(copy);
13 System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



`toString()` prints the fields of the object read back.

Step 9 / 11

VARIABLES

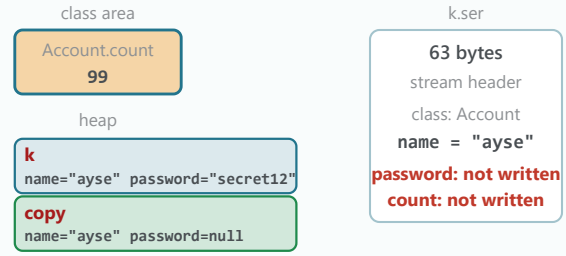
`copy = Account@2`

OUTPUT

```
Account [name=ayse, password=null]
```

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
5  c.writeObject(k);
6  c.close();
7  Account.count = 99;         // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
▶ System.out.println("count = " + Account.count);
14 System.out.println("same object? " + (copy == k));
```



`count` is the last value in memory: 99.

Step 10 / 11

VARIABLES

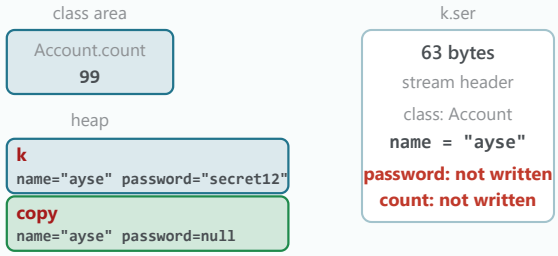
`Account.count = 99`

OUTPUT

```
Account [name=ayse, password=null]
count = 99
```

Serialization: Writing and Reading Back, Step by Step

```
1  Account k = new Account("ayse", "secret12");
2  Account.count = 5;
3  ObjectOutputStream c = new ObjectOutputStream(
4      new FileOutputStream("k.ser"));
5  c.writeObject(k);
6  c.close();
7  Account.count = 99;         // static: per class
8  ObjectInputStream g = new ObjectInputStream(
9      new FileInputStream("k.ser"));
10 Account copy = (Account) g.readObject();
11 g.close();
12 System.out.println(copy);
13 System.out.println("count = " + Account.count);
▶  System.out.println("same object? " + (copy == k));
```



Two separate objects: similar contents, different references.

Step 11 / 11

VARIABLES

`copy = = k: false`

OUTPUT

```
Account [name=ayse, password=null]
count = 99
same object? false
```

NIO (New I/O)

- Java NIO (`java.nio`, Java 1.4+) is a powerful API for advanced I/O: buffers, channels, selectors and file system integration.
- It offers more **performance** and **flexibility** than classic `java.io` streams.

Buffer

Memory areas that hold data temporarily.

Channel

Bidirectional connections that carry out I/O.

Selector

Manages many channels from **one thread**, watching for channels with pending input.

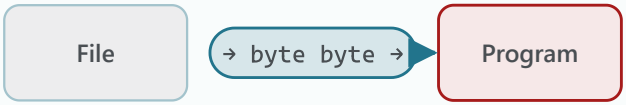
Path

Represents file and directory **paths** (Java 7+, NIO.2).

NIO.2 (`java.nio.file`, Java 7+) added the `Path` and `Files` classes for file operations.

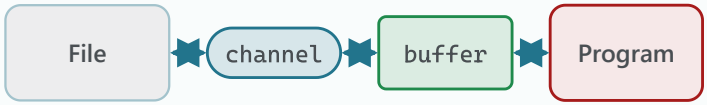
Stream or Channel?

Classic I/O: stream



one way, sequential, blocking

NIO: channel + buffer



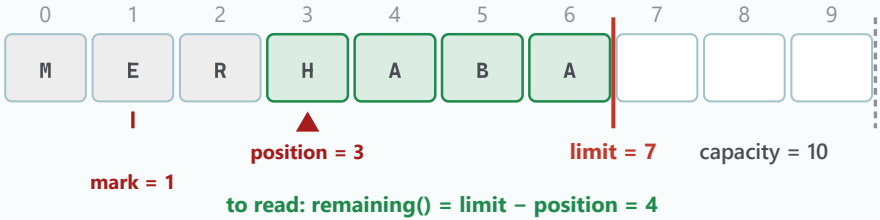
two-way, blocks, can be non-blocking

	Classic I/O (`java.io`)	**NIO (`java.nio`)**
Core idea	stream	channel + buffer
Direction	one way: input *or* output	a channel can be two-way
Unit	byte / character	a buffer-sized block
Waiting	always blocking	non-blocking mode available
Many connections	one thread per connection	one thread + selector
When?	simple file/console work	servers with many connections, large files

NIO Buffers

- A **buffer** is a fixed-size memory area that holds data temporarily (an array plus position bookkeeping).
- One class per primitive type: `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer`, `DoubleBuffer`.
- Create one with `ByteBuffer.allocate(1024)`, or wrap an existing array: `ByteBuffer.wrap(arr)`.

Property	Meaning
capacity	maximum number of elements; never changes
position	index of the next read or write
limit	first index that may **not** be read or written
mark	a position saved by `mark()`; `reset()` returns to it



Always: 0 ≤ mark ≤ position ≤ limit ≤ capacity.

ByteBuffer Example

```
public class BufferDemo {
  public static void main(String[] args) {
    // Create a 1 KB ByteBuffer
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    // Write data
    buffer.put("Hello from NIO".getBytes());
    System.out.println("after put  : " + buffer);
    // Switch to read mode
    buffer.flip();
    System.out.println("after flip : " + buffer);
    // Read the data
    while (buffer.hasRemaining()) {
      System.out.print((char) buffer.get());
    }
    System.out.println();
  }
}
```

OUTPUT

```
after put  :
java.nio.HeapByteBuffer[pos=14
lim=1024 cap=1024]
after flip :
java.nio.HeapByteBuffer[pos=0 lim=14
cap=1024]
Hello from NIO
```

- `put` writes; the position moves to 14.
- `flip()` : limit = 14, position = 0: **read** mode.
- `hasRemaining()` : is position < limit?

NIO

Buffers and Non-ASCII Characters

```
public class BufferUmlaut {
  public static void main(String[] args) {
    ByteBuffer b = ByteBuffer.allocate(64);
    b.put("Dünya".getBytes(StandardCharsets.UTF_8));
    b.flip();
    while (b.hasRemaining())
      System.out.print((char) b.get());      // wrong
    System.out.println();
    b.rewind();
    System.out.println(StandardCharsets.UTF_8.decode(b));  // right
  }
}
```

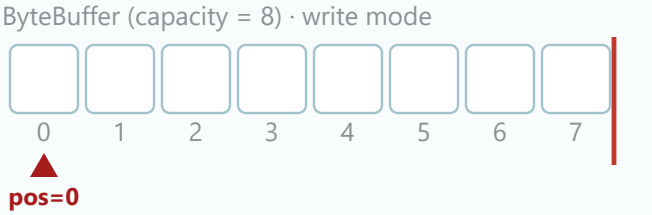
OUTPUT

```
D╠Enya
Dünya
```

`(char) buffer.get()` is only correct for ASCII text. “Dünya” (Turkish for “world”) is 6 bytes in UTF-8 (ü = `C3 BC`); casting bytes one by one to `char` mangles the ü. Decode text with `Charset.decode`.

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
▶      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```



Empty buffer: position 0, limit = capacity = 8.

Step 1 / 15

OUTPUT

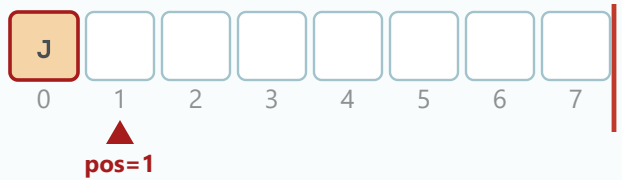
VARIABLES

`pos = 0` `lim = 8` `cap = 8`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
▶      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```

ByteBuffer (capacity = 8) · write mode



`put` writes at the position and advances it by 1.

Step 2 / 15

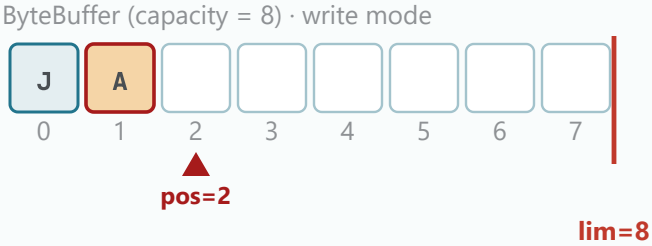
OUTPUT

VARIABLES

`pos = 1` `lim = 8` `cap = 8`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
▶      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```



Position 2.

Step 3 / 15

OUTPUT

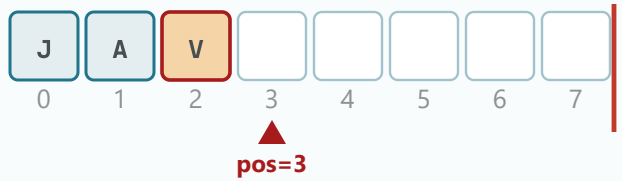
VARIABLES

`pos = 2` `lim = 8` `cap = 8`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
▶      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```

ByteBuffer (capacity = 8) · write mode



Position 3.

Step 4 / 15

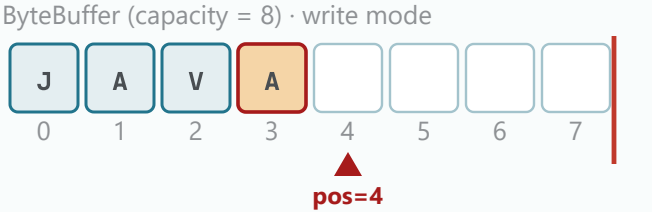
OUTPUT

VARIABLES

pos = 3 lim = 8 cap = 8

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
▶      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```



4 bytes written; position 4.

Step 5 / 15

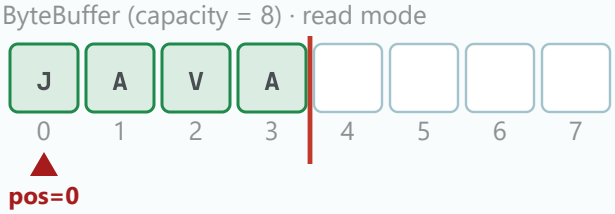
OUTPUT

VARIABLES

`pos = 4` `lim = 8` `cap = 8`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
▶      b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```



`flip()` : limit = old position (4), position = 0. Only the 4 bytes written can be read.

Step 6 / 15

OUTPUT

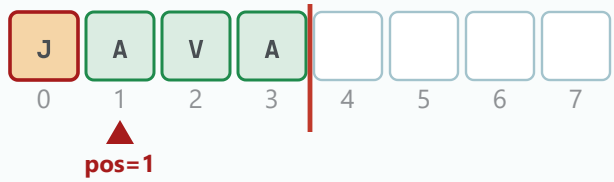
VARIABLES

`pos = 0` `lim = 4` `cap = 8`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
▶      char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```

ByteBuffer (capacity = 8) · read mode



`get()` returns the byte at the position; the position becomes 1.

Step 7 / 15

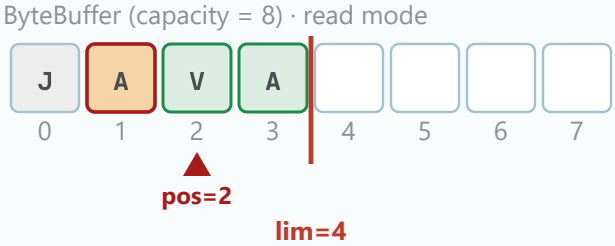
OUTPUT

VARIABLES

`pos = 1` `lim = 4` `cap = 8` `c1 = 'J'`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
▶      char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```



c2 = 'A'; 2 unread bytes remain (V, A).

Step 8 / 15

OUTPUT

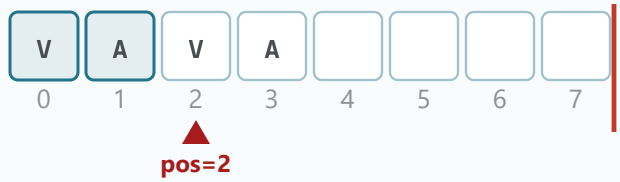
VARIABLES

`pos = 2` `lim = 4` `cap = 8` `c1 = 'J'` `c2 = 'A'`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
▶      b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```

ByteBuffer (capacity = 8) · write mode



`compact()` : the unread V, A are copied to the front; position 2, limit 8. The old bytes at 2–3 stay until overwritten.

Step 9 / 15

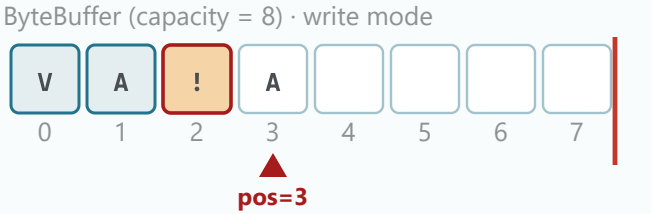
OUTPUT

VARIABLES

`pos = 2` `lim = 8` `cap = 8` `c1 = 'J'` `c2 = 'A'`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
▶      b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```



'!' is written at position 2, over the old 'V'.

Step 10 / 15

OUTPUT

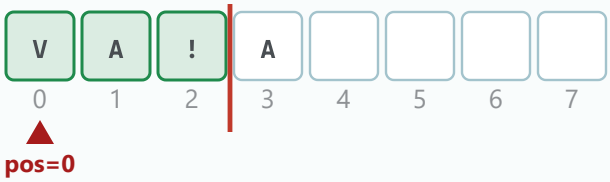
VARIABLES

`pos = 3` `lim = 8` `cap = 8` `c1 = 'J'` `c2 = 'A'`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
▶      b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```

ByteBuffer (capacity = 8) · read mode



Switch to reading: limit 3 (V, A, !).

Step 11 / 15

OUTPUT

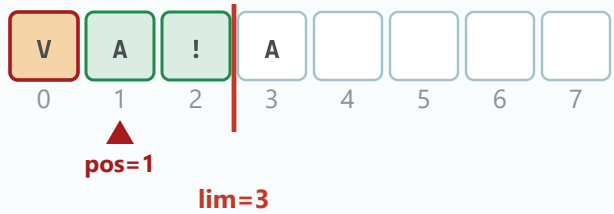
VARIABLES

pos = 0 | lim = 3 | cap = 8 | c1 = 'J' | c2 = 'A'

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
▶      b.get();
17     b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```

ByteBuffer (capacity = 8) · read mode



One byte is read ('V'); position 1.

Step 12 / 15

OUTPUT

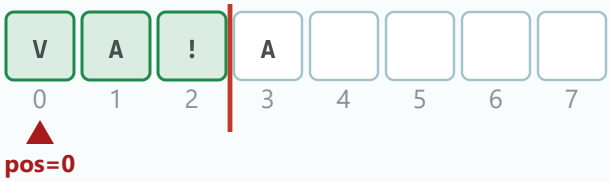
VARIABLES

`pos = 1` `lim = 3` `cap = 8` `c1 = 'J'` `c2 = 'A'`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
▶      b.rewind();
18     b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```

ByteBuffer (capacity = 8) · read mode



`rewind()` : position 0, limit **unchanged**: the same data can be read again.

Step 13 / 15

OUTPUT

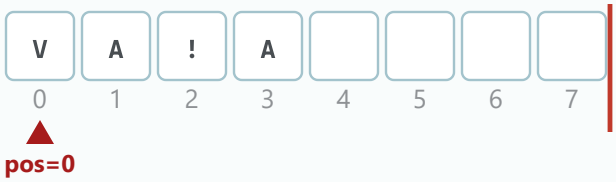
VARIABLES

`pos = 0` `lim = 3` `cap = 8` `c1 = 'J'` `c2 = 'A'`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
▶      b.clear();
19     System.out.println(c1 + "" + c2 + " " +
   b);
20   }
21 }
```

ByteBuffer (capacity = 8) · write mode



`clear()`: position 0, limit 8. The data is **not erased**; it just counts as "empty" and gets overwritten.

Step 14 / 15

OUTPUT

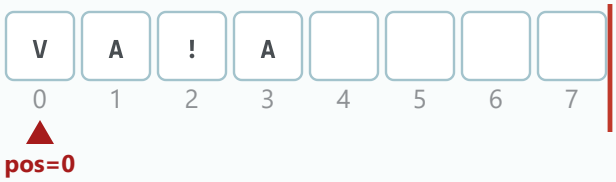
VARIABLES

`pos = 0` `lim = 8` `cap = 8` `c1 = 'J'` `c2 = 'A'`

put, flip, get, compact, rewind, clear: Step by Step

```
1  import java.nio.ByteBuffer;
2
3  public class BufferTrace {
4    public static void main(String[] args) {
5      ByteBuffer b = ByteBuffer.allocate(8);
6      b.put((byte) 'J');
7      b.put((byte) 'A');
8      b.put((byte) 'V');
9      b.put((byte) 'A');
10     b.flip();
11     char c1 = (char) b.get();
12     char c2 = (char) b.get();
13     b.compact();
14     b.put((byte) '!');
15     b.flip();
16     b.get();
17     b.rewind();
18     b.clear();
▶      System.out.println(c1 + "" + c2 + " " +
b);
20   }
21 }
```

ByteBuffer (capacity = 8) · write mode



`toString()` shows the buffer's position, limit and capacity.

Step 15 / 15

OUTPUT

```
JA java.nio.HeapByteBuffer[pos=0 lim=8
cap=8]
```

VARIABLES

`pos = 0` `lim = 8` `cap = 8` `c1 = 'J'` `c2 = 'A'`

Buffer Operations

Method	What it does	position	limit
`put(x)`	writes data at the position	+1	—
`get()`	reads the data at the position	+1	—
`flip()`	switches from writing to **reading**	0	old position
`rewind()`	to reread the same data from the start	0	unchanged
`clear()`	prepares for **writing** again; does not erase	0	capacity
`compact()`	moves unread data to the front, keeps writing	remaining count	capacity
`mark()` / `reset()`	marks the position / returns to the mark	— / mark	—

Pros

+ Flexible navigation and modification of data
+ Efficient memory use: one area is reused again and again

Cons

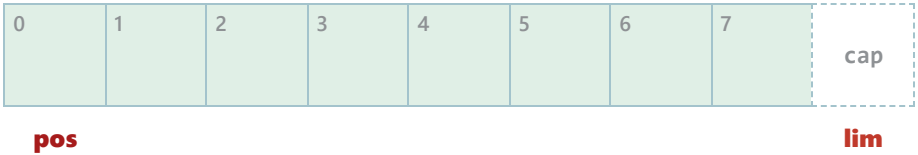
– Tricky to use (forgetting `flip` is a common bug)
– Cannot be resized later

NIO

ByteBuffer Playground

byte: `J` **put** **get** **flip** **rewind** **compact** **clear** **new buffer**

```
ByteBuffer b = ByteBuffer.allocate(8);
```



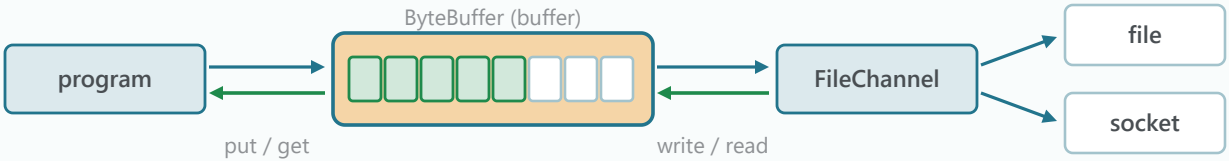
`position=0 limit=8 capacity=8 remaining=8`

Empty buffer in write mode: position = 0, limit = capacity = 8.

Green cells lie **between position and limit**: free space in write mode, data to read in read mode. Try: `put` three times, `flip`, `get` twice, `compact`; or `get` without `flip`.

NIO Channels

- A channel handles input and output: it exchanges data with a file, a socket or some protocol.
- Data is always read (`read(buffer)`) or written (`write(buffer)`) through a **buffer**.



Channel type	Use
`FileChannel`	reading and writing a file
`SocketChannel`	data over a TCP socket
`ServerSocketChannel`	accepting incoming TCP connections
`DatagramChannel`	data over a UDP socket

Opening and Closing Channels

```
public class ChannelOpenClose {
  public static void main(String[] args) throws IOException {
    // Create a FileChannel
    FileChannel fileChannel = FileChannel.open(
        Paths.get("file.txt"), StandardOpenOption.READ);
    // Create a SocketChannel
    SocketChannel socketChannel = SocketChannel.open();
    socketChannel.connect(new InetSocketAddress("example.com", 80));
    System.out.println("Connected: " + socketChannel.isConnected());
    // Close the channels
    fileChannel.close();
    socketChannel.close();
  }
}
```

OUTPUT

```
Connected: true
```

`FileChannel.open(path, options…)` opens a file channel; `SocketChannel.open()` + `connect` opens a network channel. Channels must be closed too, with `close()` or try-with-resources.

Reading with FileChannel

```
public class ChannelRead {
  public static void main(String[] args) throws IOException {
    try (FileChannel channel = FileChannel.open(
        Paths.get("file.txt"), StandardOpenOption.READ)) {
      ByteBuffer buffer = ByteBuffer.allocate(1024);
      // Read from the file
      int bytesRead = channel.read(buffer);
      System.out.println(bytesRead + " bytes read");
      // Switch to read mode
      buffer.flip();
      // Print the data
      while (buffer.hasRemaining()) {
        System.out.print((char) buffer.get());
      }
    }
  }
}
```

OUTPUT

```
17 bytes read
NIO channel demo
```

`channel.read(buffer)` reads into the buffer and returns the number of bytes read (−1 at end of file). Don't forget `flip()` before reading the buffer.

NIO

Writing with FileChannel

```
public class ChannelWrite {
  public static void main(String[] args) throws IOException {
    try (FileChannel ch = FileChannel.open(Paths.get("out.txt"),
        StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
      byte[] data = "Written via channel\n".getBytes(StandardCharsets.UTF_8);
      ByteBuffer buf = ByteBuffer.wrap(data);
      int written = ch.write(buf);
      System.out.println(written + " bytes written");
    }
  }
}
```

OUTPUT

```
20 bytes written
```

`ByteBuffer.wrap(arr)` uses an existing array as the buffer (position 0, limit = array length), so no `flip()` is needed. `CREATE`: create the file if missing.

Using SocketChannel

```
try (SocketChannel socketChannel = SocketChannel.open()) {
  socketChannel.connect(new InetSocketAddress("example.com", 80));
  ByteBuffer buffer = ByteBuffer.allocate(1024);
  buffer.put(("GET / HTTP/1.1\r\nHost: example.com\r\n"
      + "Connection: close\r\n\r\n").getBytes(StandardCharsets.US_ASCII));
  buffer.flip();
  // Write the request to the socket
  socketChannel.write(buffer);
  // Read the response
  buffer.clear();
  socketChannel.read(buffer);
  buffer.flip();
  String reply = StandardCharsets.US_ASCII.decode(buffer).toString();
  System.out.println(reply.substring(0, reply.indexOf("\r\n")));
} catch (IOException e) {
  e.printStackTrace();
}
```

OUTPUT

```
HTTP/1.1 200 OK
```

The body of `main`. A network connection is a channel too: the same `put` / `flip` / `write` pattern. HTTP/1.1 requires the `Host` header.

NIO: Pros and Cons

Pros

+ **Performance:** transfers in blocks, more efficiently than classic I/O
+ **Flexibility:** buffers and channels; two-way channels
+ One approach for both files and the network
+ **Many connections:** a selector serves many channels from one thread; only active channels are processed

Cons

– **Complexity:** harder than classic I/O
– You must manage buffer state and failures (partial reads, dropped connections)
– Selectors have a learning curve: tracking many channels' state correctly

Rule of thumb: to read one file line by line, `BufferedReader` or `Files` is enough. Channels shine with very large files and many network connections.

Path and Files (NIO.2)

```java
public class FilesNio {
  public static void main(String[] args) throws IOException {
    Path path = Path.of("data", "grades.txt");      // Java 11+
    Files.createDirectories(path.getParent());
    Files.writeString(path, "Ali 70\nAyşe 85\nCan 90\n");
    System.out.println(path.getFileName() + ": " + Files.exists(path));
    System.out.println("Size: " + Files.size(path) + " bytes");
    List<String> lines = Files.readAllLines(path);
    for (String line : lines) {
      System.out.println(line.toUpperCase(Locale.ROOT));
    }
    Files.write(path, List.of("Deniz 60"), StandardOpenOption.APPEND);
    System.out.println(Files.readAllLines(path).size() + " lines");
  }
}
```

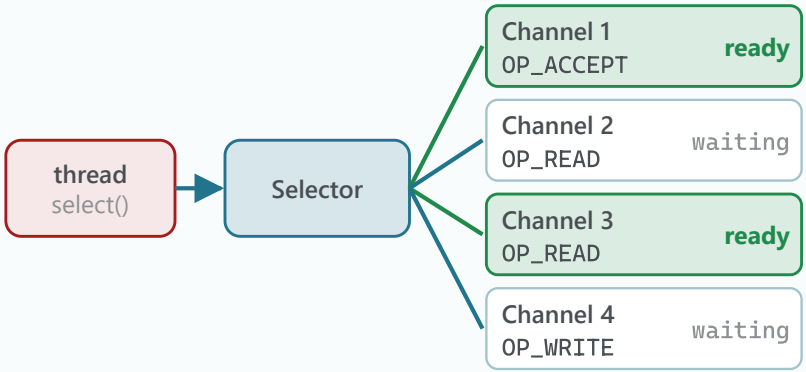
OUTPUT

```
grades.txt: true
Size: 23 bytes
ALI 70
AYŞE 85
CAN 90
4 lines
```

Method	Purpose
`Path.of(…)` / `Paths.get(…)`	builds a path (the modern `File`)
`Files.readAllLines`, `Files.readString`	reads the whole file in one call (UTF-8)
`Files.write`, `Files.writeString`	writes in one call
`Files.exists`, `size`, `copy`, `move`, `delete`	file system operations

NIO Selectors

- A mechanism for watching many channels from **a single thread**.
- `Selector.open()` creates a new selector.
- A channel is put in non-blocking mode and **registered** with the selector: `channel.register(selector, ops)`.
- `SelectionKey` constants say which events the channel waits for. `select()` blocks until at least one channel is ready.



SelectionKey	Meaning
`OP_READ`	the channel can be read
`OP_WRITE`	the channel can be written
`OP_CONNECT`	connection established
`OP_ACCEPT`	a connection can be accepted

A Server with a Selector

```
public class SelectorDemo {
  public static void main(String[] args) throws IOException {
    ServerSocketChannel server = ServerSocketChannel.open();
    Selector selector = Selector.open();
    server.bind(new InetSocketAddress(8080));
    server.configureBlocking(false);
    server.register(selector, SelectionKey.OP_ACCEPT);
    while (true) {
      int ready = selector.select();
      if (ready > 0) {
        Set<SelectionKey> keys = selector.selectedKeys();
        for (SelectionKey key : keys) {
          if (key.isAcceptable()) {
            // accept a connection
          } else if (key.isReadable()) {
            // read data
          }
        }
        keys.clear(); // drop handled keys
      }
    }
  }
}
```

- The server channel binds to port 8080, switches to non-blocking mode and registers for `OP_ACCEPT`.
- `select()` returns the number of ready channels; `selectedKeys()` gives their keys.
- Keys not removed with `clear()` come back on the next round.
- A real server registers each accepted client for `OP_READ` and reads its data.

Summary

Topic	Key class / method	Remember
Streams	`System.in`, `System.out`, `System.err`, `Scanner`	data flows from source to destination in order
Text / binary	ASCII, ISO-8859-9, UTF-8; `DataOutputStream`	127: 3 bytes as text, 4 as int; read and write with the same encoding
File system	`File`, `Path`, `Files`	`File` is just a path; it doesn't read contents
Reading/writing text	`Scanner`, `BufferedReader`, `PrintWriter`, `FileWriter`	`readLine()` gives `null` at the end; `FileWriter(name, true)` appends
Byte streams	`FileInputStream`, `ByteArray…`, `Piped…`, `Sequence…`	`read()` gives −1 at the end; keep it in an `int`
Buffering	`Buffered…`, `flush()`	batches disk access; close your streams
Copying	`read(byte[])`, `write(b, 0, n)`, `Files.copy`	write only the `n` bytes read
Random access	`RandomAccessFile`, `seek`	record k = k × record size
Serialization	`Serializable`, `ObjectOutputStream`, `transient`	`static` / `transient` are not written
NIO	`ByteBuffer`, `FileChannel`, `Selector`	`flip()` after writing; `clear()` / `compact()` before writing again

SUMMARY

Common Mistakes (1)

Ignoring a checked exception

```
error: unreported exception
FileNotFoundException;
  must be caught or declared to be thrown
```

`new FileReader("a.txt")` must sit inside `try` - `catch`, or the method must declare `throws IOException`.

Forgetting the import

```
error: cannot find symbol
  symbol:   class Scanner
```

`import java.util.Scanner;` (and `import java.io.*;` for I/O classes).

Single backslash in a Windows path

```
error: illegal escape character
  new File("C:\courses\notes.txt")
```

`\c` and `\n` count as escape sequences. Write `"C:\\courses\\notes.txt"` or `"C:/courses/notes.txt"`.

No null check in finally

```
Exception in thread "main"
java.lang.NullPointerException:
  Cannot invoke "java.util.Scanner.close()"
```

If the file can't be opened, `scanner` stays `null`. You need `if (scanner != null)` or try-with-resources.

SUMMARY

Common Mistakes (2)

nextLine after nextInt

```
Age: 19↵  Name: Ayşe↵   →   output: [] 19
```

`nextInt()` leaves the newline in the buffer, so the following `nextLine()` reads an empty line. Add a `keyboard.nextLine();` in between.

Not closing a buffered writer

```
bw.write("entry");   // no close
note.txt size: 0
```

The data stays in the buffer and the file looks empty. Call `close()` (or `flush()`); better yet, use try-with-resources.

Forgetting flip()

```
b.put((byte) 'J'); b.put((byte) 'A');
b.get() → 0   (empty cell at position 2)
```

You must `flip()` between writing and reading; otherwise you read what comes after the data, not the data.

Forgetting Serializable

```
java.io.NotSerializableException: Car
```

The class passed to `writeObject` (and its field types) must `implements Serializable`.

Storing read() in a byte

```
byte b; while ((b = (byte) in.read()) != -1)
```

A byte of value 255 (it becomes −1) is mistaken for end of file. The right type is `int`.

Writing write(buf)

```
out.write(buf);   // stale data in last block
```

The last block doesn't fill the array; the right call is `write(buf, 0, size)`.

SUMMARY

Practice Problems (1)

```
try (PrintWriter pw = new PrintWriter("s.txt")) {
  pw.print(12);
  pw.print(' ');
  pw.print(345);
}
System.out.println(new File("s.txt").length());
try (DataOutputStream d = new DataOutputStream(new FileOutputStream("s.dat"))) {
  d.writeInt(12);
  d.writeInt(345);
}
System.out.println(new File("s.dat").length());
```

1. What does this code print? `12` and `345` are written first as text, then in binary.

▶ **Show answer**

SUMMARY

Practice Problems (2)

```
try (FileWriter w = new FileWriter("a.txt")) {
  w.write("ab");
}
try (FileWriter w = new FileWriter("a.txt")) {
  w.write("cd");
}
try (FileWriter w = new FileWriter("a.txt", true)) {
  w.write("ef");
}
try (BufferedReader r = new BufferedReader(new FileReader("a.txt"))) {
  System.out.println(r.readLine());
}
```

2. What does this code print? `a.txt` does not exist at first.

▶ **Show answer**

Practice Problems (3)

```
ByteBuffer b = ByteBuffer.allocate(10);
b.put((byte) 1).put((byte) 2).put((byte) 3);
b.flip();
b.get();
System.out.println(b.position() + " " + b.limit() + " " + b.remaining());
b.compact();
System.out.println(b.position() + " " + b.limit());
b.clear();
System.out.println(b.position() + " " + b.limit() + " " + b.get(0));
```

3. What does this code print on its three lines?

▶ **Show answer**

Practice Problems (4)

```
try (RandomAccessFile f = new RandomAccessFile("r.dat", "rw")) {
  for (int i = 1; i <= 4; i++) f.writeInt(i * 10);
  f.seek(4);
  int a = f.readInt();
  f.seek(f.length() - 4);
  int b = f.readInt();
  System.out.println(a + " " + b + " " + f.getFilePointer());
}
```

4. What does this code print?

▶ **Show answer**

Exercises (1)

1. **Counter:** write a program that counts the lines, words and characters (excluding line breaks) in `text.txt`. The file: “Reading files in Java / is simple / (blank line) / and fun”.

▶ **Solution**

Exercises (2)

2. **Squares:** read `n` from the keyboard; write the numbers 1..`n` and their squares to `squares.txt`, then read the file back and compute the sum of squares.

▶ **Solution**

Exercises (3)

3. **Storing an array:** serialize `int[] arr = {3, 1, 4, 1, 5}` to `arr.ser`, read it back and print its length and sum (arrays are serializable).

▶ **Solution**

SUMMARY

Exercises (4)

4. **Grade file:** write 5 grades as `int`s with `RandomAccessFile`. Find the grade at a given rank without reading the whole file. Which position should you `seek` to?
5. **Think:** to read only the last record of a 10,000-line log file, is a sequential stream or random access better? What changes if lines have different lengths?

▶ **Answers / hints**

SUMMARY

References

- Oracle, *The Java Tutorials*: “Basic I/O” (I/O Streams, File I/O featuring NIO.2), “Scanning and Formatting”.
- Oracle, *Java SE 21 API Documentation*: the `java.io`, `java.nio`, `java.nio.channels`, `java.nio.file` packages.
- Oracle, *Java Object Serialization Specification*.
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition* (3.10.6 escape sequences, 14.20.3 try-with-resources).
- P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapter 15: Files, Input/Output Streams, NIO and XML Serialization).
- C. S. Horstmann, *Core Java, Volume II: Advanced Features*, 12th ed., 2022 (Chapter 2: Input and Output).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022 (Chapter 16: Saving Objects (and Text)).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.

The End

Chapter 8: Input and Output

Previous: Chapter 7: Testing and Debugging · Next: Chapter 9: Graphics and User Interfaces