

## LECTURE 1

# Computer Interface Concepts & File I/O

Understanding input/output operations in Java

# Today's Agenda

- 1 What is a Computer Interface?
- 2 Data Streams in Java
- 3 File Operations & I/O Classes
- 4 Exception Handling for I/O
- 5 Modern Java I/O (java.nio)
- 6 Summary & Key Takeaways

PART 1

---

# What is a Computer Interface?

Communication protocols between systems

# Computer Interface

A mechanism for communication between a computer and the external environment.

- Enables data exchange with peripherals (keyboard, mouse, display)
- Facilitates file system access and database operations
- Supports network communication and sensor integration
- Critical component of modern software architecture

# Types of Computer Interfaces

## Human-Computer Interface (HCI)

- User interactions (keyboard, mouse, touch)
- Graphical User Interface (GUI)
- Command-line Interface (CLI)
- Voice and gesture recognition
- Focus: usability and experience

## Computer-to-Computer Interface

- Machine-to-machine communication
- Network protocols (HTTP, FTP, TCP/IP)
- File system and database operations
- API interactions
- Focus: data integrity and reliability

# External Environment: Interface Targets

- Peripherals: printers, scanners, monitors, speakers, cameras
- File Systems: reading/writing to disk storage
- Databases: querying and updating persistent data
- Networks: TCP/IP, UDP, HTTP, WebSockets
- Sensors & IoT: temperature, pressure, motion detectors
- Real-time systems: embedded devices, robotics

# Why Interfaces Matter

100%

Modern apps  
need I/O

$\infty$

Possible targets

## PART 2

---

# Data Streams in Java

The `java.io` package and stream hierarchy

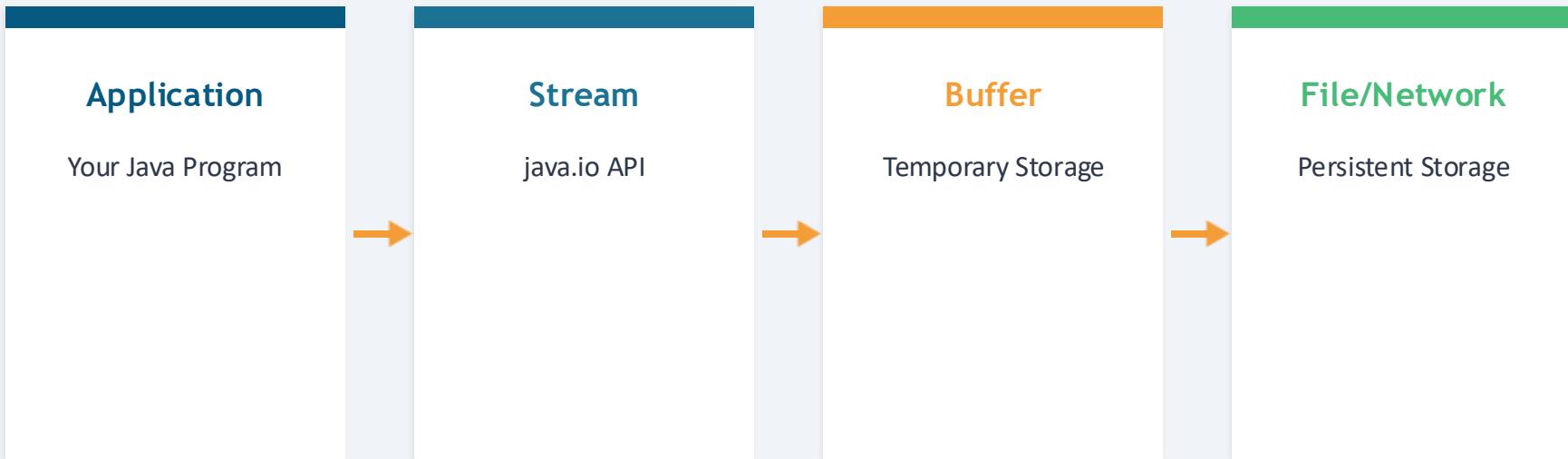
## Data Stream

A sequence of bytes or characters flowing between a program and an external source.

- Unidirectional: data flows in one direction (input or output)
- Sequential: data accessed in order, not random
- Abstract: hides complexity of underlying I/O device
- Buffered: may accumulate data before transfer

# Data Flow in Java I/O

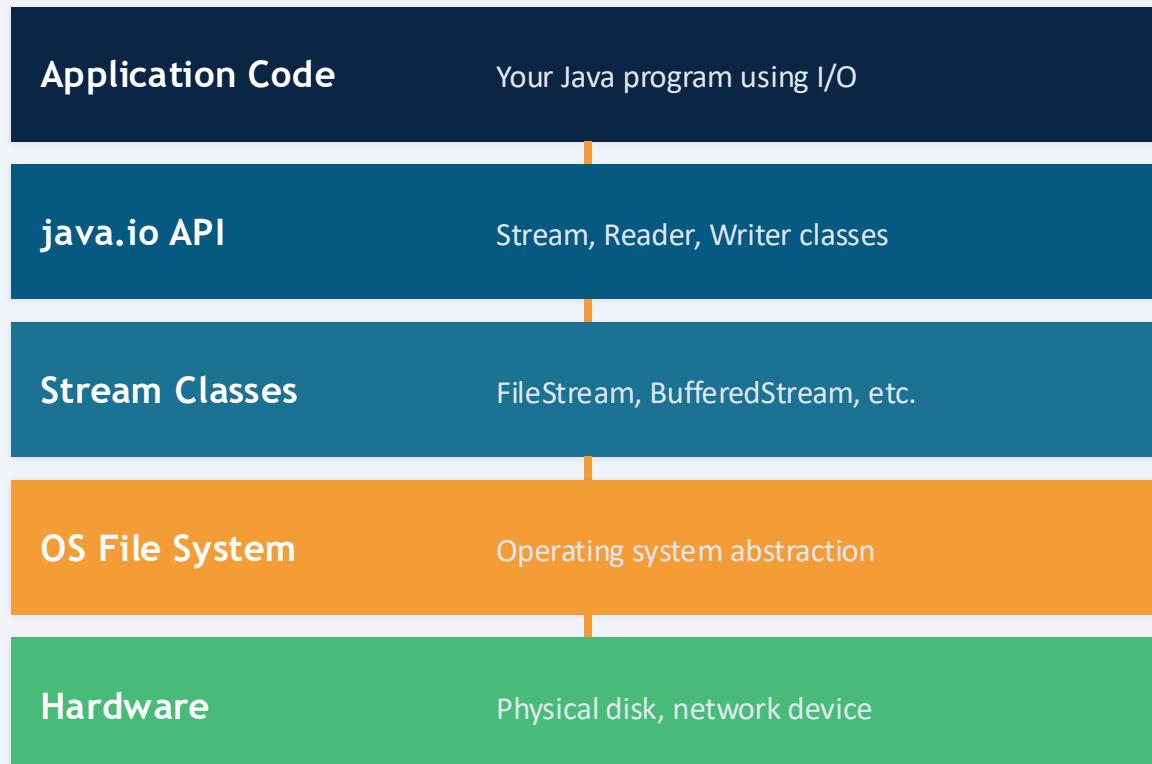
*How data travels through the I/O system*



# Java I/O Stream Hierarchy

- `InputStream/OutputStream`: Base classes for byte streams
- `Reader/Writer`: Character stream classes
- `FileInputStream/FileOutputStream`: Direct file access
- `BufferedInputStream/BufferedOutputStream`: Buffered I/O
- `DataInputStream/DataOutputStream`: Binary data (primitives)
- `InputStreamReader`: Bridge between bytes and characters

# Java I/O Architecture



## Key Points

- Layers abstract complexity
- Each layer hides lower detail
- Standard interfaces enable flexibility

# Four Key I/O Concepts

## Streams

Unidirectional flow of bytes or characters from source to destination

## Buffers

Temporary memory storage that improves I/O performance

## Files

Persistent storage on disk accessed through the file system

## Exceptions

IOException and subclasses for handling I/O errors

## PART 3

---

# File Operations & I/O Classes

Reading and writing files in Java

# File Class Basics

- `java.io.File`: Represents file path (not contents)
- Can represent directories or files
- Methods: `exists()`, `isFile()`, `isDirectory()`, `length()`
- Methods: `getName()`, `getAbsolutePath()`, `getParent()`
- Methods: `mkdir()`, `delete()`, `createNewFile()`
- Important: File object doesn't read/write data directly

# The Buffer Concept



**Buffer = Temporary Storage**

- Buffers accumulate data in memory before transfer
- Reduces number of expensive disk I/O operations
- Example: reading 100MB file with 1KB buffer = ~100K ops vs millions of byte reads
- Essential for performance optimization
- Always use buffered streams in real applications

# Reading a File: Text Content

JAVA

```
FileReader fr = new FileReader("data.txt");
BufferedReader br = new BufferedReader(fr);
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
br.close();
```

*Note: Always use try-with-resources in production code for automatic resource cleanup*

# Writing to a File: Text Content

JAVA

```
FileWriter fw = new FileWriter("output.txt");
BufferedWriter bw = new BufferedWriter(fw);
bw.write("Hello, World!");
bw.newLine();
bw.write("Second line of text");
bw.flush();
bw.close();
```

*flush() forces buffer contents to disk; close() releases resources*

# Binary File Operations: Read

JAVA

```
FileInputStream fis = new FileInputStream("binary.bin");
BufferedInputStream bis = new BufferedInputStream(fis);
int byteVal = bis.read(); // read single byte
byte[] buffer = new byte[1024];
int bytesRead = bis.read(buffer); // read into array
bis.close();
```

*read() returns -1 when EOF reached; always read into buffers for efficiency*

# Without Buffering vs With Buffering

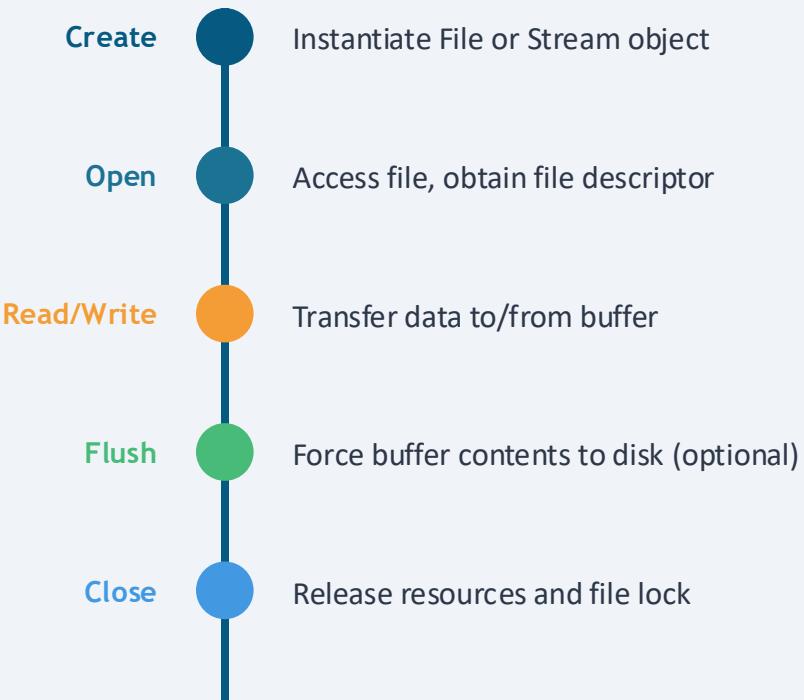
## WITHOUT BUFFERING

- Read 1 million characters one by one
- 1 million system calls to disk
- Each call has OS overhead (~microseconds)
- Total time: seconds or minutes
- CPU wastes time waiting for I/O

## WITH BUFFERING (8KB buffer)

- Read 1 million characters in ~125 buffer blocks
- Only 125 system calls to disk
- Same data volume, 8000x fewer operations
- Total time: milliseconds
- CPU processes data while disk is busy

# File Operations Sequence



# Common I/O Operation Patterns

1

**Write**

Create stream → write data  
→ flush → close

2

**Read**

Create stream → read data  
→ check EOF → close

3

**Append**

`FileWriter(file, append=true)`  
→ write → close

4

**Copy**

Read from source → write to destination → close both

# Buffering in High-Performance Systems

## SCENARIO

You are building a logging system for a web server that handles 10,000 requests per second. Each request generates a log entry (1KB) that needs to be written to a file.

## THINK ABOUT

1

Why would writing each log entry individually be a problem?

2

How would buffering help? What buffer size would you choose?

3

What happens if the server crashes before the buffer is flushed?

PART 4

---

# Exception Handling for I/O

Handling errors in file operations

## IOException

Checked exception thrown when I/O operation fails unexpectedly.

- `FileNotFoundException`: File doesn't exist (subclass of `IOException`)
- `EOFException`: Premature end of file
- `InterruptedIOException`: I/O interrupted
- Must be caught or declared in method signature (checked exception)

# Common I/O Exceptions

- `FileNotFoundException`: File not found or cannot be opened
- `SecurityException`: Permission denied
- `EOFException`: Unexpected end of stream
- `InterruptedException`: Thread interrupted during I/O
- `UnsupportedEncodingException`: Invalid character encoding
- `IOException`: General I/O failure (parent of most above)

# Try-With-Resources

JAVA

```
try (BufferedReader br = new BufferedReader(
      new FileReader("data.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

*Try-with-resources automatically closes resources (implements AutoCloseable)*

# Traditional Try-Catch-Finally

JAVA

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("data.txt"));
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (br != null) {
        try { br.close(); } catch (IOException e) { }
    }
}
```

*Verbose! Try-with-resources is preferred in modern Java*

# Best Practices: Exception Handling

- Always use try-with-resources for streams (Java 7+)
- Catch specific exceptions first (FileNotFoundException before IOException)
- Don't catch and ignore exceptions silently
- Log error details for debugging and monitoring
- Consider retry logic for transient failures
- Clean up resources in finally block if not using try-with-resources

## PART 5

---

# Modern Java I/O (java.nio)

Non-blocking I/O for high-performance applications

## java.nio (New I/O)

Modern I/O API providing non-blocking, selector-based I/O for scalability.

- Channel: Connection to file, socket, or device
- Buffer: Container for data (ByteBuffer, CharBuffer, etc.)
- Selector: Multiplexes multiple channels (handles thousands of connections)
- Key advantage: Single thread can monitor many I/O operations

# Classic I/O vs Java NIO

## Classic I/O (`java.io`)

- Stream-based (`InputStream`/`OutputStream`)
- Blocking operations
- One thread per connection
- Simpler programming model
- Good for small number of connections
- Not scalable for 10K+ connections

## Java NIO (`java.nio`)

- Channel and Buffer based
- Non-blocking operations
- Single thread with Selector
- More complex but powerful
- Excellent for many connections
- Can handle 100K+ simultaneous connections

# Java NIO: Read File with Channels

JAVA

```
RandomAccessFile raf = new RandomAccessFile("data.bin", "r");
FileChannel channel = raf.getChannel();
ByteBuffer buffer = ByteBuffer.allocate(1024);
while (channel.read(buffer) > 0) {
    buffer.flip();
    while (buffer.hasRemaining()) {
        System.out.print((char) buffer.get());
    }
    buffer.clear();
}
channel.close();
```

*flip() switches from write to read mode; clear() resets for next read*

# When to Use NIO vs Classic I/O

- Use Classic I/O: Simple file operations, small number of connections (< 100)
- Use NIO: Network servers, handling 1000s of concurrent connections
- Use NIO: Real-time applications with low-latency requirements
- Use NIO: High-throughput data processing
- Frameworks like Netty, Vert.x abstract NIO complexity for you
- **Modern Spring Boot uses non-blocking I/O by default**

# Think – Pair – Share

*Understanding the impact of buffering decisions on real-world performance*

If a Java program reads a 1GB file character by character without buffering, what happens to performance and why?

*Hints: Think about disk I/O operations | System calls per character | CPU vs I/O time*

## PART 6

---

# Summary & Key Takeaways

What you should remember

# Key Concepts Review

- 1 Computer interfaces enable communication between programs and external systems
- 2 Java I/O streams provide abstraction over physical I/O devices
- 3 Always use buffered streams to avoid performance degradation
- 4 Try-with-resources pattern automatically manages stream lifecycle
- 5 IOException is checked; must be caught or declared
- 6 Java NIO enables high-performance server applications

# Practical I/O Guidelines

- Always wrap streams with `BufferedInputStream`/`BufferedOutputStream`
- Use `try-with-resources` to ensure proper resource cleanup
- Handle exceptions appropriately—don't silently ignore
- For file size > 10MB, consider `FileChannel` or memory-mapped files
- For network I/O with 1000+ connections, use `java.nio`
- Profile your I/O operations; don't guess about bottlenecks

# Common Mistakes to Avoid

- Not closing streams (resource leak)
- Reading/writing without buffering (slow)
- Ignoring IOException (crashes in production)
- Mixing byte and character streams incorrectly
- Not using absolute paths on file system
- Assuming file exists without checking first

# Questions?

---

Next: Lecture 2 — Java Networking Fundamentals

CSCI 2450 | Computer Interface Programming | 2026