Programming in Java

3. Advanced I/O

Overview of the java.io Package

- Java’s original I/O API is contained in the java.io package,
  - Introduced in Java 1.0. It provide
- Implements
  - Input and Output Streams for reading and writing bytes
  - Readers and Writers for character-based I/O
  - Classes for working with files, object serialization, buffering, and data streams

Overview of the java.io Package

Class/Interface	Description
`InputStream`	Abstract superclass for all byte input streams
`OutputStream`	Abstract superclass for all byte output streams
`Reader` / `Writer`	Abstract classes for character streams
`File`, `FileInputStream`	File abstraction and file byte reader
`BufferedInputStream`, `BufferedReader`	Buffers data to reduce disk I/O calls
`ObjectInputStream`, `DataInputStream`	For structured or serialized data

The Stream-Based Model

- Java I/O uses a stream-based model
  - Data flows in or out one byte or character at a time.
  - Think of streams like pipes through which data flows sequentially.
- Synchronous and blocking, which means:
  - When a thread reads from a stream, it waits (blocks) until data is available.
  - When a thread writes to a stream, it blocks until the data is fully written or flushed.
- Implications of Blocking I/O:
  - Suitable for simple tasks and small numbers of concurrent users.
  - Inefficient for scalable systems (e.g., network servers) because each client typically requires a dedicated thread.
  - Can lead to thread starvation or resource exhaustion under load.

Java NIO

- Java NIO (New Input/Output)
  - Introduced overcome limitations of the traditional java.io API.
  - Provides faster, more scalable, and more flexible mechanisms for performing I/O operations
  - Especially in the context of file access, buffering, and non-blocking I/O.

Concept	Description
Channel	A bi-directional communication path to read/write data (e.g., file, socket). Unlike streams, channels can read and write at the same time.
Buffer	A container for data being read from or written to a channel. Data must go through a buffer before being processed.
Selector	Allows a single thread to monitor multiple channels (e.g., for readiness to read/write), enabling **non-blocking I/O**.
Path and Files	From NIO.2 (`java.nio.file`), replaces `File` with richer functionality for path manipulation and file operations.
Memory Mapping	Allows files to be **mapped directly into memory**, enabling high-speed access for large files.

Java IO and NIO Comparison

Feature	`java.io` (Classic)	`java.nio` (New)
Model	Stream-based	Buffer-based
Blocking	Always blocking	Supports non-blocking
Threading	One thread per connection	Single thread can manage many channels
Data Handling	Sequential byte or character streams	Uses buffers and channels
Performance	Less scalable for many clients	High performance for I/O-heavy apps
File Access	`FileInputStream`, `RandomAccessFile`	`FileChannel`, `Files`, `Paths`

Java IO and NIO Comparison

- In classic IO:
  - Reads data one byte at a time (or array), always sequentially.

```
FileInputStream fis = new FileInputStream("data.txt");
int byteData = fis.read();
```

- NIO
  - Reads data into a buffer, allowing more efficient bulk reading, positioned access, and memory mapping.

```
FileChannel channel = FileChannel.open(Paths.get("data.txt"), StandardOpenOption.READ);
ByteBuffer buffer = ByteBuffer.allocate(1024);
channel.read(buffer);
```

Channels

- Bi-directional connection to a data source or sink
  - Unlike streams, channels can both read and write data and
  - Often used with buffers.
- There are different channel types based on the data source/sing
  - FileChannel: Reads/writes from files
  - SocketChannel: TCP client
  - ServerSocketChannel: TCP server
  - DatagramChannel: UDP socket
  - SelectableChannel: Supports non-blocking with selectors (discussed later)
- The data source/sink is wrapped in a channel
  - We then interact with the channel

Buffers

- A Buffer is a container for data.
  - All data read from a channel goes into a buffer
  - All data written to a channel comes from a buffer.
  - Buffers have types based on the kind of data they hold
- A Channel is a pipe and the Buffer is a bucket
  - Channel reads into Buffer → app processes Buffer.
  - App writes into Buffer → Channel writes Buffer to destination.
  - The code must manually control buffer flow using flip(), clear(), and rewind().

Buffer Type	Description
`ByteBuffer`	Stores byte data
`CharBuffer`	Stores character data
`IntBuffer`, `LongBuffer`, etc.	For numeric data

Buffer Operations

Step	Description
`allocate()`	Creates a buffer with a fixed capacity
`put()`	Adds data to buffer
`flip()`	Switches from writing to reading mode
`get()`	Reads data from buffer
`clear()`	Resets for writing again
`rewind()`	Resets position to 0 without clearing the data

Reading a File

- Reading a file byte by byte
  - Using a 64 byte buffer
  - The read(buffer) fills the buffer with data
  - We are positioned at the end of the buffer
  - To print out the buffer, we have to switch to read the data in the buffer
  - Flip() sets the current position in the buffer to 0
  - After the buffer has been written out, it is cleared
  - Buffer.clear() implicitly does flip()

```java
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class ChannelExample {
    public static void main(String[] args) throws Exception {
        RandomAccessFile file = new RandomAccessFile("example.txt", "r");
        FileChannel channel = file.getChannel();

        ByteBuffer buffer = ByteBuffer.allocate(64);  // create buffer with capacity 64

        while (channel.read(buffer) > 0) {
            buffer.flip();  // prepare buffer for reading

            while (buffer.hasRemaining()) {
                System.out.print((char) buffer.get());  // read one byte at a time
            }

            buffer.clear();  // prepare buffer for next read
        }

        channel.close();
        file.close();
    }
}
```

Writing to a File

- Writing a file byte by byte
  - The data is but into the buffer with put()
  - We are positioned to the end of the data in the buffer
  - We use flip() to reset to the start of the buffer
  - Then we write the buffer to an channel wrapping a file

```
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class WriteExample {
    public static void main(String[] args) throws Exception {
        String text = "Hello from Java NIO!";
        ByteBuffer buffer = ByteBuffer.allocate(128);
        buffer.put(text.getBytes());   // write data into buffer

        buffer.flip();  // switch to read mode for channel to consume

        try (FileOutputStream fos = new FileOutputStream("output.txt");
             FileChannel channel = fos.getChannel()) {
            channel.write(buffer);     // write buffer to file
        }
    }
}
```

Lab 3-1

Buffers and Channels

Selectors

- A Selector is a component in Java NIO
  - Allows a single thread to monitor multiple channels for events like incoming data.
  - Used to implement scalable non-blocking I/O servers
  - Especially for handling many connections with few threads.
- In old Java IO
  - Each socket connection required a dedicated thread
  - Doesn't scale well for high concurrency.
  - A Selector allows one thread to manage hundreds or thousands of non-blocking connections

Key Selector Concepts

Concept	Description
Selector	Monitors registered channels for I/O events (e.g., read, write, connect, accept).
SelectableChannel	A channel (like `SocketChannel`) that can register with a `Selector`.
SelectionKey	Represents the registration of a channel with a selector. It tracks what events the channel is interested in.
InterestOps	Events like `OP_READ`, `OP_WRITE`, `OP_ACCEPT`, `OP_CONNECT` that the selector watches for.

Selectors and the Observer Pattern

- The Selector mechanism
  - Observer design pattern applied at the system level for I/O event handling.
  - Channels register interest in specific events (e.g., read, write, accept) with the Selector.
  - The Selector watches for those events in the background.
  - When a channel becomes ready (e.g., data available to read), the selector notifies the application by including the event in the set of SelectionKeys.
  - The application then responds to the event by checking the key and acting accordingly.
- Think of the Selector as a news broadcaster,
  - The SelectableChannels are then subscribers.
  - When an event happens (like incoming data or a new connection)
  - The Selector broadcasts that event to whoever registered interest.
- Selectors are explored more thoroughly in the next lab

Event-Driven Architecture

- Event-driven architecture (EDA)
  - Architecture design where components react to events rather than constantly polling or blocking.
  - An event handler waits for events and dispatches them to handlers.
  - Common in UI frameworks, game loops, and high-performance servers
  - Often implemented with an event handler + callback logic
    - *In other words, when an event occurs, the callback is some action the executes in response to the event*
    - *Often implemented in microservices that respond to events, like financial transactions*
- A Selector monitors many channels for I/O events
  - Channels register for specific events: OP_READ, OP_WRITE, OP_ACCEPT, etc.
  - The select() method waits for events and returns only when one or more are ready
  - The app then handles those events

Lab 3-2

Selectors

Java™