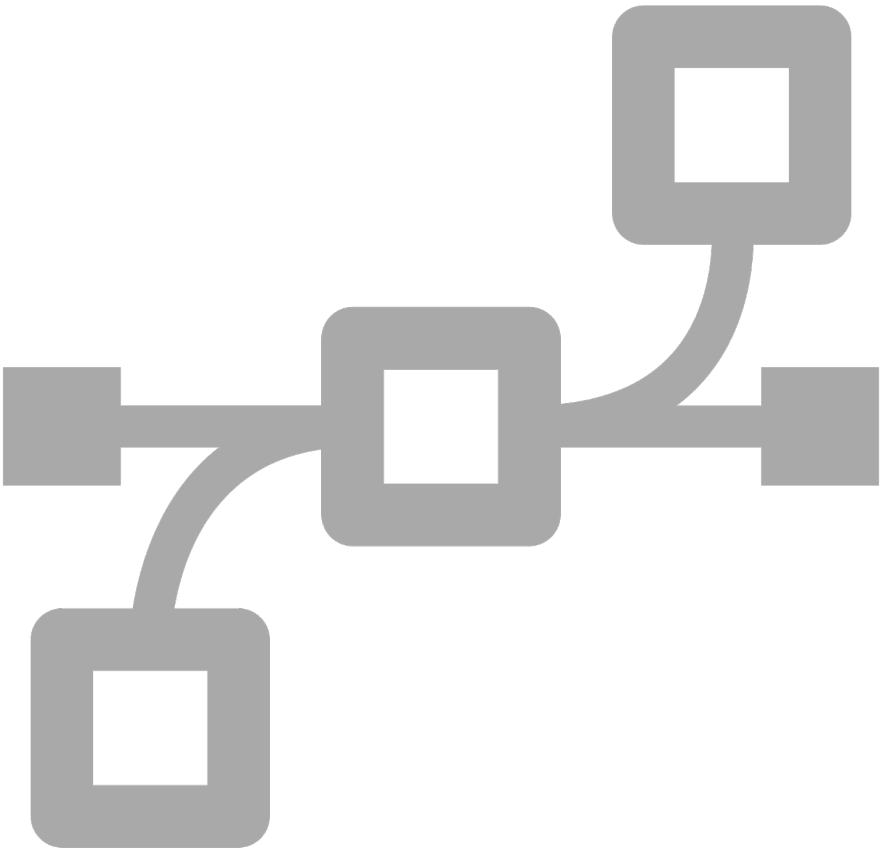
Java IO and NIO



readbytes

Java IO and NIO

Guide for Beginner to Advanced Programmers

readbytes.github.io

2025-07-14

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Java IO and NIO

1. What is Java IO?
2. History and Evolution of Java IO
3. Overview of Stream-Based IO
4. Blocking vs Non-blocking IO
5. Synchronous vs Asynchronous IO

1 Introduction to Java IO and NIO

1.1 What is Java IO?

Java IO (Input/Output) is a fundamental part of the Java programming language that enables programs to communicate with the outside world. Whether reading user input, writing data to a file, receiving data from a network, or processing binary streams, Java IO provides the tools and abstractions needed to perform these operations efficiently and reliably.

At its core, IO in Java is about **data movement**. The two main directions are:

- **Input** – receiving data from an external source (e.g., keyboard input, a file, or a network).
- **Output** – sending data to an external destination (e.g., displaying text on the screen, writing to a file, or sending data over a socket).

Without IO, Java programs would operate in isolation — like a person with no way to hear or speak. IO allows programs to be dynamic, interactive, and integrated with files, devices, and networks.

1.1.1 Purpose of Java IO

Java IO serves several key purposes:

1. **Data Communication** Java IO enables reading from and writing to different sources like files, consoles, memory buffers, and network sockets. This is essential for real-world applications that interact with users or systems.
2. **Persistence** Programs often need to save information so that it can be retrieved later. Java IO allows this persistence by supporting file operations such as saving user settings, application logs, and serialized objects.
3. **Interoperability** Java IO helps bridge communication between systems. For example, reading and writing files in various formats allows Java programs to interact with data generated by other software systems.

1.1.2 Real-World Analogy

To better understand IO, consider the analogy of a **postal system**:

- You (the program) may want to **read** a letter sent to you (input).
- You may also want to **write** and send a letter to someone else (output).

The **IO system** is the infrastructure that allows this to happen—envelopes (data streams),

addresses (file paths or URLs), and mail carriers (stream objects) all play a part in making the communication successful.

Similarly, Java uses classes and interfaces like `InputStream`, `OutputStream`, `Reader`, and `Writer` to facilitate the transfer of data between programs and external sources or destinations.

1.1.3 Java IO Architecture Overview

Java's IO system is built around the concept of **streams**, which represent a continuous flow of data. This data can be in the form of **bytes** (binary data) or **characters** (text data). The stream abstraction allows developers to work with data without needing to know the exact source or destination—it could be a file, network socket, or memory buffer.

The core Java IO library is located in the `java.io` package and includes classes for:

- **Byte streams** (e.g., `InputStream`, `OutputStream`)
- **Character streams** (e.g., `Reader`, `Writer`)
- **File handling** (e.g., `File`, `FileReader`, `FileWriter`)
- **Buffered and data streams** for optimized and structured IO
- **Object serialization** for saving and restoring objects

Later versions of Java introduced **NIO (New IO)** and **AIO (Asynchronous IO)** in the `java.nio` and `java.nio.channels` packages, which provide more efficient and scalable alternatives for high-performance applications.

1.1.4 Why Java IO is Essential

Java IO is indispensable for a variety of programming tasks:

- **Console-based applications** rely on reading user input and displaying output.
- **Desktop applications** read and write configuration files or load resources.
- **Web servers** use IO to handle requests and send responses over the network.
- **Data-driven applications** interact with logs, databases, and data files.
- **Cloud-based and distributed systems** rely heavily on asynchronous and non-blocking IO to support massive concurrency.

Mastering Java IO is not just about syntax—it's about understanding how data flows between systems and how to manage it efficiently and securely.

1.1.5 Types of IO Operations in Java

Java supports several types of IO operations, each designed for specific needs:

1. **Byte-Oriented IO** Operates at the byte level and is ideal for handling binary data such as images, audio files, or any non-text content. Classes include `InputStream`, `OutputStream`, and their subclasses.
2. **Character-Oriented IO** Designed for handling textual data using Unicode characters. It uses `Reader` and `Writer` classes to process text reliably, especially with international characters and encoding.
3. **Buffered IO** Adds a layer of buffering to IO operations, which enhances performance by reducing the number of interactions with the underlying source or destination. Classes like `BufferedReader` and `BufferedOutputStream` fall into this category.
4. **Data Streams and Object Streams** These allow reading and writing Java primitive types and objects in a portable, structured format. This includes classes like `DataInputStream`, `DataOutputStream`, `ObjectInputStream`, and `ObjectOutputStream`.
5. **NIO (New IO)** A scalable, non-blocking IO API introduced in Java 1.4 for high-performance applications. It introduces channels, buffers, and selectors.
6. **AIO (Asynchronous IO)** Introduced in Java 7 (NIO.2), this allows asynchronous, callback-based IO processing for file and socket operations.

1.2 History and Evolution of Java IO

The Java Input/Output (IO) system has evolved significantly since the language's early days. What began as a relatively simple stream-based API in Java 1.0 has grown into a comprehensive and high-performance framework capable of supporting the demands of modern applications, including asynchronous IO, file watching, memory mapping, and scalable non-blocking networking.

1.2.1 The Original IO API (java.io)

The first version of Java, released in 1996, introduced the `java.io` package — a set of classes and interfaces designed around the concept of **streams**. Streams abstract data input and output as sequences of bytes or characters, making IO operations device-agnostic. The design was elegant in its simplicity:

- **Byte streams** (`InputStream`, `OutputStream`) handled binary data.

- **Character streams** (`Reader`, `Writer`) were introduced to manage textual data with support for Unicode.

Developers could wrap streams for buffering (`BufferedInputStream`), filtering (`FilterInputStream`), or for handling data primitives (`DataInputStream`, `DataOutputStream`). For object serialization, Java provided `ObjectInputStream` and `ObjectOutputStream`.

While effective for many use cases, this IO model had **several limitations**, particularly in applications that required high concurrency, performance tuning, or low-level control over data transfer.

1.2.2 Limitations of the Original IO Model

1. **Blocking IO**: The original stream-based IO was blocking by design. When a thread performed a read or write operation, it would block (pause) until the operation was complete. In a server handling many clients, this meant a separate thread was needed for each connection — a scalability bottleneck.
2. **Lack of Multiplexing**: Java IO lacked the ability to monitor multiple data sources with a single thread. Other languages and systems (e.g., C with `select()` or `epoll`) already supported this.
3. **Limited File and Socket Control**: The original API offered minimal access to underlying system-level features like file locking, memory-mapped files, or direct buffer management.
4. **No Asynchronous IO**: There was no built-in mechanism for non-blocking or asynchronous file or socket IO, limiting responsiveness in GUI or network-heavy applications.

These shortcomings led to the introduction of a new architecture.

1.2.3 The Birth of Java NIO (New IO) Java 1.4

With the release of Java 1.4 in 2002, the **java.nio** (New IO) package was introduced. It was a major redesign focused on performance, scalability, and fine-grained control over IO operations. Java NIO was not a replacement but a supplement to the existing IO API.

Key innovations in Java NIO included:

- **Channels**: Unlike streams, channels support bi-directional data flow and are non-blocking. Examples include `FileChannel`, `SocketChannel`, and `DatagramChannel`.
- **Buffers**: NIO introduced `ByteBuffer`, `CharBuffer`, and others as containers for data. Buffers enabled efficient read/write operations, and unlike streams, they gave the programmer control over how data was stored and accessed.

- **Selectors**: Selectors allowed a single thread to monitor multiple channels for IO readiness (read, write, accept, connect), enabling scalable event-driven servers.
- **Memory-Mapped Files**: Developers could map files directly into memory using `MappedByteBuffer`, providing ultra-fast file access and manipulation.

Java NIO marked a turning point in IO programming by aligning Java with system-level capabilities offered by native OS APIs.

1.2.4 Enhancements in Java NIO.2 Java 7

Java 7 introduced **NIO.2**, an enhanced version of NIO that added several much-needed features:

- **Asynchronous IO (AIO)**: The `java.nio.channels` package was expanded to support asynchronous file and socket channels (`AsynchronousFileChannel`, `AsynchronousSocketChannel`) with `Future` and `CompletionHandler` support, allowing true non-blocking, callback-based IO.
- **Improved File Handling**: The `java.nio.file` package introduced the powerful `Path`, `Files`, and `FileSystems` classes, replacing the older `File` class for most use cases. It also included symbolic link handling, directory walking, and file attribute APIs.
- **WatchService API**: NIO.2 included the ability to monitor file system changes like create, delete, and modify events — crucial for building reactive applications and file watchers.

1.2.5 Recent Developments and Beyond

Since Java 8 and onwards, enhancements to IO and NIO have been more incremental but no less important:

- Java 9 added improved support for reactive programming via the `Flow` API.
- Java 11 introduced additional file utility methods and enhanced character set support.
- Future releases, such as Java 21 and beyond, continue to refine IO with **Project Loom**, which introduces lightweight virtual threads—making traditional blocking IO more scalable by drastically reducing the cost of thread-per-connection models.

1.3 Overview of Stream-Based IO

In Java, a **stream** is a sequence of data elements made available over time. Think of a stream as a channel or conduit through which data flows. It could represent a flow of bytes coming from a file, data being read from the keyboard, or characters being sent to a printer.

Streams in Java abstract the underlying source or destination of the data. This means that the same `InputStream` interface can be used to read data from a file, a socket, or even memory — the developer doesn't have to worry about the underlying mechanics of the data source or sink.

Streams in Java come in two major categories:

- **Input Streams** – for reading data into a program.
- **Output Streams** – for writing data out from a program.

This model allows Java to provide a flexible, extensible, and consistent way of handling IO across different types of data sources and destinations.

1.3.1 How Streams Abstract IO

The stream abstraction hides the complexity of IO operations and provides a simple programming model:

- For **input**, data flows **into** the program from an external source.
- For **output**, data flows **out** of the program to an external destination.

This model is **linear** — data is read or written sequentially, one element at a time (usually a byte or character). You don't need to load entire files into memory or manually manage file pointers. Instead, the stream interface provides high-level methods such as `read()`, `write()`, `close()`, and others.

This abstraction allows developers to work with different types of data streams using the same programming paradigm, whether reading a file from disk, a web response from a URL, or user input from the console.

1.3.2 Types of Streams in Java

Java divides its stream classes into two main groups:

1. **Byte Streams** (binary data)
    - Classes: `InputStream` and `OutputStream` (abstract base classes)
    - Used for: reading and writing raw bytes (e.g., images, audio, binary files)

2. **Character Streams** (text data)
    - Classes: `Reader` and `Writer` (abstract base classes)
    - Used for: reading and writing characters, with support for Unicode and encodings

We'll look at both byte and character stream examples below.

1.3.3 Byte Stream Example: Reading and Writing Bytes

The following example shows how to use `FileInputStream` and `FileOutputStream` to copy a file.

Full runnable code:

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

public class ByteStreamExample {
    public static void main(String[] args) {
        try (
            FileInputStream input = new FileInputStream("input.dat");
            FileOutputStream output = new FileOutputStream("output.dat");
        ) {
            int byteData;
            while ((byteData = input.read()) != -1) {
                output.write(byteData);
            }
            System.out.println("File copied successfully using byte streams.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- `FileInputStream` reads one byte at a time from `input.dat`.
- `FileOutputStream` writes that byte to `output.dat`.
- This is a simple and direct way to copy binary data.

1.3.4 Character Stream Example: Reading and Writing Text

When dealing with text, it's better to use character streams to handle encoding properly.

Full runnable code:

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

public class CharacterStreamExample {
    public static void main(String[] args) {
        try (
            FileReader reader = new FileReader("input.txt");
            FileWriter writer = new FileWriter("output.txt");
        ) {
            int charData;
            while ((charData = reader.read()) != -1) {
                writer.write(charData);
            }
            System.out.println("File copied successfully using character streams.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- `FileReader` reads characters from a text file.
- `FileWriter` writes characters to another text file.
- Character streams are ideal for handling Unicode text.

1.3.5 Stream Chaining (Wrapping Streams)

Java IO encourages **composition** of stream objects for added functionality. For example, you can wrap a `FileInputStream` with a `BufferedInputStream` for performance.

Full runnable code:

```java
import java.io.*;

public class BufferedStreamExample {
    public static void main(String[] args) {
        try (
            BufferedInputStream bufferedInput = new BufferedInputStream(new FileInputStream("input.txt"))
            BufferedOutputStream bufferedOutput = new BufferedOutputStream(new FileOutputStream("output
        ) {
            int data;
            while ((data = bufferedInput.read()) != -1) {
                bufferedOutput.write(data);
            }
            System.out.println("File copied with buffering.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Buffered streams reduce the number of disk access operations, making reading and writing faster by using an internal buffer.

1.3.6 Key Methods in Stream-Based IO

Here are some common methods used in stream-based IO:

- `read()` – reads a byte or character from an input stream.
- `write(int b)` – writes a byte or character to an output stream.
- `close()` – closes the stream and releases system resources.
- `flush()` – forces any buffered output to be written.

1.3.7 Recap

Stream-based IO in Java provides a simple and powerful way to work with data. Whether you're handling files, network sockets, or console input/output, the stream abstraction lets you focus on **how** data flows rather than **where** it comes from or goes.

Streams make IO operations consistent and composable across different sources and formats. By separating byte and character handling, Java also helps ensure correctness in processing both binary and textual data. As we move further into the Java IO ecosystem, you'll see how these foundational concepts scale into more advanced topics like buffered IO, object serialization, and non-blocking channels.

1.4 Blocking vs Non-blocking IO

When writing IO-based programs in Java, one of the most important design considerations is whether to use **blocking** or **non-blocking** IO. The distinction lies in **how a thread behaves when it attempts to read or write data** and whether it must wait for the operation to complete.

1.4.1 Blocking IO: Traditional and Simple

Definition

Blocking IO refers to an IO model where the thread making a read or write call **waits (blocks)** until the operation completes. This model is straightforward and easy to implement,

but it becomes inefficient when dealing with many concurrent IO tasks.

In Java, the classic IO streams (`InputStream`, `OutputStream`, `Reader`, `Writer`) follow the **blocking IO model**.

How It Works

Imagine you're at a bank teller window (the thread), and you're waiting for a transaction to finish (the IO operation). You **cannot leave** until the teller is done — even if it takes a long time.

Here's a simple blocking IO example:

Full runnable code:

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

public class Test {

    public static void main(String[] argv) throws Exception {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter something: ");
        String input = reader.readLine(); // This blocks until the user presses Enter
        System.out.println("You entered: " + input);

    }
}
```

In this case, the thread is blocked at `readLine()` until the user provides input.

Advantages of Blocking IO

- Simple and intuitive to implement.
- Suitable for applications with limited and predictable concurrency (e.g., desktop apps, scripts).

Drawbacks of Blocking IO

- **Scalability limits**: One thread per connection becomes expensive in terms of memory and context-switching.
- **Inefficiency under high load**: Many threads may spend most of their time idle, waiting for IO.

1.4.2 Non-Blocking IO: Scalable and Efficient

Definition

Non-blocking IO allows threads to **initiate IO operations and continue doing other work** while waiting for the operation to complete. Instead of waiting, the program is notified

when the data is ready to be read or written.

In Java, this model is supported by the **Java NIO (New IO)** package, introduced in Java 1.4. NIO uses `SelectableChannel`, `Selector`, and `Buffer` classes to achieve non-blocking behavior.

How It Works

Back to our bank analogy: with non-blocking IO, you fill out a request slip (register interest in IO), drop it in a box, and move on. The bank calls your number (via a selector) when your transaction is ready.

Here's a conceptual snippet using NIO:

Full runnable code:

```
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Set;

public class Test {

    public static void main(String[] argv) throws Exception {
        SocketChannel channel = SocketChannel.open();
        channel.configureBlocking(false);
        Selector selector = Selector.open();
        channel.register(selector, SelectionKey.OP_READ);

        // Later in event loop
        selector.select(); // Blocks until at least one channel is ready
        Set<SelectionKey> readyKeys = selector.selectedKeys();
        for (SelectionKey key : readyKeys) {
            if (key.isReadable()) {
                // Data is ready to be read from the channel
            }
        }
    }
}
```

Advantages of Non-Blocking IO

- **Highly scalable**: One thread can handle thousands of connections.
- **Efficient resource usage**: Threads aren't blocked waiting on IO.
- Ideal for **network servers**, **real-time systems**, and **high-concurrency applications**.

Drawbacks of Non-Blocking IO

- **More complex code**: Requires managing selectors, buffers, and event loops.
- **Callback and state management** can become tricky.
- Not always beneficial for small, simple applications.

1.4.3 Visualizing the Difference

Imagine an **airport check-in** scenario:

- **Blocking IO**: Every passenger waits in line at a single desk until the desk is free.
- **Non-blocking IO**: All passengers enter a lounge, check in on tablets, and are notified when their boarding pass is ready.

Feature	Blocking IO	Non-blocking IO
Thread behavior	Waits until IO is complete	Continues other work while waiting
Simplicity	Simple and straightforward	More complex (selectors, buffers)
Scalability	Limited (1 thread per connection)	High (many connections per thread)
Resource efficiency	Low (many idle threads)	High (fewer threads used efficiently)
Common usage	Desktop apps, file operations	Web servers, chat servers, real-time IO

1.4.4 Use Case Scenarios

Blocking IO Best When:

- Your application handles a **small number of users or files**.
- IO operations are quick or infrequent.
- Simplicity and readability are more important than scalability.

Example: A desktop word processor saving and loading text files. It doesn't need to handle thousands of simultaneous IO operations.

Non-blocking IO Best When:

- You are building **high-performance servers** (e.g., web servers, chat systems).
- You need to handle **thousands of client connections concurrently**.
- Latency and responsiveness are critical.

Example: A multiplayer game server that needs to serve hundreds of players over TCP with minimal delay.

1.4.5 Recap

Choosing between blocking and non-blocking IO depends on your application's needs. **Blocking IO is easier to implement** and ideal for small-scale applications. **Non-blocking IO scales far better**, especially when handling many concurrent IO operations, but requires

more effort and careful design.

In modern Java, developers can also explore **asynchronous IO (AIO)** and emerging features like **virtual threads** from Project Loom, which attempt to combine the simplicity of blocking IO with the scalability of non-blocking IO — providing even more flexibility in managing IO workloads.

1.5 Synchronous vs Asynchronous IO

In Java, how a program handles input and output (IO) operations has a profound impact on its performance and responsiveness. Two major models exist: **synchronous IO** and **asynchronous IO**. Both are essential in different scenarios and serve as the backbone of modern file and network communication.

1.5.1 What Is Synchronous IO?

Synchronous IO means that a thread initiates an IO operation and then **waits until the operation completes** before continuing. This model follows a **step-by-step, blocking execution flow**, making it predictable and easy to understand.

Execution Flow

- The program executes one instruction at a time.
- When a read/write operation is triggered, the thread **blocks** until it finishes.
- Afterward, the next instruction is executed.

This behavior is analogous to standing in line at a coffee shop: you place your order, wait until it's ready, then proceed.

Java APIs for Synchronous IO

Most of Java's classic IO classes use synchronous behavior. Examples include:

- `FileInputStream` / `FileOutputStream`
- `BufferedReader` / `BufferedWriter`
- `Socket` / `ServerSocket`
- `FileReader` / `FileWriter`

Example:

Full runnable code:

```
import java.io.BufferedReader;
import java.io.FileReader;
```

```java
public class Test {

    public static void main(String[] argv) throws Exception {
        BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
        String line = reader.readLine(); // Synchronous: thread waits until a line is read
        System.out.println(line);
        reader.close();

    }
}
```

Use Cases for Synchronous IO

- Applications with **limited concurrency**.
- Simple **desktop utilities**, scripts, or CLI tools.
- Situations where **code clarity** and **sequential flow** matter more than throughput.

Benefits of Synchronous IO

- Simpler code: Linear, easier to read and debug.
- Good performance for small workloads or isolated IO tasks.

Drawbacks

- **Inefficient under high load**: Threads spend time waiting.
- **Scalability limits**: Each IO task requires a dedicated thread.

1.5.2 What Is Asynchronous IO?

Asynchronous IO (AIO) allows a thread to initiate an IO operation and then **continue executing other tasks**. When the IO operation completes, the thread is notified via a **callback**, **future**, or **event**, rather than blocking.

Execution Flow

- The thread starts an IO operation.
- Instead of waiting, it continues other work.
- When the IO is complete, the result is processed through a notification mechanism.

This is like ordering a drink at a kiosk and receiving a text when it's ready, so you don't have to wait in line.

Java APIs for Asynchronous IO

Java introduced built-in support for asynchronous IO in **Java 7** with the `java.nio.channels` package. Key classes include:

- `AsynchronousFileChannel`
- `AsynchronousSocketChannel`

- `CompletionHandler`
- `Future` (from `java.util.concurrent`)

Example using Future:

Full runnable code:

```
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousFileChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.concurrent.Future;

public class Test {

    public static void main(String[] argv) throws Exception {
        Path path = Paths.get("data.txt");
        AsynchronousFileChannel channel = AsynchronousFileChannel.open(path, StandardOpenOption.READ);
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        Future<Integer> result = channel.read(buffer, 0); // Non-blocking
        while (!result.isDone()) {
            System.out.println("Doing other work...");
        }
        // Now read is complete
        System.out.println("Bytes read: " + result.get());
        channel.close();
    }
}
```

Example using CompletionHandler:

Full runnable code:

```
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousFileChannel;
import java.nio.channels.CompletionHandler;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class Test {

    public static void main(String[] argv) throws Exception {
        Path path = Paths.get("data.txt");
        AsynchronousFileChannel channel = AsynchronousFileChannel.open(path, StandardOpenOption.READ);
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        channel.read(buffer, 0, buffer, new CompletionHandler<Integer, ByteBuffer>() {
            @Override
            public void completed(Integer bytesRead, ByteBuffer buf) {
                System.out.println("Read completed: " + bytesRead + " bytes");
            }

            @Override
            public void failed(Throwable exc, ByteBuffer buf) {
```

```
                System.err.println("Read failed: " + exc.getMessage());
            }
        });
    }
}
```

Use Cases for Asynchronous IO

- **Scalable network servers** (e.g., web servers, chat systems).
- **Responsive GUIs** and mobile apps where UI should not freeze during IO.
- **High-throughput applications** where IO latency must be masked with parallel computation.

Benefits of Asynchronous IO

- Better **CPU utilization**: threads aren't idle while waiting on IO.
- **Highly scalable**: suitable for thousands of simultaneous IO operations.
- Supports **event-driven** and **reactive programming** models.

Drawbacks

- **Complex code**: harder to follow, especially with nested callbacks.
- Requires **careful error handling** and state management.
- Debugging and testing are more involved.

1.5.3 Key Differences at a Glance

Feature	Synchronous IO	Asynchronous IO
Execution model	Blocking	Non-blocking
Thread behavior	Waits for IO to complete	Continues immediately after starting IO
Complexity	Simple	More complex
Performance (high concurrency)	Poor	Excellent
API examples	`FileInputStream`, `BufferedReader`	`AsynchronousFileChannel`, `CompletionHandler`
Use case	Command-line tools, scripts	Network servers, UI apps, high-load systems

1.5.4 Recap

The choice between **synchronous** and **asynchronous IO** depends on your application's scale, complexity, and responsiveness requirements. **Synchronous IO** is ideal for simple tasks and sequential processing. It's easy to implement and sufficient when the number of concurrent operations is small.

In contrast, **asynchronous IO** is designed for **scalability and performance**, particularly in IO-bound, multi-client environments. Java's support for AIO with `AsynchronousChannel` classes empowers developers to write responsive, high-throughput applications — though at the cost of increased code complexity.

As you continue exploring Java IO and NIO, you'll see how combining different models — or even using newer features like **virtual threads** from Project Loom — can help you balance simplicity and performance in your applications.

Chapter 2.

Java IO Fundamentals

1. Byte Streams vs Character Streams
2. InputStream and OutputStream Classes
3. Reader and Writer Classes
4. File Handling Basics
5. Buffered Streams and Their Importance
6. Data Streams for Primitive Types

2 Java IO Fundamentals

2.1 Byte Streams vs Character Streams

Java IO provides two primary types of streams to handle input and output operations: **byte streams** and **character streams**. Understanding the distinction between these two is crucial for selecting the right tools when reading or writing data in a Java program.

2.1.1 Byte Streams

Definition

Byte streams are designed to handle **raw binary data** — such as images, audio files, or serialized objects. These streams read and write **8-bit bytes**, making them suitable for any kind of binary input or output.

Java provides the following abstract base classes for byte streams:

- `InputStream` (for reading)
- `OutputStream` (for writing)

All byte stream classes in Java are derived from these.

Typical Use Cases

- Reading and writing image files (e.g., `.jpg`, `.png`)
- Handling binary data from sockets or files
- Copying files regardless of their format

Example: Copying a Binary File Using Byte Streams

Full runnable code:

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

public class ByteStreamExample {
    public static void main(String[] args) {
        try (
            // Open input and output streams for binary file
            FileInputStream input = new FileInputStream("input.jpg");
            FileOutputStream output = new FileOutputStream("output.jpg");
        ) {
            int data;
            // Read and write one byte at a time
            while ((data = input.read()) != -1) {
                output.write(data);
            }
            System.out.println("File copied successfully using byte streams.");
```

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

Explanation:

- This example reads a `.jpg` file byte-by-byte and writes it to another file.
- It does not interpret the content — just moves raw bytes.

2.1.2 Character Streams

Definition

Character streams are designed to handle **text data**, dealing with **16-bit Unicode characters**. These streams automatically handle character encoding and decoding, making them ideal for reading and writing text files.

Java provides the following abstract base classes for character streams:

- `Reader` (for reading characters)
- `Writer` (for writing characters)

These streams are encoding-aware and useful when working with text in any language.

Typical Use Cases

- Reading and writing `.txt`, `.csv`, `.xml`, and other plain-text files
- Processing user input or log files
- Outputting human-readable reports

Example: Copying a Text File Using Character Streams

Full runnable code:

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

public class CharacterStreamExample {
    public static void main(String[] args) {
        try (
            // Open character streams for text file
            FileReader reader = new FileReader("input.txt");
            FileWriter writer = new FileWriter("output.txt");
        ) {
            int ch;
            // Read and write one character at a time
            while ((ch = reader.read()) != -1) {
                writer.write(ch);
```

```
            }
            System.out.println("File copied successfully using character streams.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- This program copies text from `input.txt` to `output.txt` using character streams.
- It understands character encoding (e.g., UTF-8, UTF-16) and works well with international text.

2.1.3 Key Differences Between Byte Streams and Character Streams

Feature	Byte Streams	Character Streams
Base Classes	`InputStream` / `OutputStream`	`Reader` / `Writer`
Data Type	Bytes (8-bit)	Characters (16-bit Unicode)
Suitable For	Binary data (images, files)	Text data (logs, documents)
Encoding Awareness	Not aware of encoding	Automatically handles encoding
Performance (text)	May corrupt text without encoding	Ideal for reading/writing text
Common Subclasses	`FileInputStream`, `BufferedInputStream`	`FileReader`, `BufferedReader`

2.1.4 When to Use Which

- Use **byte streams** when working with **non-text data** (e.g., media files, PDFs, serialized objects).
- Use **character streams** for **text files** or any situation where character encoding matters.

For example:

- Use `FileInputStream` to copy a binary `.pdf` file.
- Use `FileReader` to read a `.txt` file containing human-readable content.

2.1.5 Recap

Java's stream architecture cleanly separates IO into **byte streams** and **character streams**, each tailored to specific types of data. Byte streams provide raw access to data, making them perfect for binary files, while character streams add encoding support, making them safer and more efficient for text.

By choosing the appropriate stream type, developers can avoid common bugs such as text corruption and improve the performance and maintainability of their applications. Understanding this distinction is a foundational skill for mastering Java IO.

2.2 InputStream and OutputStream Classes

In Java's IO system, the `InputStream` and `OutputStream` classes form the foundation for all **byte stream** input and output operations. They enable reading and writing of binary data, such as images, audio files, PDF documents, or raw network data.

These two abstract classes reside in the `java.io` package and define the core API for handling **low-level byte IO**. All other byte-based stream classes in Java either extend or decorate these two base classes.

2.2.1 InputStream: Reading Bytes from a Source

`InputStream` is an **abstract class** that provides methods to read **one byte at a time**, or an array of bytes, from a source such as a file, socket, or byte array.

Key Methods

Method	Description
`int read()`	Reads one byte of data and returns it as an int (0–255), or `-1` if end of stream is reached.
`int read(byte[] b)`	Reads bytes into the provided array.
`int read(byte[] b, int off, int len)`	Reads up to `len` bytes into array `b`, starting at offset `off`.
`void close()`	Closes the stream and releases any resources.
`int available()`	Returns the number of bytes that can be read without blocking.

Common Subclasses

- `FileInputStream` – Reads from a file.
- `BufferedInputStream` – Adds buffering to reduce IO calls.

- `ByteArrayInputStream` – Reads from a byte array.
- `ObjectInputStream` – Reads serialized Java objects.
- `PipedInputStream` – Reads from a connected PipedOutputStream (used in thread communication).

Example: Reading Bytes from a File

Full runnable code:

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

public class InputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("input.dat")) {
            int byteData;
            // Read one byte at a time until end of file
            while ((byteData = fis.read()) != -1) {
                System.out.print((char) byteData); // Print byte as character (for demo)
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Note: This approach works best for small files. For larger files, use buffered streams.

2.2.2 OutputStream: Writing Bytes to a Destination

`OutputStream` is the abstract superclass for all classes that **write raw bytes** to an output destination, such as a file, byte array, or network socket.

Key Methods

Method	Description
`void write(int b)`	Writes the specified byte (lower 8 bits of int).
`void write(byte[] b)`	Writes all bytes from the given array.
`void write(byte[] b, int off, int len)`	Writes `len` bytes from the array starting at offset `off`.
`void flush()`	Forces any buffered output bytes to be written out.
`void close()`	Closes the stream and releases resources.

2.2.3 Common Subclasses

- `FileOutputStream` – Writes to a file.
- `BufferedOutputStream` – Adds buffering to improve performance.
- `ByteArrayOutputStream` – Writes to an internal byte array.
- `ObjectOutputStream` – Writes serialized objects.
- `PipedOutputStream` – Connects to a `PipedInputStream`.

2.2.4 Example: Writing Bytes to a File

Full runnable code:

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

public class OutputStreamExample {
    public static void main(String[] args) {
        String message = "Hello, this is a test message!";
        try (FileOutputStream fos = new FileOutputStream("output.dat")) {
            // Convert the string to bytes and write to file
            fos.write(message.getBytes());
            System.out.println("Message written to file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation: `getBytes()` converts the string to a byte array, which is then written to the file.

2.2.5 Combining InputStream and OutputStream: File Copy Example

Here's a practical example that demonstrates using both `InputStream` and `OutputStream` to **copy the contents of a binary file**.

Full runnable code:

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

public class FileCopyExample {
    public static void main(String[] args) {
        try (
            FileInputStream input = new FileInputStream("source.dat");
            FileOutputStream output = new FileOutputStream("copy.dat");
```

```
        ) {
            byte[] buffer = new byte[1024]; // 1KB buffer
            int bytesRead;
            // Read from source and write to destination
            while ((bytesRead = input.read(buffer)) != -1) {
                output.write(buffer, 0, bytesRead);
            }
            System.out.println("File copied successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- The file is read in chunks (1KB at a time).
- Reduces the number of IO calls for better performance.
- The use of try-with-resources ensures streams are closed automatically.

2.2.6 Best Practices When Using Streams

- **Always close streams**: Use try-with-resources to ensure automatic closing.
- **Buffer your streams**: Use `BufferedInputStream` and `BufferedOutputStream` to reduce disk access.
- **Check for available bytes**: Use `available()` carefully; it provides an estimate, not a guarantee.
- **Handle exceptions properly**: Always catch `IOException` or let it propagate.

2.2.7 Recap

`InputStream` and `OutputStream` are the backbone of Java's byte stream IO system. They provide a flexible, low-level interface for reading and writing binary data across a variety of sources and destinations. By understanding these classes and their common subclasses, you gain the tools necessary to handle a wide range of IO tasks — from simple file operations to complex network data processing. Mastering them sets the foundation for efficient, reliable IO handling in any Java application.

2.3 Reader and Writer Classes

Java provides a distinct set of stream classes for **character-based input and output**, built around the `Reader` and `Writer` abstract classes. These classes were introduced to support **Unicode** and allow seamless handling of text data in a platform- and encoding-independent manner.

While `InputStream` and `OutputStream` are designed for **byte-level** operations, `Reader` and `Writer` operate at the **character level**, abstracting away byte-to-character conversions and making it easier to work with textual content.

2.3.1 Why Character Streams?

Early Java IO relied solely on byte streams (`InputStream` and `OutputStream`). However, byte streams aren't encoding-aware — they deal with raw bytes, so developers had to manually convert bytes to characters, which led to common bugs and encoding issues.

To resolve this, the Java platform introduced `Reader` and `Writer`, which:

- Automatically handle character encoding and decoding
- Simplify reading and writing Unicode-compliant text
- Improve code clarity when dealing with textual data

2.3.2 Reader Class

`Reader` is an **abstract base class** for reading character streams. It reads **16-bit Unicode characters** from text sources such as files, memory, or network streams.

Key Methods

Method	Description
`int read()`	Reads a single character, returns -1 if end of stream.
`int read(char[] cbuf)`	Reads characters into an array.
`int read(char[] cbuf, int off, int len)`	Reads up to `len` characters into `cbuf` starting at `off`.
`void close()`	Closes the stream.
`boolean ready()`	Checks if the stream is ready to be read.

Common Subclasses

- `FileReader` – Reads characters from a file.
- `BufferedReader` – Adds buffering and line-based reading.

- `InputStreamReader` – Bridges byte streams to character streams with encoding support.
- `CharArrayReader` – Reads from a character array.
- `StringReader` – Reads from a string.

2.3.3 Writer Class

`Writer` is the **abstract superclass** for writing character streams. It writes characters, arrays, or strings to text destinations.

Key Methods

Method	Description
`void write(int c)`	Writes a single character.
`void write(char[] cbuf)`	Writes an array of characters.
`void write(String str)`	Writes a string.
`void flush()`	Flushes the stream (forces buffered data to be written).
`void close()`	Closes the stream.

Common Subclasses

- `FileWriter` – Writes characters to a file.
- `BufferedWriter` – Buffers characters to improve performance.
- `OutputStreamWriter` – Converts characters into bytes using specified encoding.
- `CharArrayWriter` – Writes to a character array in memory.
- `StringWriter` – Writes to a string buffer.

2.3.4 Example 1: Reading Text with FileReader and BufferedReader

Full runnable code:

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

public class ReaderExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            // Read line by line until end of file
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
```

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

Explanation:

- `BufferedReader` wraps `FileReader` to provide efficient, line-based reading.
- Ideal for reading text files in a readable and memory-efficient manner.

2.3.5 Example 2: Writing Text with FileWriter and BufferedWriter

Full runnable code:

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

public class WriterExample {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
            writer.write("Hello, world!");
            writer.newLine();
            writer.write("This was written using character streams.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- `BufferedWriter` improves performance by reducing direct disk writes.
- `newLine()` writes the system-dependent line separator.

2.3.6 Handling Character Encoding

Character streams like `FileReader` and `FileWriter` use the platform's default encoding, which can vary. To specify an encoding (e.g., UTF-8), use **bridging streams** like `InputStreamReader` and `OutputStreamWriter`.

Example: Reading with UTF-8 Encoding

Full runnable code:

```java
import java.io.*;

public class UTF8ReadExample {
    public static void main(String[] args) {
        try (
            InputStreamReader isr = new InputStreamReader(new FileInputStream("utf8.txt"), "UTF-8");
            BufferedReader reader = new BufferedReader(isr)
        ) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example: Writing with UTF-8 Encoding

Full runnable code:

```java
import java.io.*;

public class UTF8WriteExample {
    public static void main(String[] args) {
        try (
            OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("utf8-output.txt"), "U
            BufferedWriter writer = new BufferedWriter(osw)
        ) {
            writer.write("     "); // Japanese: "Hello, World!"
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Note: Always specify character encoding explicitly for portable, correct international text handling.

2.3.7 Key Differences Between Byte and Character Streams

Feature	Byte Streams (`InputStream`/`OutputStream`)	Character Streams (`Reader`/`Writer`)
Data Type	Bytes (8-bit)	Characters (16-bit Unicode)
Encoding Awareness	Not aware	Handles character encoding/decoding
Best For	Binary files (images, audio, PDFs)	Text files, source code, logs, CSVs

Feature	Byte Streams (`InputStream`/`OutputStream`)	Character Streams (`Reader`/`Writer`)
Example Classes	`FileInputStream`, `BufferedOutputStream`	`BufferedReader`, `FileWriter`, `PrintWriter`

2.3.8 Recap

The `Reader` and `Writer` classes bring structure and clarity to handling character data in Java. They solve the encoding challenges present in byte streams and allow developers to work naturally with Unicode and multi-language text. By choosing character streams over byte streams for text-based operations, Java developers can write more robust, maintainable, and internationalization-friendly code.

2.4 File Handling Basics

When working with files and directories in Java, the primary tool is the `java.io.File` class. Unlike input/output stream classes, `File` does not handle reading or writing the contents of a file—it **represents the abstract path** of a file or directory in the file system and allows you to **manipulate metadata and perform file-level operations** such as creating, deleting, renaming, or checking properties.

2.4.1 Understanding the `File` Class

The `File` class represents both files and directories. It is used to:

- Check if a file or directory exists
- Create new files or directories
- Delete or rename files/directories
- Get file metadata (e.g., size, path, permissions)

To use it, you simply create an instance of `File` by passing a pathname (as a `String` or `Path`):

```
import java.io.File;

File myFile = new File("example.txt");
```

This line does **not** create a physical file. It only creates a representation of the path. The actual file operations require calling methods on the `File` object.

2.4.2 Creating Files and Directories

To create a **new empty file**, use the `createNewFile()` method:

Full runnable code:

```
import java.io.File;
import java.io.IOException;

public class CreateFileExample {
    public static void main(String[] args) {
        File file = new File("sample.txt");
        try {
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- `createNewFile()` returns `true` if the file was created, `false` if it already exists.
- It throws an `IOException` if an error occurs (e.g., permissions issue).

To create a **directory**, use `mkdir()`:

```
File dir = new File("myFolder");
if (dir.mkdir()) {
    System.out.println("Directory created.");
} else {
    System.out.println("Failed to create directory.");
}
```

To create **nested directories**, use `mkdirs()`:

```
File nestedDir = new File("parent/child/grandchild");
nestedDir.mkdirs();  // creates all intermediate directories if needed
```

2.4.3 Deleting Files and Directories

To delete a file or empty directory, use the `delete()` method:

```
File file = new File("sample.txt");
if (file.delete()) {
    System.out.println("Deleted the file: " + file.getName());
} else {
    System.out.println("Failed to delete the file.");
}
```

Important:

- `delete()` does **not** delete non-empty directories.
- There's no built-in Java method to recursively delete directories in the `File` class; this must be implemented manually or by using `java.nio.file.Files`.

2.4.4 Renaming and Moving Files

To rename or move a file, use `renameTo()`:

```
File oldFile = new File("sample.txt");
File newFile = new File("renamed.txt");

if (oldFile.renameTo(newFile)) {
    System.out.println("File renamed successfully.");
} else {
    System.out.println("Rename failed.");
}
```

Note: This method can also move a file to a different directory.

2.4.5 Checking File Properties

The `File` class provides several methods to inspect the file or directory:

Full runnable code:

```
import java.io.File;

public class Test {

    public static void main(String[] argv) throws Exception {
        File file = new File("example.txt");

        System.out.println("File name: " + file.getName());
        System.out.println("Absolute path: " + file.getAbsolutePath());
        System.out.println("Parent: " + file.getParent());

        System.out.println("Exists: " + file.exists());
        System.out.println("Is directory: " + file.isDirectory());
        System.out.println("Is file: " + file.isFile());
        System.out.println("Readable: " + file.canRead());
        System.out.println("Writable: " + file.canWrite());
        System.out.println("Executable: " + file.canExecute());
        System.out.println("File size (bytes): " + file.length());
    }
}
```

These methods allow you to verify if a file exists, distinguish between files and directories, and check permissions.

2.4.6 Listing Files and Directories

To list files inside a directory, use `list()` or `listFiles()`:

Full runnable code:

```
import java.io.File;

public class Test {

    public static void main(String[] argv) throws Exception {
        File directory = new File("myFolder");

        if (directory.isDirectory()) {
            String[] fileNames = directory.list();
            for (String name : fileNames) {
                System.out.println(name);
            }
        }
    }
}
```

Or get `File` objects:

Full runnable code:

```
import java.io.File;

public class Test {

    public static void main(String[] argv) throws Exception {
        File directory = new File("myFolder");
        File[] files = directory.listFiles();
        for (File f : files) {
            System.out.println(f.getName() + (f.isDirectory() ? " (dir)" : " (file)"));
        }
    }
}
```

2.4.7 Working with Absolute and Relative Paths

- A **relative path** is relative to the program's working directory.
- An **absolute path** provides the full file location in the file system.

Full runnable code:

```
import java.io.File;

public class Test {

    public static void main(String[] argv) throws Exception {
        File relative = new File("sample.txt");
```

```
        File absolute = new File("/home/user/docs/sample.txt");

        System.out.println("Relative path: " + relative.getPath());
        System.out.println("Absolute path: " + absolute.getAbsolutePath());
    }
}
```

2.4.8 Summary of Common `File` Methods

Method	Purpose
`createNewFile()`	Creates a new file
`mkdir()` / `mkdirs()`	Creates directories
`delete()`	Deletes a file or empty directory
`renameTo(File)`	Renames or moves a file
`exists()`	Checks existence
`isFile()` / `isDirectory()`	Checks file type
`canRead()` / `canWrite()` / `canExecute()`	Checks permissions
`length()`	Gets file size in bytes
`list()` / `listFiles()`	Lists files in a directory

2.4.9 Recap

The `File` class in Java IO provides powerful tools to interact with the file system, enabling developers to create, delete, inspect, and manipulate files and directories. Although it doesn't handle file content, it is essential for managing file paths and metadata. For actual content processing, `Reader`/`Writer` or `InputStream`/`OutputStream` classes are used in conjunction. Mastering the `File` class sets the foundation for reliable and efficient file handling in Java applications.

2.5 Buffered Streams and Their Importance

In Java IO, reading and writing data directly to and from a source (like a file or socket) using unbuffered streams (e.g., `FileInputStream`, `FileOutputStream`) can be inefficient, especially when data is processed byte-by-byte or character-by-character. Buffered streams were introduced to solve this performance issue by **minimizing the number of expensive disk or network access operations** through the use of an in-memory buffer.

2.5.1 What is Buffering in IO?

Buffering refers to the technique of using a temporary memory area—a buffer—to store data before it's read or written. Instead of performing a system-level read/write operation for every byte or character, a buffered stream:

- Reads larger blocks of data into memory at once (for input)
- Writes larger blocks of data to the destination in one go (for output)

This **reduces the number of IO operations**, improving performance dramatically.

2.5.2 Buffered Streams in Java

Java provides four main buffered stream classes:

Buffered Stream Class	Base Class	Type
`BufferedInputStream`	`InputStream`	Byte-based input
`BufferedOutputStream`	`OutputStream`	Byte-based output
`BufferedReader`	`Reader`	Character-based input
`BufferedWriter`	`Writer`	Character-based output

2.5.3 How BufferedInputStream Works

`BufferedInputStream` wraps an existing `InputStream` and reads a block of bytes (default 8192 bytes) into an internal buffer. When your program reads from the stream, it accesses data from the buffer, not the file or socket directly—unless the buffer is empty.

Example: Reading a File Efficiently

Full runnable code:

```
import java.io.*;

public class BufferedInputExample {
    public static void main(String[] args) {
        try (
            FileInputStream fis = new FileInputStream("input.txt");
            BufferedInputStream bis = new BufferedInputStream(fis)
        ) {
            int byteData;
            while ((byteData = bis.read()) != -1) {
                System.out.print((char) byteData); // Convert byte to char
            }
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

Why it's efficient: Instead of one disk read per byte, a chunk of data is loaded at once, then served byte-by-byte from memory.

2.5.4 How BufferedOutputStream Works

`BufferedOutputStream` collects bytes in a memory buffer and writes them in chunks. This avoids frequent, slow writes to disk or a network stream.

Example: Writing to a File Efficiently

Full runnable code:

```
import java.io.*;

public class BufferedOutputExample {
    public static void main(String[] args) {
        try (
            FileOutputStream fos = new FileOutputStream("output.txt");
            BufferedOutputStream bos = new BufferedOutputStream(fos)
        ) {
            String message = "Buffered output stream example in Java.";
            bos.write(message.getBytes());
            bos.flush(); // Ensure data is written from buffer to file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Important: Always call `flush()` before closing to ensure remaining data in the buffer is written.

2.5.5 How BufferedReader Works

`BufferedReader` reads characters efficiently by wrapping a `Reader` (e.g., `FileReader` or `InputStreamReader`). It also provides **convenient methods like `readLine()`**, which are not available in unbuffered readers.

Example: Reading Text Line-by-Line

Full runnable code:

```java
import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (
            BufferedReader reader = new BufferedReader(new FileReader("notes.txt"))
        ) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line); // Print each line of the file
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Benefits:

- Efficient reading of characters, arrays, or lines.
- Less overhead than reading character-by-character.

2.5.6 How BufferedWriter Works

`BufferedWriter` buffers character data and writes it in bulk. It also includes a `newLine()` method to write platform-specific line separators.

Example: Writing Text with BufferedWriter

Full runnable code:

```java
import java.io.*;

public class BufferedWriterExample {
    public static void main(String[] args) {
        try (
            BufferedWriter writer = new BufferedWriter(new FileWriter("log.txt"))
        ) {
            writer.write("BufferedWriter is efficient for writing text.");
            writer.newLine();
            writer.write("It reduces the number of write operations.");
            writer.flush(); // Push remaining data to file
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Note: Like `BufferedOutputStream`, always flush the writer to avoid data loss.

2.5.7 When to Use Buffered Streams

Buffered streams are especially useful when:

- Working with large files
- Reading/writing data over slow IO channels (e.g., network)
- Processing structured text (like log files or CSVs)
- You need to minimize the overhead of IO operations

2.5.8 Default Buffer Size and Customization

By default, Java uses an **8 KB (8192 bytes)** buffer for buffered streams. You can specify a different size:

Full runnable code:

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;

public class Test {

    public static void main(String[] argv) throws Exception {
        BufferedInputStream bis = new BufferedInputStream(new FileInputStream("data.bin"), 16384); // 1
    }
}
```

Custom buffer sizes can optimize performance depending on the data volume and system architecture.

2.5.9 Recap

Buffered streams in Java are vital for **efficient IO performance**. By reading and writing data in blocks rather than byte-by-byte or character-by-character, they reduce the number of costly interactions with the file system or network. Whether you're dealing with binary or text data, buffered streams offer higher speed, lower latency, and improved application responsiveness. As a best practice, always prefer buffered streams unless working with very small data or special cases where buffering is unnecessary.

2.6 Data Streams for Primitive Types

Java's standard input and output streams (`InputStream` and `OutputStream`) operate at the byte level and lack the ability to directly read or write **Java primitive data types** (like `int`, `float`, or `boolean`). To bridge this gap, Java provides **data streams**, specifically `DataInputStream` and `DataOutputStream`, which enable applications to read and write **Java primitives and strings in a platform-independent and efficient way**.

2.6.1 Why Use Data Streams?

Data streams offer:

- **Direct support for Java primitive types**: `int`, `float`, `long`, `boolean`, `double`, etc.
- **Platform independence**: Data is written and read in a machine-neutral format.
- **Tight integration with `InputStream`/`OutputStream`**: These classes wrap around byte streams, extending their capabilities.

This makes them ideal for saving and restoring structured binary data in a format that can later be decoded accurately—without manual byte parsing.

2.6.2 DataOutputStream

`DataOutputStream` extends `FilterOutputStream` and provides methods to write Java primitives in a standardized binary format.

Key Methods

Method	Description
`writeInt(int v)`	Writes 4 bytes for an int
`writeDouble(double v)`	Writes 8 bytes for a double
`writeBoolean(boolean v)`	Writes 1 byte (`0` or `1`)
`writeUTF(String s)`	Writes a string in modified UTF-8 format
`writeChar(int v)`	Writes 2 bytes for a char

Constructor

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin"));
```

2.6.3 DataInputStream

`DataInputStream` extends `FilterInputStream` and complements `DataOutputStream` by reading data in the same format it was written.

Key Methods

Method	Description
`readInt()`	Reads 4 bytes and returns an int
`readDouble()`	Reads 8 bytes and returns a double
`readBoolean()`	Reads 1 byte and returns boolean
`readUTF()`	Reads a string in modified UTF-8
`readChar()`	Reads 2 bytes and returns a char

Constructor

```
DataInputStream dis = new DataInputStream(new FileInputStream("data.bin"));
```

Important: The order in which you read data **must exactly match** the order it was written.

2.6.4 Example: Writing Primitive Data Using DataOutputStream

Full runnable code:

```
import java.io.*;

public class DataOutputExample {
    public static void main(String[] args) {
        try (DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin"))) {
            dos.writeInt(42);              // 4 bytes
            dos.writeDouble(3.14159);      // 8 bytes
            dos.writeBoolean(true);        // 1 byte
            dos.writeUTF("Hello, Java");   // String with length prefix
            dos.writeChar('J');            // 2 bytes
            System.out.println("Data written to file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- Each method writes the respective primitive in a binary format.
- `writeUTF` prepends the string with its length in bytes, encoded in modified UTF-8.

2.6.5 Example: Reading Primitive Data Using DataInputStream

Full runnable code:

```
import java.io.*;

public class DataInputExample {
    public static void main(String[] args) {
        try (DataInputStream dis = new DataInputStream(new FileInputStream("data.bin"))) {
            int intValue = dis.readInt();
            double doubleValue = dis.readDouble();
            boolean boolValue = dis.readBoolean();
            String strValue = dis.readUTF();
            char charValue = dis.readChar();

            System.out.println("Read values:");
            System.out.println("Int: " + intValue);
            System.out.println("Double: " + doubleValue);
            System.out.println("Boolean: " + boolValue);
            System.out.println("String: " + strValue);
            System.out.println("Char: " + charValue);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- Values are read back in **exactly the same order and format** they were written.
- Mismatching read/write order will result in corrupt data or exceptions (e.g., `EOFException`).

2.6.6 Common Use Cases

- **Saving structured data**: Such as game state, user settings, or application configuration in binary form.
- **Reading/writing binary protocols**: When interacting with custom or third-party binary data formats.
- **Socket programming**: To send and receive primitive values over network streams efficiently.

2.6.7 Handling Exceptions

Data streams can throw several checked exceptions:

- `IOException`: General IO issues (file not found, access denied, etc.).

- `EOFException`: Reached the end of file unexpectedly while reading.

Always wrap stream operations in `try-with-resources` to ensure automatic resource management.

2.6.8 Important Considerations

- Data written with `DataOutputStream` can only be reliably read with `DataInputStream` (or a decoder that understands the format).
- `writeUTF()` is **not suitable** for storing very large strings (limit: 65,535 bytes).
- For **textual data**, prefer character streams (e.g., `BufferedReader`, `BufferedWriter`) unless binary encoding is required.

2.6.9 Recap

`DataInputStream` and `DataOutputStream` are powerful tools for binary IO in Java. They eliminate the complexity of manually converting primitives to and from byte arrays, ensuring accurate and portable storage of Java's basic data types. Whether you're building a low-level file format or communicating over sockets, data streams provide a clean, efficient, and consistent way to serialize and deserialize primitive values.

Chapter 3.

File and Directory Operations

1. Working with Files: File Class Overview
2. Reading and Writing Files with FileInputStream and FileOutputStream
3. Reading and Writing Files with FileReader/FileWriter
4. File Permissions and Attributes
5. Listing Files and Directories
6. File Deletion, Renaming, and Creation

3 File and Directory Operations

3.1 Working with Files: File Class Overview

In Java, the `java.io.File` class serves as an abstract representation of file and directory pathnames in the filesystem. Unlike stream classes that handle reading and writing data, the `File` class focuses on **metadata and structure**—such as verifying whether a file or directory exists, checking permissions, and creating or deleting files and directories.

The `File` class is part of the `java.io` package and plays a critical role in Java IO operations involving filesystems.

3.1.1 What is the `File` Class?

The `File` class encapsulates the **pathname** of a file or directory (absolute or relative). However, a `File` object **does not represent actual file content**—it represents a path string and provides methods to interact with the file or directory associated with that path.

Creating a `File` Object

```
File file1 = new File("example.txt");              // Relative path
File file2 = new File("/home/user/example.txt");   // Absolute path
```

This does **not create** a new file; it only creates a `File` object pointing to a path.

3.1.2 Querying File and Directory Properties

The `File` class provides several methods to inspect file properties:

Checking Existence

```
File file = new File("example.txt");
System.out.println("Exists: " + file.exists());
```

Checking Type

```
System.out.println("Is file: " + file.isFile());
System.out.println("Is directory: " + file.isDirectory());
```

Getting Size and Path

```
System.out.println("Absolute path: " + file.getAbsolutePath());
System.out.println("File name: " + file.getName());
System.out.println("Parent directory: " + file.getParent());
```

```
System.out.println("File size (bytes): " + file.length());
```

Checking Permissions

```
System.out.println("Readable: " + file.canRead());
System.out.println("Writable: " + file.canWrite());
System.out.println("Executable: " + file.canExecute());
```

These checks are particularly useful for verifying access before performing IO operations.

3.1.3 Creating Files and Directories

Creating a New File

```
File file = new File("newfile.txt");
try {
    if (file.createNewFile()) {
        System.out.println("File created.");
    } else {
        System.out.println("File already exists.");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

- `createNewFile()` creates an empty file only if it does not already exist.
- Returns `true` if a new file is created, `false` otherwise.

Creating Directories

```
File dir = new File("myDirectory");
if (dir.mkdir()) {
    System.out.println("Directory created.");
}

File nestedDir = new File("parent/child/grandchild");
if (nestedDir.mkdirs()) {
    System.out.println("Nested directories created.");
}
```

- `mkdir()` creates a single directory.
- `mkdirs()` creates a directory along with all necessary parent directories.

3.1.4 Deleting Files and Directories

Deleting a File or Directory

```
File file = new File("toDelete.txt");
if (file.delete()) {
    System.out.println("Deleted successfully.");
} else {
    System.out.println("Deletion failed.");
}
```

- `delete()` works for both files and empty directories.
- Deleting non-empty directories requires recursion or `java.nio.file.Files`.

3.1.5 Renaming and Moving Files

Renaming a File

```
File oldFile = new File("old.txt");
File newFile = new File("new.txt");

if (oldFile.renameTo(newFile)) {
    System.out.println("File renamed successfully.");
} else {
    System.out.println("Rename failed.");
}
```

- `renameTo()` can also be used to move files if the destination path differs.

3.1.6 Listing Directory Contents

You can list files and directories inside a folder using `list()` or `listFiles()`.

Listing File Names

```
File directory = new File("myDirectory");
String[] fileList = directory.list();

for (String name : fileList) {
    System.out.println(name);
}
```

Listing as File Objects

```
File[] files = directory.listFiles();
for (File f : files) {
    System.out.println(f.getName() + (f.isDirectory() ? " (dir)" : " (file)"));
}
```

These methods are helpful for building file browsers, searching files, or batch processing.

3.1.7 Working with Paths

Absolute vs Relative Paths

```
File relative = new File("data.txt");
System.out.println("Absolute path: " + relative.getAbsolutePath());

File absolute = new File("/usr/local/data.txt");
System.out.println("Is absolute: " + absolute.isAbsolute());
```

- `getCanonicalPath()` resolves symbolic links and `..` or `.` in the path.
- Use absolute paths when precise file locations are critical.

3.1.8 Practical Use Case: File Validation

```
public static void validateFile(String path) {
    File file = new File(path);
    if (!file.exists()) {
        System.out.println("File not found.");
        return;
    }
    if (!file.isFile()) {
        System.out.println("Not a regular file.");
        return;
    }
    if (!file.canRead()) {
        System.out.println("Cannot read file.");
        return;
    }

    System.out.println("File is ready for processing: " + file.getName());
}
```

This kind of validation is essential before attempting to read from or write to a file.

Full runnable code:

```
import java.io.File;
import java.io.IOException;

public class FileExample {
    public static void main(String[] args) {
        try {
            // Create a new file
            File file = new File("example.txt");
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
```

```
            } else {
                System.out.println("File already exists.");
            }

            // Check file properties
            System.out.println("Exists: " + file.exists());
            System.out.println("Is file: " + file.isFile());
            System.out.println("Is directory: " + file.isDirectory());
            System.out.println("Readable: " + file.canRead());
            System.out.println("Writable: " + file.canWrite());
            System.out.println("Executable: " + file.canExecute());
            System.out.println("Absolute path: " + file.getAbsolutePath());
            System.out.println("File size (bytes): " + file.length());

            // Create a directory
            File dir = new File("demoDir");
            if (dir.mkdir()) {
                System.out.println("Directory created: " + dir.getName());
            }

            // Rename file
            File renamed = new File("renamed_example.txt");
            if (file.renameTo(renamed)) {
                System.out.println("File renamed to: " + renamed.getName());
            }

            // List contents of current directory
            File current = new File(".");
            File[] files = current.listFiles();
            if (files != null) {
                System.out.println("\nDirectory contents:");
                for (File f : files) {
                    System.out.println(f.getName() + (f.isDirectory() ? " (dir)" : " (file)"));
                }
            }

            // Delete the renamed file
            if (renamed.delete()) {
                System.out.println("\nDeleted file: " + renamed.getName());
            }

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

3.1.9 Recap

The Java `File` class is a foundational component of the IO system, enabling applications to interact with the filesystem in a platform-independent way. While it doesn't handle file contents directly, it provides robust support for inspecting and manipulating files and directories. Mastery of `File` is essential for tasks like file validation, path resolution, directory

traversal, and metadata querying. For file content operations, it is typically used in conjunction with stream or reader/writer classes.

3.2 Reading and Writing Files with FileInputStream and FileOutputStream

In Java IO, the `FileInputStream` and `FileOutputStream` classes are fundamental for performing **byte-oriented file input and output operations**. These classes are part of the `java.io` package and are typically used when dealing with **binary data** such as images, audio files, PDFs, or raw byte sequences.

3.2.1 Understanding Byte Streams

Java's IO system is built around two types of streams:

- **Byte streams**, which deal with raw bytes (`InputStream`/`OutputStream` hierarchy)
- **Character streams**, which deal with characters (`Reader`/`Writer` hierarchy)

`FileInputStream` and `FileOutputStream` are **concrete implementations** of the byte stream classes specifically designed to work with files.

3.2.2 FileInputStream Reading Bytes from a File

`FileInputStream` allows you to **read bytes from a file**. It is ideal for reading binary files or when you need precise control over the byte-level structure of the data.

Key Methods

- `int read()` – reads a single byte, returns -1 at end of file.
- `int read(byte[] b)` – reads bytes into a buffer array.
- `int read(byte[] b, int off, int len)` – reads bytes into a buffer with offset and length.
- `void close()` – closes the stream and releases system resources.

Example: Reading a File Byte-by-Byte

Full runnable code:

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

```java
public class FileReadExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("input.bin")) {
            int byteData;
            while ((byteData = fis.read()) != -1) {
                System.out.print((char) byteData); // Cast to char for text output
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- The file `input.bin` is opened in binary mode.
- Each byte is read sequentially and printed as a character.
- The `try-with-resources` block ensures the stream is closed automatically.

3.2.3 FileOutputStream Writing Bytes to a File

`FileOutputStream` lets you **write bytes to a file**. It's commonly used for creating or modifying binary files.

Key Methods

- `void write(int b)` – writes a single byte.
- `void write(byte[] b)` – writes an entire byte array.
- `void write(byte[] b, int off, int len)` – writes part of a byte array.
- `void close()` – closes the stream.

Example: Writing Bytes to a File

Full runnable code:

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

public class FileWriteExample {
    public static void main(String[] args) {
        String data = "Java FileOutputStream Example";

        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            byte[] bytes = data.getBytes(); // Convert string to byte array
            fos.write(bytes);               // Write all bytes to file
            System.out.println("Data written to file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- The string is converted to bytes using `getBytes()`.
- The byte array is written to the file `output.txt`.
- If the file doesn't exist, it is created; if it exists, it is overwritten.

3.2.4 Writing in Append Mode

To **append data** to an existing file instead of overwriting it, use the `FileOutputStream` constructor with a `true` flag:

```
try (FileOutputStream fos = new FileOutputStream("log.txt", true)) {
    fos.write("New log entry\n".getBytes());
}
```

This mode is useful for logging or appending updates to a file.

3.2.5 Using Buffers for Efficiency

Reading or writing byte-by-byte is inefficient for large files. Use a **byte array buffer** to read/write chunks of data:

Buffered Reading and Writing

Full runnable code:

```
import java.io.*;

public class FileCopyExample {
    public static void main(String[] args) {
        try (
            FileInputStream fis = new FileInputStream("source.jpg");
            FileOutputStream fos = new FileOutputStream("copy.jpg")
        ) {
            byte[] buffer = new byte[4096]; // 4 KB buffer
            int bytesRead;

            while ((bytesRead = fis.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead); // Write only bytes read
            }

            System.out.println("File copied successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Advantages:

- Reduces the number of IO operations.
- Greatly improves performance, especially for large files.

3.2.6 Common Use Cases

- Reading and writing **binary files** (images, audio, executables)
- Implementing **file copying utilities**
- Saving **serialized binary data** (e.g., custom file formats)
- Writing **raw network data** to disk

For **text files**, `FileReader` and `FileWriter` (or their buffered counterparts) are preferred to handle character encoding.

3.2.7 Error Handling and Resource Management

Always close file streams to avoid resource leaks. The **try-with-resources** statement ensures this automatically:

```
try (FileInputStream fis = new FileInputStream("input.txt")) {
    // use the stream
} catch (IOException e) {
    e.printStackTrace();
}
```

3.2.8 Important Considerations

- **No automatic character encoding**: These streams work with raw bytes; use `OutputStreamWriter` or `InputStreamReader` if encoding matters.
- **Overwriting risk**: `FileOutputStream` will overwrite files by default—use with caution.
- **Not thread-safe**: Synchronize access if multiple threads read/write the same file.

3.2.9 Conclusion

`FileInputStream` and `FileOutputStream` are powerful classes in Java's IO toolkit, offering efficient and low-level control over reading and writing byte data to and from files. Whether you're building a file copier, manipulating binary files, or handling large datasets, these classes provide the foundation for reliable and performant IO operations. By combining them

with buffering and proper resource management, you can ensure that your file-handling code is both fast and safe.

3.3 Reading and Writing Files with FileReader/FileWriter

Java provides two broad categories of stream classes for file input and output:

- **Byte streams**: Use `InputStream` and `OutputStream` classes, designed for raw binary data.
- **Character streams**: Use `Reader` and `Writer` classes, designed for text (i.e., character data).

`FileReader` and `FileWriter` are concrete implementations of the character stream classes. They are specifically used to **read from and write to text files**, automatically handling character encoding based on the system's default or specified charset.

3.3.1 Why Use Character Streams?

When dealing with **text files**, character streams are preferred over byte streams for several reasons:

- They automatically **convert bytes to characters** using a character encoding (like UTF-8 or UTF-16).
- They allow you to process characters, lines, or strings instead of raw byte arrays.
- They reduce the need for manual conversion between bytes and characters.

For binary data such as images or audio files, **byte streams** are better suited. But for textual data—such as `.txt`, `.csv`, `.xml`, `.json`—`FileReader` and `FileWriter` are more convenient and semantically appropriate.

3.3.2 FileReader Reading Characters from a File

`FileReader` is a subclass of `InputStreamReader`, designed to read character data from a file using the **default platform encoding** (e.g., UTF-8 or ISO-8859-1, depending on the system).

Basic Usage

Full runnable code:

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

public class FileReaderExample {
    public static void main(String[] args) {
        try (FileReader reader = new FileReader("example.txt")) {
            int character;
            while ((character = reader.read()) != -1) {
                System.out.print((char) character); // Cast int to char
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- The `read()` method reads one character at a time.
- The loop continues until `read()` returns -1 (end of file).
- Each character is printed to the console.

3.3.3 FileWriter Writing Characters to a File

`FileWriter` is a subclass of `OutputStreamWriter` and allows you to write character data directly to a file.

Basic Usage

Full runnable code:

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

public class FileWriterExample {
    public static void main(String[] args) {
        String content = "This text is written using FileWriter in Java.";

        try (FileWriter writer = new FileWriter("output.txt")) {
            writer.write(content);  // Writes the entire string
            System.out.println("Data written to file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- The file `output.txt` is created (or overwritten) with the given string content.
- `FileWriter` handles the conversion from characters to bytes based on the system

encoding.

3.3.4 Writing in Append Mode

To **append** text to an existing file without overwriting:

```
try (FileWriter writer = new FileWriter("log.txt", true)) {
    writer.write("Appended text line.\n");
}
```

The second argument `true` tells `FileWriter` to append rather than overwrite.

3.3.5 Buffered Character Streams

Reading and writing character-by-character is inefficient. For better performance, wrap `FileReader` or `FileWriter` with buffered streams:

BufferedReader Example

Full runnable code:

```
import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("notes.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);  // Reads line by line
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

BufferedWriter Example

Full runnable code:

```
import java.io.*;

public class BufferedWriterExample {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("notes.txt"))) {
            writer.write("First line of text.");
            writer.newLine(); // Platform-specific line separator
            writer.write("Second line.");
```

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

Using buffered streams is highly recommended for performance and convenience (e.g., `readLine()`, `newLine()`).

3.3.6 Handling Character Encoding

By default, `FileReader` and `FileWriter` use the **platform's default encoding**, which can vary. To explicitly control encoding (such as UTF-8), use `InputStreamReader` and `OutputStreamWriter`:

Specifying Encoding

Full runnable code:

```
import java.io.*;
import java.nio.charset.StandardCharsets;

public class UTF8WriterExample {
    public static void main(String[] args) {
        try (Writer writer = new OutputStreamWriter(new FileOutputStream("utf8.txt"), StandardCharsets.U
            writer.write("Writing with UTF-8 encoding.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

This ensures consistent behavior across platforms, especially when dealing with non-ASCII characters.

3.3.7 Typical Use Cases for Character Streams

- Reading and writing **plain text files**
- Working with **CSV, XML, JSON**, or configuration files
- **Logging** to text-based log files
- Processing user-generated text content

3.3.8 Recap

`FileReader` and `FileWriter` provide a simple, high-level way to handle **text-based file input and output** in Java. Unlike byte streams, they automatically handle character encoding and make it easier to work with strings, characters, and lines. For more efficient and robust applications, it's often best to combine them with buffered streams and to explicitly set the character encoding when needed. These classes form the foundation of many file-based applications in Java that deal with human-readable data.

3.4 File Permissions and Attributes

Managing file permissions and attributes is an essential part of file handling in Java. Whether you are developing applications that need to check if files are accessible, or modify permissions for security reasons, Java provides several ways to work with these properties — from the basic `File` class to the more powerful NIO API.

3.4.1 Checking File Permissions Using the `File` Class

The `java.io.File` class offers simple methods to **check basic permissions** of files and directories:

- **`canRead()`**: Checks if the file is readable by the application.
- **`canWrite()`**: Checks if the file is writable.
- **`canExecute()`**: Checks if the file is executable (applicable mostly for scripts or executables).

Example: Checking Permissions

Full runnable code:

```
import java.io.File;

public class FilePermissionCheck {
    public static void main(String[] args) {
        File file = new File("example.txt");

        System.out.println("Readable: " + file.canRead());
        System.out.println("Writable: " + file.canWrite());
        System.out.println("Executable: " + file.canExecute());
    }
}
```

3.4.2 Modifying File Permissions Using the `File` Class

The `File` class also allows changing permissions using:

- **`setReadable(boolean readable)`**
- **`setWritable(boolean writable)`**
- **`setExecutable(boolean executable)`**

These methods attempt to change the file's permission for the **owner only** by default. You can optionally specify a second boolean parameter `ownerOnly` to restrict changes to the owner.

Example: Changing Permissions

Full runnable code:

```
import java.io.File;

public class FilePermissionModify {
    public static void main(String[] args) {
        File file = new File("example.txt");

        boolean readableSet = file.setReadable(true);
        boolean writableSet = file.setWritable(false);
        boolean executableSet = file.setExecutable(false);

        System.out.println("Readable set: " + readableSet);
        System.out.println("Writable set: " + writableSet);
        System.out.println("Executable set: " + executableSet);
    }
}
```

Note: These methods return `true` if the permission was successfully changed, or `false` otherwise. On some platforms or filesystems, permission changes may fail silently due to security policies or user rights.

3.4.3 Advanced Attributes and Permissions with NIO

Starting from **Java 7**, the `java.nio.file` package introduced a more comprehensive way to manage file permissions and attributes, including:

- Support for **POSIX file permissions** on Unix/Linux systems
- Access to file metadata such as creation time, last modified time, owner, etc.
- Atomic and secure operations via the `Files` utility class

3.4.4 Using `java.nio.file.attribute.PosixFilePermissions`

On POSIX-compliant systems (like Linux and macOS), you can get and set file permissions using the `PosixFilePermissions` class.

Checking and Setting POSIX Permissions Example:

Full runnable code:

```
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.IOException;
import java.util.Set;

public class PosixPermissionsExample {
    public static void main(String[] args) {
        Path path = Paths.get("example.txt");

        try {
            // Read current permissions
            Set<PosixFilePermission> permissions = Files.getPosixFilePermissions(path);
            System.out.println("Current permissions: " + PosixFilePermissions.toString(permissions));

            // Set new permissions: rw-r--r--
            Set<PosixFilePermission> newPermissions = PosixFilePermissions.fromString("rw-r--r--");
            Files.setPosixFilePermissions(path, newPermissions);

            System.out.println("Permissions updated.");

        } catch (UnsupportedOperationException e) {
            System.out.println("POSIX permissions not supported on this platform.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Notes:

- This will throw `UnsupportedOperationException` on Windows or non-POSIX filesystems.
- `PosixFilePermission` enums represent read, write, execute permissions for owner, group, and others.

3.4.5 Reading and Modifying Basic File Attributes

NIO also provides the `BasicFileAttributes` interface for reading attributes like:

- Creation time
- Last access time
- Last modified time

- Whether it's a regular file or directory

Example to read these attributes:

Full runnable code:

```
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.IOException;

public class BasicAttributesExample {
    public static void main(String[] args) {
        Path path = Paths.get("example.txt");

        try {
            BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttributes.class);

            System.out.println("Creation Time: " + attrs.creationTime());
            System.out.println("Last Access Time: " + attrs.lastAccessTime());
            System.out.println("Last Modified Time: " + attrs.lastModifiedTime());
            System.out.println("Is Directory: " + attrs.isDirectory());
            System.out.println("Is Regular File: " + attrs.isRegularFile());

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

3.4.6 Modifying File Timestamps

You can also update timestamps using `Files.setAttribute()`:

Full runnable code:

```
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

public class ModifyTimestampExample {
    public static void main(String[] args) {
        Path path = Paths.get("example.txt");

        try {
            FileTime now = FileTime.from(System.currentTimeMillis(), TimeUnit.MILLISECONDS);
            Files.setAttribute(path, "lastModifiedTime", now);
            System.out.println("Last modified time updated.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3.4.7 Managing File Owners

You can retrieve and modify file owners using `FileOwnerAttributeView`:

Full runnable code:

```
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.IOException;

public class FileOwnerExample {
    public static void main(String[] args) {
        Path path = Paths.get("example.txt");

        try {
            FileOwnerAttributeView ownerView = Files.getFileAttributeView(path, FileOwnerAttributeView.
            System.out.println("Current owner: " + ownerView.getOwner());

            // Example: Set owner (requires appropriate permissions)
            // UserPrincipal newOwner = path.getFileSystem().getUserPrincipalLookupService().lookupPrin
            // ownerView.setOwner(newOwner);

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

3.4.8 Recap

- The **`File` class** provides basic methods to check and modify file readability, writability, and executability.
- For **more advanced and cross-platform file attribute management**, the **NIO.2 API (`java.nio.file` and `java.nio.file.attribute`)** offers richer functionality:
  - Reading and setting **POSIX permissions** on compatible systems.
  - Accessing **file timestamps**, **owners**, and other metadata.
- Always handle exceptions, especially when working with permissions that might require elevated privileges or when running on unsupported platforms.

3.5 Listing Files and Directories

Working with files and directories often requires **listing their contents**—for example, to display files in a folder, filter files by type, or perform operations recursively on subdirectories.

Java's `java.io.File` class provides several convenient methods to list directory contents and apply filters.

3.5.1 Using the `File` Class to List Directory Contents

The `File` class provides two primary methods to list the contents of a directory:

- **`list()`**: Returns an array of `String` names (file and directory names).
- **`listFiles()`**: Returns an array of `File` objects representing files and directories.

Both methods return `null` if the `File` object is not a directory or an I/O error occurs.

3.5.2 Listing All Files and Directories

Here's a simple example that lists all names in a directory using `list()`:

Full runnable code:

```
import java.io.File;

public class ListDirectoryNames {
    public static void main(String[] args) {
        File dir = new File("myDirectory");

        // Check if directory exists and is a directory
        if (dir.exists() && dir.isDirectory()) {
            String[] contents = dir.list();

            if (contents != null) {
                for (String name : contents) {
                    System.out.println(name);
                }
            } else {
                System.out.println("Could not list directory contents.");
            }
        } else {
            System.out.println("Directory does not exist or is not a directory.");
        }
    }
}
```

Explanation:

- `list()` returns an array of file and directory names as strings.
- This is useful when you only need names, not full `File` objects.

3.5.3 Using `listFiles()` to Get `File` Objects

`listFiles()` returns an array of `File` objects, allowing you to inspect each entry's properties such as whether it is a file or directory, size, or last modified time.

Full runnable code:

```
import java.io.File;

public class ListDirectoryFiles {
    public static void main(String[] args) {
        File dir = new File("myDirectory");

        if (dir.exists() && dir.isDirectory()) {
            File[] files = dir.listFiles();

            if (files != null) {
                for (File file : files) {
                    if (file.isDirectory()) {
                        System.out.println("[DIR]  " + file.getName());
                    } else {
                        System.out.println("[FILE] " + file.getName());
                    }
                }
            } else {
                System.out.println("Could not list files.");
            }
        } else {
            System.out.println("Directory not found or invalid.");
        }
    }
}
```

This approach is more flexible because you can differentiate between files and directories and get more info.

3.5.4 Filtering Files with FilenameFilter and FileFilter

You can filter the listed files using two interfaces:

- **`FilenameFilter`**: Filters by file name strings.
- **`FileFilter`**: Filters by `File` objects.

Example: Filter Files by Extension Using `FilenameFilter`

Full runnable code:

```
import java.io.File;
import java.io.FilenameFilter;

public class ListTxtFiles {
    public static void main(String[] args) {
```

```
        File dir = new File("myDirectory");

        FilenameFilter filter = new FilenameFilter() {
            @Override
            public boolean accept(File dir, String name) {
                return name.endsWith(".txt");  // Only accept .txt files
            }
        };

        String[] txtFiles = dir.list(filter);

        if (txtFiles != null) {
            for (String fileName : txtFiles) {
                System.out.println(fileName);
            }
        }
    }
}
```

This example lists only files ending with `.txt`.

Using a Lambda Expression (Java 8)

```
String[] txtFiles = dir.list((d, name) -> name.endsWith(".txt"));
```

3.5.5 Filtering with `FileFilter`

Using `FileFilter` gives access to the full `File` object:

Full runnable code:

```
import java.io.File;
import java.io.FileFilter;

public class ListLargeFiles {
    public static void main(String[] args) {
        File dir = new File("myDirectory");

        FileFilter filter = file -> file.isFile() && file.length() > 1_000_000; // Files larger than 1M

        File[] largeFiles = dir.listFiles(filter);

        if (largeFiles != null) {
            for (File file : largeFiles) {
                System.out.println(file.getName() + " - " + file.length() + " bytes");
            }
        }
    }
}
```

3.5.6 Recursively Listing Files in Subdirectories

To list all files inside a directory and its subdirectories, you need to use **recursion**:

Full runnable code:

```
import java.io.File;

public class RecursiveFileLister {
    public static void listFilesRecursive(File dir) {
        if (dir == null || !dir.exists()) return;

        File[] files = dir.listFiles();
        if (files == null) return;

        for (File file : files) {
            if (file.isDirectory()) {
                System.out.println("[DIR]  " + file.getAbsolutePath());
                listFilesRecursive(file);  // Recursive call for subdirectory
            } else {
                System.out.println("[FILE] " + file.getAbsolutePath());
            }
        }
    }

    public static void main(String[] args) {
        File rootDir = new File("myDirectory");
        listFilesRecursive(rootDir);
    }
}
```

Explanation:

- The method checks if the current `File` is a directory.
- If so, it prints the directory path and recursively lists its contents.
- If it's a file, it prints the file path.
- This continues until all nested files and folders are processed.

3.5.7 Using `java.nio.file` for Advanced Listing (Brief Intro)

While the `File` class is simple, the newer **NIO** API (`java.nio.file`) provides more powerful directory traversal using `Files.walk()` or `Files.list()`. This can be used to filter files more flexibly and perform parallel processing.

Example with `Files.walk()` (Java 8+):

Full runnable code:

```
import java.nio.file.*;
import java.io.IOException;
```

```java
public class NIOFileWalkExample {
    public static void main(String[] args) {
        Path start = Paths.get("myDirectory");

        try {
            Files.walk(start)
                 .filter(Files::isRegularFile)
                 .filter(path -> path.toString().endsWith(".txt"))
                 .forEach(System.out::println);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3.5.8 Recap

- Use `File.list()` to get names of all entries in a directory.
- Use `File.listFiles()` for detailed `File` objects allowing inspection of files vs directories.
- Use `FilenameFilter` or `FileFilter` to filter files by name or attributes.
- Use recursion to list files in nested subdirectories.
- For more advanced features, consider the NIO API (`java.nio.file`).

3.6 File Deletion, Renaming, and Creation

Managing files often involves **creating new files**, **renaming existing ones**, or **deleting files** that are no longer needed. The Java `java.io.File` class provides straightforward methods to perform these file operations. However, because file operations depend on the underlying filesystem and operating system permissions, it's important to handle errors carefully and understand the potential pitfalls.

3.6.1 Creating Files with `createNewFile()`

The method `createNewFile()` creates a new, empty file at the path specified by the `File` object.

- If the file **does not exist**, the method attempts to create it and returns `true`.
- If the file **already exists**, it returns `false`.
- It throws an `IOException` if an I/O error occurs (e.g., invalid path, no permissions).

Example: Creating a New File

Full runnable code:

```
import java.io.File;
import java.io.IOException;

public class FileCreationExample {
    public static void main(String[] args) {
        File file = new File("newFile.txt");

        try {
            if (file.createNewFile()) {
                System.out.println("File created successfully: " + file.getName());
            } else {
                System.out.println("File already exists: " + file.getName());
            }
        } catch (IOException e) {
            System.out.println("An error occurred while creating the file.");
            e.printStackTrace();
        }
    }
}
```

Best Practices and Pitfalls:

- **Parent directories must exist** before creating a file; otherwise, `createNewFile()` will fail. Use `file.getParentFile().mkdirs()` to create parent directories if necessary.
- Always check the return value to know if the file was created or already existed.
- Handle `IOException` to catch permission errors or invalid paths.

3.6.2 Deleting Files with `delete()`

The `delete()` method deletes the file or directory denoted by the `File` object.

- Returns `true` if deletion succeeds.
- Returns `false` if the file does not exist, deletion is denied, or the file is a non-empty directory.

Example: Deleting a File

Full runnable code:

```
import java.io.File;

public class FileDeletionExample {
    public static void main(String[] args) {
        File file = new File("oldFile.txt");

        if (file.delete()) {
            System.out.println("File deleted successfully.");
```

```
        } else {
            System.out.println("Failed to delete the file. It may not exist or is in use.");
        }
    }
}
```

Important Notes and Pitfalls:

- On some operating systems (e.g., Windows), a file cannot be deleted if it is open or locked by another process.
- The `delete()` method **does not throw exceptions**; it returns a boolean, so always check its return value.
- To delete directories, the directory must be **empty**; otherwise, deletion will fail. Recursive deletion must be implemented manually.
- There is no undo — once deleted, the file is gone unless the OS or filesystem supports recovery.

3.6.3 Renaming and Moving Files with `renameTo()`

The method `renameTo(File dest)` renames the file to the new pathname specified by the `dest File` object. This method can also be used to move a file between directories.

- Returns `true` if the rename/move succeeded.
- Returns `false` if it failed.

Example: Renaming a File

Full runnable code:

```
import java.io.File;

public class FileRenameExample {
    public static void main(String[] args) {
        File oldFile = new File("oldName.txt");
        File newFile = new File("newName.txt");

        if (oldFile.renameTo(newFile)) {
            System.out.println("File renamed successfully.");
        } else {
            System.out.println("Failed to rename file.");
        }
    }
}
```

Example: Moving a File

```
File sourceFile = new File("file.txt");
File destinationFile = new File("subfolder/file.txt");
```

```java
if (sourceFile.renameTo(destinationFile)) {
    System.out.println("File moved successfully.");
} else {
    System.out.println("Failed to move the file.");
}
```

3.6.4 Potential Issues with `renameTo()`

- **Cross-filesystem moves might fail**: `renameTo()` works reliably only within the same filesystem or partition.
- **No exception thrown**: You must check the return value to detect failure.
- On some platforms, renaming might fail if the destination file exists.
- It does **not automatically overwrite** existing files at the destination.

3.6.5 Best Practices for File Creation, Deletion, and Renaming

- **Always check method return values (`boolean`)** to verify success.
- Use **try-catch blocks** for methods that throw exceptions, such as `createNewFile()`.
- Before creating a file, **ensure parent directories exist**:

```java
File file = new File("path/to/newFile.txt");
file.getParentFile().mkdirs(); // creates all nonexistent parent directories
file.createNewFile();
```

- For more robust file moves and deletes, consider using the **NIO API (`java.nio.file.Files`)** introduced in Java 7, which offers methods like `Files.move()` and `Files.delete()` that throw exceptions on failure and provide more control.
- Remember that **file operations depend on OS permissions**. Your Java program must have the necessary rights to manipulate files.
- Avoid naming conflicts by **checking if a file exists** before creating or renaming.
- When deleting directories recursively, you must manually delete all files/subdirectories first before deleting the directory.

3.6.6 Recursively Deleting a Directory Example

Because `File.delete()` won't delete non-empty directories, recursive deletion is needed:

```java
public static boolean deleteDirectory(File directory) {
    if (directory.isDirectory()) {
```

```
        File[] entries = directory.listFiles();
        if (entries != null) {
            for (File entry : entries) {
                if (!deleteDirectory(entry)) {
                    return false;  // Failed to delete a file/subdir
                }
            }
        }
    }
    return directory.delete();
}
```

Use with care, as recursive deletion is **destructive and irreversible**.

3.6.7 Recap

- **`createNewFile()`** creates new files but requires existing parent directories.
- **`delete()`** removes files/directories but returns `false` on failure, without throwing exceptions.
- **`renameTo()`** renames or moves files but can be unreliable across filesystems and doesn't overwrite by default.
- Always check return values and handle exceptions where applicable.
- For more powerful and reliable file operations, consider Java NIO's `Files` class.
- Manage parent directories and consider permissions when performing file operations.

Chapter 4.

Advanced Java IO Concepts

1. Object Serialization and Deserialization
2. Using ObjectInputStream and ObjectOutputStream
3. Externalizable Interface
4. Piped Streams for Inter-thread Communication
5. PushbackInputStream and Mark/Reset Methods

4 Advanced Java IO Concepts

4.1 Object Serialization and Deserialization

Java provides a powerful mechanism called **serialization** that allows you to convert an object into a sequence of bytes, which can then be saved to a file, transmitted over a network, or stored for later retrieval. The reverse process, **deserialization**, reconstructs the object from these bytes back into memory.

4.1.1 What is Serialization?

Serialization is the process of transforming the state of an object into a byte stream. This byte stream captures the object's data, and sometimes metadata, so it can be stored or transmitted. The key benefit is that the object's entire state can be saved and restored later, enabling:

- **Object persistence**: Saving objects to files or databases for later use.
- **Communication**: Sending objects over a network between different Java virtual machines (JVMs).
- **Caching**: Storing objects in memory or disk caches for fast access.

Without serialization, saving or transmitting complex objects would require manual conversion to a suitable format.

4.1.2 The `Serializable` Interface

In Java, an object must explicitly indicate that it can be serialized by implementing the **marker interface** `java.io.Serializable`. This interface has no methods; it simply marks the class as serializable.

```
import java.io.Serializable;

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

    private String name;
    private int age;

    // Constructor, getters, setters...
}
```

- The **`serialVersionUID`** is a unique identifier for the class version. It helps during deserialization to ensure that a serialized object corresponds to a compatible class version. If not provided, Java generates one automatically, but explicitly defining it is

best practice for version control.

4.1.3 How Serialization Works in Java

The standard serialization mechanism uses two classes from the `java.io` package:

- **`ObjectOutputStream`** — Writes serialized objects to an output stream (e.g., file or socket).
- **`ObjectInputStream`** — Reads serialized objects from an input stream and reconstructs them.

4.1.4 Basic Serialization Example

This example shows how to serialize a `Person` object to a file.

```
import java.io.*;

public class SerializeExample {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);

        try (FileOutputStream fileOut = new FileOutputStream("person.ser");
             ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

            out.writeObject(person);  // Serialize the person object
            System.out.println("Object serialized to person.ser");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- The object is converted into bytes and written to the file `person.ser`.
- Note the use of **try-with-resources** to automatically close streams.

4.1.5 Basic Deserialization Example

To restore the serialized object from the file:

```
import java.io.*;

public class DeserializeExample {
    public static void main(String[] args) {
        try (FileInputStream fileIn = new FileInputStream("person.ser");
             ObjectInputStream in = new ObjectInputStream(fileIn)) {
```

```
            Person person = (Person) in.readObject();  // Deserialize object
            System.out.println("Deserialized Person:");
            System.out.println("Name: " + person.getName());
            System.out.println("Age: " + person.getAge());

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

- The `readObject()` method reads the byte stream and reconstructs the `Person` object.
- You need to cast the returned object to the appropriate class.
- Handle both `IOException` and `ClassNotFoundException`.

4.1.6 Important Considerations

Transient Fields

If a field should **not** be serialized (e.g., sensitive information or fields that can be recalculated), declare it as `transient`:

```
private transient String password;
```

Such fields are ignored during serialization and restored with default values (`null` for objects, zero for primitives) on deserialization.

Object Graph Serialization

Java serialization handles entire **object graphs**. If a serialized object references other objects (fields holding other objects), those referenced objects must also implement `Serializable`. The whole graph is serialized recursively.

Customization via `writeObject` and `readObject`

Classes can customize the serialization process by defining these private methods:

```
private void writeObject(ObjectOutputStream out) throws IOException {
    // Custom serialization logic
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    // Custom deserialization logic
}
```

This allows, for example, encryption, compression, or special handling of certain fields.

Serial Version UID

Changing a class structure without updating `serialVersionUID` can lead to **`InvalidClassException`** during deserialization. Always update or maintain consistent version UIDs when evolving

classes.

4.1.7 Why Serialization is Useful

- **Persistence:** Saving program state easily without converting objects manually.
- **Network Communication:** Transmitting Java objects in distributed systems (e.g., RMI, messaging).
- **Caching:** Storing objects in serialized form to disk or memory for later reuse.
- **Deep Cloning:** Creating deep copies of objects via serialization.

4.1.8 Limitations and Alternatives

- Java serialization can be **slow and produce large output**.
- It is **Java-specific**, so serialized files may not be portable across languages.
- Security risks if deserializing untrusted data (can lead to exploits).
- Alternatives include **JSON, XML, Protocol Buffers**, or **custom serialization** formats for interoperability and control.

4.1.9 Recap

- Serialization converts Java objects into byte streams for storage or transmission.
- Deserialization reconstructs objects from these byte streams.
- Implement `Serializable` to mark classes for serialization.
- Use `ObjectOutputStream` and `ObjectInputStream` for serialization/deserialization.
- Manage versioning with `serialVersionUID`.
- Use `transient` fields to exclude sensitive or non-serializable data.
- Be aware of security and performance implications.

4.2 Using ObjectInputStream and ObjectOutputStream

Serialization in Java revolves around two core classes: `ObjectOutputStream` and `ObjectInputStream`. These classes provide the functionality to convert Java objects into a byte stream and vice versa, enabling easy persistence and communication of complex data structures.

4.2.1 What Are `ObjectOutputStream` and `ObjectInputStream`?

- **`ObjectOutputStream`**: Wraps around an underlying `OutputStream` and writes Java objects as a serialized stream of bytes.
- **`ObjectInputStream`**: Wraps around an `InputStream` and reads serialized bytes, reconstructing Java objects.

Together, they simplify the process of writing and reading serializable objects to/from files, network sockets, or any stream.

4.2.2 How Do These Classes Work?

- `ObjectOutputStream` serializes objects implementing the `Serializable` interface, including their entire object graph (all referenced objects).
- `ObjectInputStream` reads the byte stream and recreates the objects in memory.
- Both streams handle the metadata necessary to maintain object identity, versioning, and type information.

4.2.3 Basic Usage: Writing Objects to a File

The common pattern is to wrap a `FileOutputStream` with an `ObjectOutputStream`. This allows writing objects directly to a file.

Example: Serializing a `Person` Object

```
import java.io.*;

public class SerializeDemo {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);

        try (FileOutputStream fos = new FileOutputStream("person.ser");
             ObjectOutputStream oos = new ObjectOutputStream(fos)) {

            oos.writeObject(person);  // Serialize the person object
            System.out.println("Object serialized successfully.");

        } catch (IOException e) {
            System.err.println("Serialization failed:");
            e.printStackTrace();
        }
    }
}
```

- **`writeObject(Object obj)`** serializes the given object.
- The **try-with-resources** block ensures streams are closed automatically.

- If `person` references other serializable objects, they are also serialized recursively.

4.2.4 Reading Objects from a File

To deserialize, wrap a `FileInputStream` with an `ObjectInputStream` and call `readObject()`:

```
import java.io.*;

public class DeserializeDemo {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("person.ser");
             ObjectInputStream ois = new ObjectInputStream(fis)) {

            Person person = (Person) ois.readObject();  // Deserialize object
            System.out.println("Deserialized Person:");
            System.out.println("Name: " + person.getName());
            System.out.println("Age: " + person.getAge());

        } catch (IOException | ClassNotFoundException e) {
            System.err.println("Deserialization failed:");
            e.printStackTrace();
        }
    }
}
```

- **`readObject()`** reads and reconstructs the object.
- It returns `Object`, so a cast is required.
- Handle both `IOException` and `ClassNotFoundException`:
  - `IOException` for stream or file errors.
  - `ClassNotFoundException` if the class of the serialized object isn't found.

4.2.5 Handling Versioning with `serialVersionUID`

Java serialization includes a **version control mechanism** using the `serialVersionUID` field, which helps detect class mismatches between serialized objects and the current class version.

```
import java.io.Serializable;

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

    private String name;
    private int age;

    // Constructors, getters, setters...
}
```

- If the `serialVersionUID` of the serialized class and the class at deserialization differ, an `InvalidClassException` is thrown.
- Explicitly declaring `serialVersionUID` protects against accidental incompatibilities caused by compiler-generated values.
- When evolving classes (adding/removing fields), update the `serialVersionUID` accordingly.

4.2.6 Important Considerations When Using These Streams

Closing Streams

Always close streams after use to free system resources and avoid data corruption. Use try-with-resources or explicit `close()` calls.

Transient Fields

Fields marked `transient` are not serialized and will have default values after deserialization.

Custom Serialization

Classes can override `writeObject` and `readObject` private methods to customize serialization (e.g., encrypting data or handling transient fields).

```
private void writeObject(ObjectOutputStream out) throws IOException {
    // Custom serialization logic
    out.defaultWriteObject();
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    // Custom deserialization logic
    in.defaultReadObject();
}
```

Serializing Collections

Common Java collections like `ArrayList`, `HashMap`, etc., implement `Serializable` and can be serialized directly.

4.2.7 Full Example: Serialize and Deserialize Multiple Objects

```
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class SerializeMultipleObjects {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
```

```
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));

        // Serialize list of people
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("people.ser"))) {
            oos.writeObject(people);
            System.out.println("List serialized.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize list of people
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("people.ser"))) {
            List<Person> deserializedPeople = (List<Person>) ois.readObject();
            System.out.println("Deserialized people:");
            for (Person p : deserializedPeople) {
                System.out.println(p.getName() + ", Age: " + p.getAge());
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

4.2.8 Recap

- **`ObjectOutputStream`** and **`ObjectInputStream`** provide seamless Java object serialization and deserialization.
- Use **`writeObject()`** to serialize objects and **`readObject()`** to deserialize.
- Always handle exceptions (`IOException`, `ClassNotFoundException`) properly.
- Use **`serialVersionUID`** to maintain version compatibility.
- Customize serialization if needed using `writeObject` and `readObject`.
- Close streams to avoid resource leaks, preferably using try-with-resources.
- These classes enable saving, transmitting, and restoring complex Java objects efficiently.

4.3 Externalizable Interface

Java provides two main interfaces for object serialization: **`Serializable`** and **`Externalizable`**. While both enable object serialization, `Externalizable` offers greater control over the serialization process, allowing developers to customize exactly how an object's data is written and read. This section explains the `Externalizable` interface, how it differs from `Serializable`, and when and how to use it effectively.

4.3.1 What is `Externalizable`?

`Externalizable` is a subinterface of `Serializable` defined in the `java.io` package. It requires the implementing class to explicitly define **how its fields are serialized and deserialized** by overriding two methods:

- `void writeExternal(ObjectOutput out) throws IOException`
- `void readExternal(ObjectInput in) throws IOException, ClassNotFoundException`

In contrast, `Serializable` uses **default serialization**, which automatically serializes all non-transient fields.

4.3.2 Key Differences Between `Serializable` and `Externalizable`

Aspect	`Serializable`	`Externalizable`
Serialization Logic	Automatic by JVM, serializes all non-transient fields	Manual; developer writes explicit serialization code
Methods to Override	None mandatory (optional `writeObject/readObject`)	Must implement `writeExternal` and `readExternal`
Control Over Data Written	Limited	Complete control over what and how to serialize
Performance	Simpler, but can be slower due to extra metadata	Can be faster and more compact, if implemented carefully
Use Case	General-purpose serialization	When custom serialization is needed or optimized serialization required

4.3.3 Why Use `Externalizable`?

`Externalizable` is ideal when:

- You want **fine-grained control** over the serialized format.
- You need to **optimize performance** or **minimize serialized data size**.
- You want to **exclude certain fields selectively** without relying on `transient`.
- You want to **serialize only parts of the object** or include external resources.
- You need to maintain **compatibility across different versions** of a class by managing the serialized form explicitly.

4.3.4 Implementing the `Externalizable` Interface

When a class implements `Externalizable`, it **must** provide implementations for both `writeExternal` and `readExternal`. The serialization runtime will not automatically serialize any fields.

4.3.5 Example: Implementing `Externalizable`

```
import java.io.*;

public class Person implements Externalizable {
    private String name;
    private int age;
    private transient String password; // will not be serialized

    // Mandatory no-arg constructor for deserialization
    public Person() {}

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

    // Serialize object data manually
    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(name);   // Write name as UTF string
        out.writeInt(age);    // Write age as int
        // Intentionally exclude password for security
    }

    // Deserialize object data manually
    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        name = in.readUTF();
        age = in.readInt();
        // password remains null after deserialization
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + ", password='" + password + "'}";
    }
}
```

4.3.6 Testing Serialization and Deserialization

```
public class ExternalizableTest {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30, "secretPass");

        // Serialize to file
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("personExt.ser"))) {
            oos.writeObject(person);
            System.out.println("Person serialized.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize from file
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("personExt.ser"))) {
            Person deserializedPerson = (Person) ois.readObject();
            System.out.println("Deserialized: " + deserializedPerson);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Person serialized.
Deserialized: Person{name='Alice', age=30, password='null'}
```

The password is not serialized because it was deliberately excluded in `writeExternal`.

4.3.7 Important Notes

- **No-arg Constructor Requirement:** The class must have a **public no-argument constructor**. This constructor is called during deserialization before `readExternal`.
- **Complete Responsibility:** The developer is fully responsible for writing and reading every field, including handling data types and order. Failing to do so causes corrupt or inconsistent data.
- **Versioning:** Since you control serialization, you can design a custom versioning scheme (e.g., write a version number first) to maintain backward compatibility.
- **Security:** You can exclude sensitive data, or even encrypt data during serialization.

4.3.8 When to Prefer `Externalizable`

- When you want **full control** over serialized data format for optimization or compliance with a protocol.

- When you want to **exclude transient data explicitly** or handle it in a custom way.
- When implementing **custom serialization schemes**, such as compressing or encrypting data.
- When working in environments where you need to **interoperate with non-Java systems** by defining your own serialization format.

For typical use cases where default serialization suffices, `Serializable` is easier and less error-prone.

4.3.9 Recap

- `Externalizable` extends `Serializable` but requires explicit implementation of `writeExternal` and `readExternal`.
- It offers **total control** over what and how an object is serialized.
- You must provide a **no-argument constructor**.
- Used for **performance tuning**, **security**, and **custom serialization logic**.
- It is more complex but powerful when the default `Serializable` mechanism is insufficient.

4.4 Piped Streams for Inter-thread Communication

In Java IO, **piped streams** provide a simple yet powerful mechanism for threads to communicate by sending data through a stream, similar to how one thread writes data that another thread reads. The classes `PipedInputStream` and `PipedOutputStream` are designed to work together to create a **pipe** — a one-way communication channel — between threads.

4.4.1 What Are Piped Streams?

- `PipedOutputStream` acts as the **producer** end, where data is written.
- `PipedInputStream` acts as the **consumer** end, where data is read.
- These streams are connected such that bytes written to the `PipedOutputStream` are available to read from the connected `PipedInputStream`.

This mechanism is analogous to a physical pipe: data flows in one end and emerges from the other.

4.4.2 Why Use Piped Streams?

- **Thread communication:** Piped streams are primarily used to connect two threads where one thread produces data and the other consumes it.
- **Data streaming:** Unlike shared variables or queues, piped streams provide stream-based communication, making it suitable for IO-oriented workflows.
- **Decoupling:** Producer and consumer threads can operate independently but remain synchronized through the pipe.

4.4.3 How Do Piped Streams Work?

To establish a pipe:

1. Create a `PipedOutputStream`.
2. Create a `PipedInputStream`.
3. Connect them using the constructor or the `connect()` method.

After connection, writing bytes to the `PipedOutputStream` makes those bytes available for reading from the `PipedInputStream`.

4.4.4 Thread Safety and Synchronization

- The piped streams **internally synchronize** data transfer between threads.
- However, the producer and consumer threads must handle their respective stream's lifecycle properly.
- If the consumer reads faster than the producer writes, it will block waiting for data.
- If the producer writes faster than the buffer capacity, it will block until space becomes available.

4.4.5 Example: Producer-Consumer Using Piped Streams

The example demonstrates two threads:

- A **Producer** thread writes messages to a `PipedOutputStream`.
- A **Consumer** thread reads messages from the connected `PipedInputStream`.

Full runnable code:

```
import java.io.*;

public class PipedStreamExample {
```

```
    public static void main(String[] args) {
        try {
            // Create piped input and output streams and connect them
            PipedOutputStream pos = new PipedOutputStream();
            PipedInputStream pis = new PipedInputStream(pos);

            // Producer thread writes to PipedOutputStream
            Thread producer = new Thread(() -> {
                try (PrintWriter writer = new PrintWriter(pos)) {
                    String[] messages = {"Hello", "from", "the", "Producer", "thread!"};
                    for (String msg : messages) {
                        writer.println(msg);
                        writer.flush(); // Ensure data is sent immediately
                        System.out.println("Producer sent: " + msg);
                        Thread.sleep(500); // Simulate delay
                    }
                } catch (InterruptedException e) {
                    System.err.println("Producer error: " + e.getMessage());
                }
            });

            // Consumer thread reads from PipedInputStream
            Thread consumer = new Thread(() -> {
                try (BufferedReader reader = new BufferedReader(new InputStreamReader(pis))) {
                    String line;
                    while ((line = reader.readLine()) != null) {
                        System.out.println("Consumer received: " + line);
                    }
                } catch (IOException e) {
                    System.err.println("Consumer error: " + e.getMessage());
                }
            });

            // Start both threads
            consumer.start();
            producer.start();

            // Wait for threads to finish
            producer.join();
            // Closing the output stream signals end of data, consumer will exit read loop
            pos.close();
            consumer.join();

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

4.4.6 Explanation of the Example

- **Connection:** The `PipedInputStream` is constructed with the `PipedOutputStream` as an argument, connecting the two.

- **Producer thread:** Uses a `PrintWriter` wrapped around the `PipedOutputStream` to write strings line-by-line.
- **Consumer thread:** Uses a `BufferedReader` wrapped around an `InputStreamReader` on the `PipedInputStream` to read lines.
- Both threads run concurrently; the consumer blocks waiting for input when none is available.
- The producer flushes the writer after each message to ensure data is sent promptly.
- When the producer finishes, it closes the `PipedOutputStream`. This causes the consumer's read loop to terminate since `readLine()` returns `null` on end-of-stream.
- Proper exception handling captures IO errors or interruptions.
- The main thread waits for both threads to complete using `join()`.

4.4.7 Benefits and Limitations

Benefits:

- Provides a direct, stream-based communication channel between threads.
- Useful for decoupling producer-consumer workflows.
- Easy to use with existing stream-based APIs.

Limitations:

- Only supports one-way communication per pair of piped streams.
- Buffer size is fixed internally (default 1024 bytes); can block if buffer fills or empties.
- Not suitable for high-performance or complex thread coordination; consider using higher-level concurrency utilities (e.g., `BlockingQueue`) for complex scenarios.
- Can be prone to deadlocks if not managed carefully (e.g., both threads waiting on each other).

4.4.8 Best Practices

- Always **connect** `PipedInputStream` and `PipedOutputStream` before use.
- Use **try-with-resources** or explicitly close streams to avoid resource leaks.
- Properly **handle exceptions** in both producer and consumer threads.
- Avoid long blocking operations while holding the pipe to prevent deadlocks.
- Consider buffer size adjustments by using the constructor `PipedInputStream(int pipeSize)` if necessary.
- For **bidirectional communication**, use two pairs of piped streams or higher-level constructs.

4.4.9 Recap

- `PipedInputStream` and `PipedOutputStream` allow one thread to send data directly to another using a stream.
- They enable simple inter-thread communication using standard IO stream paradigms.
- The producer writes bytes; the consumer reads those bytes, synchronized by the underlying pipe buffer.
- Proper connection, synchronization, and resource management are essential to avoid deadlocks and errors.
- Piped streams are ideal for lightweight producer-consumer scenarios but less suited for complex concurrency.

4.5 PushbackInputStream and Mark/Reset Methods

Java IO streams provide versatile tools for reading data sequentially. However, in some scenarios—such as parsing or processing complex data formats—you may need the ability to **"unread"** bytes or **go back** to a previously read position in the stream. This is where **`PushbackInputStream`** and the **`mark()` / `reset()`** methods come into play.

4.5.1 What is `PushbackInputStream`?

`PushbackInputStream` is a subclass of `FilterInputStream` that allows you to **push bytes back** into the input stream, effectively "unreading" them. This is especially useful in parsing scenarios where you read ahead some bytes to decide how to process them but realize that some of those bytes belong to the next data unit.

4.5.2 How Does `PushbackInputStream` Work?

- When you read bytes normally, they are consumed and lost from the stream.
- With a `PushbackInputStream`, you can call the `unread()` method to push one or more bytes back into an internal buffer.
- The next read operations will first consume the bytes in the pushback buffer before reading new bytes from the underlying stream.

4.5.3 Common Use Case: Lookahead in Parsing

Imagine reading a stream where the next few bytes determine how to interpret the data, but you do not want to lose these bytes permanently if you decide to process them differently. `PushbackInputStream` lets you peek and then push bytes back so they can be reread.

4.5.4 Example: Using `PushbackInputStream`

Full runnable code:

```
import java.io.*;

public class PushbackExample {
    public static void main(String[] args) throws IOException {
        byte[] data = { 'a', 'b', 'c', 'd' };
        ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(data);
        PushbackInputStream pushbackInputStream = new PushbackInputStream(byteArrayInputStream);

        int firstByte = pushbackInputStream.read();
        System.out.println("Read byte: " + (char) firstByte); // Output: a

        int secondByte = pushbackInputStream.read();
        System.out.println("Read byte: " + (char) secondByte); // Output: b

        // Decide to "unread" the second byte
        pushbackInputStream.unread(secondByte);
        System.out.println("Pushed back byte: " + (char) secondByte);

        // Read again, should get the same byte
        int rereadByte = pushbackInputStream.read();
        System.out.println("Reread byte: " + (char) rereadByte); // Output: b

        pushbackInputStream.close();
    }
}
```

Output:

```
Read byte: a
Read byte: b
Pushed back byte: b
Reread byte: b
```

4.5.5 Important Notes About `PushbackInputStream`

- The constructor can take a **pushback buffer size**, allowing multiple bytes to be pushed back. The default size is 1 byte.

- Attempting to unread more bytes than the buffer size causes an `IOException`.
- It works only for **byte streams** (`InputStream`), not character streams (`Reader`).

4.5.6 The `mark()` and `reset()` Methods

While `PushbackInputStream` lets you "unread" bytes, many streams support **marking a position** and later **resetting** the stream back to that position, allowing multiple bytes to be reread without pushing them back individually.

4.5.7 How `mark()` and `reset()` Work

- Calling **`mark(int readlimit)`** tells the stream to remember the current position and keep a buffer of up to `readlimit` bytes for possible reset.
- Later, calling **`reset()`** resets the stream back to that marked position.
- This is useful for lookahead or tentative reading: you can read ahead, and if needed, rewind to the mark to reread or discard.

4.5.8 Which Streams Support mark/reset?

- Not all input streams support `mark` and `reset`.
- To check, call `markSupported()`; if it returns `true`, you can safely use `mark()` and `reset()`.
- Common supporting streams include `BufferedInputStream`, `ByteArrayInputStream`, and many readers like `BufferedReader`.

4.5.9 Example: Using mark() and reset()

Full runnable code:

```
import java.io.*;

public class MarkResetExample {
    public static void main(String[] args) throws IOException {
        byte[] data = { 'x', 'y', 'z' };
        ByteArrayInputStream bais = new ByteArrayInputStream(data);
        BufferedInputStream bis = new BufferedInputStream(bais);

        System.out.println("Read: " + (char) bis.read()); // x
```

```
        if (bis.markSupported()) {
            bis.mark(10); // mark current position with a buffer limit
            System.out.println("Read after mark: " + (char) bis.read()); // y
            System.out.println("Read after mark: " + (char) bis.read()); // z

            bis.reset(); // reset to marked position
            System.out.println("After reset: " + (char) bis.read()); // y (again)
        }

        bis.close();
    }
}
```

Output:

```
Read: x
Read after mark: y
Read after mark: z
After reset: y
```

4.5.10 Use Cases for mark/reset

- **Parsing protocols or file formats** where you need to peek ahead without losing data.
- **Conditional processing**, where you read a segment to decide on a strategy, then rewind.
- Implementing **tokenizers** or **lexers** that require lookahead.
- Avoids manual pushback when multiple bytes need to be reread.

4.5.11 Comparing Pushback and mark/reset

Feature	`PushbackInputStream`	`mark()` / `reset()`
Usage	Manually unread bytes	Mark and rewind stream position
Buffer Size	Fixed buffer size specified in constructor (default 1)	Managed internally by the stream, up to readlimit
Flexibility	Precise byte-level unread	Allows rewinding to a mark position
Stream Support	Only byte streams (`InputStream`)	Depends on stream; many support it
Typical Use	Single or few bytes lookahead/unread	Larger lookahead with automatic rewind

4.5.12 Recap

- **`PushbackInputStream`** allows unread bytes to be pushed back into the stream for subsequent re-reading, useful for simple lookahead or correcting read decisions.
- **`mark()` and `reset()`** let you mark a stream position and rewind to it later, enabling flexible multi-byte lookahead and reprocessing.
- Both are essential tools in building parsers, tokenizers, and protocols requiring conditional reading.
- Always check `markSupported()` before using `mark/reset`.
- Use `PushbackInputStream` when you want explicit unread control with a small buffer; use `mark/reset` for more extensive or automatic rewind functionality.

Chapter 5.

Introduction to Java NIO (New IO)

1. Motivation Behind NIO
2. Core Concepts: Buffers, Channels, and Selectors
3. Differences Between IO and NIO
4. Non-blocking IO and Selectors Overview

5 Introduction to Java NIO (New IO)

5.1 Motivation Behind NIO

Java's original IO (Input/Output) API, introduced in the early versions of Java, provides a straightforward, stream-based approach to reading and writing data. While it served the needs of many applications, especially desktop and simple server programs, it soon showed limitations in scalability, performance, and flexibility — particularly for modern networked and high-throughput applications. This led to the development and introduction of **Java NIO (New IO)** in Java 1.4, a major overhaul designed to address these challenges.

5.1.1 Limitations of the Original Java IO API

The classic Java IO API is built on the concept of **blocking, stream-oriented IO**:

- **Blocking IO**: When a thread reads from or writes to a stream, it blocks — that is, it waits until the operation completes before continuing execution. For example, reading from a socket or file input stream suspends the thread until data is available or the end of the stream is reached.
- **Stream-Oriented**: Data is read or written sequentially as a flow of bytes or characters.

While this model is simple and intuitive, it has several drawbacks in high-performance or large-scale applications:

Blocking Threads Wastes Resources

Because IO operations block the calling thread, each connection or file operation typically requires its own thread. In server applications handling thousands of simultaneous connections, this can lead to:

- **Thread proliferation**: Creating and managing a large number of threads consumes system memory and CPU time.
- **Context switching overhead**: The operating system must frequently switch between many threads, degrading performance.
- **Difficult concurrency management**: Writing thread-safe code with many threads is complex and error-prone.

Poor Scalability

Blocking IO works well for applications with a small number of IO channels, but as the number of concurrent connections grows, performance and scalability suffer:

- Servers using one thread per client eventually reach resource limits.
- High latency and lower throughput result from thread contention and blocking waits.

Lack of Fine-Grained Control

The old IO API offers limited control over buffering, multiplexing, or non-blocking operations. Developers often have to rely on platform-specific or third-party tools to handle efficient, scalable IO.

5.1.2 Why Was Java NIO Introduced?

Java NIO was introduced in Java 1.4 to provide a **new foundation for scalable, high-performance IO operations** in Java applications. Its design goals include:

- **Non-blocking IO**: Allow threads to initiate IO operations without waiting for completion, enabling a single thread to manage many IO channels.
- **Selectors for multiplexing**: Efficiently monitor multiple channels for readiness, avoiding the need for many threads.
- **Buffer-oriented data handling**: Use explicit buffers for reading and writing, enabling fine-grained control of data transfer.
- **Improved performance and scalability**: Reduce overhead and improve responsiveness, especially in server and network applications.
- **Platform independence**: Provide a consistent, portable API that abstracts underlying OS features like epoll, kqueue, or IOCP.

5.1.3 Core Concepts and Features of Java NIO

- **Buffers**: Unlike stream IO, NIO uses **buffers**—containers for fixed-size data arrays—which must be explicitly flipped, cleared, or compacted. This explicit data management improves efficiency and control.
- **Channels**: Channels are like bidirectional IO pipes representing connections to files, sockets, or other IO entities. They can be non-blocking and allow asynchronous operations.
- **Selectors**: Selectors let a single thread **monitor multiple channels** for events such as readiness to read or write, enabling multiplexed, non-blocking IO.

5.1.4 How NIO Improves Scalability and Efficiency

- **Single-thread multiplexing**: Instead of dedicating one thread per connection, one thread can handle thousands of connections by using selectors to react when channels are ready for IO.

- **Reduced context switching**: Fewer threads mean less overhead from context switches and lower memory consumption.
- **Non-blocking mode**: Threads don't block waiting on IO; they continue executing, enhancing throughput and responsiveness.
- **Direct buffers**: NIO supports direct buffers that interact more efficiently with the underlying OS and hardware, reducing copying and improving performance.
- **Asynchronous file and network IO**: Some NIO APIs support asynchronous operations for even greater concurrency.

5.1.5 Real-World Scenarios Where NIO Shines

High-Performance Servers

Web servers, chat servers, game servers, and other network applications that handle thousands or millions of simultaneous connections benefit from NIO's ability to multiplex many channels with a limited number of threads.

For example, a web server using NIO can manage many thousands of client sockets without allocating one thread per client, conserving system resources and improving response times.

Event-Driven Architectures

NIO's selector mechanism fits well with event-driven programming models, where IO readiness triggers events that drive application logic, such as in frameworks like Netty or Vert.x.

File Processing Applications

NIO's memory-mapped files and efficient channel operations improve performance in applications that process large files or perform random access reads/writes.

Real-Time or Interactive Applications

Applications that require low latency and high throughput, such as financial trading platforms or multimedia streaming, use NIO to avoid blocking delays and improve responsiveness.

5.1.6 Recap

Java's original IO API is simple and stream-based but suffers from blocking behavior and scalability issues when handling many simultaneous IO operations. **Java NIO** was introduced to overcome these limitations by providing:

- Non-blocking IO capabilities,
- Efficient multiplexing through selectors,

- Explicit buffer management,
- Better scalability and resource efficiency.

These improvements make NIO especially suitable for modern server applications, event-driven systems, and high-performance file processing, enabling Java applications to meet today's demanding performance and scalability requirements.

5.2 Core Concepts: Buffers, Channels, and Selectors

Java NIO (New IO) introduces a new architecture for handling IO in Java applications, designed to improve scalability, efficiency, and control. The foundation of this architecture rests on three core components: **Buffers**, **Channels**, and **Selectors**. Understanding these concepts and how they interact is essential to effectively leveraging Java NIO's power, especially in non-blocking IO operations.

5.2.1 Buffers: Containers for Data

At the heart of NIO's data handling is the **Buffer**. Unlike the classic Java IO's stream-based sequential access, NIO adopts a **buffer-oriented** model.

What is a Buffer?

A **Buffer** is a fixed-size block of memory that holds data to be read or written. It acts as a container or workspace where bytes or other primitive data types are stored before they are transferred to or from an IO source.

Buffers in NIO come in several types corresponding to different primitive data:

- `ByteBuffer` (most common)
- `CharBuffer`
- `IntBuffer`
- `FloatBuffer`
- …and more.

Buffer Structure and Key Properties

Each buffer maintains:

- **Capacity**: The total size of the buffer.
- **Position**: The current index where the next read or write will occur.
- **Limit**: The index one past the last valid data element (for reading) or the maximum writable position.

Before reading from or writing to a buffer, you manipulate these properties via methods like:

- `clear()`: Prepares the buffer for writing (position set to 0, limit set to capacity).
- `flip()`: Prepares the buffer for reading after writing (limit set to current position, position reset to 0).
- `rewind()`: Resets position to 0 to reread data.
- `compact()`: Moves unread data to the start for additional writing.

5.2.2 Analogy: Buffer as a Container

Think of a buffer like a glass container:

- **Capacity** = size of the container.
- **Position** = where you're currently pouring liquid in or scooping liquid out.
- **Limit** = the maximum level of liquid that you're allowed to read/write.

Before drinking (reading) from the glass, you first pour (write) some liquid, then flip your intention to drinking by resetting the position to the start.

5.2.3 Channels: The Data Pathways

While buffers hold data, **Channels** are the conduits or pipes that connect buffers to IO devices like files, network sockets, or pipes.

What is a Channel?

A **Channel** is a bi-directional communication channel for reading, writing, or both, between a Java program and an IO source/sink.

- Channels can be **readable**, **writable**, or both.
- They represent entities such as files (`FileChannel`), sockets (`SocketChannel`), and datagram connections (`DatagramChannel`).

How Channels Work with Buffers

Channels do not operate on streams of bytes like old IO; instead, they **transfer data to or from buffers**.

- To **read** data, a channel fills a buffer.
- To **write** data, a channel drains data from a buffer.

Because buffers are containers, this separation allows for efficient, flexible data handling.

5.2.4 Example: Reading from a File Using Channel and Buffer

Full runnable code:

```
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class FileReadExample {
    public static void main(String[] args) throws Exception {
        Path path = Paths.get("example.txt");
        try (FileChannel fileChannel = FileChannel.open(path, StandardOpenOption.READ)) {
            ByteBuffer buffer = ByteBuffer.allocate(1024);

            int bytesRead = fileChannel.read(buffer);
            while (bytesRead != -1) {
                buffer.flip(); // Prepare buffer for reading

                while (buffer.hasRemaining()) {
                    System.out.print((char) buffer.get()); // Read bytes from buffer
                }
                buffer.clear(); // Prepare buffer for next read
                bytesRead = fileChannel.read(buffer);
            }
        }
    }
}
```

This example shows how the `FileChannel` reads data into a `ByteBuffer`, then the program reads from the buffer.

5.2.5 Selectors: Multiplexing Multiple Channels

One of the most powerful features of NIO is the **Selector**, which enables a single thread to monitor multiple channels for events, such as readiness to read or write.

5.2.6 What is a Selector?

A **Selector** is an object that can **monitor multiple channels** simultaneously, waiting for events on any of them without blocking the thread.

Instead of dedicating one thread per connection or file operation (as in traditional blocking IO), a Selector lets a single thread react when any registered channel is ready for IO.

5.2.7 How Does a Selector Work?

1. You register multiple channels with the selector, specifying interest operations (e.g., OP_READ, OP_WRITE).
2. The selector blocks until at least one channel is ready.
3. It provides a set of **SelectionKeys**, representing channels ready for IO.
4. Your program handles these ready channels accordingly.

5.2.8 Analogy: Selector as a Traffic Controller

Imagine a traffic controller watching multiple roads (channels). Instead of blocking at each road, the controller waits until a vehicle arrives on any road and directs traffic accordingly. This approach avoids idle waiting at each road and efficiently manages multiple lanes with fewer resources.

5.2.9 Example: Registering a Channel with a Selector

Full runnable code:

```
import java.nio.channels.*;
import java.net.InetSocketAddress;

public class SelectorExample {
    public static void main(String[] args) throws Exception {
        Selector selector = Selector.open();

        // Open a socket channel and configure non-blocking mode
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.configureBlocking(false);

        // Connect to server
        socketChannel.connect(new InetSocketAddress("example.com", 80));

        // Register channel with selector for connect events
        socketChannel.register(selector, SelectionKey.OP_CONNECT);

        while (true) {
            selector.select(); // Wait for ready channels

            for (SelectionKey key : selector.selectedKeys()) {
                if (key.isConnectable()) {
                    // Finish connection process...
                }
                // Handle other events...
            }
            selector.selectedKeys().clear(); // Clear handled keys
        }
```

```
    }
}
```

5.2.10 How Buffers, Channels, and Selectors Work Together

- **Buffers** hold the data being read or written.
- **Channels** transfer data between buffers and IO sources/sinks.
- **Selectors** allow a single thread to efficiently manage multiple channels, waiting for events and enabling non-blocking IO.

Together, they enable scalable, high-performance applications where threads do not block waiting for IO but instead react to readiness events, reading/writing data in controlled buffer chunks.

5.2.11 Summary

- **Buffers**: Fixed-size containers managing data with explicit control over reading/writing positions.
- **Channels**: Data conduits connecting buffers with IO sources or destinations.
- **Selectors**: Multiplexers that monitor multiple channels for IO readiness, allowing efficient non-blocking IO with few threads.

This architecture contrasts with classic blocking IO and enables modern applications such as high-performance servers, real-time systems, and event-driven frameworks to handle large numbers of simultaneous IO operations efficiently.

5.3 Differences Between IO and NIO

Java provides two main APIs for input/output operations: **Traditional Java IO** (introduced in Java 1.0) and **Java NIO** (New IO, introduced in Java 1.4). Both enable reading and writing data, but they differ significantly in design philosophy, performance characteristics, and typical use cases.

Understanding these differences is crucial when choosing the right approach for your application, especially when dealing with scalable or high-performance IO needs.

5.3.1 Blocking vs Non-Blocking IO

Traditional Java IO: Blocking IO

Traditional Java IO is built on **blocking IO**. When a thread calls a read or write operation on an InputStream or OutputStream, it **blocks**, meaning the thread waits until the data is fully read or written before proceeding.

Example:

```
InputStream input = new FileInputStream("file.txt");
int data = input.read();  // Blocks until a byte is available or EOF
```

Implications:

- Each blocking operation consumes a thread, which must wait idly.
- For network servers, this often means one thread per client connection.
- Excessive threads cause overhead from context switching and memory usage.
- Scaling to thousands of concurrent connections is challenging.

Java NIO: Non-Blocking IO

Java NIO introduces **non-blocking IO** and **selectors**, enabling a single thread to manage multiple channels (connections) without waiting.

Channels can be configured to non-blocking mode, so calls like `read()` or `write()` return immediately:

- If data is available, they read/write some or all data.
- If no data is available, they return zero or a special value, letting the thread continue other work.

A **Selector** monitors many channels, notifying when one or more are ready for IO, avoiding thread-blocking.

Example:

Full runnable code:

```
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Set;

public class Test {

    public static void main(String[] argv) throws Exception {
        SocketChannel channel = SocketChannel.open();
        channel.configureBlocking(false);
        Selector selector = Selector.open();
        channel.register(selector, SelectionKey.OP_READ);

        while (true) {
            selector.select();  // Blocks until at least one channel is ready
            Set<SelectionKey> keys = selector.selectedKeys();
```

```
            for (SelectionKey key : keys) {
                if (key.isReadable()) {
                    // Read data without blocking
                }
            }
            keys.clear();
        }
    }
}
```

Implications:

- One thread can manage thousands of connections efficiently.
- Greatly improves scalability and resource utilization.
- Requires more complex programming to handle readiness events.

5.3.2 Stream-Based vs Buffer-Based Data Handling

Traditional IO: Stream-Based

Java IO models data as **streams** — continuous flows of bytes or characters.

- Data is read or written **sequentially**, one byte/char at a time.
- The API abstracts away the internal buffering or memory management.
- Developers typically read or write bytes in a loop.

Example:

```
byte[] buffer = new byte[1024];
int bytesRead = inputStream.read(buffer);
```

The `read()` method blocks until at least one byte is available and fills the buffer.

Java NIO: Buffer-Based

Java NIO operates with **buffers**—fixed-size containers that explicitly hold data.

- Data is **read into** a buffer or **written from** a buffer.
- Buffers have positions and limits that must be managed manually.
- This explicit control enables more efficient, flexible data processing.

Example:

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
int bytesRead = channel.read(buffer);
buffer.flip();  // Prepare to read from buffer
while(buffer.hasRemaining()) {
    byte b = buffer.get();
}
buffer.clear(); // Prepare buffer for next write
```

5.3.3 Synchronous vs Asynchronous Processing

Traditional IO: Mostly Synchronous

Traditional Java IO operations are synchronous and blocking — the program flow waits for IO completion.

- Simpler to program and reason about.
- May lead to thread starvation or performance bottlenecks under heavy load.

Java NIO: Supports Asynchronous and Synchronous Non-Blocking

NIO supports:

- **Non-blocking synchronous IO** via selectors (as described above).
- **Asynchronous IO** (introduced later as NIO.2 in Java 7) with classes like `AsynchronousFileChannel` that allow the OS to notify completion via callbacks or futures.

This enables event-driven architectures, where the program reacts to IO events without being stuck waiting.

5.3.4 Data Flow Differences

Traditional IO Data Flow:

```
Application Thread
    V (blocking read)
InputStream/File/Socket
    V sequential data flow
```

- Each IO call blocks the thread until data arrives.
- One thread per IO operation.

NIO Data Flow:

```
Application Thread
    V
Selector <-- multiple Channels (non-blocking)
    V
Buffers <--> Channels
```

- The selector notifies when channels are ready.
- One thread handles many channels.
- Buffers explicitly hold data in memory.

5.3.5 Impact on Application Design

- **Traditional IO** is easier for simple or small-scale applications.
- **NIO** requires managing buffers, selectors, and readiness events but excels in scalable, high-concurrency scenarios like servers or real-time applications.
- NIO fits well with **event-driven architectures** and frameworks like Netty, enabling highly responsive network applications.

5.3.6 Summary Table

Feature	Traditional Java IO	Java NIO
IO Model	Blocking, stream-based	Non-blocking, buffer-based
Thread Usage	One thread per IO operation	One thread manages many channels
Data Handling	Sequential byte/char streams	Explicit buffers with position/limit
Scalability	Limited by thread overhead	High scalability via selectors
Programming Complexity	Simpler API	More complex, event-driven
Asynchronous Support	No native async	Supports async with NIO.2
Typical Use Cases	Simple file IO, small apps	High-performance servers, network apps

5.3.7 Recap

While the traditional Java IO API remains useful for simple, blocking IO tasks, Java NIO offers a modern, scalable alternative designed for the demands of today's networked, concurrent, and high-performance applications. By moving from blocking streams to non-blocking buffers and selectors, NIO allows Java developers to build efficient and scalable systems without the overhead of thread-per-connection models.

5.4 Non-blocking IO and Selectors Overview

Java NIO (New IO) introduced a revolutionary way to handle IO operations that vastly improves scalability and performance for applications dealing with multiple simultaneous IO channels, such as servers handling many client connections.

At the core of this improvement lies the concept of **non-blocking IO** combined with **selectors**, allowing a **single thread to efficiently manage many IO channels** without blocking or dedicating one thread per connection. This section explores how selectors work, their relationship with channels, and how they enable scalable non-blocking IO.

5.4.1 Understanding Non-blocking IO

In traditional IO, when you read from or write to a channel (e.g., a socket or file), the thread blocks until the operation completes. For example, reading from a socket input stream blocks until data arrives. This is simple but inefficient for high concurrency — threads spend much time waiting and consume resources.

Non-blocking IO changes this behavior:

- When a thread attempts to read from or write to a channel in non-blocking mode, the operation returns immediately.
- If data is available, some or all of it is processed.
- If no data is available (e.g., no bytes to read), the method returns zero or a special value indicating no action.
- The thread can then continue doing other work instead of waiting.

This non-blocking mode is **enabled by setting a channel to non-blocking**, typically via:

```
channel.configureBlocking(false);
```

5.4.2 Selectors: Multiplexing Multiple Channels

Non-blocking IO alone is useful, but applications often need to manage many channels simultaneously (e.g., thousands of client sockets). Managing many channels in a single thread requires a mechanism to **know which channels are ready for reading, writing, or connecting without busy-waiting or polling inefficiently**.

This is where the `Selector` class comes in.

What is a Selector?

A **Selector** is a multiplexing tool that allows a single thread to monitor multiple channels for various IO events, such as:

- **Read readiness** — data available to read.
- **Write readiness** — channel ready to accept data.
- **Connect readiness** — a connection operation finished.
- **Accept readiness** — new incoming connection ready to be accepted.

The Selector **blocks the thread until at least one registered channel is ready for**

one of the requested operations, enabling efficient waiting without wasting CPU cycles.

5.4.3 How Selectors Work

1. **Open a Selector**

```
Selector selector = Selector.open();
```

2. **Configure Channels as Non-blocking and Register with Selector**

Each channel you want to monitor must be configured to non-blocking mode and registered with the selector, specifying which operations you want to listen for:

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

3. **Waiting for Ready Channels**

The selector's `select()` method blocks until one or more channels are ready:

```
int readyChannels = selector.select();
```

4. **Processing Selected Keys**

After `select()` returns, you retrieve the set of **`SelectionKey`** objects representing ready channels:

Full runnable code:

```
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class Test {

    public static void main(String[] argv) throws Exception {
        SocketChannel channel = SocketChannel.open();
        channel.configureBlocking(false);
        Selector selector = Selector.open();
        Set<SelectionKey> selectedKeys = selector.selectedKeys();
        Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

        while (keyIterator.hasNext()) {
            SelectionKey key = keyIterator.next();

            if (key.isReadable()) {
                // Read data from channel
            } else if (key.isWritable()) {
                // Write data to channel
            } else if (key.isAcceptable()) {
                // Accept a new connection
            } else if (key.isConnectable()) {
```

```
                // Finish connection process
            }

            keyIterator.remove(); // Important: Remove the key to avoid processing it again
        }
    }
}
```

5.4.4 SelectionKey: The Channels Registration Token

When you register a channel with a selector, you get a `SelectionKey` representing the relationship. It tracks:

- Which operations the channel is interested in (`interestOps`).
- Which operations are ready (`readyOps`).
- Attachment objects for storing related data or state.

5.4.5 Advantages of Using Selectors and Non-blocking IO

- **Scalability:** A single thread can manage thousands of channels efficiently, avoiding the overhead of one thread per connection.
- **Resource Efficiency:** Threads aren't blocked and don't waste CPU cycles polling or waiting.
- **Responsiveness:** Applications can respond immediately when channels are ready for IO.
- **Fits Event-driven Models:** The selector pattern aligns well with event-driven programming, facilitating reactive and asynchronous designs.

5.4.6 Conceptual Example: Echo Server Using Selectors

Full runnable code:

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class Test {
```

```
    public static void main(String[] argv) throws Exception {
        SocketChannel channel = SocketChannel.open();
        channel.configureBlocking(false);
        Selector selector = Selector.open();
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.bind(new InetSocketAddress(8080));
        serverChannel.configureBlocking(false);
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);

        while (true) {
            selector.select(); // Wait for events

            Set<SelectionKey> keys = selector.selectedKeys();
            Iterator<SelectionKey> iter = keys.iterator();

            while (iter.hasNext()) {
                SelectionKey key = iter.next();

                if (key.isAcceptable()) {
                    // Accept connection
                    ServerSocketChannel server = (ServerSocketChannel) key.channel();
                    SocketChannel client = server.accept();
                    client.configureBlocking(false);
                    client.register(selector, SelectionKey.OP_READ);
                }

                if (key.isReadable()) {
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(256);
                    int bytesRead = client.read(buffer);
                    if (bytesRead == -1) {
                        client.close(); // Client closed connection
                    } else {
                        buffer.flip();
                        client.write(buffer); // Echo data back
                    }
                }

                iter.remove(); // Remove handled key
            }
        }
    }
}
```

This simple echo server handles multiple clients concurrently on a single thread without blocking.

5.4.7 Summary

- **Non-blocking IO** allows IO operations to return immediately, avoiding thread blocking.
- **Selectors** multiplex many non-blocking channels, notifying when channels are ready for IO.

- A single thread can efficiently monitor and service many channels using a selector.
- `SelectionKey`s represent channel registrations and track IO readiness.
- This model enables scalable, resource-efficient, and responsive IO applications, especially suitable for servers and event-driven systems.

Chapter 6.

Buffers and Channels

1. ByteBuffer and Other Buffer Types
2. Buffer Operations: Read, Write, Flip, Clear, Compact
3. FileChannel for File Operations
4. SocketChannel and DatagramChannel
5. Memory-Mapped Files

6 Buffers and Channels

6.1 ByteBuffer and Other Buffer Types

In Java NIO, the concept of **buffers** is central to handling data during IO operations. Unlike the traditional stream-based IO model, where data flows sequentially byte-by-byte or character-by-character, NIO uses buffers as fixed-size containers to hold data explicitly. Understanding buffers, especially `ByteBuffer` and its siblings, is essential for efficient and controlled data processing in Java NIO.

6.1.1 What Are Buffers in Java NIO?

A **Buffer** is a block of memory used for reading and writing data. It acts as an intermediary storage area between the application and the IO channel. Buffers provide a structured way to handle data with explicit control over the read/write process.

Buffers come with a fixed **capacity**, and they maintain two important pointers:

- **Position:** The index of the next element to be read or written.
- **Limit:** The boundary that marks the end of the readable or writable data.

Additionally, buffers have a **mark** feature to save and reset positions during complex operations.

6.1.2 The Role of Buffers

Buffers serve as the workspace for data moving between Java programs and IO devices (files, sockets, etc.). The workflow usually involves:

- Reading data **from a channel into a buffer**.
- Processing or manipulating data inside the buffer.
- Writing data **from the buffer back to a channel**.

This explicit buffering model allows for more efficient IO by reducing the overhead of system calls and enabling batch processing of data.

6.1.3 The ByteBuffer Class

Among all buffer types, **`ByteBuffer`** is the most fundamental and widely used. It stores data as raw bytes (`byte` values). Because virtually all data — text, images, multimedia — can be

represented as bytes, `ByteBuffer` acts as the base for many IO operations.

6.1.4 Internal Structure of ByteBuffer

`ByteBuffer` extends the abstract `Buffer` class and provides several key methods for manipulating data.

- **Allocation:**

  You create a `ByteBuffer` by allocating a fixed amount of space.

```
ByteBuffer buffer = ByteBuffer.allocate(1024);  // 1 KB buffer
```

- **Position, Limit, Capacity:**
  - `capacity()` returns the buffer's total size (e.g., 1024 bytes).
  - `position()` shows the current index for reading/writing.
  - `limit()` marks the end of data for read/write operations.
- **Writing to the Buffer:**

  You write data using `put()` methods, which write bytes and advance the position.

```
buffer.put((byte)65);           // Writes ASCII 'A'
buffer.put(new byte[] {1, 2}); // Writes multiple bytes
```

- **Flipping the Buffer:**

  Before reading data back, you call `flip()` to prepare the buffer:

```
buffer.flip();
```

  This sets the limit to the current position and resets position to zero, so you can read from the start up to the written data.

- **Reading from the Buffer:**

  You read data using `get()` methods:

```
byte b = buffer.get();       // Reads one byte
byte[] data = new byte[buffer.remaining()];
buffer.get(data);            // Reads multiple bytes
```

- **Clearing and Compacting:**

  After reading or when you want to reuse the buffer, you use:
  - `clear()`: Resets position and limit to capacity for writing new data.
  - `compact()`: Moves unread data to the beginning and prepares for writing more data.

6.1.5 Example: Basic ByteBuffer Usage

Full runnable code:

```
import java.nio.ByteBuffer;

public class ByteBufferExample {
    public static void main(String[] args) {
        // Allocate a ByteBuffer with capacity 10 bytes
        ByteBuffer buffer = ByteBuffer.allocate(10);

        // Write bytes into the buffer
        buffer.put((byte) 'H');
        buffer.put((byte) 'i');

        // Prepare the buffer for reading
        buffer.flip();

        // Read bytes from the buffer and print as characters
        while (buffer.hasRemaining()) {
            System.out.print((char) buffer.get());
        }
        // Output: Hi
    }
}
```

6.1.6 Other Buffer Types in Java NIO

Java NIO provides specialized buffers for different primitive data types, each subclassing the abstract `Buffer` class:

Buffer Type	Stores Data of Type	Common Use Case
`CharBuffer`	`char` (16-bit Unicode)	Handling character data, text processing
`ShortBuffer`	`short`	Working with 16-bit integers
`IntBuffer`	`int`	Working with 32-bit integers
`LongBuffer`	`long`	Handling 64-bit integers
`FloatBuffer`	`float`	Working with floating-point numbers
`DoubleBuffer`	`double`	Handling double-precision floating-point numbers

6.1.7 Why Use Different Buffer Types?

Using typed buffers allows you to:

- Work directly with higher-level data types without manually converting them to bytes.
- Simplify code when working with structured binary data formats.

- Benefit from type-specific `get()` and `put()` methods.

For example, an `IntBuffer` lets you read/write integers rather than manually packing and unpacking bytes.

6.1.8 Example: Using an IntBuffer

Full runnable code:

```
import java.nio.IntBuffer;

public class IntBufferExample {
    public static void main(String[] args) {
        // Allocate an IntBuffer with capacity for 5 integers
        IntBuffer intBuffer = IntBuffer.allocate(5);

        // Put some integers into the buffer
        intBuffer.put(10);
        intBuffer.put(20);
        intBuffer.put(30);

        // Prepare buffer for reading
        intBuffer.flip();

        // Read integers from the buffer
        while (intBuffer.hasRemaining()) {
            System.out.println(intBuffer.get());
        }
        // Output:
        // 10
        // 20
        // 30
    }
}
```

6.1.9 Buffer Lifecycle Recap

1. **Allocate** the buffer with a fixed size.
2. **Write data** using `put()` methods.
3. **Flip** the buffer to switch from writing to reading mode.
4. **Read data** using `get()` methods.
5. Optionally, **compact** or **clear** the buffer for reuse.

6.1.10 Summary

- **Buffers** are core to Java NIO's approach to data handling, offering explicit control over memory and data transfer.
- `ByteBuffer` is the fundamental buffer storing raw bytes and is used extensively in IO operations.
- Other buffer types (`CharBuffer`, `IntBuffer`, `DoubleBuffer`, etc.) allow direct handling of primitive data types, simplifying code and improving clarity.
- Understanding buffer states (position, limit, capacity) and lifecycle methods (`flip()`, `clear()`, `compact()`) is crucial for correct usage.
- Using buffers improves IO efficiency by allowing batch operations and better control over data flow.

6.2 Buffer Operations: Read, Write, Flip, Clear, Compact

Buffers are fundamental to Java NIO's data handling, acting as containers that hold data for reading from or writing to IO channels. To use buffers effectively, you must understand the key operations that control their internal state: the **position**, **limit**, and **capacity**. These operations govern where data is written or read, how much data can be accessed, and how the buffer can be reused efficiently.

This section explains the most important buffer operations: `put()` (write), `get()` (read), `flip()`, `clear()`, and `compact()`, detailing their effects and showing common usage patterns.

6.2.1 Buffer Anatomy Recap

Before diving into operations, remember these core properties of a buffer:

- **Capacity:** The fixed size of the buffer; the maximum number of elements it can hold.
- **Position:** The index of the next element to read or write.
- **Limit:** The index marking the end of readable or writable data.

At any moment, the buffer's state determines how much data can be read or written.

6.2.2 Writing to Buffers: `put()`

The `put()` method writes data into the buffer at the current position and advances the position by the number of elements written.

- You can write single elements or arrays of data.

- Writing is only allowed up to the buffer's capacity (or limit, depending on state).
- Attempting to write beyond the limit throws a `BufferOverflowException`.

Example:

```
ByteBuffer buffer = ByteBuffer.allocate(10);
buffer.put((byte)10);
buffer.put((byte)20);

System.out.println("Position after writing: " + buffer.position());  // Outputs 2
```

At this point, the buffer's position is 2 (two bytes written), limit is 10 (capacity).

6.2.3 Preparing to Read: `flip()`

After writing data, you call `flip()` to switch the buffer from **write mode** to **read mode**. What `flip()` does:

- Sets the limit to the current position (marks the end of valid data).
- Resets the position to zero (start reading from the beginning).

This prepares the buffer to be read from the data just written.

Example:

```
buffer.flip();
System.out.println("Position after flip: " + buffer.position());  // 0
System.out.println("Limit after flip: " + buffer.limit());        // 2
```

Now, the buffer is ready to read 2 bytes from position 0 up to limit 2.

6.2.4 Reading from Buffers: `get()`

The `get()` method reads data from the current position and advances it.

- Reading continues until position reaches limit.
- Reading beyond limit throws a `BufferUnderflowException`.

Example:

```
byte first = buffer.get();
byte second = buffer.get();
System.out.println("Bytes read: " + first + ", " + second);
System.out.println("Position after reading: " + buffer.position()); // 2
```

6.2.5 Clearing the Buffer: `clear()`

After you finish reading and want to write new data, call `clear()`.

- Resets position to 0.
- Sets limit to capacity.
- The buffer is ready for writing again.
- **Note:** It does NOT erase data, but marks the entire buffer as writable.

Example:

```
buffer.clear();
System.out.println("Position after clear: " + buffer.position());  // 0
System.out.println("Limit after clear: " + buffer.limit());        // 10
```

6.2.6 Compacting the Buffer: `compact()`

`compact()` is used when you have read some data from the buffer but still have unread data that you want to keep before writing more.

What `compact()` does:

- Copies unread data from current position to the beginning of the buffer.
- Sets position to just after the copied data.
- Sets limit to capacity, so you can write more data after the unread portion.

This avoids overwriting unread data while allowing new data to be appended.

Example:

```
// Suppose buffer has limit=10, position=4 after reading some bytes
buffer.compact();
System.out.println("Position after compact: " + buffer.position());
// Now position is set after unread data, ready for writing more
```

6.2.7 Step-by-Step Example: Using Buffer Operations in a Typical Read/Write Cycle

Full runnable code:

```
import java.nio.ByteBuffer;

public class Test {

    public static void main(String[] argv) throws Exception {
        ByteBuffer buffer = ByteBuffer.allocate(8);

        // Step 1: Write data into buffer
```

```
        buffer.put((byte) 1);
        buffer.put((byte) 2);
        buffer.put((byte) 3);
        System.out.println("After writing, position: " + buffer.position()); // 3

        // Step 2: Prepare buffer for reading
        buffer.flip();
        System.out.println("After flip, position: " + buffer.position() + ", limit: " + buffer.limit())

        // Step 3: Read one byte
        byte b = buffer.get();
        System.out.println("Read byte: " + b);
        System.out.println("Position after read: " + buffer.position()); // 1

        // Step 4: Compact the buffer (keep unread bytes)
        buffer.compact();
        System.out.println("After compact, position: " + buffer.position() + ", limit: " + buffer.limit

        // Step 5: Write more data after compacting
        buffer.put((byte) 4);
        buffer.put((byte) 5);
        System.out.println("After writing more, position: " + buffer.position()); // 4

        // Step 6: Prepare to read all available data again
        buffer.flip();
        while (buffer.hasRemaining()) {
            System.out.println("Reading: " + buffer.get());
        }
    }
}
```

Output breakdown:

- After initial write, position is 3 (bytes 1,2,3).
- `flip()` sets position to 0 and limit to 3 for reading.
- One byte read advances position to 1.
- `compact()` shifts unread bytes (2,3) to front and sets position to 2 for writing more.
- New bytes (4,5) added; position moves to 4.
- Another `flip()` prepares buffer for reading all 4 bytes.
- Reads output: 2, 3, 4, 5.

6.2.8 Summary of Buffer Operations

Operation	Position (pos)	Limit (lim)	Capacity (cap)	Purpose
`put()`	Advances by number written	Unchanged	Unchanged	Write data into buffer at current position
`flip()`	Set to 0	Set to current position	Unchanged	Prepare buffer for reading data just written

Operation	Position (pos)	Limit (lim)	Capacity (cap)	Purpose
`get()`	Advances by number read	Unchanged	Unchanged	Read data from buffer at current position
`clear()`	Set to 0	Set to capacity	Unchanged	Prepare buffer for writing new data (discard old markers)
`compact()`	Set to unread data length	Set to capacity	Unchanged	Keep unread data, move to front, prepare for writing more

6.2.9 Why These Operations Matter

- **`flip()`** is essential to switch modes between writing and reading.
- **`clear()`** resets the buffer for reuse without erasing data physically.
- **`compact()`** is useful for partial reads where you want to keep leftover data and still write new data without losing anything.
- Using these operations correctly avoids common bugs like reading unwritten data or overwriting unread bytes.

6.2.10 Conclusion

Mastering buffer operations like `put()`, `get()`, `flip()`, `clear()`, and `compact()` is crucial for efficient data management in Java NIO. These operations give fine-grained control over buffer state, enabling you to handle complex IO workflows and maximize performance.

With practice, these methods become intuitive and enable building scalable, non-blocking IO applications that efficiently manage data flow.

6.3 FileChannel for File Operations

Java NIO (New IO) introduced several new abstractions for more efficient and flexible input/output operations. One of the core classes for file handling in NIO is **`FileChannel`**. It provides an efficient way to read from, write to, and manipulate files at a low level, improving upon the traditional Java IO classes like `FileInputStream` and `FileOutputStream`.

6.3.1 What is FileChannel?

`FileChannel` is a part of the `java.nio.channels` package and represents a connection to a file that supports reading, writing, mapping, and manipulating file content. Unlike stream-based IO, which processes data sequentially, `FileChannel` allows random access to file content and can read/write data at specific positions.

6.3.2 Advantages of FileChannel over Traditional IO

1. **Random Access:** Unlike traditional `FileInputStream` or `FileOutputStream`, which only allow sequential reading or writing, `FileChannel` supports random access, enabling you to read/write at any position in the file without having to process all preceding bytes.
2. **Efficient Bulk Data Transfer:** `FileChannel` can transfer data directly between channels using `transferTo()` and `transferFrom()`, which can leverage lower-level OS optimizations and reduce overhead.
3. **Memory Mapping:** You can map files directly into memory with `FileChannel.map()`, allowing you to treat file content as a part of memory — improving performance for large files.
4. **Non-blocking and Asynchronous IO Compatibility:** As part of NIO, `FileChannel` fits well into non-blocking IO paradigms and works smoothly with buffers and selectors.
5. **Explicit Buffer Management:** Instead of relying on streams, `FileChannel` requires buffers for reading and writing, offering more precise control over data flow.

6.3.3 Creating a FileChannel

You obtain a `FileChannel` instance from file streams or from the `java.nio.file.Files` utility:

```
// From FileInputStream or FileOutputStream
FileInputStream fis = new FileInputStream("example.txt");
FileChannel channel = fis.getChannel();

// Or from RandomAccessFile for read-write mode
RandomAccessFile raf = new RandomAccessFile("example.txt", "rw");
FileChannel rafChannel = raf.getChannel();

// Or using Files.newByteChannel (Java 7+)
Path path = Paths.get("example.txt");
FileChannel fileChannel = FileChannel.open(path, StandardOpenOption.READ, StandardOpenOption.WRITE);
```

6.3.4 Key FileChannel Methods

- **`int read(ByteBuffer dst)`**: Reads bytes from the channel into the buffer starting at the current file position. Advances the position by the number of bytes read.
- **`int write(ByteBuffer src)`**: Writes bytes from the buffer into the channel starting at the current file position. Advances the position by the number of bytes written.
- **`long position()` / `position(long newPosition)`**: Gets or sets the file position for the next read or write.
- **`long size()`**: Returns the current size of the file.
- **`FileLock lock(long position, long size, boolean shared)`**: Locks a region of the file for exclusive or shared access.
- **`long transferTo(long position, long count, WritableByteChannel target)`**: Transfers bytes directly from this channel to another writable channel.
- **`long transferFrom(ReadableByteChannel src, long position, long count)`**: Transfers bytes from a readable channel into this file channel.
- **`MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`**: Maps a region of the file into memory.

6.3.5 Reading a File Using FileChannel

Reading data from a file using `FileChannel` requires a `ByteBuffer` to hold the data read:

Full runnable code:

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

public class FileChannelReadExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt");
             FileChannel fileChannel = fis.getChannel()) {

            ByteBuffer buffer = ByteBuffer.allocate(1024); // 1 KB buffer

            int bytesRead = fileChannel.read(buffer);
            while (bytesRead != -1) {
                buffer.flip();  // Prepare buffer for reading

                while (buffer.hasRemaining()) {
                    System.out.print((char) buffer.get());  // Print characters read
                }

                buffer.clear(); // Prepare buffer for writing
```

```
                bytesRead = fileChannel.read(buffer);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- We allocate a buffer of size 1024 bytes.
- We read data into the buffer until EOF (`read()` returns -1).
- Before reading data from the buffer, we call `flip()` to switch from writing mode to reading mode.
- After processing, `clear()` resets the buffer to receive more data.

6.3.6 Writing to a File Using FileChannel

Similarly, writing requires a buffer filled with data to be written:

Full runnable code:

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

public class FileChannelWriteExample {
    public static void main(String[] args) {
        String data = "Hello, FileChannel!";

        try (FileOutputStream fos = new FileOutputStream("output.txt");
             FileChannel fileChannel = fos.getChannel()) {

            ByteBuffer buffer = ByteBuffer.allocate(1024);
            buffer.put(data.getBytes());

            buffer.flip(); // Prepare buffer for writing to channel

            while (buffer.hasRemaining()) {
                fileChannel.write(buffer);
            }

            System.out.println("Data written to file successfully.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- We put the byte representation of a string into the buffer.

- `flip()` switches buffer from write mode to read mode.
- We write the buffer contents to the channel in a loop until all bytes are written.

6.3.7 Random Access with FileChannel

Because `FileChannel` supports setting the file position, you can read or write data at arbitrary locations:

Full runnable code:

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

public class Test {

    public static void main(String[] argv) throws Exception {
        RandomAccessFile raf = new RandomAccessFile("data.bin", "rw");
        FileChannel channel = raf.getChannel();

        ByteBuffer buffer = ByteBuffer.allocate(4);
        buffer.putInt(12345);
        buffer.flip();

        // Write at position 10
        channel.position(10);
        channel.write(buffer);

        // Read back the integer at position 10
        buffer.clear();
        channel.position(10);
        channel.read(buffer);
        buffer.flip();

        System.out.println("Read integer: " + buffer.getInt());

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

6.3.8 Summary

- `FileChannel` is a powerful NIO class for file operations supporting efficient, random access to file data.
- It offers advantages over traditional IO, such as bulk transfers, memory mapping, and position-based access.
- Operations revolve around `ByteBuffer`s to read and write data.

- Key methods include `read()`, `write()`, `position()`, and `map()`.
- Using `FileChannel` with buffers enables high-performance and flexible file IO suited for modern Java applications.

6.4 SocketChannel and DatagramChannel

Java NIO provides powerful networking capabilities through channels designed for scalable, efficient IO operations. Two primary channel classes for network communication are:

- **`SocketChannel`** — for TCP-based stream communication.
- **`DatagramChannel`** — for UDP-based datagram communication.

These classes support **non-blocking IO**, allowing Java applications to handle many simultaneous network connections with minimal threads and overhead.

6.4.1 SocketChannel: TCP Communication in Java NIO

Role and Characteristics:

`SocketChannel` is a selectable channel for stream-oriented TCP connections. It allows you to open a TCP socket connection to a remote server, read from and write to the connection, and optionally configure non-blocking behavior.

TCP (Transmission Control Protocol) provides a reliable, ordered, and error-checked delivery of a stream of bytes between applications.

Key Features:

- Supports **connection-oriented** communication.
- Can operate in **blocking** or **non-blocking** modes.
- Integrates with NIO's **Selector** for multiplexed IO.
- Reads and writes use `ByteBuffer`s.
- Allows random access to the stream position only in blocking mode (non-blocking reads/writes are always sequential).

6.4.2 Basic Usage of SocketChannel

1. **Opening and Connecting:**

Full runnable code:

```java
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class SocketChannelExample {
    public static void main(String[] args) {
        try {
            // Open a SocketChannel
            SocketChannel socketChannel = SocketChannel.open();

            // Configure non-blocking mode
            socketChannel.configureBlocking(false);

            // Connect to server at localhost:5000
            socketChannel.connect(new InetSocketAddress("localhost", 5000));

            // Wait or check for connection completion (non-blocking)
            while (!socketChannel.finishConnect()) {
                System.out.println("Connecting...");
                // Do something else or sleep briefly
            }

            // Prepare data to send
            String message = "Hello, Server!";
            ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());

            // Write data to server
            while (buffer.hasRemaining()) {
                socketChannel.write(buffer);
            }

            // Read response
            buffer.clear();
            int bytesRead = socketChannel.read(buffer);
            if (bytesRead > 0) {
                buffer.flip();
                byte[] received = new byte[buffer.remaining()];
                buffer.get(received);
                System.out.println("Received: " + new String(received));
            }

            socketChannel.close();

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

Explanation:

- We open a `SocketChannel` and configure it to non-blocking mode.
- The connection attempt is initiated but may not complete immediately; `finishConnect()` confirms when connected.
- Data is written and read using `ByteBuffers`.
- Non-blocking mode allows the application to perform other tasks while waiting for the

connection or IO readiness.

6.4.3 DatagramChannel: UDP Communication in Java NIO

Role and Characteristics:

`DatagramChannel` provides a selectable channel for sending and receiving UDP packets.

UDP (User Datagram Protocol) is connectionless and message-oriented, meaning it sends discrete packets without establishing a dedicated connection, with no guarantee of delivery or order.

Key Features:

- Supports **connectionless** communication.
- Can send and receive **datagrams** (packets).
- Works in blocking or non-blocking modes.
- Uses `ByteBuffer` for packet data.
- Supports multicast via the `join()` method.

6.4.4 Basic Usage of DatagramChannel

1. **Opening, Sending, and Receiving Packets:**

Full runnable code:

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class DatagramChannelExample {
    public static void main(String[] args) {
        try {
            // Open DatagramChannel and bind to a local port
            DatagramChannel datagramChannel = DatagramChannel.open();
            datagramChannel.bind(new InetSocketAddress(9999));

            // Configure non-blocking mode
            datagramChannel.configureBlocking(false);

            // Prepare a message to send
            String message = "Hello UDP!";
            ByteBuffer sendBuffer = ByteBuffer.wrap(message.getBytes());

            // Send the packet to a remote address
            InetSocketAddress remoteAddress = new InetSocketAddress("localhost", 8888);
            datagramChannel.send(sendBuffer, remoteAddress);
```

```
            // Prepare buffer for receiving
            ByteBuffer receiveBuffer = ByteBuffer.allocate(1024);

            // Receive packets (non-blocking, returns null if none)
            InetSocketAddress senderAddress = (InetSocketAddress) datagramChannel.receive(receiveBuffer)

            if (senderAddress != null) {
                receiveBuffer.flip();
                byte[] data = new byte[receiveBuffer.remaining()];
                receiveBuffer.get(data);
                System.out.println("Received from " + senderAddress + ": " + new String(data));
            } else {
                System.out.println("No packet received");
            }

            datagramChannel.close();

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

Explanation:

- We open and bind the `DatagramChannel` to a local port to receive packets.
- The channel is set to non-blocking mode.
- We send a UDP packet using `send()`.
- We attempt to receive a packet using `receive()`, which returns immediately in non-blocking mode.
- Received data is read from the `ByteBuffer`.

6.4.5 Non-Blocking IO and Selectors

Both `SocketChannel` and `DatagramChannel` can be used with Java NIO's **Selector** class to handle multiple network connections or datagram sockets efficiently using a single thread.

Non-blocking IO avoids the thread-per-connection model, greatly improving scalability for servers or clients managing many simultaneous connections.

Example Selector usage highlights:

- Register channels with the selector for interested events (OP_READ, OP_WRITE, OP_CONNECT).
- The selector blocks until one or more registered channels are ready for IO.
- The application processes ready channels, reading or writing as appropriate.

6.4.6 Summary

Feature	SocketChannel	DatagramChannel
Protocol	TCP (stream, connection-oriented)	UDP (datagram, connectionless)
Communication model	Reliable, ordered byte streams	Unreliable, discrete packets
Blocking/non-blocking	Supports both	Supports both
Usage scenario	Web servers, chat clients, file transfer	DNS, real-time data, multicast
Key methods	`connect()`, `read()`, `write()`, `finishConnect()`	`send()`, `receive()`

6.4.7 Conclusion

`SocketChannel` and `DatagramChannel` are powerful tools in Java NIO for building scalable network applications. Their support for non-blocking IO and integration with selectors enables efficient multiplexed IO operations. `SocketChannel` is ideal for reliable TCP stream communication, while `DatagramChannel` suits fast, connectionless UDP messaging.

Mastering these classes equips Java developers to build high-performance servers, clients, and real-time networked applications that can handle thousands of connections efficiently.

6.5 Memory-Mapped Files

When dealing with large files or performance-critical applications, traditional read/write operations may become bottlenecks. Java NIO offers an advanced feature called **memory-mapped files** that can significantly improve IO efficiency by leveraging the operating system's virtual memory subsystem. This section explains what memory mapping is, how it works in Java, and how to use `MappedByteBuffer` to perform fast file operations.

6.5.1 What is Memory Mapping?

Memory mapping a file means associating a portion of a file directly with a region of memory. Instead of explicitly reading or writing bytes via system calls, the file's contents are mapped into the process's address space. This allows a program to access file contents just like normal

memory arrays.

How it works:

- The operating system loads the requested file region into RAM on demand.
- Reads and writes to the memory region are automatically synchronized with the file on disk.
- The OS handles paging in/out of data, caching, and flushing changes back to disk.
- Access to the file content becomes very fast and efficient, often faster than traditional IO streams.

6.5.2 Advantages of Memory-Mapped Files

1. **High Performance:** Because file content is accessed via memory pointers, reading and writing can avoid many copies and system calls.
2. **Random Access Efficiency:** You can read or write any part of the file instantly by simply accessing the memory region at an offset.
3. **Reduced Buffering Overhead:** Eliminates the need for manual buffering since the OS handles paging.
4. **Simplified Code:** Treat file data as a simple array, simplifying complex IO logic.
5. **Large File Handling:** Suitable for working with files larger than the available heap space, since the OS pages data transparently.

6.5.3 MappedByteBuffer: The Java API

In Java NIO, the `FileChannel` class provides a `map()` method that returns a `MappedByteBuffer`. This buffer represents the memory-mapped region of the file.

Signature:

```
MappedByteBuffer map(FileChannel.MapMode mode, long position, long size) throws IOException;
```

- **`mode`**: Specifies access type — read-only, read-write, or private (copy-on-write).
  - `MapMode.READ_ONLY`: Read-only mapping.
  - `MapMode.READ_WRITE`: Read-write mapping.
  - `MapMode.PRIVATE`: Changes are private to the process (not written back).
- **`position`**: The starting byte offset in the file.
- **`size`**: The number of bytes to map.

Once mapped, you can read/write the buffer like any other NIO buffer. Changes to a `READ_WRITE` buffer are written back to the file, either immediately or when the buffer is flushed.

6.5.4 Example: Mapping a File and Reading Data

Full runnable code:

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

public class MemoryMappedFileRead {
    public static void main(String[] args) {
        try (RandomAccessFile file = new RandomAccessFile("largefile.txt", "r");
             FileChannel channel = file.getChannel()) {

            // Map the first 1024 bytes of the file into memory (read-only)
            MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, 1024);

            // Read bytes from the buffer
            for (int i = 0; i < 1024; i++) {
                byte b = buffer.get(i);
                System.out.print((char) b);
            }

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

Explanation:

- Opens a file in read-only mode.
- Maps the first 1024 bytes into memory.
- Reads bytes directly from the mapped buffer as if reading an array.
- No explicit `read()` calls; the OS manages loading pages on demand.

6.5.5 Example: Modifying a File Using Memory Mapping

Full runnable code:

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

```java
public class MemoryMappedFileWrite {
    public static void main(String[] args) {
        try (RandomAccessFile file = new RandomAccessFile("data.bin", "rw");
             FileChannel channel = file.getChannel()) {

            // Map the first 128 bytes of the file into memory (read-write)
            MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_WRITE, 0, 128);

            // Write some bytes into the buffer at specific positions
            buffer.put(0, (byte) 10);
            buffer.put(1, (byte) 20);
            buffer.put(2, (byte) 30);

            // Sequential write example
            buffer.position(3);
            buffer.put((byte) 40);
            buffer.put((byte) 50);

            // Force changes to be written to disk (optional, as OS flushes eventually)
            buffer.force();

            System.out.println("Data written successfully.");

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

Explanation:

- Opens the file in read-write mode.
- Maps 128 bytes starting at the beginning of the file.
- Writes data at specific offsets within the mapped region.
- Calls `force()` to flush changes to disk immediately (optional).

6.5.6 Best Practices and Considerations

- **Size Matters:** The size parameter controls how much of the file is mapped. Mapping very large files in one go may cause `OutOfMemoryError` on 32-bit JVMs. Consider mapping in chunks.
- **OS-Dependent Behavior:** Memory mapping relies heavily on the OS's virtual memory system. Performance and behavior may vary across platforms.
- **Resource Management:** Unlike streams, `MappedByteBuffers` are managed by the OS, and explicit unmapping is not directly exposed in Java. This can cause the file to remain locked until the buffer is garbage collected, which can affect file deletion on some systems.
- **Concurrency:** Multiple threads can safely read from a `MappedByteBuffer`. Writing

should be carefully synchronized.

- **File Size Changes:** If the file size changes on disk after mapping, behavior is undefined. Ideally, map the file once when size is stable.

6.5.7 When to Use Memory-Mapped Files

- Reading or writing very large files where performance matters.
- Random access patterns requiring low latency.
- Applications like databases, multimedia processing, or large-scale file transformations.
- Avoiding overhead of frequent system calls in traditional IO.

6.5.8 Summary

- **Memory mapping** maps a file's content directly into memory, allowing fast and flexible IO.
- Java NIO's `FileChannel.map()` returns a `MappedByteBuffer` to access mapped file regions.
- It improves performance by minimizing system calls, enabling random access, and leveraging OS virtual memory.
- Use `MappedByteBuffer` for efficient reading and writing of files, especially large or performance-critical ones.
- Remember platform-specific behavior and resource management considerations.

Chapter 7.

Selectors and Non-blocking IO

1. Understanding Selectors
2. Registering Channels with Selectors
3. Handling SelectionKeys and Events
4. Building a Simple Non-blocking Server
5. Performance Considerations

7 Selectors and Non-blocking IO

7.1 Understanding Selectors

Java NIO (New IO) revolutionized the way Java applications handle input/output by introducing non-blocking IO and scalable architectures. At the heart of this scalable non-blocking model lies the `Selector`, an object that allows a single thread to monitor multiple channels (e.g., `SocketChannel`, `ServerSocketChannel`) for events like **read**, **write**, and **connect** readiness.

This mechanism enables efficient resource usage and is the cornerstone for building high-performance servers and event-driven applications.

7.1.1 What is a Selector?

A **Selector** is a Java NIO component that can monitor **multiple channels** simultaneously and detect when one or more of them are **ready** for a certain type of IO operation (such as accepting a connection, reading, or writing). Instead of dedicating a thread per socket connection, a single thread can use a selector to manage **hundreds or thousands of connections**.

Selectors support the following **channel types**:

- `SocketChannel` (client connections)
- `ServerSocketChannel` (server sockets)
- `DatagramChannel` (UDP sockets)

These channels must be in **non-blocking mode** to be used with a `Selector`.

7.1.2 Why Use Selectors?

Traditionally, handling multiple client connections meant using one thread per connection. This model does not scale well, especially in environments with many idle or slow connections. Selectors solve this problem by enabling **multiplexed IO** — a single thread checks multiple channels to see which ones are ready, then acts accordingly.

Real-world Analogy:

Imagine a **security guard** monitoring multiple **doors** (channels). Instead of standing at each door waiting (blocking), the guard walks through a hallway (selector) and checks which doors have visitors (read/write events). This is far more efficient than assigning a guard per door.

7.1.3 Key Concepts

- **Channel registration**: A channel is registered with a selector and specifies the kind of operation to monitor (`OP_ACCEPT`, `OP_CONNECT`, `OP_READ`, `OP_WRITE`).
- **SelectionKey**: When a channel is registered, a `SelectionKey` is returned. This key represents the registration and holds the interest and readiness sets.
- **Selected keys**: When the selector detects that a channel is ready, it returns a set of selection keys representing the channels that are ready for operations.

7.1.4 Common Selection Operations

- `OP_ACCEPT`: A server socket is ready to accept a new connection.
- `OP_CONNECT`: A socket channel finished its connection process.
- `OP_READ`: A channel is ready for reading.
- `OP_WRITE`: A channel is ready for writing.

7.1.5 Basic Workflow

1. **Create a selector**
2. **Configure channels to non-blocking mode**
3. **Register channels with the selector**
4. **Call `select()` to wait for readiness**
5. **Process selected keys and perform IO**
6. **Repeat the loop**

7.1.6 Basic Code Example: Server Using Selector

Full runnable code:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.Iterator;
import java.util.Set;

public class SelectorExample {
    public static void main(String[] args) throws IOException {
        // Step 1: Create Selector
        Selector selector = Selector.open();
```

```
        // Step 2: Create ServerSocketChannel and bind port
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.bind(new InetSocketAddress(5000));
        serverChannel.configureBlocking(false);

        // Step 3: Register ServerChannel with Selector for OP_ACCEPT
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
        System.out.println("Server started. Listening on port 5000...");

        // Step 4: Event loop
        while (true) {
            // Wait for events (blocking with timeout optional)
            selector.select();

            // Get keys for channels that are ready
            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> iter = selectedKeys.iterator();

            while (iter.hasNext()) {
                SelectionKey key = iter.next();

                // Step 5: Check what event occurred
                if (key.isAcceptable()) {
                    // Accept the new client connection
                    ServerSocketChannel server = (ServerSocketChannel) key.channel();
                    SocketChannel client = server.accept();
                    client.configureBlocking(false);
                    client.register(selector, SelectionKey.OP_READ);
                    System.out.println("New client connected.");
                } else if (key.isReadable()) {
                    // Read data from client
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(256);
                    int bytesRead = client.read(buffer);
                    if (bytesRead == -1) {
                        client.close();
                        System.out.println("Client disconnected.");
                    } else {
                        String msg = new String(buffer.array()).trim();
                        System.out.println("Received: " + msg);
                    }
                }

                // Remove processed key to avoid reprocessing
                iter.remove();
            }
        }
    }
}
```

7.1.7 Explanation of the Code:

- A `Selector` is created to monitor events.

- A `ServerSocketChannel` is configured for non-blocking mode and registered with the selector for `OP_ACCEPT`.
- In the event loop, `select()` blocks until one or more channels are ready.
- `SelectionKey.isAcceptable()` detects new incoming connections.
- `SelectionKey.isReadable()` handles incoming data from clients.
- Each client `SocketChannel` is also set to non-blocking mode and registered with the selector.

7.1.8 Benefits of Using Selectors

- **Scalability**: A single thread can manage thousands of connections.
- **Efficiency**: Reduces thread context switching and memory usage.
- **Flexibility**: Fine-grained control over how IO is handled for each channel.
- **Asynchronous Behavior**: Applications can perform other tasks while waiting for IO readiness.

7.1.9 Summary

- A **Selector** allows a single thread to monitor multiple channels for IO readiness.
- Channels must be in **non-blocking** mode to work with selectors.
- Events like `OP_ACCEPT`, `OP_READ`, and `OP_WRITE` allow you to handle different types of IO operations.
- **SelectionKeys** track the readiness and registration status of channels.
- This model is ideal for building **high-performance servers** such as chat apps, HTTP servers, proxies, or real-time data processors.

Selectors enable event-driven IO models that are **scalable**, **efficient**, and **well-suited** for modern networked applications.

7.2 Registering Channels with Selectors

One of the most powerful features of Java NIO is its support for **non-blocking IO** using **selectors**. Selectors allow a single thread to manage multiple IO channels efficiently. To use this mechanism, channels must first be registered with a `Selector`. This section explains how registration works, the meaning of selection operations like `OP_READ` and `OP_WRITE`, and how to configure interest sets to monitor specific IO events.

7.2.1 Selectable Channels in Java NIO

Not all channels in Java NIO are selectable. Only those that implement the `SelectableChannel` interface can be registered with a `Selector`. The key channel types that support this include:

- `SocketChannel` – for client TCP connections.
- `ServerSocketChannel` – for server-side listening sockets.
- `DatagramChannel` – for UDP-based communication.
- `Pipe.SourceChannel` and `Pipe.SinkChannel` – for inter-thread communication.

All these channels must be configured to **non-blocking mode** before being registered with a selector.

7.2.2 What Happens During Registration?

When you register a channel with a selector, you're asking the selector to watch that channel for specific events, such as when it's ready to read data or accept a new connection.

This is done using the channel's `register()` method:

```
SelectionKey key = channel.register(selector, ops);
```

- `channel` – A `SelectableChannel` such as `SocketChannel`.
- `selector` – The `Selector` instance that will monitor this channel.
- `ops` – A set of interest operations (like `OP_READ` or `OP_WRITE`) the channel wants to be notified about.
- `key` – A `SelectionKey` object representing this registration.

7.2.3 Selection Operations (Interest Ops)

When registering a channel, you specify the **interest set** — a bitmask that tells the selector which operations to monitor on the channel.

The selection operations include:

Constant	Description
`SelectionKey.OP_ACCEPT`	Channel is ready to accept a new connection (for `ServerSocketChannel`)
`SelectionKey.OP_CONNECT`	Channel has completed connection process (for `SocketChannel`)
`SelectionKey.OP_READ`	Channel is ready for reading data
`SelectionKey.OP_WRITE`	Channel is ready for writing data

These constants are bit flags and can be **combined** using the bitwise OR (`|`) operator.

Example:

```
int ops = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
channel.register(selector, ops);
```

This tells the selector to notify us when the channel is either ready to read **or** write.

7.2.4 Configuring a Channel and Registering with a Selector

Before registration, the channel **must** be configured as **non-blocking**:

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.configureBlocking(false);
```

Then it can be registered:

```
Selector selector = Selector.open();
socketChannel.register(selector, SelectionKey.OP_CONNECT);
```

7.2.5 Code Example: Registering Multiple Channels

Here's a complete example showing how to register `ServerSocketChannel` and accept incoming connections, then register each new `SocketChannel` for reading:

Full runnable code:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.Iterator;
import java.util.Set;

public class ChannelRegistrationExample {
    public static void main(String[] args) throws IOException {
        // Step 1: Create a selector
        Selector selector = Selector.open();

        // Step 2: Create a non-blocking ServerSocketChannel
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.bind(new InetSocketAddress(4000));
        serverChannel.configureBlocking(false);

        // Step 3: Register server channel with selector for ACCEPT operation
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);

        System.out.println("Server listening on port 4000...");
```

```
        // Step 4: Event loop
        while (true) {
            selector.select(); // Block until at least one channel is ready

            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> iter = selectedKeys.iterator();

            while (iter.hasNext()) {
                SelectionKey key = iter.next();

                if (key.isAcceptable()) {
                    // Accept connection
                    ServerSocketChannel server = (ServerSocketChannel) key.channel();
                    SocketChannel client = server.accept();
                    client.configureBlocking(false);

                    // Register client for READ operation
                    client.register(selector, SelectionKey.OP_READ);
                    System.out.println("Accepted new client connection.");
                } else if (key.isReadable()) {
                    // Read data from client
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(256);
                    int bytesRead = client.read(buffer);

                    if (bytesRead == -1) {
                        client.close(); // Client closed connection
                        System.out.println("Client disconnected.");
                    } else {
                        buffer.flip();
                        byte[] data = new byte[buffer.remaining()];
                        buffer.get(data);
                        System.out.println("Received: " + new String(data));
                    }
                }

                iter.remove(); // Remove processed key
            }
        }
    }
}
```

7.2.6 Explanation of the Code

- **ServerSocketChannel** is registered for `OP_ACCEPT`, allowing the selector to notify when a client attempts to connect.
- On accepting a connection, the **SocketChannel** is configured to non-blocking mode and registered with the selector for `OP_READ`.
- The selector monitors both the server and client channels, dispatching events accordingly.
- Only one thread handles all the connections efficiently.

7.2.7 The SelectionKey Object

When a channel is registered, a `SelectionKey` is returned. This object provides:

- `channel()` – The registered channel.
- `selector()` – The selector managing the key.
- `interestOps()` – The set of operations the key is interested in.
- `readyOps()` – The set of operations the channel is ready for.

You can also **attach objects** to the key (e.g., buffers or session info):

```
SelectionKey key = clientChannel.register(selector, SelectionKey.OP_READ);
key.attach(ByteBuffer.allocate(1024));
```

Later, you can retrieve the attachment:

```
ByteBuffer buffer = (ByteBuffer) key.attachment();
```

7.2.8 Summary

- Channels must be in **non-blocking** mode to register with a `Selector`.
- The **`register()`** method binds a channel to a selector with a specified interest set (`OP_READ`, `OP_WRITE`, etc.).
- You can register **multiple channels** with one selector to efficiently manage multiple IO operations.
- **`SelectionKey`** represents each registration and provides APIs to inspect readiness and attach metadata.

Using selectors and channel registration together forms the backbone of scalable, non-blocking network servers in Java.

7.3 Handling SelectionKeys and Events

In Java NIO's non-blocking IO system, the `Selector` class works closely with the `SelectionKey` class to monitor and handle IO readiness events across multiple channels. Understanding how to work with `SelectionKey` objects is essential for building efficient and scalable applications.

This section explores what `SelectionKey` represents, the types of events it tracks, and how to process these keys during event-driven IO operations.

7.3.1 What is a `SelectionKey`?

A `SelectionKey` represents the registration of a **channel** with a **selector**. It is created when a `SelectableChannel` (such as a `SocketChannel`) is registered to a `Selector` using the `register()` method:

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

This key maintains information about:

- The **channel** it is associated with
- The **selector** managing it
- The **interest set** (what events the application is interested in)
- The **ready set** (what events the channel is ready for)
- Optional **attachment** (a user-defined object for tracking context)

7.3.2 Selection Operations (Event Types)

The `SelectionKey` class defines four constants representing IO readiness events:

Constant	Meaning
`OP_ACCEPT`	Ready to accept a new incoming connection (server socket)
`OP_CONNECT`	A non-blocking connection has finished establishing
`OP_READ`	Channel has data available to read
`OP_WRITE`	Channel is ready to accept data for writing

These are referred to as **interest ops** when registering the channel and as **ready ops** when the event has occurred.

7.3.3 Inspecting and Handling Events

Once the `Selector.select()` method is called and returns, the `selectedKeys()` method gives you the set of keys for channels that are ready.

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> iterator = selectedKeys.iterator();
```

You loop through this set to handle events:

```
while (iterator.hasNext()) {
    SelectionKey key = iterator.next();

    if (key.isAcceptable()) {
        // Handle new connection
```

```
    } else if (key.isConnectable()) {
        // Handle client connection finish
    } else if (key.isReadable()) {
        // Handle read from client
    } else if (key.isWritable()) {
        // Handle write to client
    }

    iterator.remove(); // Important: remove the processed key
}
```

7.3.4 Example: Full Event Handling Logic

Here's a complete code example of a simple server handling accept and read events using `SelectionKey`.

Full runnable code:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.Iterator;
import java.util.Set;

public class SelectionKeyExample {
    public static void main(String[] args) throws IOException {
        Selector selector = Selector.open();

        // Create server channel and register for accept
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.bind(new InetSocketAddress(5000));
        serverChannel.configureBlocking(false);
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
        System.out.println("Server started on port 5000.");

        while (true) {
            selector.select(); // Wait for events
            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> iter = selectedKeys.iterator();

            while (iter.hasNext()) {
                SelectionKey key = iter.next();

                if (key.isAcceptable()) {
                    ServerSocketChannel server = (ServerSocketChannel) key.channel();
                    SocketChannel client = server.accept();
                    client.configureBlocking(false);
                    client.register(selector, SelectionKey.OP_READ);
                    System.out.println("Accepted new client.");
                }

                else if (key.isReadable()) {
```

```
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(256);
                    int bytesRead = client.read(buffer);

                    if (bytesRead == -1) {
                        client.close();
                        System.out.println("Client disconnected.");
                    } else {
                        buffer.flip();
                        byte[] data = new byte[buffer.remaining()];
                        buffer.get(data);
                        System.out.println("Received: " + new String(data));
                    }
                }

                iter.remove(); // Important to avoid reprocessing
            }
        }
    }
}
```

7.3.5 Attaching Context with `SelectionKey`

You can store additional context (such as a buffer or session object) using the `attach()` method when registering the channel:

```
SelectionKey key = clientChannel.register(selector, SelectionKey.OP_READ);
key.attach(ByteBuffer.allocate(1024)); // Attach a buffer
```

Later, you can retrieve it during event handling:

```
ByteBuffer buffer = (ByteBuffer) key.attachment();
```

This approach is useful when each client needs its own data buffer or state tracker.

7.3.6 Best Practices for Using `SelectionKey`

- **Always remove keys after processing** using `iterator.remove()` to avoid handling them again.
- **Use attachments** to store per-channel data such as buffers or user sessions.
- **Check for multiple readiness states**: A key may be ready for more than one operation at a time. Always use `if-else` or `switch` to check each.
- **Gracefully handle closed connections**: When `read()` returns `-1`, it means the client has disconnected. Always close the channel.

7.3.7 Summary

- `SelectionKey` objects represent the link between a channel and a selector.
- They track what events the channel is interested in and what it is currently ready for.
- You retrieve and process these keys from the `Selector` using `selectedKeys()`.
- Events like `OP_ACCEPT`, `OP_READ`, and `OP_WRITE` allow efficient, event-driven IO.
- You can attach custom objects to a key to manage per-channel context.
- Proper handling of `SelectionKey` objects is essential for building scalable non-blocking servers.

By mastering how to work with `SelectionKey`, you unlock the full power of Java NIO selectors and can build high-performance applications with minimal thread usage.

7.4 Building a Simple Non-blocking Server

In this tutorial, we'll build a **simple non-blocking TCP server** that:

- Listens for client connections
- Reads messages from connected clients
- Echoes messages back to them (echo server)
- Handles multiple clients using **one thread**

7.4.1 Step 1: Imports and Setup

We'll begin by importing the necessary Java NIO classes:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.Iterator;
import java.util.Set;
```

7.4.2 Step 2: Create and Configure the Server

We need a `ServerSocketChannel` and a `Selector`. The server channel must be non-blocking and registered with the selector to watch for **OP_ACCEPT** events (ready to accept new connections).

```
public class NonBlockingTCPServer {
    public static void main(String[] args) {
        try {
```

```
            // 1. Open a selector
            Selector selector = Selector.open();

            // 2. Open a server socket channel
            ServerSocketChannel serverChannel = ServerSocketChannel.open();
            serverChannel.bind(new InetSocketAddress(5000));
            serverChannel.configureBlocking(false); // Non-blocking mode

            // 3. Register the server channel with selector for ACCEPT operations
            serverChannel.register(selector, SelectionKey.OP_ACCEPT);
            System.out.println("Server listening on port 5000...");

            // 4. Event loop
            while (true) {
                selector.select(); // Blocking call - waits for events

                // 5. Get the set of keys representing ready channels
                Set<SelectionKey> selectedKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectedKeys.iterator();

                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();

                    // 6. Acceptable event (new client connection)
                    if (key.isAcceptable()) {
                        handleAccept(key, selector);
                    }

                    // 7. Readable event (client sent data)
                    else if (key.isReadable()) {
                        handleRead(key);
                    }

                    // Remove the key from the set to avoid processing again
                    iterator.remove();
                }
            }

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

7.4.3 Step 3: Handle New Client Connections

When a client attempts to connect, the server channel becomes "acceptable". We then accept the connection and register the new client channel for **OP_READ** (read readiness).

```
    private static void handleAccept(SelectionKey key, Selector selector) throws IOException {
        ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
        SocketChannel clientChannel = serverChannel.accept(); // Accept client
        clientChannel.configureBlocking(false);

        // Register client channel for READ events
```

```
        clientChannel.register(selector, SelectionKey.OP_READ);

        System.out.println("New client connected from " + clientChannel.getRemoteAddress());
    }
```

7.4.4 Step 4: Handle Incoming Data

When a client sends data, the channel becomes "readable". We read the data from the channel using a `ByteBuffer`, and in this example, we **echo** the data back to the client.

```
    private static void handleRead(SelectionKey key) throws IOException {
        SocketChannel clientChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        int bytesRead = -1;

        try {
            bytesRead = clientChannel.read(buffer);
        } catch (IOException e) {
            System.out.println("Client forcibly closed the connection.");
            clientChannel.close();
            key.cancel();
            return;
        }

        if (bytesRead == -1) {
            // Client closed the connection cleanly
            System.out.println("Client disconnected.");
            clientChannel.close();
            key.cancel();
            return;
        }

        // Echo back the received message
        buffer.flip(); // Prepare buffer for reading
        String received = new String(buffer.array(), 0, buffer.limit());
        System.out.println("Received: " + received.trim());

        // Echo it back
        clientChannel.write(buffer); // Buffer still in read mode
    }
}
```

Full runnable code:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.Iterator;
import java.util.Set;

public class NonBlockingTCPServer {
    public static void main(String[] args) {
```

```
        try {
            // 1. Open a selector
            Selector selector = Selector.open();

            // 2. Open a server socket channel
            ServerSocketChannel serverChannel = ServerSocketChannel.open();
            serverChannel.bind(new InetSocketAddress(5000));
            serverChannel.configureBlocking(false); // Non-blocking mode

            // 3. Register the server channel with selector for ACCEPT operations
            serverChannel.register(selector, SelectionKey.OP_ACCEPT);
            System.out.println("Server listening on port 5000...");

            // 4. Event loop
            while (true) {
                selector.select(); // Blocking call - waits for events

                Set<SelectionKey> selectedKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectedKeys.iterator();

                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();

                    if (key.isAcceptable()) {
                        handleAccept(key, selector);
                    } else if (key.isReadable()) {
                        handleRead(key);
                    }

                    iterator.remove(); // Prevent processing the same key again
                }
            }

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

    private static void handleAccept(SelectionKey key, Selector selector) throws IOException {
        ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
        SocketChannel clientChannel = serverChannel.accept();
        clientChannel.configureBlocking(false);
        clientChannel.register(selector, SelectionKey.OP_READ);
        System.out.println("New client connected from " + clientChannel.getRemoteAddress());
    }

    private static void handleRead(SelectionKey key) throws IOException {
        SocketChannel clientChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        int bytesRead;

        try {
            bytesRead = clientChannel.read(buffer);
        } catch (IOException e) {
            System.out.println("Client forcibly closed the connection.");
            clientChannel.close();
            key.cancel();
            return;
```

```
        }

        if (bytesRead == -1) {
            System.out.println("Client disconnected.");
            clientChannel.close();
            key.cancel();
            return;
        }

        buffer.flip();
        String received = new String(buffer.array(), 0, buffer.limit());
        System.out.println("Received: " + received.trim());

        clientChannel.write(buffer); // Echo back
    }
}
```

7.4.5 How It Works

- The **selector** waits for channels to become ready.
- When a channel is **ready to accept**, we accept the connection and register the new socket.
- When a client sends data, the channel becomes **readable**, and we read the message.
- After reading, we **write it back**, effectively creating an echo server.

7.4.6 Testing the Server

You can test the server using multiple `telnet` sessions:

```
telnet localhost 5000
```

Type a message and press Enter. The server will echo the message back to you.

7.4.7 Benefits of Non-blocking NIO

- **Single thread handles many connections**: Unlike traditional blocking IO, we don't need one thread per connection.
- **Low memory and CPU usage**: Threads are expensive. Fewer threads = less overhead.
- **Highly scalable**: Ideal for chat servers, proxies, and real-time apps.

7.4.8 Limitations and Enhancements

This server is a good starting point, but there are areas for improvement:

- Add write buffering and support for partial writes.
- Add a command protocol for structured communication.
- Use attachments (`SelectionKey.attach()`) to store session state.
- Support concurrent reads/writes using a thread pool for intensive tasks.

7.4.9 Summary

In this tutorial, you've learned how to:

- Set up a **non-blocking TCP server** using Java NIO
- Use a **Selector** to handle multiple client connections in one thread
- Respond to **OP_ACCEPT** and **OP_READ** events
- Echo data back to clients efficiently

Java NIO selectors are a powerful tool for building scalable network applications. With just a bit more work, this basic server can evolve into a production-grade framework.

7.5 Performance Considerations

Java NIO's selector-based non-blocking IO model is designed for high performance and scalability. Unlike traditional IO, which dedicates a thread per connection, NIO allows a **single thread** to handle **thousands of connections** by multiplexing readiness events across channels. While this design provides significant performance benefits, achieving optimal efficiency requires careful attention to several areas: **threading, resource management, event handling**, and **common pitfalls**.

In this section, we'll explore these performance considerations in depth and provide actionable best practices for optimizing selector-based applications.

7.5.1 Scalability Through Fewer Threads

A primary advantage of non-blocking IO is the ability to support **many concurrent connections** using **a small number of threads**. Since a selector can monitor thousands of channels, the server can scale horizontally without spawning a thread per client.

Why It Matters:

- Traditional blocking IO uses one thread per connection, which doesn't scale well under heavy load.
- Each thread consumes memory (stack space) and CPU time for context switching.
- NIO avoids these costs by handling IO events only when data is ready.

Best Practice:

- Use a single **Selector thread** to handle network IO.
- Offload CPU-intensive or blocking operations (like database access) to a **separate thread pool**.

7.5.2 Thread Management and Design Patterns

To avoid performance bottlenecks, organize your application into logical thread roles:

- **Selector Thread**: Handles accept/read/write events and dispatches them.
- **Worker Threads**: Process data, execute business logic, or perform blocking tasks.

This pattern ensures the selector loop remains responsive and doesn't block on operations like disk IO or long-running tasks.

```
[Selector Thread] → [Worker Pool] → [Processing Logic]
```

Best Practice:

- Keep the selector thread fast and non-blocking.
- Use `ExecutorService` to manage a pool of workers for background processing.

7.5.3 Minimizing Memory Allocation and Garbage Collection

Non-blocking servers can experience **frequent memory allocation** from buffers and temporary objects, which leads to **garbage collection (GC)** pressure. GC pauses can introduce latency and jitter.

Tips to reduce GC overhead:

- **Reuse `ByteBuffer` objects**: Allocate buffers once and reuse them per connection.
- **Use attachments on `SelectionKey`** to store buffer and session objects.
- Avoid per-request object creation in the selector loop.

Example:

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
key.attach(buffer); // Reuse per client
```

7.5.4 Efficient Buffer and Channel Management

Efficient use of `ByteBuffer` is critical for performance. Understand buffer methods like `clear()`, `flip()`, `compact()` to avoid redundant copying or unnecessary allocations.

Best Practice:

- Use **direct buffers** (`ByteBuffer.allocateDirect()`) for large or long-lived buffers.
- Use **heap buffers** (`ByteBuffer.allocate()`) for short-lived or small objects where GC is negligible.

Caution: Direct buffers avoid heap GC but are more expensive to allocate and harder to monitor. Use them judiciously.

7.5.5 Selector Wakeup Overhead

A common pitfall is **unnecessary wakeups** of the selector, which can degrade performance, especially under high load.

Avoid:

- Calling `selector.wakeup()` too frequently.
- Waking the selector from multiple threads unless absolutely required.

Best Practice:

- Use a **single-threaded event loop** when possible.
- When you need to register channels from other threads, use a **task queue** and call `wakeup()` only when needed.

7.5.6 Handling Write Readiness Properly

Another common pitfall is treating `OP_WRITE` like `OP_READ`. Unlike read events, **write events are always ready** unless the buffer is full. Constantly registering for write events can cause busy loops and CPU spikes.

Best Practice:

- Only register `OP_WRITE` when you have actual data to write.
- Remove `OP_WRITE` interest once all data is written.

```
if (buffer.hasRemaining()) {
    key.interestOps(key.interestOps() | SelectionKey.OP_WRITE);
} else {
    key.interestOps(key.interestOps() & ~SelectionKey.OP_WRITE);
}
```

7.5.7 Avoid Blocking Operations in Event Loop

Blocking the selector thread (e.g., by reading from disk, calling `Thread.sleep()`, or synchronizing on shared objects) severely reduces responsiveness.

Best Practice:

- Offload slow or blocking operations to a **dedicated thread pool**.
- Never call blocking methods like `readLine()` or database queries in the selector thread.

7.5.8 Monitoring and Profiling

To tune performance, you must **observe your application** under load:

- Use profilers (VisualVM, JFR) to track GC, thread usage, and heap allocations.
- Log channel lifecycle events (connect, read, write, disconnect) for debugging.
- Instrument IO throughput and latency.

7.5.9 Use Selectors Correctly

NIO selectors are powerful but fragile. Misuse can lead to **stale keys**, **dropped connections**, or **busy spinning**.

Common Issues:

- Not calling `iterator.remove()` causes keys to be reprocessed.
- Not handling `IOException` can leave channels in an inconsistent state.
- Forgetting to close channels leads to resource leaks.

Best Practice:

- Always `remove()` keys after processing.
- Properly close and cancel keys on error.
- Use try-catch around all IO operations.

7.5.10 Summary

Using selectors and non-blocking IO in Java NIO can yield high-performance, scalable servers when used correctly. Key takeaways include:

- Use minimal threads with clear roles.
- Reuse buffers and avoid memory churn.

- Register `OP_WRITE` only when needed.
- Keep selector threads free of blocking calls.
- Profile and monitor your server to find bottlenecks.

By adhering to these practices, you can build NIO-based applications that serve thousands of clients efficiently while minimizing CPU, memory, and thread usage.

Below is a **complete example of a multi-threaded non-blocking TCP server** using Java NIO with:

- A **single selector thread** (for accepting and reading client connections)
- A **worker thread pool** (for offloading business logic)
- **ByteBuffer reuse via SelectionKey attachments**
- **Proper handling of `OP_WRITE` readiness**
- **Thread-safe task submission using a queue**

7.5.11 Features Demonstrated

- Non-blocking server with `Selector`
- `ExecutorService` for background processing
- Channel registration from multiple threads using `Selector.wakeup()`
- Efficient buffer reuse and write registration logic

7.5.12 Full Java Example

Full runnable code:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.Iterator;
import java.util.Set;
import java.util.concurrent.*;

public class MultiThreadedNIOServer {
    private static final int PORT = 5000;
    private static final int BUFFER_SIZE = 1024;

    private final Selector selector;
    private final ServerSocketChannel serverChannel;
    private final ExecutorService workerPool;
    private final ConcurrentLinkedQueue<Runnable> pendingTasks;

    public MultiThreadedNIOServer() throws IOException {
        this.selector = Selector.open();
```

```
        this.serverChannel = ServerSocketChannel.open();
        this.workerPool = Executors.newFixedThreadPool(4); // Worker thread pool
        this.pendingTasks = new ConcurrentLinkedQueue<>();

        serverChannel.bind(new InetSocketAddress(PORT));
        serverChannel.configureBlocking(false);
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    }

    public void start() throws IOException {
        System.out.println("Server started on port " + PORT);

        while (true) {
            // Execute any tasks added from other threads
            Runnable task;
            while ((task = pendingTasks.poll()) != null) {
                task.run();
            }

            selector.select();

            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> iter = selectedKeys.iterator();

            while (iter.hasNext()) {
                SelectionKey key = iter.next();

                try {
                    if (key.isAcceptable()) {
                        handleAccept(key);
                    } else if (key.isReadable()) {
                        handleRead(key);
                    } else if (key.isWritable()) {
                        handleWrite(key);
                    }
                } catch (IOException e) {
                    System.err.println("Closing broken connection: " + e.getMessage());
                    closeKey(key);
                }

                iter.remove();
            }
        }
    }

    private void handleAccept(SelectionKey key) throws IOException {
        ServerSocketChannel server = (ServerSocketChannel) key.channel();
        SocketChannel client = server.accept();
        client.configureBlocking(false);

        // Attach a buffer for each client connection
        ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
        client.register(selector, SelectionKey.OP_READ, buffer);

        System.out.println("Accepted connection from " + client.getRemoteAddress());
    }

    private void handleRead(SelectionKey key) throws IOException {
```

```
        SocketChannel client = (SocketChannel) key.channel();
        ByteBuffer buffer = (ByteBuffer) key.attachment();

        int bytesRead = client.read(buffer);

        if (bytesRead == -1) {
            closeKey(key);
            return;
        }

        buffer.flip();
        byte[] data = new byte[buffer.limit()];
        buffer.get(data);
        buffer.clear(); // Ready for next read

        String message = new String(data).trim();
        System.out.println("Received: " + message);

        // Offload processing to worker pool
        workerPool.submit(() -> {
            String response = "[Echo] " + message;
            ByteBuffer responseBuffer = ByteBuffer.wrap(response.getBytes());

            // Schedule write back in selector thread
            scheduleTask(() -> {
                key.attach(responseBuffer);
                key.interestOps(SelectionKey.OP_WRITE);
                selector.wakeup(); // Ensure selector notices change
            });
        });
    }

    private void handleWrite(SelectionKey key) throws IOException {
        SocketChannel client = (SocketChannel) key.channel();
        ByteBuffer buffer = (ByteBuffer) key.attachment();

        client.write(buffer);
        if (!buffer.hasRemaining()) {
            // Done writing; switch back to read
            ByteBuffer readBuffer = ByteBuffer.allocate(BUFFER_SIZE);
            key.attach(readBuffer);
            key.interestOps(SelectionKey.OP_READ);
        }
    }

    private void scheduleTask(Runnable task) {
        pendingTasks.add(task);
        selector.wakeup(); // Wake up selector to execute task
    }

    private void closeKey(SelectionKey key) {
        try {
            key.channel().close();
        } catch (IOException ignored) {}
        key.cancel();
    }

    public static void main(String[] args) throws IOException {
```

```
        new MultiThreadedNIOServer().start();
    }
}
```

7.5.13 How This Works

- **Selector Thread**: Accepts new clients and handles IO readiness events (`OP_READ`, `OP_WRITE`)
- **Worker Pool**: Handles processing of messages outside the selector thread
- **Attachments**: Each channel carries its own buffer, reused for read/write
- **Selector Wakeup**: Ensures safe registration and state changes from other threads
- **Pending Task Queue**: Thread-safe way to modify selector state from workers

7.5.14 Test the Server

Open multiple terminal tabs and run:

```
telnet localhost 5000
```

Each message sent will be echoed back with `[Echo]` prefix.

7.5.15 Best Practices Applied

- One thread handles thousands of connections (selector thread)
- Worker threads only process logic, keeping selector responsive
- Buffer reuse via attachments reduces GC pressure
- Selector wakeup ensures thread-safe channel updates

Chapter 8.

Asynchronous IO

1. Introduction to Asynchronous IO in Java
2. AsynchronousFileChannel
3. CompletionHandler and Future
4. Using AIO for Network Communication

8 Asynchronous IO

8.1 Introduction to Asynchronous IO in Java

Asynchronous IO (AIO) is a powerful programming model designed to improve **scalability**, **performance**, and **responsiveness** in applications that perform intensive input/output operations. Java introduced support for AIO in Java 7 as part of the `java.nio.channels` package, offering developers an alternative to blocking and non-blocking IO models.

In this section, we'll explore what asynchronous IO is, how it differs from traditional synchronous models, why it's useful in modern software systems, and how Java's AIO API fits into the larger IO ecosystem.

8.1.1 Understanding Asynchronous vs Synchronous IO

At a high level, the key distinction between **synchronous** and **asynchronous** IO lies in how **control is managed during an IO operation**.

Synchronous IO

In synchronous IO (used by both traditional IO and some forms of NIO), when a program initiates a read or write operation, it **waits** for the operation to complete before continuing. This behavior is simple and predictable but can lead to performance issues in high-concurrency scenarios.

```
// Traditional synchronous IO example
int bytesRead = inputStream.read(buffer);
```

In this case, the thread is **blocked** until data is available.

Asynchronous IO

In asynchronous IO, the program initiates an IO operation and immediately regains control. The actual work is performed **in the background**, and a **callback** or **future** is used to notify the program when the operation is complete.

```
// Pseudo-AIO concept
channel.read(buffer, attachment, completionHandler);
```

The calling thread does **not block**, enabling it to manage many other tasks while IO is performed by the OS or a background thread.

8.1.2 Motivation for Using Asynchronous IO

The primary motivation for AIO is to **increase concurrency without increasing the number of threads**. This is especially critical in scenarios like:

- **High-performance servers** handling thousands of simultaneous connections
- **GUI applications** that must remain responsive while performing background IO
- **Cloud services** that need to scale under unpredictable load
- **Network-intensive systems** that perform multiple IO operations concurrently

With AIO:

- You avoid **thread-per-connection** overhead
- Reduce **CPU time** spent in context switching
- Gain better **hardware utilization**, especially on multi-core systems

8.1.3 How AIO Improves Scalability and Responsiveness

Let's look at a typical server use case:

Synchronous Server:

- Each client connection is handled by a dedicated thread.
- 1000 clients = 1000 threads.
- Results in increased memory consumption and context switching overhead.

Asynchronous Server:

- A small number of threads initiate and manage thousands of IO operations.
- IO is delegated to the OS or a thread pool and only resumes when necessary.
- The server is highly **scalable**, lightweight, and responsive.

GUI Applications:

In GUI environments like JavaFX or Swing, blocking the main UI thread can cause freezing or lag. AIO allows data to be read or written **in the background**, keeping the UI fluid and interactive.

8.1.4 Overview of Javas AIO API

Java introduced AIO support in Java 7 through the `java.nio.channels` package. The primary interfaces and classes include:

`AsynchronousChannel`

The base interface for channels supporting asynchronous operations. Two main implementations exist:

- `AsynchronousSocketChannel` (for TCP network IO)
- `AsynchronousFileChannel` (for file IO)

`CompletionHandlerV, A`

A callback interface for handling the result of an asynchronous operation.

```
channel.read(buffer, attachment, new CompletionHandler<Integer, Object>() {
    @Override
    public void completed(Integer result, Object attachment) {
        // Handle success
    }

    @Override
    public void failed(Throwable exc, Object attachment) {
        // Handle error
    }
});
```

`FutureV`

Alternatively, AIO methods can return a `Future<V>` that represents the result of the operation. This allows for polling or blocking to retrieve results.

```
Future<Integer> future = channel.read(buffer);
while (!future.isDone()) {
    // do something else
}
int bytesRead = future.get(); // blocks if not done
```

Thread Pool Backing

Asynchronous channels often rely on an **underlying thread pool**, which can be:

- The system default
- Custom-provided via `AsynchronousChannelGroup`

This allows flexibility in managing IO operation execution.

8.1.5 Where AIO Fits in the Java IO Ecosystem

Java now offers multiple IO paradigms:

Model	API	Use Case
Blocking IO	`InputStream`, `Reader`	Simple applications, low concurrency

Model	API	Use Case
Non-blocking IO	`SocketChannel`, `Selector`	Scalable servers, requires manual control
Asynchronous IO	`Asynchronous*Channel`	Highly scalable, callback/future-based

AIO is ideal when:

- You need to perform multiple simultaneous IO operations without blocking.
- You want to offload IO latency without writing a full selector/event loop manually.
- You prefer a **callback-driven or future-based programming style**.

8.1.6 Real-world Use Cases for AIO

- **Web Servers** (e.g., asynchronous HTTP or WebSocket handlers)
- **Chat Applications** with many idle connections
- **File Watchers or Backup Systems** reading/writing large files in the background
- **Microservices** that fetch data from multiple services concurrently

8.1.7 Summary

Asynchronous IO in Java provides a modern, scalable solution for applications that demand high performance and low latency under concurrent loads. By decoupling IO operations from thread blocking, AIO enables developers to handle massive workloads using a small, efficient thread pool. Java's AIO API, introduced in Java 7, is a natural complement to traditional and non-blocking IO and plays a crucial role in building responsive and scalable Java applications today.

In the next section, we'll explore how to use `AsynchronousFileChannel` for asynchronous file reading and writing in real-world scenarios.

8.2 AsynchronousFileChannel

Java NIO.2, introduced in Java 7, brought powerful file I/O enhancements, among them the `AsynchronousFileChannel` class. This class enables **non-blocking, asynchronous file operations**, allowing developers to read from and write to files without stalling the executing thread. It is part of the `java.nio.channels` package and leverages the underlying operating system's asynchronous I/O capabilities where available.

8.2.1 Overview

Traditional file I/O in Java—whether through `java.io` or even the `FileChannel` class in the original NIO—tends to be **blocking**. This means that if a thread starts a read or write operation, it must wait for that operation to complete before doing anything else. In contrast, `AsynchronousFileChannel` allows I/O operations to be executed in the background, enabling the thread to continue with other tasks or respond to I/O completion events via callbacks.

This capability is particularly valuable in high-performance, scalable applications such as web servers, file processors, and database engines.

8.2.2 Creating an AsynchronousFileChannel

To use `AsynchronousFileChannel`, you typically open a file with the appropriate read/write permissions and optionally provide an `ExecutorService` for managing asynchronous tasks.

```
Path path = Paths.get("example.txt");

// Open for asynchronous writing
AsynchronousFileChannel channel = AsynchronousFileChannel.open(
    path,
    StandardOpenOption.WRITE,
    StandardOpenOption.CREATE
);
```

You can also provide a custom thread pool:

```
ExecutorService executor = Executors.newFixedThreadPool(2);

AsynchronousFileChannel channel = AsynchronousFileChannel.open(
    path,
    EnumSet.of(StandardOpenOption.READ, StandardOpenOption.WRITE),
    executor
);
```

8.2.3 Key Methods

- `read(ByteBuffer dst, long position, A attachment, CompletionHandlerInteger, ? super A handler)` Reads bytes from the file into the given buffer starting at a given file position. The operation is non-blocking and handled by a `CompletionHandler`.
- `write(ByteBuffer src, long position, A attachment, CompletionHandlerInteger, ? super A handler)` Writes bytes from the buffer into the file starting at the given position, using a completion handler for notification.
- `FutureInteger read(ByteBuffer dst, long position)` Starts an asynchronous read operation and returns a `Future`, which can be queried or blocked on.

- `FutureInteger write(ByteBuffer src, long position)` Initiates an asynchronous write and returns a `Future`.

8.2.4 Asynchronous Read Example

Below is a complete example demonstrating how to asynchronously read data from a file using a `CompletionHandler`:

Full runnable code:

```
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.util.concurrent.*;

public class AsyncFileRead {
    public static void main(String[] args) {
        try {
            Path path = Paths.get("input.txt");
            AsynchronousFileChannel channel = AsynchronousFileChannel.open(path, StandardOpenOption.REA

            ByteBuffer buffer = ByteBuffer.allocate(1024);
            long position = 0;

            channel.read(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {
                @Override
                public void completed(Integer result, ByteBuffer attachment) {
                    System.out.println("Read completed: " + result + " bytes");
                    attachment.flip();
                    byte[] data = new byte[attachment.limit()];
                    attachment.get(data);
                    System.out.println("Data: " + new String(data));
                }

                @Override
                public void failed(Throwable exc, ByteBuffer attachment) {
                    System.err.println("Read failed");
                    exc.printStackTrace();
                }
            });

            // Let the async read complete
            Thread.sleep(1000);

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

8.2.5 Asynchronous Write Example

Here's how you can asynchronously write data to a file:

Full runnable code:

```
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.util.concurrent.*;

public class AsyncFileWrite {
    public static void main(String[] args) {
        try {
            Path path = Paths.get("output.txt");
            AsynchronousFileChannel channel = AsynchronousFileChannel.open(
                path, StandardOpenOption.WRITE, StandardOpenOption.CREATE
            );

            ByteBuffer buffer = ByteBuffer.wrap("Hello, asynchronous world!".getBytes());
            long position = 0;

            channel.write(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {
                @Override
                public void completed(Integer result, ByteBuffer attachment) {
                    System.out.println("Write completed: " + result + " bytes");
                }

                @Override
                public void failed(Throwable exc, ByteBuffer attachment) {
                    System.err.println("Write failed");
                    exc.printStackTrace();
                }
            });

            // Wait for async write to complete
            Thread.sleep(1000);

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

8.2.6 Comparison to Traditional FileChannel

Feature	`FileChannel`	`AsynchronousFileChannel`
Blocking behavior	Blocking	Non-blocking
Thread responsiveness	Limited	High

Feature	`FileChannel`	`AsynchronousFileChannel`
Scalability	Lower (one thread per I/O)	Higher (event-driven, fewer threads)
Use case suitability	Simple, synchronous I/O	High-performance, async applications
Completion notification	None (call returns on completion)	Via `Future` or `CompletionHandler`

8.2.7 Use Cases

- File servers handling many concurrent requests
- Applications with a non-blocking I/O architecture (e.g., Netty)
- Log processors writing data asynchronously
- Background tasks that should not block the main thread

8.2.8 Conclusion

The `AsynchronousFileChannel` class provides a powerful mechanism for performing file I/O operations without blocking the thread. It is especially beneficial in high-concurrency environments where efficient use of threads is critical. By utilizing Java's asynchronous I/O APIs, developers can build more responsive and scalable applications. Whether through `Future` objects or `CompletionHandler` callbacks, `AsynchronousFileChannel` provides flexible options for integrating asynchronous file access into Java programs.

8.3 CompletionHandler and Future

Java's asynchronous IO (NIO.2), introduced in Java 7, enables non-blocking I/O operations that allow a program to continue executing other tasks while waiting for potentially slow IO processes—like reading from or writing to files or network sockets—to complete. Two core mechanisms are provided to handle the completion of these asynchronous operations: the **callback-based** approach using the `CompletionHandler` interface and the **future-based** approach using the `Future` class. Understanding these two paradigms is essential for effectively working with Java's asynchronous IO API.

8.3.1 Roles in Handling Asynchronous Operation Completion

When an asynchronous operation is initiated—such as reading from a file—Java returns control immediately to the caller, allowing the current thread to do other work instead of blocking. However, the program must still handle the result or status of the IO operation once it completes.

Java provides two main ways to handle this:

1. **Callback-based Handling with `CompletionHandler`** The `CompletionHandler` interface allows you to pass a callback object to the asynchronous IO method. This callback is notified when the operation completes (either successfully or with failure). This pattern fits well with event-driven programming and is very efficient for high-concurrency environments.
2. **Future-based Handling with `Future`** The asynchronous IO methods can also return a `Future` object. This object represents the result of the asynchronous computation and can be queried or blocked on to retrieve the result once ready. This model is closer to the traditional synchronous model but allows you to defer waiting on the result until it is needed.

8.3.2 The `CompletionHandler` Interface

The `CompletionHandler` interface resides in the `java.nio.channels` package and is defined as:

```
public interface CompletionHandler<V, A> {
    void completed(V result, A attachment);
    void failed(Throwable exc, A attachment);
}
```

- `V` represents the result type of the I/O operation, typically an `Integer` indicating the number of bytes read or written.
- `A` is an attachment object provided by the caller, useful for passing context or state into the callback.

8.3.3 How it Works

When you start an asynchronous operation (e.g., `AsynchronousFileChannel.read()`), you pass a `CompletionHandler` implementation that defines what should happen on:

- **`completed`**: Called when the operation completes successfully.
- **`failed`**: Called when the operation fails with an exception.

This callback approach is highly efficient because it does not require any thread blocking or

polling—your program simply reacts to events as they happen.

8.3.4 The `Future` Class

The `Future` interface (in `java.util.concurrent`) represents the result of an asynchronous computation. It provides methods such as:

- `get()`: Waits (blocks) if necessary for the computation to complete, then returns the result.
- `isDone()`: Checks if the computation has completed.
- `cancel()`: Attempts to cancel the operation.

8.3.5 How it Works

When you invoke an asynchronous IO operation that returns a `Future`, you receive a placeholder object immediately. You can then:

- Check periodically if the operation is done (polling).
- Block and wait for the operation result when needed.
- Cancel the operation if desired.

While easier to reason about, this model may introduce blocking and is generally less performant for large numbers of concurrent operations compared to callbacks.

8.3.6 Differences Between `CompletionHandler` and `Future`

Aspect	`CompletionHandler`	`Future`
Notification style	Event-driven callback	Polling or blocking to get the result
Thread blocking	No blocking; the callback runs on completion	May block if `get()` is called before completion
Complexity	Requires implementing callback methods	Simpler to use, similar to synchronous calls
Suitability	High concurrency, reactive/event-driven apps	Simpler tasks or when blocking is acceptable
Error handling	In `failed()` callback	Via exceptions thrown from `get()`
Cancellation support	Managed via `cancel()` on the channel/future	Direct cancellation on the `Future`

8.3.7 When to Use Each

- Use **`CompletionHandler`** if you want your application to remain fully non-blocking and responsive, especially in servers or GUI apps where waiting on IO is undesirable.
- Use **`Future`** when you prefer simpler control flow or when blocking for completion at some point in your logic is acceptable or easier.

8.3.8 Example 1: Using `CompletionHandler` for Asynchronous File Reading

Full runnable code:

```
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class AsyncReadWithCompletionHandler {
    public static void main(String[] args) throws Exception {
        Path file = Paths.get("example.txt");
        AsynchronousFileChannel channel = AsynchronousFileChannel.open(file, StandardOpenOption.READ);

        ByteBuffer buffer = ByteBuffer.allocate(1024);
        channel.read(buffer, 0, buffer, new CompletionHandler<Integer, ByteBuffer>() {
            @Override
            public void completed(Integer result, ByteBuffer attachment) {
                System.out.println("Read " + result + " bytes.");
                attachment.flip();
                byte[] data = new byte[attachment.limit()];
                attachment.get(data);
                System.out.println("Data: " + new String(data));
            }

            @Override
            public void failed(Throwable exc, ByteBuffer attachment) {
                System.err.println("Read failed:");
                exc.printStackTrace();
            }
        });

        // Keep main thread alive to allow async operation to complete
        Thread.sleep(1000);
        channel.close();
    }
}
```

8.3.9 Example 2: Using `Future` for Asynchronous File Writing

Full runnable code:

```java
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.util.concurrent.Future;

public class AsyncWriteWithFuture {
    public static void main(String[] args) throws Exception {
        Path file = Paths.get("output.txt");
        AsynchronousFileChannel channel = AsynchronousFileChannel.open(
            file,
            StandardOpenOption.WRITE, StandardOpenOption.CREATE
        );

        ByteBuffer buffer = ByteBuffer.wrap("Hello, Future!".getBytes());
        Future<Integer> writeResult = channel.write(buffer, 0);

        // Do some other work here if needed

        // Wait for completion and get the result
        int bytesWritten = writeResult.get();  // This call blocks until done
        System.out.println("Written bytes: " + bytesWritten);

        channel.close();
    }
}
```

8.3.10 Summary

- **`CompletionHandler`** enables **callback-driven**, fully non-blocking handling of asynchronous operations. It is well-suited for reactive programming models and high scalability.
- **`Future`** offers a simpler, **polling/blocking** model where you can check or wait for completion explicitly, at the expense of possible blocking.
- Understanding the differences allows you to select the right approach based on your application's concurrency and responsiveness needs.

By mastering both mechanisms, Java developers can efficiently handle asynchronous IO, improving performance and scalability in modern applications.

8.4 Using AIO for Network Communication

Java NIO.2 (introduced in Java 7) significantly enhanced Java's IO capabilities by introducing asynchronous IO (AIO) APIs, which allow non-blocking, event-driven network communication. Unlike traditional blocking IO where threads wait idly for operations to complete, asynchronous IO enables efficient, scalable networking by delegating operations to the operating system or a thread pool and notifying the application upon completion. This approach

reduces thread contention, improves resource utilization, and is ideal for high-performance network applications such as servers and clients handling many simultaneous connections.

8.4.1 Asynchronous Socket Channels in Java

The core class for asynchronous network communication in Java NIO.2 is `AsynchronousSocketChannel` for TCP/IP sockets. It represents a socket channel capable of non-blocking connect, read, and write operations.

There is also `AsynchronousServerSocketChannel` which is used for accepting incoming connections asynchronously.

Both classes reside in `java.nio.channels` and provide methods for starting asynchronous operations and handling their completion either via `CompletionHandler` callbacks or `Future` objects.

8.4.2 How AsynchronousSocketChannel Works

`AsynchronousSocketChannel` supports three main asynchronous operations:

1. **Connect**: Initiate a non-blocking connection to a remote server.
2. **Read**: Read data from the channel into a buffer without blocking.
3. **Write**: Write data from a buffer to the channel asynchronously.

Each operation returns immediately, and you can be notified when it completes via:

- A `CompletionHandler` — an event-driven callback interface.
- A `Future` — to block or poll for completion at your discretion.

Using `CompletionHandler` is the preferred idiomatic way for truly asynchronous, non-blocking networking.

8.4.3 Performing Non-blocking Connect, Read, and Write

Connecting

To establish a connection asynchronously:

```
AsynchronousSocketChannel socketChannel = AsynchronousSocketChannel.open();
socketChannel.connect(new InetSocketAddress("host", port), attachment, new CompletionHandler<Void, Atta
    @Override
    public void completed(Void result, AttachmentType attachment) {
        // Connection successful, proceed with reading or writing
    }
```

```
    @Override
    public void failed(Throwable exc, AttachmentType attachment) {
        // Handle connection failure
    }
});
```

8.4.4 Reading

After the connection is established, you can initiate asynchronous reads:

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
socketChannel.read(buffer, buffer, new CompletionHandler<Integer, ByteBuffer>() {
    @Override
    public void completed(Integer bytesRead, ByteBuffer buf) {
        if (bytesRead == -1) {
            // Channel closed by peer
            return;
        }
        buf.flip();
        // Process data in buffer here
        // Optionally start another read for continuous data
        buf.clear();
        socketChannel.read(buf, buf, this);
    }

    @Override
    public void failed(Throwable exc, ByteBuffer buf) {
        // Handle read failure
    }
});
```

8.4.5 Writing

Similarly, writing is done asynchronously:

```
ByteBuffer buffer = ByteBuffer.wrap("Hello server!".getBytes());
socketChannel.write(buffer, buffer, new CompletionHandler<Integer, ByteBuffer>() {
    @Override
    public void completed(Integer bytesWritten, ByteBuffer buf) {
        if (buf.hasRemaining()) {
            // Not all data was written, write the rest
            socketChannel.write(buf, buf, this);
        } else {
            // Write complete, proceed as needed
        }
    }

    @Override
    public void failed(Throwable exc, ByteBuffer buf) {
        // Handle write failure
```

```
    }
});
```

8.4.6 Event-driven Networking with `CompletionHandler`

The power of asynchronous IO is realized fully when integrated with `CompletionHandlers`. Each network operation passes a `CompletionHandler` implementation to handle success or failure events, enabling event-driven programming:

- When an operation completes, the JVM invokes the handler's `completed()` method with the result.
- If an error occurs, `failed()` is invoked with an exception.
- You can chain subsequent operations inside these callbacks for continuous, non-blocking IO workflows.

8.4.7 Sample Asynchronous TCP Echo Server

Below is an example of a simple asynchronous TCP echo server that accepts client connections and echoes back any received data.

Full runnable code:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;

public class AsyncEchoServer {
    public static void main(String[] args) throws IOException {
        int port = 5000;
        AsynchronousServerSocketChannel serverChannel =
            AsynchronousServerSocketChannel.open()
                .bind(new InetSocketAddress(port));

        System.out.println("Echo server listening on port " + port);

        serverChannel.accept(null, new CompletionHandler<AsynchronousSocketChannel, Void>() {
            @Override
            public void completed(AsynchronousSocketChannel clientChannel, Void att) {
                // Accept the next connection
                serverChannel.accept(null, this);

                // Handle client communication
                handleClient(clientChannel);
            }
```

```
            @Override
            public void failed(Throwable exc, Void att) {
                System.err.println("Failed to accept a connection");
                exc.printStackTrace();
            }
        });

        // Keep the server running
        try {
            Thread.currentThread().join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static void handleClient(AsynchronousSocketChannel clientChannel) {
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        clientChannel.read(buffer, buffer, new CompletionHandler<Integer, ByteBuffer>() {
            @Override
            public void completed(Integer bytesRead, ByteBuffer buf) {
                if (bytesRead == -1) {
                    // Client closed connection
                    try {
                        clientChannel.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                    return;
                }

                buf.flip();
                // Echo back the data
                clientChannel.write(buf, buf, new CompletionHandler<Integer, ByteBuffer>() {
                    @Override
                    public void completed(Integer bytesWritten, ByteBuffer buf) {
                        if (buf.hasRemaining()) {
                            clientChannel.write(buf, buf, this);
                        } else {
                            buf.clear();
                            // Read more data from client
                            clientChannel.read(buf, buf, this);
                        }
                    }

                    @Override
                    public void failed(Throwable exc, ByteBuffer buf) {
                        System.err.println("Write failed");
                        exc.printStackTrace();
                        try {
                            clientChannel.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                });
            }
```

```
            @Override
            public void failed(Throwable exc, ByteBuffer buf) {
                System.err.println("Read failed");
                exc.printStackTrace();
                try {
                    clientChannel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

How it works:

- The server opens an `AsynchronousServerSocketChannel` and binds it to a port.
- It calls `accept()` with a `CompletionHandler` to asynchronously wait for client connections.
- When a client connects, the handler is invoked. The server immediately calls `accept()` again to handle new incoming connections concurrently.
- The connected client is handled by the `handleClient()` method, which performs asynchronous reads.
- Upon receiving data, the server writes the same data back (echo), and once writing completes, it continues to read more data.
- All operations are non-blocking, and callbacks handle all I/O events.

8.4.8 Sample Asynchronous Client Using CompletionHandler

Full runnable code:

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.nio.charset.StandardCharsets;

public class AsyncEchoClient {
    public static void main(String[] args) throws Exception {
        AsynchronousSocketChannel clientChannel = AsynchronousSocketChannel.open();

        clientChannel.connect(new InetSocketAddress("localhost", 5000), null, new CompletionHandler<Voi
            @Override
            public void completed(Void result, Void attachment) {
                System.out.println("Connected to server");

                String message = "Hello, Asynchronous Server!";
                ByteBuffer buffer = ByteBuffer.wrap(message.getBytes(StandardCharsets.UTF_8));

                clientChannel.write(buffer, buffer, new CompletionHandler<Integer, ByteBuffer>() {
```

```
                    @Override
                    public void completed(Integer bytesWritten, ByteBuffer buf) {
                        if (buf.hasRemaining()) {
                            clientChannel.write(buf, buf, this);
                        } else {
                            buf.clear();
                            // Read response from server
                            clientChannel.read(buf, buf, new CompletionHandler<Integer, ByteBuffer>() {
                                @Override
                                public void completed(Integer bytesRead, ByteBuffer buf) {
                                    buf.flip();
                                    byte[] data = new byte[buf.limit()];
                                    buf.get(data);
                                    System.out.println("Received from server: " + new String(data));
                                    try {
                                        clientChannel.close();
                                    } catch (Exception e) {
                                        e.printStackTrace();
                                    }
                                }

                                @Override
                                public void failed(Throwable exc, ByteBuffer attachment) {
                                    System.err.println("Read failed");
                                    exc.printStackTrace();
                                }
                            });
                        }
                    }

                    @Override
                    public void failed(Throwable exc, ByteBuffer buf) {
                        System.err.println("Write failed");
                        exc.printStackTrace();
                    }
                });
            }

            @Override
            public void failed(Throwable exc, Void attachment) {
                System.err.println("Connection failed");
                exc.printStackTrace();
            }
        });

        Thread.sleep(2000); // Keep main thread alive to complete async operations
    }
}
```

8.4.9 Summary

Java's asynchronous IO network communication model via `AsynchronousSocketChannel` and `AsynchronousServerSocketChannel`:

- Enables **non-blocking connect, read, and write** operations.
- Uses **`CompletionHandler` callbacks** to receive notifications on operation completion or failure, enabling event-driven programming.
- Scales efficiently by avoiding thread blocking and allowing concurrent handling of many connections.
- Can be combined with `Future` objects when blocking on completion is acceptable.

This approach is ideal for modern network servers and clients that require scalability, responsiveness, and efficient resource use. The event-driven style with `CompletionHandler`s encourages a clean separation of IO logic from processing logic, making it a powerful tool in the Java networking toolkit.

Chapter 9.

Working with Character Sets and Encodings

1. Charset and CharsetDecoder/CharsetEncoder
2. Reading and Writing Text with Charset Support
3. Handling Unicode and UTF Variants
4. Charset Conversion Examples

9 Working with Character Sets and Encodings

9.1 Charset and CharsetDecoder/CharsetEncoder

When working with text in Java, understanding how characters are represented and converted to bytes (and vice versa) is crucial. This is especially important for applications dealing with file IO, network communication, or interoperability with systems that may use different encodings. Java's `Charset`, `CharsetEncoder`, and `CharsetDecoder` classes provide a robust framework to handle these conversions reliably and efficiently.

9.1.1 What is a Character Set?

A **character set** (or **charset**) defines a mapping between a collection of characters (letters, digits, symbols) and their corresponding numeric values (code points). These numeric values are then encoded into sequences of bytes to store or transmit text.

Common character sets include:

- **ASCII:** 7-bit, represents basic English letters and control characters.
- **ISO-8859-1:** An 8-bit encoding covering Western European languages.
- **UTF-8:** A variable-length Unicode encoding capable of representing any character.
- **UTF-16:** A Unicode encoding using 2 or 4 bytes per character.

Because different systems and protocols may use different charsets, **converting between bytes and characters requires specifying which charset to use**. Incorrect charset assumptions often lead to mojibake (garbled text).

9.1.2 The `Charset` Class in Java

The `Charset` class (in `java.nio.charset`) represents a named mapping between sequences of 16-bit Unicode characters (`char`) and sequences of bytes. Java's core platform includes support for many standard charsets, and you can obtain a `Charset` instance for any supported charset.

9.1.3 How to Obtain a Charset

You can get a `Charset` instance via:

```
Charset charset1 = Charset.forName("UTF-8");          // Standard charset by name
Charset charset2 = StandardCharsets.UTF_8;            // Preferred constant from Java 7+
```

Java also provides constants for other popular charsets like `US_ASCII`, `ISO_8859_1`, and `UTF_16`.

9.1.4 Role of Charset in Java Text Encoding and Decoding

- **Encoding:** Converting a `CharBuffer` (characters) into a `ByteBuffer` (bytes) using a `CharsetEncoder`.
- **Decoding:** Converting a `ByteBuffer` back into a `CharBuffer` using a `CharsetDecoder`.

This two-way transformation is essential because:

- Internally, Java strings and characters use UTF-16.
- External data (files, network data) is often represented as bytes encoded in a specific charset.

9.1.5 `CharsetEncoder` and `CharsetDecoder` Classes

CharsetEncoder

`CharsetEncoder` converts characters into bytes according to a specific charset encoding scheme.

- Created from a `Charset` by calling `.newEncoder()`.
- Provides methods to encode characters in bulk or incrementally.
- Handles character-to-byte conversion, including error handling for unmappable or malformed characters.

CharsetDecoder

`CharsetDecoder` converts bytes into characters.

- Created from a `Charset` by calling `.newDecoder()`.
- Reads bytes from a `ByteBuffer` and outputs decoded characters into a `CharBuffer`.
- Handles invalid byte sequences gracefully.

9.1.6 Why Use CharsetEncoder/Decoder Instead of Convenience Methods?

While Java offers convenience methods like `String.getBytes(Charset)` and `new String(byte[], Charset)`, the encoder/decoder classes give:

- **More control:** You can manage incremental encoding/decoding (streaming).
- **Error handling:** Configure how to respond to malformed or unmappable input.

- **Performance benefits:** Avoid unnecessary intermediate objects in bulk operations.

9.1.7 Example: Using CharsetEncoder and CharsetDecoder

Full runnable code:

```
import java.nio.*;
import java.nio.charset.*;

public class CharsetExample {
    public static void main(String[] args) throws CharacterCodingException {
        // Obtain a Charset instance for UTF-8
        Charset charset = StandardCharsets.UTF_8;

        // Create encoder and decoder
        CharsetEncoder encoder = charset.newEncoder();
        CharsetDecoder decoder = charset.newDecoder();

        // The original string
        String original = "Hello,  ";  // Includes Unicode characters

        // Encode: Convert characters to bytes
        CharBuffer charBuffer = CharBuffer.wrap(original);
        ByteBuffer byteBuffer = encoder.encode(charBuffer);

        System.out.println("Encoded bytes:");
        while (byteBuffer.hasRemaining()) {
            System.out.printf("%02X ", byteBuffer.get());
        }
        System.out.println();

        // Reset buffer position for reading
        byteBuffer.flip();

        // Decode: Convert bytes back to characters
        CharBuffer decodedCharBuffer = decoder.decode(byteBuffer);
        String decodedString = decodedCharBuffer.toString();

        System.out.println("Decoded string:");
        System.out.println(decodedString);
    }
}
```

Output:

```
Encoded bytes:
48 65 6C 6C 6F 2C 20 E4 B8 96 E7 95 8C
Decoded string:
Hello,
```

9.1.8 Handling Malformed and Unmappable Characters

`CharsetEncoder` and `CharsetDecoder` allow configuring actions on errors:

```
encoder.onMalformedInput(CodingErrorAction.REPLACE);
encoder.onUnmappableCharacter(CodingErrorAction.IGNORE);
decoder.onMalformedInput(CodingErrorAction.REPORT);
```

Options include:

- `REPORT`: Throw an exception on error.
- `IGNORE`: Skip malformed/unmappable sequences.
- `REPLACE`: Replace with a default character (e.g., `?`).

9.1.9 Summary

- The **`Charset`** class models the concept of a named character encoding scheme and provides access to encoders and decoders.
- The **`CharsetEncoder`** converts characters (Java's internal UTF-16) into bytes according to the charset.
- The **`CharsetDecoder`** converts bytes back into characters.
- These classes are essential for correctly reading and writing text data in different encodings, avoiding data corruption.
- They support fine-grained control over incremental encoding/decoding and error handling, making them suitable for performance-critical and robust applications.

By mastering `Charset`, `CharsetEncoder`, and `CharsetDecoder`, Java developers can handle text data across diverse platforms and protocols with confidence and precision.

9.2 Reading and Writing Text with Charset Support

Handling text files correctly in Java requires careful attention to character encoding. A common source of bugs and data corruption arises when the character encoding used to read a file does not match the encoding in which the file was written. This mismatch can cause unreadable characters (mojibake), lost data, or exceptions. To avoid such problems, **explicitly specifying the character set (charset) during text file IO is crucial**.

9.2.1 Why Specifying Charset Matters

Every text file is essentially a sequence of bytes interpreted as characters according to an encoding scheme or charset. Examples of common charsets are UTF-8, ISO-8859-1, UTF-16,

etc. Different charsets encode characters differently:

- For example, the character 'é' in UTF-8 is two bytes (`0xC3 0xA9`), but in ISO-8859-1 it is one byte (`0xE9`).
- A file encoded in UTF-8 but read as ISO-8859-1 will produce corrupted characters or unexpected results.

If you rely on Java's platform default charset implicitly (e.g., `FileReader` or `FileWriter`), your code becomes non-portable and fragile because default charset varies by platform, locale, and JVM settings.

Explicit charset specification ensures that your program reads and writes text consistently, regardless of platform or environment.

9.2.2 Java Classes for Charset-Aware Text IO

The key classes to perform charset-aware reading and writing of text files are:

- **`InputStreamReader`**: Converts bytes from an input stream into characters using a specified charset.
- **`OutputStreamWriter`**: Converts characters into bytes and writes them to an output stream using a specified charset.

Both classes bridge between byte streams (`InputStream`/`OutputStream`) and character streams (`Reader`/`Writer`).

9.2.3 Reading Text Files with `InputStreamReader`

The typical pattern to read a text file with a specific charset is:

Full runnable code:

```
import java.io.*;
import java.nio.charset.Charset;

public class CharsetAwareFileRead {
    public static void main(String[] args) {
        File file = new File("example.txt");
        Charset charset = Charset.forName("UTF-8");

        try (InputStream inputStream = new FileInputStream(file);
             Reader reader = new InputStreamReader(inputStream, charset);
             BufferedReader bufferedReader = new BufferedReader(reader)) {

            String line;
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
```

```
            }

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

9.2.4 Explanation:

- `FileInputStream` reads raw bytes from the file.
- `InputStreamReader` converts these bytes into characters using the specified charset (`UTF-8` in this example).
- `BufferedReader` provides efficient buffered reading and convenient methods like `readLine()`.

This method guarantees that bytes are interpreted according to the specified charset, avoiding platform default pitfalls.

9.2.5 Writing Text Files with `OutputStreamWriter`

Similarly, to write text safely with explicit charset:

Full runnable code:

```
import java.io.*;
import java.nio.charset.Charset;

public class CharsetAwareFileWrite {
    public static void main(String[] args) {
        File file = new File("output.txt");
        Charset charset = Charset.forName("UTF-8");

        try (OutputStream outputStream = new FileOutputStream(file);
             Writer writer = new OutputStreamWriter(outputStream, charset);
             BufferedWriter bufferedWriter = new BufferedWriter(writer)) {

            bufferedWriter.write("Hello,  !");
            bufferedWriter.newLine();
            bufferedWriter.write("This file is written with UTF-8 charset.");

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

9.2.6 Explanation:

- `FileOutputStream` writes raw bytes to the file.
- `OutputStreamWriter` encodes characters into bytes using the specified charset.
- `BufferedWriter` provides buffered writing with convenient methods such as `newLine()`.

This ensures the written bytes accurately represent the characters in UTF-8 encoding.

9.2.7 Example: Round-Trip Reading and Writing

To illustrate safe and portable IO, consider reading a file in UTF-8 and writing its contents to another file in UTF-8 explicitly:

Full runnable code:

```
import java.io.*;
import java.nio.charset.StandardCharsets;

public class RoundTripTextIO {
    public static void main(String[] args) {
        File inputFile = new File("input.txt");
        File outputFile = new File("output.txt");

        try (BufferedReader reader = new BufferedReader(
                new InputStreamReader(new FileInputStream(inputFile), StandardCharsets.UTF_8));
             BufferedWriter writer = new BufferedWriter(
                new OutputStreamWriter(new FileOutputStream(outputFile), StandardCharsets.UTF_8))) {

            String line;
            while ((line = reader.readLine()) != null) {
                writer.write(line);
                writer.newLine();
            }

            System.out.println("File copied successfully with UTF-8 charset.");

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

This example:

- Reads `input.txt` assuming it's encoded in UTF-8.
- Writes its contents to `output.txt` in UTF-8.
- Avoids data corruption by explicitly specifying the charset for both reading and writing.
- Is portable and reliable across different platforms and JVM settings.

9.2.8 Common Pitfalls and Best Practices

1. **Never rely on default charset for file IO.** Always specify charset explicitly unless you have very specific reasons and control over the environment.
2. **Match the charset on both reading and writing.** When reading files you wrote yourself, use the same charset consistently to avoid surprises.
3. **Use `StandardCharsets` constants when possible.** E.g., `StandardCharsets.UTF_8` is preferred over `Charset.forName("UTF-8")` to avoid typos and `UnsupportedCharsetException`.
4. **Wrap streams in buffered readers/writers.** For performance and convenient line-based operations.
5. **Be cautious with legacy APIs like `FileReader` and `FileWriter`.** They use platform default charset internally and are discouraged for portable code.

9.2.9 Summary

- Text files are sequences of bytes interpreted as characters via a charset.
- Mismatched charset assumptions between reading and writing cause data corruption.
- Java's `InputStreamReader` and `OutputStreamWriter` classes allow explicit charset specification bridging byte streams to character streams.
- Always specify charset explicitly for safe, portable, and correct text file IO.
- Use buffered wrappers for efficient and convenient reading/writing.

By adhering to these principles and using the proper classes with explicit charset parameters, Java developers can avoid many common pitfalls and ensure their applications handle text files robustly across environments and locales.

9.3 Handling Unicode and UTF Variants

In today's globalized digital world, software often needs to handle text from many languages and scripts. This requirement has made Unicode the fundamental standard for representing text characters consistently across platforms and systems. Understanding Unicode and its encoding schemes—UTF-8, UTF-16, and UTF-32—is critical for developers, including Java programmers, to ensure correct, efficient, and interoperable text processing.

9.3.1 What Is Unicode?

Unicode is a **universal character set** designed to encode all the characters used in writing systems worldwide—letters, digits, symbols, emoji, and control characters—into a single standard. Unlike older character encodings that supported only limited alphabets (like ASCII or ISO-8859-1), Unicode can represent **over one million code points**, though currently fewer than 150,000 are assigned.

- Each character in Unicode is assigned a **code point**, a unique integer value typically written in hexadecimal, e.g., the Latin capital letter A is `U+0041`.
- Code points are organized in **planes** of 65,536 characters each; the **Basic Multilingual Plane (BMP)** is plane 0 and contains most commonly used characters.

Unicode itself is an abstract mapping of characters to numbers. To store or transmit text, these numbers need to be encoded as bytes—this is where **UTF encodings** come in.

9.3.2 Why Different UTF Encodings Exist

Unicode code points are abstract; they do not specify how to represent characters as bytes. Different encoding schemes—UTF-8, UTF-16, and UTF-32—define how to convert these code points to byte sequences.

Each UTF encoding offers trade-offs in terms of:

- **Storage size**
- **Compatibility with legacy encodings**
- **Processing complexity**
- **Ease of random access to characters**

No single encoding fits all use cases perfectly, so the Unicode standard provides multiple UTF variants to meet diverse needs.

9.3.3 The UTF Encodings Explained

UTF-8

- **Variable-length encoding:** 1 to 4 bytes per character.
- Uses **1 byte** for ASCII characters (`U+0000` to `U+007F`), making it fully backward-compatible with ASCII.
- Non-ASCII characters use 2, 3, or 4 bytes.
- Widely used on the web and many modern applications due to its efficiency for texts dominated by ASCII characters.
- Byte order is unambiguous (no endianness issues).

Example:

- The character 'A' (`U+0041`) is encoded as `0x41` (1 byte).
- The character '中' (`U+4E16`) is encoded as `0xE4 0xB8 0x96` (3 bytes).

9.3.4 UTF-16

- **Variable-length encoding:** 2 or 4 bytes per character.
- Characters in the BMP (most common characters) are encoded as **2 bytes**.
- Characters outside the BMP (supplementary characters) use **surrogate pairs**—two 2-byte code units (total 4 bytes).
- Commonly used in Windows APIs and Java internally.
- Requires consideration of **byte order (endianness)** — UTF-16LE and UTF-16BE variants exist.

9.3.5 UTF-32

- **Fixed-length encoding:** 4 bytes per character.
- Each Unicode code point is stored as a 4-byte integer.
- Simple for indexing and processing since every character is a fixed size.
- Inefficient for storage compared to UTF-8 and UTF-16, used mostly in internal processing or environments where fixed-width encoding is beneficial.

9.3.6 How Java Supports Unicode and UTF Encodings

Java's native `char` type is a **16-bit UTF-16 code unit**, meaning Java strings are internally encoded as UTF-16 sequences. This design enables Java to handle all BMP characters in a single `char`, but supplementary characters (outside BMP) are represented using **two `char` units called surrogate pairs**.

Java provides rich support for encoding and decoding Unicode text:

- Classes like `java.nio.charset.Charset`, `CharsetEncoder`, and `CharsetDecoder` support UTF-8, UTF-16 (both endian variants), and UTF-32 (less common).
- StandardCharsets class includes constants like `StandardCharsets.UTF_8`, `StandardCharsets.UTF_16`, and `StandardCharsets.UTF_16BE`.
- Input/output streams and readers/writers accept charset parameters for reading and writing Unicode data correctly.
- Unicode-aware APIs like `String`, `Character`, and `CodePoint` methods help manage surrogate pairs and supplementary characters.

9.3.7 Practical Advice on Choosing the Right Encoding

1. **Default to UTF-8 when possible** UTF-8 is the de facto standard on the internet and in most modern software because it is compact for ASCII-heavy text, compatible with ASCII, and byte-order safe. For new projects and cross-platform interoperability, UTF-8 is generally the best choice.
2. **Use UTF-16 when interacting with systems or protocols that expect it** For example, Windows and Java internally use UTF-16, and some APIs or file formats require UTF-16 encoded data. But be aware of byte order and surrogate pairs.
3. **UTF-32 is rarely needed except for specialized processing** Fixed-width UTF-32 simplifies character indexing but uses much more space. Use it only if the application demands fixed-width encoding for performance reasons.
4. **Always specify the encoding explicitly** Never rely on platform default charset when reading or writing text, as this can cause data corruption or incompatibility.

9.3.8 Common Pitfalls When Handling Unicode Data

- **Assuming one `char` equals one character:** In Java, `char` is a UTF-16 code unit, not a full Unicode character. Supplementary characters are represented by surrogate pairs (`char` pairs), so methods like `String.length()` may not correspond to the number of actual Unicode characters (code points). Use methods like `codePointCount()`, `codePointAt()`, and `offsetByCodePoints()` for proper handling.
- **Not specifying charset on IO:** Reading a UTF-8 file as ISO-8859-1 will produce garbled output. Always specify charset explicitly, e.g., `new InputStreamReader(inputStream, StandardCharsets.UTF_8)`.
- **Ignoring byte order in UTF-16:** UTF-16 encoded files can be little-endian or big-endian. The presence of a BOM (byte order mark) can help detect the order, but some files omit it. Make sure to use the correct variant or detect BOM properly.
- **Incorrectly handling surrogate pairs:** String operations that manipulate characters by index can break surrogate pairs and corrupt text. Use Unicode-aware APIs.

9.3.9 Summary

Unicode is the universal standard for representing text from all languages, and UTF encodings specify how to serialize those characters into bytes:

- **UTF-8**: Compact, ASCII-compatible, variable length (1-4 bytes), widely used.

- **UTF-16**: Variable length (2 or 4 bytes), used internally by Java and Windows, sensitive to byte order.
- **UTF-32**: Fixed length (4 bytes), simple but space-inefficient.

Java's internal string representation uses UTF-16, and it offers comprehensive support for all UTF encodings through its `Charset` APIs. Choosing the right encoding involves balancing compatibility, efficiency, and application requirements, with UTF-8 being the safest default for most cases.

By understanding these fundamentals and pitfalls, developers can correctly handle Unicode text, ensuring applications are robust, internationalized, and interoperable in a multilingual world.

9.4 Charset Conversion Examples

In many real-world Java applications, you often need to convert text data from one character encoding to another—for example, reading a file encoded in ISO-8859-1 and saving it as UTF-8, or processing data streams with mixed encodings. Java's `CharsetDecoder` and `CharsetEncoder` classes, along with utility classes in `java.nio.charset`, provide robust tools to perform such conversions reliably.

This section explains how to convert text between different charsets in Java and demonstrates practical code examples.

9.4.1 Why Charset Conversion is Important

Different systems and files may use different encodings, and misinterpreting bytes as the wrong charset can lead to corrupted text (mojibake) or exceptions. Charset conversion ensures that:

- Text data can be interoperably exchanged between systems.
- Legacy data encoded with older encodings can be converted to modern Unicode-based formats like UTF-8.
- Your application can display or store text correctly according to user or system requirements.

9.4.2 Core Classes: CharsetDecoder and CharsetEncoder

- **`CharsetDecoder`** converts bytes from a specific charset into Java characters (`char`), resulting in a `CharBuffer`.

- **`CharsetEncoder`** converts Java characters (`char`) into bytes according to a target charset, resulting in a `ByteBuffer`.

To convert text from one charset to another, you:

1. **Decode** the original bytes using the source charset into characters.
2. **Encode** these characters using the target charset back into bytes.

9.4.3 Example 1: Basic Charset Conversion Using CharsetDecoder and CharsetEncoder

This example reads a byte array encoded in ISO-8859-1 and converts it to a UTF-8 encoded byte array.

Full runnable code:

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.*;

public class CharsetConversionExample {
    public static void main(String[] args) throws CharacterCodingException {
        // Original text encoded in ISO-8859-1 bytes (for demo)
        byte[] iso8859Bytes = {(byte)0xE9, (byte)0x20, (byte)0x6C, (byte)0xE0, (byte)0x20, (byte)0x63,

        // Step 1: Decode ISO-8859-1 bytes to characters
        Charset sourceCharset = Charset.forName("ISO-8859-1");
        CharsetDecoder decoder = sourceCharset.newDecoder();
        ByteBuffer sourceBytes = ByteBuffer.wrap(iso8859Bytes);
        CharBuffer chars = decoder.decode(sourceBytes);

        System.out.println("Decoded characters:");
        System.out.println(chars.toString());  // prints: é là cé car

        // Step 2: Encode characters into UTF-8 bytes
        Charset targetCharset = StandardCharsets.UTF_8;
        CharsetEncoder encoder = targetCharset.newEncoder();
        ByteBuffer utf8Bytes = encoder.encode(chars);

        System.out.println("Re-encoded UTF-8 bytes:");
        while (utf8Bytes.hasRemaining()) {
            System.out.printf("%02X ", utf8Bytes.get());
        }
        // Output: C3 A9 20 6C C3 A0 20 63 C3 A9 20 63 61 72
    }
}
```

9.4.4 Explanation

- We start with a byte array `iso8859Bytes` containing text encoded in ISO-8859-1, including accented characters like `é` and `à`.
- Using `CharsetDecoder` for ISO-8859-1, we decode these bytes into Java characters (`CharBuffer`).
- Then, we encode these characters into UTF-8 bytes using `CharsetEncoder`.
- This two-step decode-encode approach converts text from one charset to another safely.

9.4.5 Example 2: Converting Text File Encoding

Suppose you have a file encoded in Windows-1252 and want to convert it to UTF-8.

Full runnable code:

```
import java.io.*;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.*;

public class FileEncodingConverter {
    public static void main(String[] args) {
        File inputFile = new File("input-win1252.txt");
        File outputFile = new File("output-utf8.txt");

        Charset sourceCharset = Charset.forName("windows-1252");
        Charset targetCharset = StandardCharsets.UTF_8;

        try (InputStream inStream = new FileInputStream(inputFile);
             OutputStream outStream = new FileOutputStream(outputFile)) {

            // Decoder and encoder
            CharsetDecoder decoder = sourceCharset.newDecoder();
            CharsetEncoder encoder = targetCharset.newEncoder();

            // Read all bytes from source file
            byte[] inputBytes = inStream.readAllBytes();
            ByteBuffer sourceBuffer = ByteBuffer.wrap(inputBytes);

            // Decode bytes to chars
            CharBuffer charBuffer = decoder.decode(sourceBuffer);

            // Encode chars to target charset bytes
            ByteBuffer targetBuffer = encoder.encode(charBuffer);

            // Write bytes to output file
            outStream.write(targetBuffer.array(), 0, targetBuffer.limit());

            System.out.println("File encoding converted successfully.");

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

```
        }
    }
}
```

9.4.6 Explanation

- This program reads all bytes from a file encoded in Windows-1252.
- It uses `CharsetDecoder` to decode the bytes into characters.
- It then encodes the characters into UTF-8 bytes with `CharsetEncoder`.
- Finally, it writes the UTF-8 bytes to the output file.
- This approach preserves the text correctly, regardless of special characters present.

9.4.7 Utility Approach: Using `new String()` and `getBytes()` for Quick Conversions

For simpler scenarios, Java's `String` constructors and `getBytes()` methods can be used for conversion:

Full runnable code:

```
import java.io.UnsupportedEncodingException;

public class Test {

    public class SimpleConversion {
        public static void main(String[] args) throws UnsupportedEncodingException {
            byte[] windows1252Bytes = { (byte) 0xE9, (byte) 0x20, (byte) 0x6C, (byte) 0xE0 }; // é là

            // Decode bytes into String with Windows-1252
            String text = new String(windows1252Bytes, "windows-1252");
            System.out.println("Decoded text: " + text);

            // Encode String into UTF-8 bytes
            byte[] utf8Bytes = text.getBytes("UTF-8");

            System.out.println("UTF-8 bytes:");
            for (byte b : utf8Bytes) {
                System.out.printf("%02X ", b);
            }
        }
    }
}
```

9.4.8 Note

While convenient, this approach doesn't provide fine-grained control over error handling or streaming conversion and may be less efficient for large data.

9.4.9 Handling Malformed and Unmappable Characters

When converting between charsets, you may encounter characters that cannot be mapped from source to target charset. Both `CharsetDecoder` and `CharsetEncoder` allow configuring error handling strategies:

```
decoder.onMalformedInput(CodingErrorAction.REPLACE);
decoder.onUnmappableCharacter(CodingErrorAction.IGNORE);

encoder.onMalformedInput(CodingErrorAction.REPORT);
encoder.onUnmappableCharacter(CodingErrorAction.REPLACE);
```

Options include:

- `REPORT`: Throw an exception on error (default).
- `REPLACE`: Substitute with a replacement character (usually `?`).
- `IGNORE`: Skip the problematic input.

Use these settings depending on your tolerance for data loss or corruption.

9.4.10 Summary

- Charset conversion in Java involves **decoding bytes to characters** from the source charset and **encoding characters to bytes** in the target charset.
- Use `CharsetDecoder` and `CharsetEncoder` for controlled, streaming-capable, and robust conversions.
- For quick tasks, `new String(byte[], charset)` and `String.getBytes(charset)` may suffice.
- Always be mindful of error handling policies to deal with malformed or unmappable characters.
- Explicitly specifying charsets ensures your application correctly handles multilingual data and avoids text corruption.

Mastering charset conversion techniques ensures your Java applications remain compatible with diverse data sources and produce reliably encoded output across environments.

Chapter 10.

Practical Java IO and NIO Examples

1. File Copying and Moving
2. Implementing a File Watcher
3. Building a Simple HTTP Server with NIO
4. Logging and Debugging IO Operations
5. Best Practices for Efficient IO

10 Practical Java IO and NIO Examples

10.1 File Copying and Moving

File manipulation—copying and moving files—is a common requirement in many Java applications, whether for backup, organization, or processing workflows. Java provides multiple APIs to accomplish these tasks: the traditional I/O streams (`FileInputStream`/`FileOutputStream`) and the modern NIO (`java.nio.file.Files`) utilities introduced since Java 7.

This tutorial covers both approaches with practical examples, explaining their differences, advantages, and use cases.

10.1.1 Traditional IO Approach: Copying Files Using Streams

Before Java 7, the common way to copy a file was to open input and output streams and manually transfer bytes. This approach gives you low-level control but requires careful resource management and more code.

10.1.2 Copying a File Using `FileInputStream` and `FileOutputStream`

Full runnable code:

```
import java.io.*;

public class FileCopyTraditional {
    public static void copyFile(File source, File destination) throws IOException {
        try (FileInputStream fis = new FileInputStream(source);
             FileOutputStream fos = new FileOutputStream(destination)) {

            byte[] buffer = new byte[8192]; // 8KB buffer
            int bytesRead;

            while ((bytesRead = fis.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead);
            }
        }
    }

    public static void main(String[] args) {
        File src = new File("source.txt");
        File dest = new File("destination.txt");

        try {
            copyFile(src, dest);
            System.out.println("File copied successfully using traditional IO.");
```

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

10.1.3 Explanation

- `FileInputStream` reads raw bytes from the source file.
- `FileOutputStream` writes bytes to the destination file.
- The `buffer` temporarily stores chunks of bytes during transfer, improving efficiency.
- The `try-with-resources` statement ensures streams are closed automatically, avoiding resource leaks.
- This method works on any JVM version, but it requires manual buffer management and error handling.

10.1.4 Moving Files Using Traditional IO

Moving a file by streams requires copying the file contents and then deleting the original file:

Full runnable code:

```
import java.io.*;

public class FileMoveTraditional {
    public static void moveFile(File source, File destination) throws IOException {
        copyFile(source, destination);  // Copy the file
        if (!source.delete()) {         // Delete the original file
            throw new IOException("Failed to delete original file: " + source.getAbsolutePath());
        }
    }

    // Reuse copyFile from previous example
    public static void copyFile(File source, File destination) throws IOException {
        try (FileInputStream fis = new FileInputStream(source);
             FileOutputStream fos = new FileOutputStream(destination)) {

            byte[] buffer = new byte[8192];
            int bytesRead;

            while ((bytesRead = fis.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead);
            }
        }
    }

    public static void main(String[] args) {
```

```
        File src = new File("source.txt");
        File dest = new File("moved.txt");

        try {
            moveFile(src, dest);
            System.out.println("File moved successfully using traditional IO.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

10.1.5 Explanation

- Move is done by copying and deleting the source file.
- This approach is less efficient and more error-prone because deletion might fail or leave duplicates.
- It does not handle atomic moves or filesystem-specific optimizations.

10.1.6 Modern NIO Approach: Using `java.nio.file.Files`

Java 7 introduced the NIO.2 package (`java.nio.file`), which simplifies file operations with the `Files` utility class. It provides methods such as `Files.copy()` and `Files.move()`, which are easier to use, safer, and more powerful.

10.1.7 Copying Files Using `Files.copy()`

Full runnable code:

```
import java.io.*;
import java.nio.file.*;

public class FileCopyNIO {
    public static void main(String[] args) {
        Path source = Paths.get("source.txt");
        Path destination = Paths.get("destination.txt");

        try {
            Files.copy(source, destination, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("File copied successfully using NIO.");
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }
}
```

10.1.8 Explanation

- `Files.copy()` copies the file content atomically when supported by the file system.
- The third argument is a varargs of `CopyOption`s:
  - `StandardCopyOption.REPLACE_EXISTING` overwrites the destination file if it exists.
  - Other options include `COPY_ATTRIBUTES` (copy file metadata).
- `Files.copy()` can also copy directories recursively with additional logic.
- It automatically handles resource management and buffering internally.

10.1.9 Moving Files Using `Files.move()`

Full runnable code:

```
import java.io.*;
import java.nio.file.*;

public class FileMoveNIO {
    public static void main(String[] args) {
        Path source = Paths.get("source.txt");
        Path destination = Paths.get("moved.txt");

        try {
            Files.move(source, destination, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("File moved successfully using NIO.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

10.1.10 Explanation

- `Files.move()` moves or renames a file.
- It attempts to perform the move atomically if the file system supports it, which is faster and safer.

- You can specify options similar to `Files.copy()`, e.g., `REPLACE_EXISTING`.
- If moving across different file systems, it falls back to copy-and-delete, handling errors gracefully.

10.1.11 When to Use Each Approach?

Feature	Traditional IO (`FileInputStream` / `FileOutputStream`)	NIO (`Files.copy()`, `Files.move()`)
Simplicity	Verbose and manual	Simple and concise
Performance	Buffer size configurable but requires manual code	Optimized, atomic operations where supported
Resource management	Must manage streams explicitly	Automatic and safer
Atomic moves (rename)	Not supported	Supported if file system allows
Metadata copying	Not handled	Supported (`COPY_ATTRIBUTES` option)
Portability and modern API	Legacy, less portable	Modern, recommended since Java 7
Error handling	More complex to handle all corner cases	Robust and consistent

10.1.12 Summary and Best Practices

- For **new projects**, always prefer NIO's `Files.copy()` and `Files.move()`. These methods are simpler, safer, and usually more efficient.
- Use **traditional IO** only when you need explicit control over byte streams or need compatibility with very old Java versions.
- When using NIO, specify options like `StandardCopyOption.REPLACE_EXISTING` to control behavior explicitly.
- Handle exceptions gracefully, especially for moves, since atomicity depends on the underlying filesystem.
- Remember to consider permissions and filesystem differences when copying or moving files, especially across different volumes or network drives.

10.1.13 Conclusion

Copying and moving files in Java can be done either with low-level stream-based APIs or the modern NIO utilities. While traditional IO is still valid, NIO's `Files` API is the recommended approach for most use cases, offering simplicity, reliability, and enhanced functionality.

By mastering both, you'll be equipped to handle file operations in any Java environment, ensuring your applications manipulate files efficiently and correctly.

10.2 Implementing a File Watcher

Monitoring file system changes—such as file creation, modification, or deletion—is a common need in many applications, including logging systems, IDEs, synchronization tools, and auto-reloaders. Java's NIO.2 API, introduced in Java 7, provides a powerful and efficient way to watch file system events through the `WatchService` API.

This guide walks you through implementing a file watcher step-by-step, explaining key concepts, and providing a complete, well-commented Java example.

10.2.1 Understanding the `WatchService` API

The `WatchService` API allows you to monitor one or more directories for changes such as:

- **ENTRY_CREATE:** A new file or directory is created.
- **ENTRY_MODIFY:** An existing file or directory is modified.
- **ENTRY_DELETE:** A file or directory is deleted.

The API works by:

1. Registering a directory (`Path`) with a `WatchService`.
2. Listening asynchronously for events on the registered directory.
3. Retrieving and processing the events as they occur.

10.2.2 Step 1: Obtain a WatchService

You create a `WatchService` from the file system's default provider:

```
WatchService watchService = FileSystems.getDefault().newWatchService();
```

This service will be used to register directories and poll for events.

10.2.3 Step 2: Register a Directory with the WatchService

You register a `Path` representing a directory with the `WatchService`, specifying which event kinds to watch:

```
Path dir = Paths.get("path/to/directory");

dir.register(watchService,
             StandardWatchEventKinds.ENTRY_CREATE,
             StandardWatchEventKinds.ENTRY_DELETE,
             StandardWatchEventKinds.ENTRY_MODIFY);
```

You can register multiple directories if needed.

10.2.4 Step 3: Poll and Process Events

Events are retrieved as `WatchKey` instances from the `WatchService`. You can use blocking or non-blocking polling:

- `watchService.take()` blocks until an event occurs.
- `watchService.poll()` returns immediately, possibly returning `null` if no event is available.

Each `WatchKey` contains one or more `WatchEvents`, each describing an event kind and the affected file relative to the registered directory.

10.2.5 Complete Java Example: Directory File Watcher

Full runnable code:

```
import java.io.IOException;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;
import java.util.List;

public class DirectoryWatcher {

    public static void main(String[] args) {
        // Directory to monitor
        Path dir = Paths.get("watched-dir");

        // Create the watcher service
        try (WatchService watchService = FileSystems.getDefault().newWatchService()) {

            // Register the directory for CREATE, DELETE, and MODIFY events
            dir.register(watchService, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);

            System.out.println("Watching directory: " + dir.toAbsolutePath());
```

```
            // Infinite loop to wait and process events
            while (true) {
                WatchKey key;
                try {
                    // Wait for a key to be signaled (blocking call)
                    key = watchService.take();
                } catch (InterruptedException e) {
                    System.out.println("Interrupted. Exiting.");
                    return;
                }

                // Retrieve all pending events for the key
                List<WatchEvent<?>> events = key.pollEvents();

                for (WatchEvent<?> event : events) {
                    // Get event kind
                    WatchEvent.Kind<?> kind = event.kind();

                    // The context for directory entry event is the relative path to the file
                    WatchEvent<Path> ev = (WatchEvent<Path>) event;
                    Path filename = ev.context();

                    System.out.printf("Event kind: %s. File affected: %s%n", kind.name(), filename);

                    // You can add custom logic here, e.g. react to specific files
                }

                // Reset the key - this step is critical to receive further watch events
                boolean valid = key.reset();
                if (!valid) {
                    System.out.println("WatchKey no longer valid, directory might be inaccessible.");
                    break;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

10.2.6 Explanation of the Code

- **WatchService Creation:** We create a new `WatchService` instance from the default filesystem.
- **Registering the Directory:** The `dir.register()` method registers the directory and specifies the event kinds to monitor: create, delete, and modify.
- **Event Loop:** The `while(true)` loop calls `watchService.take()`, which blocks until an event is available.
- **Processing Events:** Each `WatchKey` may have multiple events; we iterate over them.

Each event provides:

  - The **kind** of event (`ENTRY_CREATE`, `ENTRY_MODIFY`, or `ENTRY_DELETE`).
  - The **context** — the relative path of the affected file/directory (relative to the registered path).

- **Resetting the Key:** Calling `key.reset()` re-enables the key for further event notifications. If it returns `false`, the key is invalid, possibly because the directory was deleted or is inaccessible.
- **Exception Handling:** IO exceptions and interruptions are handled gracefully.

10.2.7 Important Notes and Best Practices

- **Only directories can be registered.** You cannot register individual files. To monitor multiple directories, register each one separately.
- **Events are relative paths.** The event's `context()` is relative to the directory registered, so combine it with the directory path if you need the full path.
- **WatchService behavior is platform-dependent.** Some platforms may coalesce multiple events or behave slightly differently. For example, on Linux with `inotify`, modify events may trigger multiple times.
- **Long-running processes should handle interruptions gracefully.** When shutting down the watcher thread, interrupting the thread waiting on `take()` is the recommended way to stop.
- **Performance considerations:** For directories with heavy changes, events may be lost or coalesced. Monitor carefully for missed changes if that matters.
- **File renames:** A rename may appear as a delete event followed by a create event.

10.2.8 Extending the File Watcher

You can extend the watcher to:

- **Monitor multiple directories** by registering each directory's `Path` with the same `WatchService`.
- **React to specific files** or patterns by adding conditional logic inside the event loop.
- **Run the watcher in a separate thread** for non-blocking operation in GUI or server applications.
- **Use logging frameworks** instead of `System.out.println` for production use.

10.2.9 Summary

- Java NIO's `WatchService` API provides an efficient way to monitor directories for file creation, modification, and deletion.
- You create a `WatchService`, register directories with it, and then loop, waiting for events.
- Handling the `WatchKey` and calling `reset()` is essential to continue receiving events.
- This approach is more efficient and cleaner compared to periodic polling for file changes.

By following this guide and using the example code, you can implement your own directory monitoring solution in Java that responds in near real-time to file system changes.

10.3 Building a Simple HTTP Server with NIO

Traditional Java networking with `ServerSocket` and `Socket` classes uses blocking IO, where a thread waits for each client. While simple, this doesn't scale well with many concurrent clients because each connection consumes a thread.

Java NIO (New IO) introduced **non-blocking IO** and the **selector pattern** to handle many connections efficiently with a small number of threads. This tutorial walks you through creating a minimal HTTP server using NIO's `ServerSocketChannel`, `SocketChannel`, and `Selector`.

10.3.1 What Youll Learn

- How to open a `ServerSocketChannel` in non-blocking mode.
- How to create and use a `Selector` to multiplex multiple client connections.
- How to accept new client connections.
- How to read and parse simple HTTP requests.
- How to write basic HTTP responses.
- Understanding the non-blocking design pattern with selectors.

10.3.2 Non-blocking IO and Selector Overview

Non-blocking IO

In non-blocking mode, read/write calls on channels **do not block** if the data is not immediately available. Instead, they return immediately with how much data was read or written. This means a single thread can:

- Check which channels are ready for reading or writing.
- Read/write only the ready channels without blocking.
- Move on to check other channels.

10.3.3 Selector

A **Selector** allows a single thread to monitor multiple channels for IO events:

- Channels register with the selector, specifying which operations they're interested in (e.g., accept, read, write).
- The selector's `select()` method blocks until one or more registered channels are ready for any of the requested operations.
- The thread processes ready channels and then calls `select()` again.

10.3.4 Step 1: Setting up the ServerSocketChannel

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.bind(new InetSocketAddress(8080));
serverChannel.configureBlocking(false);  // Non-blocking mode
```

- Open the server socket channel.
- Bind it to port 8080.
- Set it to non-blocking mode to work with the selector.

10.3.5 Step 2: Creating the Selector and Registering the Server Channel

```
Selector selector = Selector.open();
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

- Open a selector.
- Register the server socket channel with the selector for **accept** events (new incoming connections).

10.3.6 Step 3: Event Loop with Selector

The server runs an event loop, calling `selector.select()` to wait for events, then handling each ready channel accordingly.

10.3.7 Step 4: Accepting Connections and Registering Client Channels

When the server socket channel is ready to accept, you call `accept()`, configure the new socket channel as non-blocking, and register it with the selector for **read** events.

10.3.8 Step 5: Reading Data from Client Channels

When a socket channel is ready for reading, you read bytes into a buffer. For simplicity, we assume the request fits into one read. (In production, you'd accumulate and parse incrementally.)

10.3.9 Step 6: Parsing HTTP Requests and Writing Responses

We parse the request line (e.g., `GET / HTTP/1.1`), ignore headers for simplicity, and respond with a simple HTTP 200 OK message with a plain text body.

10.3.10 Full Working Java Code

Full runnable code:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.nio.charset.StandardCharsets;
import java.util.Iterator;
import java.util.Set;

public class SimpleNIOServer {

    private static final int PORT = 8080;
    private static final String RESPONSE_BODY = "Hello from NIO HTTP Server!";
    private static final String RESPONSE_TEMPLATE =
        "HTTP/1.1 200 OK\r\n" +
        "Content-Length: %d\r\n" +
        "Content-Type: text/plain\r\n" +
        "Connection: close\r\n" +
        "\r\n%s";

    public static void main(String[] args) throws IOException {
        // Open server socket channel
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.bind(new InetSocketAddress(PORT));
        serverChannel.configureBlocking(false);
```

```
        // Open selector
        Selector selector = Selector.open();

        // Register server channel for accept events
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);

        System.out.println("Server listening on port " + PORT);

        while (true) {
            // Wait for events
            selector.select();

            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> iter = selectedKeys.iterator();

            while (iter.hasNext()) {
                SelectionKey key = iter.next();
                iter.remove();

                if (key.isAcceptable()) {
                    handleAccept(key, selector);
                }

                if (key.isReadable()) {
                    handleRead(key);
                }

                if (key.isWritable()) {
                    handleWrite(key);
                }
            }
        }
    }

    private static void handleAccept(SelectionKey key, Selector selector) throws IOException {
        ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
        SocketChannel clientChannel = serverChannel.accept();  // Accept connection
        clientChannel.configureBlocking(false);

        System.out.println("Accepted connection from " + clientChannel.getRemoteAddress());

        // Register client channel for reading
        clientChannel.register(selector, SelectionKey.OP_READ, ByteBuffer.allocate(1024));
    }

    private static void handleRead(SelectionKey key) throws IOException {
        SocketChannel clientChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = (ByteBuffer) key.attachment();

        int bytesRead = clientChannel.read(buffer);
        if (bytesRead == -1) {
            // Client closed connection
            System.out.println("Client closed connection: " + clientChannel.getRemoteAddress());
            clientChannel.close();
            key.cancel();
            return;
        }
```

```
        // Check if we have received a full HTTP request (simplistic check: look for double CRLF)
        String request = new String(buffer.array(), 0, buffer.position(), StandardCharsets.US_ASCII);
        if (request.contains("\r\n\r\n")) {
            System.out.println("Received request:\n" + request.split("\r\n")[0]);  // Print request lin

            // Prepare response
            String response = String.format(RESPONSE_TEMPLATE, RESPONSE_BODY.length(), RESPONSE_BODY);

            // Attach response bytes to the key for writing
            key.attach(ByteBuffer.wrap(response.getBytes(StandardCharsets.US_ASCII)));

            // Change interest to write
            key.interestOps(SelectionKey.OP_WRITE);
        }
    }

    private static void handleWrite(SelectionKey key) throws IOException {
        SocketChannel clientChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = (ByteBuffer) key.attachment();

        clientChannel.write(buffer);

        if (!buffer.hasRemaining()) {
            // Response fully sent; close connection
            System.out.println("Response sent. Closing connection: " + clientChannel.getRemoteAddress()
            clientChannel.close();
            key.cancel();
        }
    }
}
```

10.3.11 Explanation of the Code

- **ServerSocketChannel & Selector:** We create and bind a non-blocking server socket channel, then register it with a selector for accept events.
- **Event Loop:** The server thread loops calling `selector.select()`, which blocks until events occur.
- **Accepting Connections:** When `isAcceptable()` is true, a new client connection is accepted, configured as non-blocking, and registered for read events with a 1024-byte buffer attached.
- **Reading Requests:** When a channel is readable, we read bytes into the buffer. We do a basic check for the end of the HTTP headers (double CRLF) in the buffer.
- **Parsing Request:** For simplicity, we just print the first request line. Real servers would parse method, path, headers, and body.
- **Preparing the Response:** We create a simple HTTP 200 OK response with plain text body. The response bytes are wrapped into a new `ByteBuffer` and attached to

the key.

- **Writing Response:** When the channel is writable, we write bytes from the buffer to the client socket channel.
- **Connection Close:** After sending the response fully, the connection is closed and the key canceled.

10.3.12 Key Points About Non-blocking Design Pattern

- **Single-threaded, multiplexed IO:** One thread manages multiple client connections, reacting only when channels are ready, avoiding busy waiting or many threads.
- **Stateful keys:** Each `SelectionKey` holds state: a `ByteBuffer` for reading and later a buffer for writing response bytes.
- **InterestOps Switching:** We switch interest ops between reading and writing as appropriate. When done writing, the channel is closed.
- **Efficiency:** Non-blocking IO with selectors scales much better than one-thread-per-connection blocking IO, especially under many simultaneous connections.

10.3.13 How to Run and Test

- Compile and run the Java program.
- Open a browser or use `curl` to access `http://localhost:8080/`.
- You should see the response text `"Hello from NIO HTTP Server!"`.
- Multiple clients can connect concurrently with minimal threads.

10.3.14 Limitations and Next Steps

- **Basic request parsing only:** This server just reads until it sees `\r\n\r\n` and does not handle HTTP methods, headers, or request bodies.
- **No concurrency or thread pool:** The single-threaded event loop handles all IO. For CPU-bound tasks, you may want a worker thread pool.
- **No HTTPS or advanced HTTP features:** This is a minimal proof of concept.
- **Error handling:** Production servers need more robust error handling and resource cleanup.

10.3.15 Summary

This tutorial demonstrated building a simple HTTP server using Java NIO's non-blocking IO:

- Open a `ServerSocketChannel` in non-blocking mode.
- Use a `Selector` to multiplex multiple connections efficiently.
- Accept new connections and register for reading.
- Read and parse basic HTTP requests.
- Write HTTP responses asynchronously.
- Close connections after response.

With this foundation, you can explore building more advanced servers with richer HTTP support and better scalability.

10.4 Logging and Debugging IO Operations

Input/output (IO) operations are fundamental to many Java applications—reading files, writing logs, communicating over networks, or processing streams. However, IO operations are often a common source of bugs and issues, such as silent failures, resource leaks, or encoding mismatches. Effective logging and debugging practices are essential to identify, diagnose, and fix these problems efficiently.

This section explores common IO issues, how to trace them, and how to implement logging strategies using Java's built-in logging (`java.util.logging`) as well as the popular SLF4J facade, illustrated with IO-specific examples.

10.4.1 Silent Failures

A very frequent problem is when IO operations fail silently. For example, failing to close a stream due to swallowed exceptions or ignoring error return codes can cause resource leaks or data corruption.

10.4.2 Encoding Mismatches

When reading or writing text files or network data, incorrect or inconsistent character encodings cause garbled text or data loss. For example, reading UTF-8 encoded data using ISO-8859-1 results in corrupted characters.

10.4.3 Buffering and Partial Reads/Writes

Reading or writing in incorrect buffer sizes or misunderstanding the non-blocking nature of some channels can cause incomplete data processing or infinite loops.

10.4.4 Concurrency and Resource Contention

Accessing the same file or stream from multiple threads without synchronization can lead to unpredictable behavior and hard-to-reproduce bugs.

10.4.5 File Not Found or Permission Issues

Common IOExceptions due to missing files or permission denied can halt program execution if not handled or logged properly.

10.4.6 Why Logging is Important in IO

- **Trace Errors and Exceptions:** Logging stack traces and exception messages provides insight into the root cause of failures.
- **Monitor Resource Usage:** Logs can indicate if streams or channels were properly closed.
- **Detect Encoding Issues Early:** Logging the charset used during read/write helps ensure encoding consistency.
- **Track Data Flow:** Debug logs can show what data was read/written, aiding in diagnosing corruption or unexpected input.

10.4.7 Using Javas Built-in Logging (`java.util.logging`)

Java provides a built-in logging API in the `java.util.logging` package. Here's an example demonstrating logging around an IO operation with proper error handling:

Full runnable code:

```
import java.io.*;
import java.nio.charset.Charset;
import java.util.logging.*;
```

```java
public class LoggingIOExample {
    private static final Logger logger = Logger.getLogger(LoggingIOExample.class.getName());

    public static void readFile(String path, Charset charset) {
        logger.info("Starting to read file: " + path + " with charset: " + charset);

        try (BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(path)
            String line;
            while ((line = reader.readLine()) != null) {
                logger.fine("Read line: " + line);
            }
        } catch (FileNotFoundException e) {
            logger.log(Level.SEVERE, "File not found: " + path, e);
        } catch (IOException e) {
            logger.log(Level.SEVERE, "Error reading file: " + path, e);
        }

        logger.info("Finished reading file: " + path);
    }

    public static void main(String[] args) {
        // Set log level to show info and above; fine logs won't appear by default
        Logger rootLogger = Logger.getLogger("");
        rootLogger.setLevel(Level.INFO);

        readFile("example.txt", Charset.forName("UTF-8"));
    }
}
```

10.4.8 Explanation:

- Use `Logger.getLogger()` to create a logger for the class.
- Log entry and exit of important IO methods (`info` level).
- Log each line read with `fine` level (debug-level logs; can be enabled in detailed debug mode).
- Log exceptions with full stack traces using `logger.log(Level.SEVERE, message, exception)`.
- Properly close resources using try-with-resources.

10.4.9 Configuring Logging Levels

By default, `java.util.logging` shows logs of level INFO and above. To see DEBUG-level logs (`FINE`), configure the logging level programmatically or via a properties file.

10.4.10 Using SLF4J for IO Logging

SLF4J (Simple Logging Facade for Java) is widely used in enterprise applications for flexible logging abstraction. SLF4J works with popular backends like Logback or Log4j.

10.4.11 Example of Logging IO Operations with SLF4J

Full runnable code:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.*;
import java.nio.charset.StandardCharsets;

public class Slf4jLoggingIOExample {

    private static final Logger logger = LoggerFactory.getLogger(Slf4jLoggingIOExample.class);

    public static void writeFile(String path, String content) {
        logger.info("Writing to file: {}", path);

        try (BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(pat
            writer.write(content);
            logger.debug("Written content length: {}", content.length());
        } catch (IOException e) {
            logger.error("Error writing to file: {}", path, e);
        }

        logger.info("Finished writing to file: {}", path);
    }

    public static void main(String[] args) {
        writeFile("output.txt", "Hello SLF4J IO logging!");
    }
}
```

10.4.12 Key Points:

- Parameterized logging (`{}` placeholders) avoids unnecessary string concatenation when logs are disabled.
- Separate log levels (`info`, `debug`, `error`) provide flexible control.
- Log exceptions with the exception object to get stack traces.
- SLF4J makes it easy to switch logging implementations without changing application code.

Debugging Tips for IO Issues

10.4.13 Log Byte Buffers and Content Dumps

When reading raw bytes (e.g., from network sockets or files), logging hex dumps or base64 representations of data helps spot corruption.

```
private static void logBufferContent(ByteBuffer buffer, Logger logger) {
    byte[] bytes = new byte[buffer.remaining()];
    buffer.get(bytes);
    String hexDump = javax.xml.bind.DatatypeConverter.printHexBinary(bytes);
    logger.debug("Buffer content (hex): {}", hexDump);
    buffer.position(buffer.position() - bytes.length); // reset position after reading
}
```

10.4.14 Log Charset Details

Always log which charset you are using to read or write text, as mismatches often cause hard-to-detect bugs.

10.4.15 Use Stack Traces and Cause Chains

IOExceptions often wrap root causes. Use `logger.log()` to print full stack traces and investigate nested exceptions.

10.4.16 Watch for Resource Leaks

Log when streams or channels are opened and closed to detect if resources are not properly freed.

10.4.17 Enable More Verbose Logging Temporarily

In troubleshooting, increase log verbosity to `FINE` or `DEBUG` for IO classes or packages.

10.4.18 Summary

- IO operations are prone to subtle bugs like silent failures, encoding issues, and resource leaks.

- Effective logging is critical to surface and diagnose these problems.
- Use `java.util.logging` or SLF4J to log entry/exit points, data content, exceptions, and charset details.
- Employ parameterized logging and log at appropriate levels to avoid performance overhead.
- Always close resources safely and log their lifecycle events.
- Logging combined with careful exception handling and charset awareness will make your IO operations robust and easier to debug.

By following these guidelines and using logging strategically, you can confidently trace and fix issues in Java IO code—turning black-box problems into understandable and manageable events.

10.5 Best Practices for Efficient IO

Input/output (IO) is a critical aspect of many Java applications, ranging from file processing to network communication. However, IO operations can become bottlenecks if not handled efficiently. Writing efficient IO code helps reduce latency, improve throughput, and conserve system resources.

This guide covers essential best practices to write performant and robust IO code in Java, with practical examples and tips.

10.5.1 Use Buffering to Improve Performance

Reading or writing data byte-by-byte or character-by-character is extremely inefficient because each call may result in an expensive system call.

10.5.2 Why Buffer?

- **Reduce system calls:** Buffering batches many small reads or writes into fewer larger ones.
- **Lower overhead:** Less interaction with the underlying OS reduces context switches and CPU usage.

10.5.3 How to Buffer in Java?

Java provides buffered wrappers:

- `BufferedInputStream` and `BufferedOutputStream` for byte streams.
- `BufferedReader` and `BufferedWriter` for character streams.

10.5.4 Example: Buffered File Copy

Full runnable code:

```
import java.io.*;

public class BufferedFileCopy {
    public static void copyFile(File source, File dest) throws IOException {
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(source));
             BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(dest))) {
            byte[] buffer = new byte[8192];  // 8KB buffer
            int bytesRead;
            while ((bytesRead = bis.read(buffer)) != -1) {
                bos.write(buffer, 0, bytesRead);
            }
        }
    }
}
```

10.5.5 Performance Tip:

Use at least an 8KB buffer size (`8192` bytes) for disk IO; smaller buffers may increase overhead, while excessively large buffers waste memory and may cause GC pressure.

10.5.6 Always Use Try-With-Resources to Avoid Resource Leaks

Open IO resources (streams, readers, sockets) must be closed promptly to release system resources like file descriptors.

10.5.7 Why Avoid Leaks?

- Unclosed resources can cause application slowdowns, file locking issues, or even crashes due to exhaustion of handles.

10.5.8 Javas Solution: Try-With-Resources

Since Java 7, the **try-with-resources** statement ensures automatic closing:

```
try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}  // reader is automatically closed here, even if exceptions occur
```

Avoid manual closing or ignoring exceptions in finally blocks. Try-with-resources is safer, cleaner, and less error-prone.

10.5.9 Choose Proper Buffer Sizes Based on Context

Buffer size affects latency and throughput:

- **Too small:** Frequent system calls degrade performance.
- **Too large:** Wastes memory and may slow down GC, especially in memory-constrained environments.

10.5.10 General Recommendations:

- **File IO:** 8KB to 64KB buffers typically work well.
- **Network IO:** Depending on the protocol, 4KB to 16KB buffers are common.
- **Memory-mapped files (NIO):** Use buffers aligned to system page sizes (usually 4KB or 8KB).

Experiment with buffer sizes using benchmarks specific to your workload.

10.5.11 Know When to Use Traditional IO vs NIO

Traditional IO (`java.io`)

- Easier to use, great for simple tasks.
- Blocking by nature—each call waits until data is ready.
- Uses stream-based APIs.
- Good for straightforward file reading/writing or small-scale applications.

NIO (`java.nio`)

- Non-blocking, scalable for high-concurrency scenarios.

- Uses channels, buffers, and selectors.
- Ideal for network servers or apps handling many simultaneous connections.
- Can perform memory-mapped file IO (`FileChannel.map()`) for large files with efficient OS paging.

Choosing Between Them

Scenario	Recommended API
Simple file read/write	Traditional IO
High-performance, large files	NIO FileChannel
Network servers with many clients	NIO with Selectors
Low concurrency, simple apps	Traditional IO

10.5.12 Minimize Disk or Network IO When Possible

IO is often orders of magnitude slower than CPU and memory access. Minimizing the number of IO operations can drastically improve performance.

10.5.13 Strategies:

- **Batch IO operations:** Read/write larger blocks instead of many small chunks.
- **Cache data in memory:** When data is reused frequently, avoid repeated disk/network access.
- **Avoid unnecessary flushes:** Flushing buffers too frequently forces OS-level writes.
- **Use lazy loading:** Only load data when necessary.
- **Compress data:** To reduce the amount transferred over networks or stored on disk.

10.5.14 Use Direct Byte Buffers with NIO for Network IO

Java NIO provides **direct byte buffers** (`ByteBuffer.allocateDirect()`) that allocate memory outside the JVM heap and can improve IO performance by avoiding an extra copy between Java heap and OS buffers.

10.5.15 Example:

```
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
```

Use direct buffers cautiously—they are more expensive to create and clean up, but beneficial for long-lived buffers in high-performance networking.

10.5.16 Avoid Blocking Calls in Critical Threads

If you use traditional IO in UI or event-driven applications, blocking calls can freeze the program.

- Use asynchronous IO (`AsynchronousFileChannel`, `AsynchronousSocketChannel`).
- Offload blocking IO calls to separate worker threads.

10.5.17 Examples Combining Best Practices

Reading a Text File Efficiently

Full runnable code:

```
import java.io.*;
import java.nio.charset.StandardCharsets;

public class EfficientFileReader {

    public static void readFile(String path) {
        try (BufferedReader reader = new BufferedReader(
                new InputStreamReader(new FileInputStream(path), StandardCharsets.UTF_8))) {
            String line;
            while ((line = reader.readLine()) != null) {
                // Process line
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Uses buffering (`BufferedReader`).
- Specifies charset explicitly.
- Uses try-with-resources for safety.

Writing to a Network Socket with NIO

Full runnable code:

```java
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class NIONetworkWriteExample {

    public static void sendMessage(String host, int port, String message) throws IOException {
        try (SocketChannel socketChannel = SocketChannel.open()) {
            socketChannel.connect(new InetSocketAddress(host, port));
            socketChannel.configureBlocking(true);

            ByteBuffer buffer = ByteBuffer.allocate(1024);
            buffer.put(message.getBytes());
            buffer.flip();

            while (buffer.hasRemaining()) {
                socketChannel.write(buffer);
            }
        }
    }
}
```

- Allocates a properly sized buffer.
- Uses try-with-resources.
- Writes entire message in a loop to ensure complete transmission.

10.5.18 Summary of Best Practices

Practice	Why It Matters
Use buffering	Reduces costly system calls, improves throughput
Use try-with-resources	Prevents resource leaks, simplifies cleanup
Choose appropriate buffer sizes	Balances memory use and IO efficiency
Pick IO vs NIO based on need	Simplifies code or enables scalability
Minimize IO operations	Reduces latency and resource consumption
Use direct buffers for NIO	Enhances performance in networking and large file IO
Avoid blocking calls on main threads	Keeps UI and event loops responsive

By following these best practices, you can write IO code in Java that is not only functionally correct but also efficient and scalable—helping your applications run smoothly in real-world environments.

Chapter 11.

Performance and Tuning

1. Buffer Sizes and Their Impact
2. Direct Buffers vs Heap Buffers
3. Reducing Garbage Collection Overhead
4. Profiling and Monitoring IO Performance

11 Performance and Tuning

11.1 Buffer Sizes and Their Impact

Buffering is a fundamental concept in IO programming, and the size of the buffer you choose can have a significant impact on the performance of your Java applications. Whether you're reading from files, writing to sockets, or streaming data, understanding how buffer size influences throughput, latency, and resource utilization is key to building efficient IO solutions.

This explanation dives into the trade-offs of small vs large buffers, how buffer size affects IO performance metrics, and practical guidance to determine the best buffer sizes for various scenarios.

11.1.1 The Role of Buffers in IO

When Java programs read or write data, buffers serve as temporary storage areas holding chunks of bytes or characters. Instead of interacting with the underlying system one byte at a time (which is extremely costly), data is transferred in blocks. This batching:

- Reduces the number of system calls.
- Decreases context switches between user and kernel space.
- Improves CPU efficiency.

Buffers are used in both traditional IO (`BufferedInputStream`, `BufferedReader`, etc.) and in Java NIO's `ByteBuffer`.

11.1.2 Small Buffers: Low Latency but Lower Throughput

Characteristics

- **Size:** Often 1–512 bytes.
- **Advantages:**
  - Lower latency for applications requiring immediate processing of data, such as interactive programs or real-time systems.
  - Reduced memory footprint.
- **Disadvantages:**
  - Increased number of IO calls, leading to higher overhead.
  - Poor throughput because many small system calls create performance bottlenecks.

Impact

For example, reading a large file one byte at a time results in thousands or millions of system calls, each incurring kernel/user mode transitions and associated costs. This drastically reduces throughput and wastes CPU cycles.

Scenario: Reading a file with a 128-byte buffer may cause hundreds of thousands of read operations for a multi-megabyte file, limiting throughput.

11.1.3 Large Buffers: High Throughput but Potentially Higher Latency

Characteristics

- **Size:** Typically 8 KB (8192 bytes) up to 64 KB or more.
- **Advantages:**
  - Fewer, larger IO operations mean less overhead and higher data throughput.
  - Reduced CPU usage per byte transferred.
- **Disadvantages:**
  - Higher latency per individual IO operation because more data must be accumulated or flushed.
  - Increased memory usage, which can affect heap pressure and garbage collection.
  - For latency-sensitive applications, buffering too much data may delay processing.

Impact

Large buffers can read or write tens of thousands of bytes per system call, dramatically improving IO throughput. For instance, a file copy operation using a 64 KB buffer can be orders of magnitude faster than one using a 512-byte buffer.

Scenario: Copying a 100 MB file with a 64 KB buffer performs approximately 1,600 read/write calls compared to over 200,000 calls with a 512-byte buffer.

11.1.4 Buffer Size and IO Performance Metrics

Throughput

Throughput is the amount of data processed per unit time (e.g., MB/s). Large buffers improve throughput by amortizing system call overhead across many bytes.

Latency

Latency is the delay between requesting data and receiving it. Smaller buffers reduce the wait to fill a buffer before processing, lowering latency. Larger buffers might increase latency

since the system waits to fill or flush the buffer.

CPU Utilization

Smaller buffers cause the CPU to spend more time managing system calls and context switches. Larger buffers improve CPU efficiency but can increase memory usage and possibly cause GC pauses if many large buffers are allocated frequently.

Practical Benchmark Example

Consider a simple benchmark copying a large file with different buffer sizes:

Full runnable code:

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Test {
    public static void copyFile(File src, File dest, int bufferSize) throws IOException {
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(src), bufferSize);
                BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(dest), bufferS

            byte[] buffer = new byte[bufferSize];
            int read;
            long start = System.currentTimeMillis();

            while ((read = bis.read(buffer)) != -1) {
                bos.write(buffer, 0, read);
            }

            long duration = System.currentTimeMillis() - start;
            System.out.println("Buffer size: " + bufferSize + " bytes, Time taken: " + duration + " ms"
        }
    }
}
```

11.1.5 Hypothetical Results

Buffer Size (bytes)	Time Taken (ms)
256	1200
1024	600
8192 (8 KB)	220
65536 (64 KB)	180
262144 (256 KB)	175

11.1.6 Interpretation

- Increasing from 256 bytes to 8 KB yields significant performance gains.
- Beyond 64 KB, returns diminish—large buffers may not improve much further.
- Extremely large buffers (256 KB+) increase memory usage with little added benefit.
- This plateau suggests an optimal buffer size around 8 KB to 64 KB for disk IO on typical systems.

11.1.7 Guidelines to Determine Optimal Buffer Size

1. **Start with 8 KB (8192 bytes):** This is the default buffer size for many Java IO classes and works well in most cases.
2. **Consider the IO medium:**
   - **Disk IO:** Use 8 KB to 64 KB buffers, aligned with filesystem block sizes.
   - **Network IO:** Use smaller buffers (4 KB to 16 KB) to reduce latency.
   - **Memory-mapped IO:** Buffer size depends on system page size (often 4 KB or 8 KB).
3. **Profile and Benchmark:** Measure your application's throughput and latency using different buffer sizes under realistic workloads.
4. **Adjust Based on Latency Sensitivity:** For real-time or interactive applications, smaller buffers might be justified even at throughput cost.
5. **Beware of JVM and OS tuning:** Sometimes buffer sizes are less impactful if underlying OS or JVM buffers dominate performance.

11.1.8 Buffer Size Considerations in Java NIO

In Java NIO, `ByteBuffer` size controls how much data is read or written per operation. While NIO supports non-blocking IO, inefficient buffer sizes still degrade performance:

- **Too small:** Frequent `read()` or `write()` calls that may return zero bytes and waste CPU.
- **Too large:** Increased heap memory usage, slower garbage collection, and possible memory fragmentation.

Use direct buffers (`ByteBuffer.allocateDirect`) wisely, as they are more expensive to allocate but can reduce copies between Java and OS.

11.1.9 Summary and Best Practices

- **Buffering is essential for performance** — never read or write one byte at a time.
- **Small buffers reduce latency but hurt throughput and CPU efficiency.**
- **Large buffers improve throughput by batching IO, but increase latency and memory use.**
- **8 KB to 64 KB is a practical range for most disk IO tasks.**
- **Benchmark your application with your actual workload and hardware** to find the best buffer size.
- **Consider the trade-offs between latency and throughput** depending on your application needs.
- **Monitor memory usage** to avoid over-allocating buffers causing GC issues.
- **In network programming, smaller buffers may reduce latency but consider protocol behavior.**

By understanding and tuning buffer sizes thoughtfully, you can optimize your Java applications to achieve the best balance of performance and resource usage in your IO operations.

11.2 Direct Buffers vs Heap Buffers

Java NIO introduced the `ByteBuffer` class and related buffer types to enable efficient, flexible data handling for IO operations. Among `ByteBuffer`s, two main categories exist based on memory allocation and management: **heap buffers** and **direct buffers**. Understanding the differences between these buffer types is crucial for writing high-performance Java applications, especially when working with files, networks, or native code.

11.2.1 What Are Heap Buffers?

Heap buffers are `ByteBuffer`s backed by regular Java heap memory arrays. When you create a heap buffer, Java allocates a byte array inside the JVM heap, and the buffer's API methods operate on this array.

11.2.2 Allocation and Management

Heap buffers are allocated via:

```
ByteBuffer heapBuffer = ByteBuffer.allocate(int capacity);
```

This allocates a `byte[]` internally on the heap. The JVM's garbage collector (GC) manages this memory automatically. Accessing the buffer's content is essentially array access, which is

fast and straightforward.

11.2.3 Access

Heap buffers expose their backing array, so you can retrieve it with:

```
byte[] array = heapBuffer.array();
```

This is convenient for interoperability with legacy code or APIs requiring arrays.

11.2.4 What Are Direct Buffers?

Direct buffers are allocated outside the JVM heap, in **native memory** managed by the operating system. They are intended to provide a buffer that can be passed more efficiently to native IO operations, such as OS-level read/write or DMA transfers.

11.2.5 Allocation and Management

You allocate a direct buffer using:

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(int capacity);
```

This creates a buffer whose memory is outside the heap. The JVM manages the buffer's lifecycle, but the actual memory is allocated by native OS calls (e.g., `malloc`).

Direct buffers don't have an accessible backing array. Access is done through JNI (Java Native Interface) or direct memory pointers internally.

The JVM periodically frees direct buffers via finalization or uses explicit cleaner mechanisms, but you cannot rely on immediate reclamation, which can lead to higher native memory usage if not carefully managed.

11.2.6 Performance Characteristics

Heap Buffers

- **Pros:**
  - Fast access within Java since data is stored in the heap array.
  - Efficient for frequent, small read/write operations inside JVM.

- Easy to use and compatible with any Java API requiring arrays.
- Garbage-collected automatically with low overhead.

- **Cons:**
  - When passed to native IO operations, the JVM must copy data between heap memory and native buffers. This copy incurs overhead, reducing performance in IO-heavy applications.
  - Limited for zero-copy or direct OS IO optimizations.

11.2.7 Direct Buffers

- **Pros:**
  - Bypasses copying to native buffers, enabling **zero-copy** IO operations on supported platforms.
  - Improved throughput for large, frequent IO operations, such as network servers or file channels.
  - Preferred when interfacing with native libraries or performing high-performance networking.
- **Cons:**
  - Allocating and deallocating direct buffers is more expensive than heap buffers.
  - Native memory is not managed by GC, so improper use can cause native memory leaks.
  - Accessing data is slower within Java code due to JNI overhead and no backing array access.
  - Requires more careful memory management and sometimes explicit cleanup.

11.2.8 Use Cases

Buffer Type	Recommended For
Heap Buffer	Short-lived buffers, small to medium data, frequent JVM-side manipulation, legacy code needing arrays
Direct Buffer	Large buffers, long-lived, IO-heavy applications, network servers, file channels, zero-copy requirements

11.2.9 Example Code: Allocating and Using Both Buffers

Full runnable code:

```
import java.nio.ByteBuffer;

public class BufferExample {
    public static void main(String[] args) {
        // Heap Buffer allocation
        ByteBuffer heapBuffer = ByteBuffer.allocate(1024);
        System.out.println("Heap Buffer: isDirect = " + heapBuffer.isDirect());

        // Put some data
        heapBuffer.put("Hello Heap Buffer".getBytes());
        heapBuffer.flip();  // Prepare for reading

        byte[] heapData = new byte[heapBuffer.remaining()];
        heapBuffer.get(heapData);
        System.out.println("Heap Buffer content: " + new String(heapData));

        // Access backing array (only for heap buffers)
        if (heapBuffer.hasArray()) {
            byte[] backingArray = heapBuffer.array();
            System.out.println("Backing array length: " + backingArray.length);
        }

        // Direct Buffer allocation
        ByteBuffer directBuffer = ByteBuffer.allocateDirect(1024);
        System.out.println("Direct Buffer: isDirect = " + directBuffer.isDirect());

        directBuffer.put("Hello Direct Buffer".getBytes());
        directBuffer.flip();

        byte[] directData = new byte[directBuffer.remaining()];
        directBuffer.get(directData);
        System.out.println("Direct Buffer content: " + new String(directData));

        // Direct buffers do NOT expose a backing array
        System.out.println("Direct Buffer has array? " + directBuffer.hasArray());
    }
}
```

11.2.10 Output:

```
Heap Buffer: isDirect = false
Heap Buffer content: Hello Heap Buffer
Backing array length: 1024
Direct Buffer: isDirect = true
Direct Buffer content: Hello Direct Buffer
Direct Buffer has array? false
```

11.2.11 Key Points to Remember

- **`isDirect()` method:** Use this to check if a buffer is direct or heap-backed.
- **`hasArray()` and `array()`:** Available only for heap buffers.
- **Performance tuning:** For network and file IO where large volumes of data are transferred, prefer direct buffers for efficiency.
- **Garbage collection:** Heap buffers are managed naturally by the JVM GC, while direct buffers rely on less predictable native memory cleanup.
- **Resource management:** If you allocate many large direct buffers, consider explicit cleanup strategies or reuse buffers to avoid native memory exhaustion.

11.2.12 Summary

Aspect	Heap Buffers	Direct Buffers
Memory location	JVM heap	Native OS memory
Allocation	Fast, low overhead	Slower, more expensive
Access speed	Fast JVM access	Slower JVM access, faster native IO
Backed by array?	Yes	No
Garbage collection	Managed by JVM GC	Managed outside JVM, less predictable
Use case	Frequent JVM-side data manipulation, small buffers	High throughput network/file IO, zero-copy needs
Risks	None significant	Native memory leaks if mismanaged

Understanding these differences lets you choose the right buffer type depending on your IO patterns, data size, and performance requirements. For typical Java applications, heap buffers suffice, but when optimizing network servers or large file transfers, direct buffers often unlock better performance.

11.3 Reducing Garbage Collection Overhead

Java's automatic memory management via Garbage Collection (GC) simplifies development but can introduce unpredictable pauses—especially problematic in IO-heavy applications where low latency and high throughput are critical. Excessive object allocation during IO leads to frequent GC cycles, increasing pause times and reducing overall performance.

This discussion explores practical techniques to reduce GC overhead during IO operations by minimizing object creation, reusing buffers, leveraging direct memory, and employing object pooling. Understanding and applying these approaches helps maintain smoother application performance and lower latency.

11.3.1 Why Garbage Collection Overhead Matters in IO-Heavy Applications

IO-intensive Java programs often perform many short-lived operations—reading and writing data chunks, creating temporary objects for buffers or wrappers, and handling protocol parsing. Each allocation adds pressure on the JVM heap and triggers GC cycles when memory runs low.

11.3.2 Effects of GC Overhead:

- **Stop-the-world pauses:** JVM suspends all application threads to reclaim memory, causing latency spikes.
- **Throughput reduction:** Frequent GC consumes CPU cycles that could otherwise serve business logic.
- **Unpredictable performance:** Makes tuning latency-sensitive systems (network servers, real-time apps) difficult.

Reducing GC pressure helps maintain consistent response times and improves scalability.

11.3.3 Technique 1: Minimize Object Allocation in IO Paths

Avoid creating new objects inside tight IO loops or per-request processing. Common culprits include:

- Creating new byte arrays or buffers every time data is read or written.
- Constructing temporary wrappers like `ByteArrayInputStream` or strings unnecessarily.
- Instantiating small immutable objects repeatedly (e.g., wrapper classes, protocol headers).

11.3.4 How to Minimize Allocations:

- **Use reusable buffers:** Allocate byte buffers once and reuse them for multiple read/write cycles.

- **Avoid unnecessary boxing/unboxing:** Prefer primitive arrays or `ByteBuffer` over wrapper classes.
- **Prefer streaming APIs:** For example, use `InputStream` and `OutputStream` directly rather than converting data back and forth to strings or objects.
- **Use `StringBuilder` for string concatenations instead of repeated `String` creation.**

11.3.5 Technique 2: Reuse Buffers to Avoid Allocation Overhead

IO operations frequently require byte or char buffers to hold data temporarily. Allocating a new buffer per operation leads to many short-lived objects.

11.3.6 Buffer Reuse Strategies:

- **Thread-local buffers:** Each thread holds its own reusable buffer, avoiding contention and repeated allocation.

```
private static final ThreadLocal<byte[]> threadLocalBuffer =
    ThreadLocal.withInitial(() -> new byte[8192]);

public void readData(InputStream in) throws IOException {
    byte[] buffer = threadLocalBuffer.get();
    int bytesRead;
    while ((bytesRead = in.read(buffer)) != -1) {
        // Process bytesRead from buffer
    }
}
```

- **Buffer pools:** Maintain a pool of preallocated buffers shared across threads, checked out and returned after use.
- **Avoid resizing:** Use fixed-size buffers when possible to prevent costly reallocations.

11.3.7 Technique 3: Use Direct Buffers to Reduce GC Impact

Java NIO direct buffers allocate memory outside the Java heap, managed by the OS. This means their allocation and deallocation do not directly contribute to GC pressure.

11.3.8 Benefits of Direct Buffers for IO:

- **Reduced heap footprint:** Less memory pressure on the JVM heap reduces GC frequency.
- **Zero-copy IO:** Native OS calls can access buffers directly without copying, improving throughput.
- **Long-lived buffers:** Holding on to direct buffers minimizes frequent allocations.

11.3.9 Caveats:

- Direct buffers are more expensive to create and reclaim.
- Native memory is not managed by GC, so excessive direct buffer allocation without release risks native memory leaks.

11.3.10 Example:

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(8192);
```

Reuse this buffer across IO calls to maximize benefit.

11.3.11 Technique 4: Employ Object and Buffer Pooling

Pooling reuses objects instead of creating new instances, reducing GC overhead and allocation costs.

11.3.12 Common Pooling Patterns:

- **Buffer Pools:** Implemented with `BlockingQueue<ByteBuffer>` or specialized libraries like Netty's `ByteBuf` allocator.
- **Object Pools:** For frequently created objects, such as protocol parsers, message objects, or event wrappers.

11.3.13 Pool Implementation Example:

```
import java.util.concurrent.ArrayBlockingQueue;

public class BufferPool {
    private final ArrayBlockingQueue<byte[]> pool;

    public BufferPool(int size, int bufferSize) {
        pool = new ArrayBlockingQueue<>(size);
        for (int i = 0; i < size; i++) {
            pool.offer(new byte[bufferSize]);
        }
    }

    public byte[] acquire() throws InterruptedException {
        return pool.take();
    }

    public void release(byte[] buffer) {
        pool.offer(buffer);
    }
}
```

11.3.14 Advantages:

- Reduces GC by avoiding repeated allocations.
- Can improve cache locality by reusing warm objects.
- Helps control maximum memory usage by bounding the pool size.

11.3.15 Performance Tips and Best Practices

- **Avoid premature optimization:** Measure allocation rates and GC behavior using profilers (e.g., VisualVM, Java Flight Recorder).
- **Prefer off-heap buffers for large, long-lived data:** Large heap buffers increase GC pause times due to longer marking and sweeping.
- **Combine pooling with thread-local buffers:** Minimizes synchronization overhead on pools.
- **Avoid global mutable state:** Pools must be thread-safe to avoid concurrency bugs.
- **Monitor native memory usage:** When using direct buffers or off-heap memory, track native allocations to prevent leaks.

11.3.16 How These Techniques Help Reduce GC Pauses

- **Lower allocation rate:** Fewer objects created means less frequent garbage collection.
- **Less heap fragmentation:** Reusing buffers reduces heap churn and fragmentation, which improves GC efficiency.
- **Reduced GC pause duration:** Smaller heaps or heaps with fewer live objects enable faster GC cycles.
- **Better predictability:** Applications with stable object usage patterns have more predictable latency.

11.3.17 Summary

Technique	What It Does	GC Impact
Minimize object allocation	Avoid unnecessary temporary objects	Lower allocation rate
Buffer reuse	Reuse pre-allocated byte arrays or buffers	Reduce short-lived objects
Use direct buffers	Allocate buffers off-heap for native IO	Reduces heap usage and GC load
Pooling	Maintain reusable buffer/object pools	Limits new allocations

11.3.18 Conclusion

GC overhead is a major performance factor in IO-heavy Java applications, but it can be significantly mitigated by conscious programming techniques:

- Minimize object creation during IO.
- Reuse buffers aggressively.
- Use direct buffers to reduce heap pressure.
- Implement pooling strategies for buffers and frequently created objects.

These methods reduce GC frequency and pause times, leading to smoother, more responsive, and scalable applications. Careful profiling and tuning are essential to strike the right balance between memory usage and performance.

11.4 Profiling and Monitoring IO Performance

Efficient IO operations are critical for Java applications, especially those dealing with files, databases, or networks. However, IO performance problems—such as bottlenecks, excessive garbage collection (GC), or slow disk/network access—can be difficult to detect and diagnose without the right tools and methodology.

This guide introduces popular Java profiling and monitoring tools, explains how to spot IO-related issues, and provides step-by-step instructions to set up basic IO performance tracking and analyze results effectively.

11.4.1 Why Profile IO Performance?

IO performance issues often manifest as:

- Increased latency or slow response times.
- High CPU utilization with little throughput gain.
- Excessive garbage collection due to temporary object churn.
- Thread contention or blocking on IO operations.

Profiling helps to:

- Understand where your application spends time.
- Identify hotspots related to IO calls.
- Detect inefficient buffer usage or frequent allocations.
- Pinpoint slow external systems (disks, networks).
- Correlate GC activity with IO workloads.

11.4.2 Key Profiling and Monitoring Tools for Java IO

Java Flight Recorder (JFR)

Overview:

JFR is a low-overhead profiling and event collection framework built into the JVM (Oracle JDK and OpenJDK 11+). It captures detailed runtime data, including thread states, IO events, allocations, and GC activity.

Why use JFR?

- Minimal performance impact, suitable for production environments.
- Rich event data tailored for IO operations (e.g., file read/write, socket events).
- Integrated with tools like Java Mission Control (JMC) for visual analysis.

Basic Setup:

Enable JFR when launching your app:

```
java -XX:StartFlightRecording=filename=recording.jfr,duration=60s,settings=profile -jar yourapp.jar
```

After the recording completes, open the `.jfr` file with Java Mission Control to analyze IO events, thread states, and GC behavior.

VisualVM

Overview:

VisualVM is a free visual profiling tool bundled with the JDK (or available standalone). It supports heap dumps, CPU profiling, thread analysis, and monitoring.

Why use VisualVM?

- Easy setup with a GUI interface.
- Real-time monitoring of memory, CPU, and threads.
- Plugins available for advanced profiling.

Using VisualVM for IO profiling:

- Start your Java application.
- Launch VisualVM and connect to the running JVM.
- Use the **Sampler** or **Profiler** tab to record CPU usage and allocations.
- Observe which methods dominate CPU time—look for IO methods such as `FileInputStream.read()`, `SocketChannel.write()`.
- Monitor memory usage and GC activity in the **Monitor** tab.

11.4.3 async-profiler

Overview:

`async-profiler` is a low-overhead, sampling-based profiler for Linux and macOS that supports CPU, allocation, and lock profiling. It can capture detailed stack traces without stopping your application.

Why use async-profiler?

- Extremely low overhead (1-2% CPU).
- Supports native and Java stacks for IO syscall analysis.
- Flame graphs and other visualizations help locate bottlenecks quickly.

Basic usage:

Download and build from async-profiler GitHub, then attach to a running JVM:

```
./profiler.sh -d 30 -f profile.html <pid>
```

Open `profile.html` in a browser and examine hotspots related to IO syscalls or Java IO

APIs.

11.4.4 Identifying IO Bottlenecks and GC Pressure

Step 1: Detect High IO Latency or Blocking

- Use thread analysis in VisualVM or JFR to find threads blocked in IO calls (`read()`, `write()`, `select()`, `poll()`).
- Look for thread states like `WAITING` or `BLOCKED` on IO-related system calls.
- Identify if threads spend excessive time waiting on slow disks or network.

Step 2: Analyze IO Method Hotspots

- Profile CPU usage to see which IO methods consume the most time.
- In async-profiler or VisualVM, look for repeated calls to inefficient buffer handling or blocking IO methods.
- Confirm if small buffer sizes or frequent system calls contribute to overhead.

Step 3: Monitor GC Activity Related to IO

- In JFR or VisualVM, monitor allocation rates and GC pause durations.
- High allocation rates in IO threads suggest excessive temporary object creation (e.g., creating new buffers every read).
- Look for Full GCs that coincide with IO bursts, indicating memory pressure caused by IO buffers.

Step 4: Measure Disk and Network Throughput

- JFR records OS-level IO events including bytes read/written and IO wait times.
- Identify if IO throughput is low due to slow disks, network latency, or contention.
- Compare IO throughput against CPU usage—high CPU but low throughput may indicate IO bottlenecks.

11.4.5 Setting Up Basic IO Performance Tracking

Using Java Flight Recorder

1. Start JFR recording with profiling enabled:

```
java -XX:StartFlightRecording=filename=myapp.jfr,duration=2m,settings=profile -jar myapp.jar
```

2. Perform your typical IO workload.
3. Open `myapp.jfr` with Java Mission Control.
4. Navigate to the **IO** tab:

- Review **File I/O** and **Socket I/O** events.
- Examine average latency and bytes transferred.
- Look for long-running IO operations.

5. Check the **Threads** tab for blocked or waiting threads.
6. Inspect **GC** statistics to understand allocation impact.

11.4.6 Using VisualVM for Real-Time Monitoring

1. Launch VisualVM and attach to the JVM process.
2. Select the **Monitor** tab to watch CPU, memory, and thread states live.
3. Use the **Profiler** tab to start CPU profiling.
4. Run your IO scenario, then stop profiling.
5. Analyze call trees for IO hotspots.
6. Monitor memory allocations and GC frequency under the **Sampler** tab.

11.4.7 Interpreting Profiling Results

- **High CPU in IO methods:** Indicates heavy IO processing or inefficient buffer handling.
- **Many threads blocked on IO:** May suggest slow external devices or insufficient concurrency.
- **Frequent GC during IO:** Implies excessive temporary object creation during data read/write.
- **Long IO latency in JFR:** Points to disk/network bottlenecks or contention.
- **Low throughput with high CPU:** Indicates CPU-bound processing around IO, possibly due to data transformations or inefficient code.

11.4.8 Summary and Next Steps

Profiling IO performance requires correlating multiple metrics—CPU, memory, thread states, and system IO events. Using tools like Java Flight Recorder, VisualVM, and async-profiler, you can pinpoint:

- Where IO is slowing down your app.
- How GC impacts IO responsiveness.

- Whether your buffers and threading strategy are efficient.

Start by enabling lightweight JFR profiles in production or use VisualVM for quick local debugging. When deeper insight is needed, async-profiler's flame graphs give a low-overhead, detailed view.

After identifying bottlenecks, optimize by:

- Reusing buffers to reduce allocations.
- Increasing buffer sizes to reduce syscall overhead.
- Using direct buffers or async IO.
- Adjusting thread pools for IO concurrency.
- Improving external device throughput.

Monitoring and profiling should be a continuous part of your development and operations process to maintain optimal IO performance.

Chapter 12.

Security and IO

1. IO Security Basics
2. Secure File Access and Permissions
3. Secure Network Communication with NIO
4. Handling Sensitive Data Streams

12 Security and IO

12.1 IO Security Basics

Input/output (IO) operations are foundational to nearly every Java application, enabling programs to read and write files, communicate over networks, and serialize data for storage or transmission. However, IO operations also introduce significant security risks if not handled carefully. Attackers often exploit insecure IO practices to gain unauthorized access, execute malicious code, or compromise data integrity and confidentiality.

This introduction explores why IO is a common source of security vulnerabilities in Java applications, highlights fundamental security principles like least privilege and input validation, and provides examples of common IO vulnerabilities alongside mitigation techniques.

12.1.1 Why IO Operations Present Security Risks

IO operations in Java interact with external resources outside the JVM's internal control—such as the filesystem, network sockets, and external data streams. These interactions inherently increase the attack surface:

- **File system access** can expose sensitive files or allow unauthorized file modification.
- **Network communication** can be intercepted, manipulated, or spoofed.
- **Serialization and deserialization** of objects can be exploited to inject malicious data or execute arbitrary code.

Since IO often deals with untrusted data—files uploaded by users, network input, or serialized objects—any lapse in validating or securing these operations can lead to serious security breaches.

12.1.2 Core Principles for Securing IO in Java

Least Privilege

Limit the permissions of your Java application and its components to the minimum necessary for their IO tasks. For example:

- Use Java Security Manager policies (where applicable) to restrict file or network access.
- Run your application under an OS user account with limited file system privileges.
- Avoid granting write access to directories unless absolutely required.

Least privilege reduces the potential impact if an attacker exploits an IO vulnerability.

Input Validation and Sanitization

All data read through IO channels must be treated as untrusted. Rigorously validate and sanitize input before processing:

- For **file paths**, prevent directory traversal attacks by canonicalizing paths and restricting access to safe directories.
- For **network data**, check message formats, lengths, and character sets.
- For **serialized objects**, use safe deserialization techniques (see below).

Reject or sanitize malformed or unexpected data to prevent injection attacks, buffer overflows, or crashes.

Secure Defaults and Explicit Configuration

Java IO APIs often have default behaviors that may not be secure in all contexts. Adopt secure defaults such as:

- Using **secure protocols** (e.g., TLS) for network communication.
- Setting file permissions explicitly after creating files or directories.
- Avoiding deserialization of untrusted data or using validation hooks.
- Closing IO resources promptly to avoid resource leaks that attackers can exploit.

12.1.3 Common IO-Related Vulnerabilities and Mitigation Strategies

Vulnerability 1: Directory Traversal

Description: An attacker crafts a filename containing sequences like `../` to access files outside the intended directory.

Example:

```
String baseDir = "/app/data/";
String requestedFile = "../../etc/passwd";
File file = new File(baseDir, requestedFile);
```

If unchecked, this may read the system's password file.

Mitigation:

- Use `File#getCanonicalPath()` to resolve the absolute path.
- Verify that the resolved path starts with the intended base directory.

```
File requested = new File(baseDir, requestedFile);
String canonicalBase = new File(baseDir).getCanonicalPath();
String canonicalRequested = requested.getCanonicalPath();

if (!canonicalRequested.startsWith(canonicalBase)) {
    throw new SecurityException("Invalid file path");
}
```

12.1.4 Vulnerability 2: Insecure Network Communication

Description: Data sent over plaintext sockets can be intercepted or altered by attackers.

Mitigation:

- Use secure channels like **TLS/SSL** via `SSLSocket` or `HttpsURLConnection`.
- Validate server certificates to prevent man-in-the-middle attacks.
- Authenticate and authorize clients as needed.

```
SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
try (SSLSocket socket = (SSLSocket) factory.createSocket(host, port)) {
    // Secure communication
}
```

12.1.5 Vulnerability 3: Unsafe Deserialization

Description: Java deserialization allows reconstruction of objects from byte streams. Attackers can craft malicious byte streams to execute arbitrary code during deserialization.

Mitigation:

- Avoid deserializing untrusted data.
- Use a whitelist of allowed classes with `ObjectInputFilter` (Java 9+).
- Consider safer alternatives like JSON or XML parsing with strict schemas.

```
ObjectInputStream ois = new ObjectInputStream(inputStream);
ObjectInputFilter filter = ObjectInputFilter.Config.createFilter("com.example.MySafeClass;!*");
ois.setObjectInputFilter(filter);
Object obj = ois.readObject();
```

12.1.6 Vulnerability 4: Resource Exhaustion

Description: Poor IO handling can cause resource leaks, such as open file descriptors or sockets, leading to denial of service.

Mitigation:

- Use **try-with-resources** to ensure streams and channels close properly.
- Limit buffer sizes and validate data lengths to avoid memory exhaustion.
- Monitor and log resource usage.

```
try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
    // Read file safely
}
```

12.1.7 Summary: Best Practices for Secure IO in Java

Principle	Best Practice
Least Privilege	Limit filesystem and network permissions
Input Validation	Sanitize paths, verify data formats
Secure Defaults	Use encrypted communication and safe deserialization
Resource Management	Close IO resources promptly with try-with-resources

12.1.8 Conclusion

IO operations in Java are inherently risky due to their interaction with external resources and untrusted data. Following security principles—least privilege, rigorous input validation, secure defaults, and proper resource management—greatly reduces the attack surface.

Common vulnerabilities like directory traversal, insecure network communication, unsafe deserialization, and resource exhaustion can be mitigated through careful coding and the use of modern Java security features.

By adopting these security-conscious IO practices, Java developers can build applications that are robust, reliable, and resistant to common IO-related attacks.

12.2 Secure File Access and Permissions

File access is a common yet sensitive operation in Java applications. Improper handling of file permissions can lead to unauthorized reading, modification, or deletion of critical data, exposing the application and its environment to security risks. Securing file access means ensuring that your Java program only accesses files it is authorized to, and that files are protected against unintended or malicious changes.

This section covers how to check and enforce file permissions using Java's `File` class and the more advanced `java.nio.file.attribute` package, the role of security managers in access control, and practical coding techniques to prevent unauthorized file operations.

12.2.1 Understanding File Permissions in Java

Java provides multiple layers for managing file access:

- **File system permissions:** These are OS-level permissions that restrict who can read, write, or execute files.

- **Java API-level checks:** Methods that check the accessibility of files before attempting operations.
- **Security Manager (deprecated in newer Java versions):** A Java-level policy mechanism to restrict file access within the JVM.

12.2.2 Checking File Permissions with the `File` Class

The `java.io.File` class offers simple methods to check file permissions:

- `canRead()` — Returns `true` if the file is readable.
- `canWrite()` — Returns `true` if the file is writable.
- `canExecute()` — Returns `true` if the file is executable.

12.2.3 Example: Checking Permissions

```
File file = new File("/path/to/file.txt");

if (file.exists()) {
    System.out.println("File exists");
    System.out.println("Readable: " + file.canRead());
    System.out.println("Writable: " + file.canWrite());
    System.out.println("Executable: " + file.canExecute());
} else {
    System.out.println("File does not exist.");
}
```

While these methods help to inspect permissions, they are often insufficient for enforcing strict security policies because they reflect the current OS permissions, and you may want more fine-grained control or to modify permissions programmatically.

12.2.4 Managing File Permissions with `java.nio.file.attribute`

Java 7 introduced the `java.nio.file` package, including the `attribute` subpackage, which provides a more flexible and powerful way to inspect and modify file attributes and permissions.

12.2.5 Using `PosixFilePermission`

On POSIX-compliant systems (Linux, Unix, macOS), you can read and change file permissions using `PosixFilePermission`:

```
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.Set;

Path path = Paths.get("/path/to/file.txt");

// Read permissions
Set<PosixFilePermission> perms = Files.getPosixFilePermissions(path);
System.out.println("Current permissions: " + PosixFilePermissions.toString(perms));

// Add owner write permission
perms.add(PosixFilePermission.OWNER_WRITE);

// Remove group write permission
perms.remove(PosixFilePermission.GROUP_WRITE);

// Set the new permissions
Files.setPosixFilePermissions(path, perms);
System.out.println("Permissions updated.");
```

This approach allows you to enforce minimum necessary permissions explicitly and prevent unwanted access.

12.2.6 Using ACLs on Windows

On Windows, the `AclFileAttributeView` can be used to manipulate Access Control Lists (ACLs):

```
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.List;

Path path = Paths.get("C:\\path\\to\\file.txt");

AclFileAttributeView aclView = Files.getFileAttributeView(path, AclFileAttributeView.class);
List<AclEntry> aclEntries = aclView.getAcl();

for (AclEntry entry : aclEntries) {
    System.out.println(entry.principal() + ": " + entry.permissions());
}

// Modify ACLs as needed (requires detailed knowledge of Windows ACL)
```

This allows precise control over which users or groups can access or modify the file.

12.2.7 Enforcing File Access Using the Security Manager (Legacy)

Note: The `SecurityManager` and its associated file permission checks are deprecated and slated for removal in future Java versions, but understanding them is helpful in legacy

contexts.

The `SecurityManager` can restrict file operations by enforcing policies based on `java.io.FilePermission`. For example, you can configure a security policy file to grant or deny read/write access to specific files or directories.

Example security policy entry:

```
grant codeBase "file:/myapp/-" {
    permission java.io.FilePermission "/myapp/data/-", "read,write";
    permission java.io.FilePermission "/myapp/config/config.xml", "read";
};
```

With the Security Manager enabled, Java checks these permissions before allowing file operations, throwing `SecurityException` if denied.

12.2.8 Preventing Unauthorized File Access: Best Practices

1. **Validate and sanitize file paths:** Prevent directory traversal by canonicalizing paths and restricting access to allowed directories.

```
File baseDir = new File("/app/data/");
File requested = new File(baseDir, userInputPath);

String canonicalBase = baseDir.getCanonicalPath();
String canonicalRequested = requested.getCanonicalPath();

if (!canonicalRequested.startsWith(canonicalBase)) {
    throw new SecurityException("Access denied: invalid file path");
}
```

2. **Check permissions before operations:** Use `File.canRead()` and `File.canWrite()` to verify access, but don't rely solely on these—handle exceptions robustly.

3. **Set secure file permissions after creating files:** For example, create a file and then restrict access to owner only.

```
Path newFile = Files.createFile(Paths.get("/app/data/newfile.txt"));
Set<PosixFilePermission> perms = PosixFilePermissions.fromString("rw-------");
Files.setPosixFilePermissions(newFile, perms);
```

4. **Use try-with-resources and handle exceptions:** Ensure streams and channels close properly to avoid resource leaks that might cause file locks.

5. **Avoid running as an administrator or root:** Run your Java process under a least-privilege OS user to limit potential damage.

12.2.9 Secure File Handling Code Example

Here is an example demonstrating secure reading of a user-requested file, with path validation and permission checking:

Full runnable code:

```
import java.io.*;
import java.nio.file.*;

public class SecureFileReader {

    private final Path baseDirectory;

    public SecureFileReader(String baseDir) {
        this.baseDirectory = Paths.get(baseDir).toAbsolutePath().normalize();
    }

    public String readFile(String userProvidedPath) throws IOException {
        Path requestedFile = baseDirectory.resolve(userProvidedPath).normalize();

        if (!requestedFile.startsWith(baseDirectory)) {
            throw new SecurityException("Unauthorized file access attempt");
        }

        if (!Files.exists(requestedFile) || !Files.isReadable(requestedFile)) {
            throw new FileNotFoundException("File not found or not readable");
        }

        try (BufferedReader reader = Files.newBufferedReader(requestedFile)) {
            StringBuilder content = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                content.append(line).append("\n");
            }
            return content.toString();
        }
    }

    public static void main(String[] args) {
        try {
            SecureFileReader sfr = new SecureFileReader("/app/data");
            String content = sfr.readFile("example.txt");
            System.out.println("File content:\n" + content);
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

This code ensures the requested file resides under a base directory, checks read permissions, and safely reads the file.

12.2.10 Conclusion

Securing file access in Java requires a combination of OS-level permissions and application-level checks. The `File` class provides simple permission inspection, while the `java.nio.file.attribute` package offers advanced control over file attributes and access rights. Although the Security Manager can enforce security policies, modern applications rely more on OS user permissions and explicit permission handling.

By validating file paths, verifying permissions before access, setting restrictive file attributes, and carefully managing file IO operations, you can greatly reduce the risk of unauthorized file access and modification in your Java applications.

12.3 Secure Network Communication with NIO

Network communication in Java NIO offers scalable, non-blocking IO operations, which are ideal for high-performance servers and clients. However, by default, NIO channels transmit data in plaintext, exposing them to eavesdropping, tampering, and man-in-the-middle attacks. To secure NIO-based networking, Java provides the `SSLEngine` API, which allows integrating SSL/TLS encryption and authentication while preserving the non-blocking nature of NIO.

This section explains how to implement secure network communication using Java NIO and `SSLEngine`. We cover:

- The role of `SSLEngine` in TLS integration.
- Setting up a secure `SocketChannel` with `SSLEngine`.
- Performing the SSL/TLS handshake in a non-blocking context.
- Encrypting and decrypting application data.
- Key Java code examples for each step.

12.3.1 Why Use `SSLEngine` for TLS in Java NIO?

Traditional SSL/TLS support in Java (e.g., `SSLSocket`) is blocking and tightly coupled to socket IO. `SSLEngine` separates TLS protocol handling from actual IO and is designed for integration with non-blocking transport layers like `SocketChannel`. It allows developers to:

- Perform TLS handshake and data encryption/decryption manually.
- Manage encrypted and decrypted buffers explicitly.
- Integrate with selectors and non-blocking IO workflows.
- Support scalable secure servers or clients.

12.3.2 Overview: How `SSLEngine` Works with NIO

The key concept is that `SSLEngine` handles TLS protocol logic on byte buffers:

- **Application data** buffers contain plaintext data to send or receive.
- **Network data** buffers contain encrypted TLS packets to be sent or received on the network.

The developer drives the handshake and data processing by repeatedly calling `wrap()` and `unwrap()` methods on `SSLEngine`, moving data between these buffers and the underlying `SocketChannel`.

12.3.3 Setting Up a Secure NIO Channel Using `SocketChannel` and `SSLEngine`

Step 1: Initialize SSL Context and Create SSLEngine

First, create an `SSLContext` initialized with key and trust managers, then create an `SSLEngine` configured as client or server.

```
import javax.net.ssl.*;

SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(keyManagers, trustManagers, null);

SSLEngine sslEngine = sslContext.createSSLEngine(host, port);
sslEngine.setUseClientMode(true); // or false if server
```

Here, `keyManagers` and `trustManagers` manage certificates and trust chains.

Step 2: Create ByteBuffers for Encrypted and Plain Data

You must allocate buffers for outgoing (encrypted) and incoming (encrypted) network data, and for outgoing and incoming application (plaintext) data.

```
SSLSession session = sslEngine.getSession();

ByteBuffer appData = ByteBuffer.allocate(session.getApplicationBufferSize());
ByteBuffer netData = ByteBuffer.allocate(session.getPacketBufferSize());
ByteBuffer peerAppData = ByteBuffer.allocate(session.getApplicationBufferSize());
ByteBuffer peerNetData = ByteBuffer.allocate(session.getPacketBufferSize());
```

- `appData`: plaintext data to send.
- `netData`: encrypted data to send.
- `peerNetData`: encrypted data received from network.
- `peerAppData`: decrypted data received from peer.

12.3.4 Step 3: Establish Non-Blocking SocketChannel

Create and configure the `SocketChannel` for non-blocking mode and connect it.

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress(host, port));

// Use Selector to wait for connect completion and subsequent read/write readiness.
```

12.3.5 Step 4: Perform the TLS Handshake

The handshake involves several `SSLEngineResult.HandshakeStatus` states and requires driving `wrap()` and `unwrap()` calls:

```
sslEngine.beginHandshake();
SSLEngineResult.HandshakeStatus handshakeStatus = sslEngine.getHandshakeStatus();

while (handshakeStatus != SSLEngineResult.HandshakeStatus.FINISHED &&
       handshakeStatus != SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {

    switch (handshakeStatus) {
        case NEED_UNWRAP:
            if (socketChannel.read(peerNetData) < 0) {
                throw new IOException("Channel closed during handshake");
            }
            peerNetData.flip();
            SSLEngineResult unwrapResult = sslEngine.unwrap(peerNetData, peerAppData);
            peerNetData.compact();
            handshakeStatus = unwrapResult.getHandshakeStatus();
            break;

        case NEED_WRAP:
            netData.clear();
            SSLEngineResult wrapResult = sslEngine.wrap(appData, netData);
            handshakeStatus = wrapResult.getHandshakeStatus();
            netData.flip();
            while (netData.hasRemaining()) {
                socketChannel.write(netData);
            }
            break;

        case NEED_TASK:
            Runnable task;
            while ((task = sslEngine.getDelegatedTask()) != null) {
                task.run();
            }
            handshakeStatus = sslEngine.getHandshakeStatus();
            break;

        default:
            throw new IllegalStateException("Invalid handshake status: " + handshakeStatus);
    }
}
```

- **NEED_UNWRAP:** Read encrypted data from the socket and decrypt it.
- **NEED_WRAP:** Encrypt data and send it.
- **NEED_TASK:** Run delegated tasks (CPU-intensive operations like certificate validation).
- Loop until handshake finishes.

12.3.6 Step 5: Secure Data Exchange After Handshake

After a successful handshake, application data can be exchanged securely using `wrap()` and `unwrap()`:

Sending data:

```
appData.clear();
appData.put("Hello secure world".getBytes(StandardCharsets.UTF_8));
appData.flip();

netData.clear();
SSLEngineResult result = sslEngine.wrap(appData, netData);
netData.flip();

while (netData.hasRemaining()) {
    socketChannel.write(netData);
}
```

Receiving data:

```
peerNetData.clear();
int bytesRead = socketChannel.read(peerNetData);
if (bytesRead > 0) {
    peerNetData.flip();
    peerAppData.clear();
    SSLEngineResult result = sslEngine.unwrap(peerNetData, peerAppData);
    peerNetData.compact();

    peerAppData.flip();
    byte[] receivedBytes = new byte[peerAppData.remaining()];
    peerAppData.get(receivedBytes);
    System.out.println("Received: " + new String(receivedBytes, StandardCharsets.UTF_8));
}
```

The `wrap()` method encrypts plaintext data into TLS packets, and `unwrap()` decrypts received TLS packets into plaintext.

Here's a **complete, single runnable Java example** that:

- Creates an SSL/TLS connection using `SocketChannel` and `SSLEngine`
- Performs a full handshake
- Sends and receives secure data
- Uses a self-contained trust manager (for demo purposes only)

WARNING **Note:** For a real secure deployment, you'd load proper certificates. This example

uses a **dummy trust manager** that accepts all certificates for simplicity.

12.3.7 Complete Runnable Java Example (Client-Side)

Full runnable code:

```
import javax.net.ssl.*;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.security.cert.X509Certificate;

public class SecureNioClient {

    public static void main(String[] args) throws Exception {
        String host = "localhost";
        int port = 8443;

        // Step 1: Create SSLContext with dummy TrustManager (INSECURE - for demo only)
        TrustManager[] trustAll = new TrustManager[] {
            new X509TrustManager() {
                public X509Certificate[] getAcceptedIssuers() { return null; }
                public void checkClientTrusted(X509Certificate[] certs, String authType) { }
                public void checkServerTrusted(X509Certificate[] certs, String authType) { }
            }
        };

        SSLContext sslContext = SSLContext.getInstance("TLS");
        sslContext.init(null, trustAll, new SecureRandom());

        SSLEngine sslEngine = sslContext.createSSLEngine(host, port);
        sslEngine.setUseClientMode(true);
        sslEngine.beginHandshake();

        SSLSession session = sslEngine.getSession();

        // Step 2: Allocate Buffers
        ByteBuffer appData = ByteBuffer.allocate(session.getApplicationBufferSize());
        ByteBuffer netData = ByteBuffer.allocate(session.getPacketBufferSize());
        ByteBuffer peerAppData = ByteBuffer.allocate(session.getApplicationBufferSize());
        ByteBuffer peerNetData = ByteBuffer.allocate(session.getPacketBufferSize());

        // Step 3: Open SocketChannel
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.configureBlocking(false);
        socketChannel.connect(new InetSocketAddress(host, port));

        while (!socketChannel.finishConnect()) {
            Thread.sleep(50); // Wait for connection
        }
```

```
        // Step 4: TLS Handshake
        SSLEngineResult.HandshakeStatus handshakeStatus = sslEngine.getHandshakeStatus();

        while (handshakeStatus != SSLEngineResult.HandshakeStatus.FINISHED &&
               handshakeStatus != SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {

            switch (handshakeStatus) {
                case NEED_UNWRAP:
                    if (socketChannel.read(peerNetData) < 0) {
                        throw new IOException("Channel closed during handshake");
                    }
                    peerNetData.flip();
                    SSLEngineResult unwrapResult = sslEngine.unwrap(peerNetData, peerAppData);
                    peerNetData.compact();
                    handshakeStatus = unwrapResult.getHandshakeStatus();
                    break;

                case NEED_WRAP:
                    netData.clear();
                    SSLEngineResult wrapResult = sslEngine.wrap(ByteBuffer.allocate(0), netData);
                    handshakeStatus = wrapResult.getHandshakeStatus();
                    netData.flip();
                    while (netData.hasRemaining()) {
                        socketChannel.write(netData);
                    }
                    break;

                case NEED_TASK:
                    Runnable task;
                    while ((task = sslEngine.getDelegatedTask()) != null) {
                        task.run();
                    }
                    handshakeStatus = sslEngine.getHandshakeStatus();
                    break;

                default:
                    throw new IllegalStateException("Unexpected handshake status: " + handshakeStatus);
            }
        }

        System.out.println("TLS Handshake completed.");

        // Step 5: Send Secure Data
        String message = "Hello secure world!";
        appData.clear();
        appData.put(message.getBytes(StandardCharsets.UTF_8));
        appData.flip();

        netData.clear();
        SSLEngineResult wrapResult = sslEngine.wrap(appData, netData);
        netData.flip();

        while (netData.hasRemaining()) {
            socketChannel.write(netData);
        }

        System.out.println("Sent: " + message);
```

```
        // Step 6: Receive Secure Response
        peerNetData.clear();
        int bytesRead = socketChannel.read(peerNetData);

        if (bytesRead > 0) {
            peerNetData.flip();
            peerAppData.clear();
            SSLEngineResult result = sslEngine.unwrap(peerNetData, peerAppData);
            peerNetData.compact();

            peerAppData.flip();
            byte[] receivedBytes = new byte[peerAppData.remaining()];
            peerAppData.get(receivedBytes);
            System.out.println("Received: " + new String(receivedBytes, StandardCharsets.UTF_8));
        }

        // Cleanup
        sslEngine.closeOutbound();
        socketChannel.close();
    }
}
```

12.3.8 Requirements to Run

- Java 11+
- A TLS server running at `localhost:8443` (can be a simple echo server)
- Add real trust/key managers for production use (instead of the dummy trust manager)

12.3.9 Warning

This demo uses a **dummy `X509TrustManager` that accepts all certificates**, which is insecure and should **never** be used in production. Always validate certificates using a properly configured trust store.

12.3.10 Key Points and Considerations

- **Buffer sizing:** Use buffer sizes from `sslEngine.getSession()` to avoid buffer overflows or underflows.
- **Handling `BUFFER_OVERFLOW` and `BUFFER_UNDERFLOW`:** Check `SSLEngineResult.Status` and adjust buffer sizes or read more data accordingly.
- **Non-blocking and selector integration:** Coordinate channel readiness (read/write) with handshake and data wrap/unwrap cycles.

- **Exception handling:** Handle IOExceptions and `SSLException` to gracefully close connections on failure.
- **Closing SSL connections:** Use `sslEngine.closeOutbound()` and perform a proper SSL/TLS close_notify handshake.

12.3.11 Minimal Secure Client Example Outline

```
// 1. Setup SSLContext, SSLEngine (client mode)
// 2. Create non-blocking SocketChannel and connect
// 3. Perform handshake loop as described
// 4. After handshake, wrap application data and write to channel
// 5. Read encrypted data, unwrap, and process plaintext
// 6. Close connection gracefully with SSL/TLS close_notify
```

12.3.12 Summary

Integrating SSL/TLS into Java NIO applications requires explicit management of encrypted and decrypted buffers via the `SSLEngine` API. While this adds complexity compared to traditional blocking SSL sockets, it enables scalable, secure network applications with non-blocking IO.

The process involves:

- Creating and configuring an `SSLEngine`.
- Allocating appropriate buffers for TLS packet and application data.
- Driving the SSL/TLS handshake with coordinated wrap and unwrap calls.
- Encrypting and decrypting data securely during communication.

Mastering `SSLEngine` unlocks the ability to build high-performance, secure Java servers and clients that benefit from both TLS security and NIO scalability.

12.4 Handling Sensitive Data Streams

Java IO streams are essential for reading and writing data, but handling sensitive information—like passwords, encryption keys, or personal user data—requires extra care. Without proper precautions, sensitive data may be inadvertently exposed through plaintext storage, memory leaks, or insecure transmission channels.

This guide covers best practices for securely handling sensitive data in Java IO streams, focusing on:

- Avoiding plaintext persistence of secrets.
- Encrypting data before storage or transmission.
- Securely clearing buffers to minimize in-memory data exposure.
- Using Java's cryptographic APIs, including `CipherInputStream` and `CipherOutputStream`.
- Example code demonstrating encryption and decryption with streams.

12.4.1 Why Secure Handling Matters

Sensitive data such as passwords, API keys, credit card information, or personally identifiable information (PII) are attractive targets for attackers. Common risks when handling sensitive data via IO streams include:

- Writing secrets directly to disk in plaintext, leaving them accessible to unauthorized users.
- Transmitting sensitive data over unencrypted channels, exposing it to interception.
- Leaving sensitive data in memory buffers, increasing the risk of extraction through memory dumps or side-channel attacks.
- Poorly managing keys or cryptographic material, weakening overall security.

Following secure IO practices reduces these risks and helps comply with privacy regulations and security standards.

12.4.2 Best Practices for Secure IO Handling of Sensitive Data

Avoid Writing Plaintext Secrets to Disk

Never store raw passwords, keys, or tokens in plaintext files. If persistent storage is necessary:

- Use strong encryption to protect data at rest.
- Restrict file permissions to limit access.
- Avoid logging sensitive data accidentally.

Encrypt Data Before Writing to Disk or Transmitting

Encrypt sensitive data with a robust symmetric cipher (e.g., AES) before writing it to disk or sending over the network. Decrypt only when necessary.

- Use a well-tested cryptographic provider (`javax.crypto`).
- Manage encryption keys securely (never hardcode keys).
- Consider authenticated encryption (e.g., AES-GCM) to ensure confidentiality and integrity.

Securely Clear Buffers After Use

Buffers holding sensitive data in memory should be cleared immediately after use to prevent lingering secrets:

- Overwrite byte arrays or char arrays with zeros or random data.
- Avoid using immutable objects (like `String`) for sensitive data, since they cannot be erased from memory.

Use `CipherInputStream` and `CipherOutputStream` for Stream Encryption

These classes allow you to transparently encrypt or decrypt data as it flows through Java IO streams.

12.4.3 Encrypting and Decrypting Data with Java IO Streams

Here's how to implement stream encryption and decryption securely with `CipherOutputStream` and `CipherInputStream`.

12.4.4 Setup: Create an AES Key and Cipher

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import java.io.*;
import java.security.SecureRandom;

public class SecureStreamExample {

    private static final String AES = "AES";
    private static final String TRANSFORMATION = "AES/GCM/NoPadding";
    private static final int GCM_TAG_LENGTH = 16; // bytes
    private static final int GCM_IV_LENGTH = 12;  // bytes

    // Generate a random AES key
    public static SecretKey generateKey() throws Exception {
        KeyGenerator keyGen = KeyGenerator.getInstance(AES);
        keyGen.init(256); // Use 256-bit AES key (requires JCE Unlimited Strength)
        return keyGen.generateKey();
    }

    // Generate a secure random IV (nonce)
    public static byte[] generateIV() {
        byte[] iv = new byte[GCM_IV_LENGTH];
        new SecureRandom().nextBytes(iv);
```

```
        return iv;
    }
```

12.4.5 Encrypt Data to File Using `CipherOutputStream`

```
    public static void encryptToFile(byte[] plaintext, File outputFile, SecretKey key) throws Exception
        byte[] iv = generateIV();

        Cipher cipher = Cipher.getInstance(TRANSFORMATION);
        GCMParameterSpec spec = new GCMParameterSpec(GCM_TAG_LENGTH * 8, iv);
        cipher.init(Cipher.ENCRYPT_MODE, key, spec);

        try (FileOutputStream fos = new FileOutputStream(outputFile);
             // Write IV at the beginning of the file for later decryption
             BufferedOutputStream bos = new BufferedOutputStream(fos);
             CipherOutputStream cos = new CipherOutputStream(bos, cipher)) {

            bos.write(iv);  // prepend IV for use during decryption
            cos.write(plaintext);
        }

        // Securely clear plaintext buffer
        java.util.Arrays.fill(plaintext, (byte) 0);
    }
```

This method:

- Generates a secure random IV.
- Prepends the IV to the output file (needed for decryption).
- Wraps the `FileOutputStream` in a `CipherOutputStream` for transparent encryption.
- Overwrites the plaintext buffer after writing.

12.4.6 Decrypt Data from File Using `CipherInputStream`

```
    public static byte[] decryptFromFile(File inputFile, SecretKey key) throws Exception {
        try (FileInputStream fis = new FileInputStream(inputFile);
             BufferedInputStream bis = new BufferedInputStream(fis)) {

            // Read the IV from the file header
            byte[] iv = new byte[GCM_IV_LENGTH];
            if (bis.read(iv) != GCM_IV_LENGTH) {
                throw new IllegalStateException("Invalid input file format");
            }

            Cipher cipher = Cipher.getInstance(TRANSFORMATION);
            GCMParameterSpec spec = new GCMParameterSpec(GCM_TAG_LENGTH * 8, iv);
            cipher.init(Cipher.DECRYPT_MODE, key, spec);
```

```
            try (CipherInputStream cis = new CipherInputStream(bis, cipher);
                 ByteArrayOutputStream baos = new ByteArrayOutputStream()) {

                byte[] buffer = new byte[4096];
                int bytesRead;
                while ((bytesRead = cis.read(buffer)) != -1) {
                    baos.write(buffer, 0, bytesRead);
                }

                byte[] decrypted = baos.toByteArray();

                // Securely clear intermediate buffer
                java.util.Arrays.fill(buffer, (byte) 0);

                return decrypted;
            }
        }
    }
}
```

This method:

- Reads the IV from the start of the file.
- Initializes a cipher for decryption.
- Wraps the input stream in a `CipherInputStream` to decrypt transparently.
- Reads decrypted data into a byte array.
- Clears temporary buffers after use.

Full runnable code:

```
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;
import java.io.*;
import java.security.SecureRandom;
import java.util.Arrays;

public class SecureStreamExample {

    private static final String AES = "AES";
    private static final String TRANSFORMATION = "AES/GCM/NoPadding";
    private static final int GCM_TAG_LENGTH = 16; // in bytes
    private static final int GCM_IV_LENGTH = 12;  // in bytes

    public static void main(String[] args) throws Exception {
        SecretKey key = generateKey();
        File file = new File("encrypted.dat");

        String message = "This is a top-secret message.";
        byte[] plaintext = message.getBytes();

        encryptToFile(plaintext, file, key);
        byte[] decrypted = decryptFromFile(file, key);
```

```
        System.out.println("Decrypted message: " + new String(decrypted));
    }

    // Generate a 256-bit AES key
    public static SecretKey generateKey() throws Exception {
        KeyGenerator keyGen = KeyGenerator.getInstance(AES);
        keyGen.init(256);
        return keyGen.generateKey();
    }

    // Generate secure random IV
    public static byte[] generateIV() {
        byte[] iv = new byte[GCM_IV_LENGTH];
        new SecureRandom().nextBytes(iv);
        return iv;
    }

    // Encrypt data and write to file
    public static void encryptToFile(byte[] plaintext, File outputFile, SecretKey key) throws Exception
        byte[] iv = generateIV();

        Cipher cipher = Cipher.getInstance(TRANSFORMATION);
        GCMParameterSpec spec = new GCMParameterSpec(GCM_TAG_LENGTH * 8, iv);
        cipher.init(Cipher.ENCRYPT_MODE, key, spec);

        try (FileOutputStream fos = new FileOutputStream(outputFile);
             BufferedOutputStream bos = new BufferedOutputStream(fos)) {

            bos.write(iv); // prepend IV

            try (CipherOutputStream cos = new CipherOutputStream(bos, cipher)) {
                cos.write(plaintext);
            }
        }

        Arrays.fill(plaintext, (byte) 0); // Clear sensitive data
    }

    // Decrypt data from file
    public static byte[] decryptFromFile(File inputFile, SecretKey key) throws Exception {
        try (FileInputStream fis = new FileInputStream(inputFile);
             BufferedInputStream bis = new BufferedInputStream(fis)) {

            byte[] iv = new byte[GCM_IV_LENGTH];
            if (bis.read(iv) != GCM_IV_LENGTH) {
                throw new IllegalStateException("Invalid file format: IV missing");
            }

            Cipher cipher = Cipher.getInstance(TRANSFORMATION);
            GCMParameterSpec spec = new GCMParameterSpec(GCM_TAG_LENGTH * 8, iv);
            cipher.init(Cipher.DECRYPT_MODE, key, spec);

            try (CipherInputStream cis = new CipherInputStream(bis, cipher);
                 ByteArrayOutputStream baos = new ByteArrayOutputStream()) {

                byte[] buffer = new byte[4096];
                int bytesRead;
                while ((bytesRead = cis.read(buffer)) != -1) {
```

```
                    baos.write(buffer, 0, bytesRead);
                }

                Arrays.fill(buffer, (byte) 0);
                return baos.toByteArray();
            }
        }
    }
}
```

12.4.7 Additional Tips for Secure Stream Handling

- **Avoid using `String` for secrets:** Store sensitive data in mutable `char[]` or `byte[]` so you can overwrite it.
- **Use try-with-resources:** Always close streams promptly to flush and release resources.
- **Use authenticated encryption:** Modes like AES-GCM provide both confidentiality and integrity.
- **Secure key management:** Store keys securely (e.g., hardware security modules, encrypted key stores) and never hardcode them.
- **Limit buffer sizes:** Large buffers increase memory footprint; choose sizes balancing performance and security.

12.4.8 Summary

Handling sensitive data securely in Java IO streams involves:

- Never writing secrets in plaintext to disk or network.
- Using Java Cryptography Architecture (`javax.crypto`) to encrypt and decrypt data with `CipherOutputStream` and `CipherInputStream`.
- Generating secure random IVs/nonces and using authenticated encryption modes.
- Securely clearing memory buffers after use to minimize exposure.
- Managing encryption keys carefully and restricting access to encrypted files.

By following these practices and using the Java cryptographic APIs properly, you can protect sensitive data throughout its lifecycle in your Java applications.

Chapter 13.

New Features and Future of Java IO/NIO

1. Updates in Java 11 and Later Versions
2. Project Loom and Virtual Threads Impact
3. Reactive Streams and IO
4. Upcoming Improvements and Alternatives

13 New Features and Future of Java IO/NIO

13.1 Updates in Java 11 and Later Versions

Since Java 11's release in 2018, the Java platform has steadily introduced improvements and new features related to IO (Input/Output) and NIO (New IO) APIs. These enhancements aim to simplify file and stream handling, improve performance, expand support for modern file formats, and offer more robust tools for developers working with IO-intensive applications.

This overview covers key IO/NIO API enhancements from Java 11 through the latest stable Java release (Java 21 at time of writing), focusing on:

- Enhancements to `java.nio.file` package
- Additions and improvements to `InputStream` and related stream classes
- Support for new file types and attributes
- Performance and usability improvements

13.1.1 Enhancements in `java.nio.file` Package

a) Reading/Writing Small Files Conveniently (Java 11)

Java 11 introduced new methods in the `Files` utility class to read and write small files with ease.

- **`Files.readString(Path path)`** — Reads all content from a file into a `String` with UTF-8 encoding by default.
- **`Files.writeString(Path path, CharSequence csq, OpenOption... options)`** — Writes a `String` directly to a file.

These methods simplify common IO patterns, reducing boilerplate code.

```
Path path = Path.of("example.txt");

// Read entire file as a String
String content = Files.readString(path);
System.out.println(content);

// Write a String to a file
Files.writeString(path, "Hello, Java 11+", StandardOpenOption.CREATE);
```

This is more concise than using `BufferedReader` or `BufferedWriter` and avoids explicit charset specification for UTF-8.

b) New File Attribute Views and File Types (Java 12)

Java 12 introduced improvements to file attribute handling, including support for additional file attributes such as:

- **`DosFileAttributes`** and **`DosFileAttributeView`** extended for finer DOS/Windows

file attribute manipulation.
- Support for symbolic links and junction points improved in Windows environments.
- **`UnixFileAttributes`** enhanced for better POSIX compliance.

This enables developers to write more portable file-handling code, dealing with platform-specific file features more seamlessly.

c) `Path.of` and Related Factory Methods (Java 11)

Java 11 introduced static factory methods for `Path`, making it easier and cleaner to obtain `Path` instances:

```
Path p = Path.of("dir", "subdir", "file.txt");
```

This replaces older `Paths.get(...)` calls, offering a more fluent and intuitive API.

d) Improved Support for Temporary Files and Directories (Java 12)

Java 12 enhanced the `Files` API with better handling of temporary files and directories, including options for setting file attributes atomically during creation, and better default permissions.

```
Path tempDir = Files.createTempDirectory("myapp", PosixFilePermissions.asFileAttribute(
    PosixFilePermissions.fromString("rwx------")));
```

This ensures secure temporary storage, preventing race conditions or unauthorized access.

13.1.2 `InputStream` and Related Stream Improvements

a) `InputStream.transferTo(OutputStream)` Method (Java 9, used widely post-Java 11)

While introduced in Java 9, `InputStream.transferTo()` became a widely adopted utility in Java 11+ projects.

This method copies all bytes from an `InputStream` to an `OutputStream` efficiently and with minimal code:

```
try (InputStream in = Files.newInputStream(path);
     OutputStream out = System.out) {
    in.transferTo(out);
}
```

This simplifies stream copying tasks, replacing verbose buffer copy loops with a single call.

b) `InputStream.readAllBytes()` and `readNBytes()` (Java 9)

Similarly, `InputStream.readAllBytes()` reads all bytes into a byte array, simplifying common tasks like reading entire files or network streams.

```
byte[] data = Files.newInputStream(path).readAllBytes();
```

The `readNBytes(int len)` method allows reading a specific number of bytes safely.

13.1.3 Support for New File Types and Enhanced File System Features

a) ZIP File System Enhancements

Java's support for ZIP file systems (`java.nio.file.FileSystem` provider for ZIP/JAR files) was enhanced in recent releases:

- Better support for reading and modifying ZIP entries.
- Ability to mount ZIP files as file systems using `FileSystems.newFileSystem()` with extended options.

This allows applications to treat ZIP/JAR files like regular file systems, improving flexibility.

```
try (FileSystem zipfs = FileSystems.newFileSystem(zipPath, null)) {
    Path fileInsideZip = zipfs.getPath("/doc/readme.txt");
    String content = Files.readString(fileInsideZip);
    System.out.println(content);
}
```

b) Improved Symlink Handling and File Copy Options

New options for copying symbolic links and better symlink support help avoid common pitfalls when working across different platforms and file systems.

13.1.4 IO Performance and Usability Improvements

a) Improved Buffer Allocation and Management

Later Java releases included internal improvements in NIO buffer management and memory allocation, reducing overhead and improving throughput in high-load IO scenarios. While these are mostly transparent to the developer, they enhance performance in applications using `ByteBuffer` extensively.

b) Better Asynchronous File IO

Enhancements in asynchronous file IO APIs (`AsynchronousFileChannel`) improved scalability and integration with `CompletableFuture`, simplifying asynchronous programming patterns.

13.1.5 Benefits for Developers

- **Less boilerplate:** Methods like `Files.readString()` and `writeString()` reduce code verbosity and improve readability.
- **Better platform support:** Extended file attribute APIs and symlink support make code more portable and reliable.
- **Enhanced security:** More control over file permissions and secure temp file handling help write safer code.
- **Modern API style:** Factory methods like `Path.of()` offer cleaner, more intuitive coding patterns.
- **Performance gains:** Internal buffer and async IO improvements benefit applications handling large volumes of data or requiring high concurrency.

13.1.6 Summary

Since Java 11, the IO/NIO ecosystem has evolved with a focus on developer productivity, security, and performance:

Feature	Description	Java Version
`Files.readString()` and `writeString()`	Simplify text file IO with default UTF-8	11
`Path.of()` factory methods	Cleaner creation of Path instances	11
Enhanced file attribute views	Better POSIX/DOS file attribute support	12+
Improved temporary file handling	Secure temp file/directory creation	12+
`InputStream.transferTo()` and `readAllBytes()`	Easier stream data copying and reading	9+ (commonly used post-11)
ZIP FileSystem improvements	Treat ZIP files as file systems more flexibly	11+
Async IO enhancements	Better asynchronous file channel support	12+
Buffer and memory optimizations	Improved NIO buffer management and throughput	11+

These improvements collectively modernize Java's IO APIs, enabling developers to write concise, efficient, and secure IO code aligned with today's application demands.

13.2 Project Loom and Virtual Threads Impact

Java's concurrency and IO models have long relied on **platform (OS) threads** and either blocking or non-blocking IO APIs. While this model has been powerful, it also introduces challenges in scalability and complexity, especially for IO-heavy applications such as web servers, microservices, or reactive systems.

Project Loom, an ongoing OpenJDK project, aims to revolutionize Java concurrency by introducing **virtual threads**—lightweight user-mode threads that enable massive concurrency with a familiar, simple programming model. This fundamentally changes how Java handles IO, impacting both traditional blocking IO and NIO's non-blocking mechanisms.

13.2.1 The Problem with Traditional Thread-Based IO

In traditional Java IO, each blocking operation (like reading from a socket or file) blocks the underlying operating system thread until the operation completes. OS threads are relatively heavy-weight:

- They consume significant memory (stack space per thread).
- They require costly context switches managed by the OS.
- Their numbers are limited by OS and hardware constraints.

13.2.2 Scalability Bottleneck

Suppose a server needs to handle thousands of concurrent connections. Using one OS thread per connection quickly becomes unmanageable due to:

- Excessive memory consumption.
- Scheduling overhead and context switching.
- Potential for thread starvation and increased latency.

To mitigate this, Java introduced **NIO (Non-blocking IO)** with selectors and multiplexing. NIO lets one or few threads manage many connections by polling readiness events and using callbacks or futures. While scalable, NIO's model brings complexity:

- Requires intricate state machines and callback logic.
- Often leads to more complex, harder-to-maintain code.
- Can suffer from head-of-line blocking and fairness issues.

13.2.3 What is Project Loom?

Project Loom introduces **virtual threads**—lightweight threads managed by the Java runtime rather than the OS. They aim to make concurrency scalable **without sacrificing the simplicity of blocking code**.

13.2.4 Key Characteristics of Virtual Threads

- **Extremely lightweight:** Can have hundreds of thousands or even millions of virtual threads.
- **Managed by JVM:** The JVM schedules them onto a smaller pool of OS threads.
- **Blocking operations are non-blocking under the hood:** When a virtual thread blocks (e.g., on IO), the JVM parks that virtual thread and frees the OS thread to do other work.
- **Familiar API:** Virtual threads use the same `java.lang.Thread` API, so existing blocking code works without modification.

13.2.5 How Virtual Threads Affect Java IO and NIO

Traditional Blocking IO with Virtual Threads

With virtual threads, you can write simple, blocking IO code per connection without worrying about OS thread exhaustion.

```
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

executor.submit(() -> {
    try (Socket socket = serverSocket.accept();
         BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {
        String line = in.readLine(); // Blocking call but lightweight
        System.out.println("Received: " + line);
    } catch (IOException e) {
        e.printStackTrace();
    }
});
```

- Each connection runs in its own virtual thread.
- Blocking calls like `readLine()` only block the virtual thread.
- The underlying OS thread is freed during blocking, allowing other virtual threads to run.
- This greatly simplifies code while enabling massive concurrency.

Full runnable code:

```java
import java.io.*;
import java.net.*;
import java.util.concurrent.*;

public class VirtualThreadEchoServer {
    public static void main(String[] args) throws IOException {
        ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Server started on port 5000");

            while (true) {
                Socket clientSocket = serverSocket.accept(); // Blocking, lightweight on virtual thread
                executor.submit(() -> handleClient(clientSocket));
            }
        }
    }

    private static void handleClient(Socket socket) {
        try (socket;
             BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
             BufferedWriter out = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())))

            String line;
            while ((line = in.readLine()) != null) {
                System.out.println("Received: " + line);
                out.write("Echo: " + line + "\n");
                out.flush();
            }

        } catch (IOException e) {
            System.err.println("Connection error: " + e.getMessage());
        }
    }
}
```

NIOs Role in a Loom World

NIO's non-blocking model still exists and is useful, especially in legacy or performance-critical applications. However:

- Virtual threads reduce the need to use complex selector-based IO.
- Non-blocking IO's complexity can often be avoided by writing straightforward blocking code atop virtual threads.
- You still get scalability without intricate event-driven state machines.

Conceptual Diagram

```
Traditional Model:

[ OS Threads (limited, heavy) ]
     |-- blocking IO --> OS thread blocks, wastes resource

NIO Model:
```

```
[ Few OS Threads ]
    |-- Selector polls events
    |-- Callbacks or futures handle readiness
    |-- Complex state machine

Project Loom Model:

[ Many Virtual Threads (lightweight) ]
    |-- Blocking IO in virtual thread
    |-- JVM parks virtual thread during blocking
    |-- OS thread reused for other virtual threads
    |-- Simple sequential code, massive concurrency
```

13.2.6 Comparing Virtual Threads and NIOs Non-Blocking IO

Aspect	Virtual Threads	NIO Non-Blocking IO
Programming Model	Simple, imperative, blocking calls	Complex, callback/future-based, event-driven
Code Complexity	Low — straightforward sequential code	High — requires explicit state management
Scalability	Very high—millions of virtual threads	High—few threads multiplex many connections
Performance	Slight overhead from scheduling virtual threads	High throughput but overhead managing readiness
Use Cases	General-purpose concurrency, legacy blocking APIs, rapid development	Performance critical apps, custom event loops
Error Handling	Simple try/catch, natural stack traces	Harder to track errors across callbacks

13.2.7 When to Use Virtual Threads vs. NIO

Use Virtual Threads When:

- You want to write simple, blocking-style code.
- You're modernizing or writing new IO-heavy applications.
- You want rapid development without complex callback management.
- You need to scale to many concurrent connections or tasks.

Use NIO When:

- You need extreme fine-tuned performance and minimal latency.
- You want to integrate with existing NIO-based frameworks or libraries.
- You have special requirements for multiplexing or protocol-level customization.

In many cases, Loom's virtual threads can replace NIO, making code easier and safer without sacrificing scalability.

13.2.8 Example: Traditional NIO vs. Loom Virtual Threads

Traditional NIO Server Skeleton (simplified)

```
Selector selector = Selector.open();
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.bind(new InetSocketAddress(8080));
serverChannel.configureBlocking(false);
serverChannel.register(selector, SelectionKey.OP_ACCEPT);

while (true) {
    selector.select();
    Set<SelectionKey> keys = selector.selectedKeys();
    for (SelectionKey key : keys) {
        if (key.isAcceptable()) {
            SocketChannel client = serverChannel.accept();
            client.configureBlocking(false);
            client.register(selector, SelectionKey.OP_READ);
        } else if (key.isReadable()) {
            SocketChannel client = (SocketChannel) key.channel();
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            int read = client.read(buffer);
            // Handle data (non-blocking, complex)
        }
    }
    keys.clear();
}
```

13.2.9 Loom Virtual Threads Server Skeleton

```
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
ServerSocket serverSocket = new ServerSocket(8080);

while (true) {
    Socket socket = serverSocket.accept(); // blocking, but lightweight virtual thread
    executor.submit(() -> {
        try (BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()))
            String line = reader.readLine(); // blocking call per connection
            System.out.println("Received: " + line);
        }
```

```
    });
}
```

The Loom example is shorter, easier to understand, and scales easily without manual multiplexing.

13.2.10 Summary

Project Loom fundamentally changes Java concurrency by introducing **virtual threads**, a lightweight, scalable alternative to OS threads. This innovation allows developers to write simple blocking IO code while achieving scalability that previously required complex NIO-based non-blocking code.

Traditional IO	Heavy OS threads, limited concurrency, blocking IO limits scalability
NIO (non-blocking)	Efficient multiplexing, complex programming model, event-driven
Project Loom	Massive virtual threads, simple blocking code, scalable and easy

In many new applications, Loom's virtual threads simplify development, maintain readability, and deliver performance comparable to NIO's non-blocking approach. However, NIO remains relevant for legacy codebases and specialized use cases requiring explicit control over IO multiplexing.

13.3 Reactive Streams and IO

Modern applications increasingly demand efficient, scalable, and responsive data processing pipelines—especially when handling asynchronous IO such as network requests, file streaming, or user interactions. **Reactive streams** and the **reactive programming model** have emerged as powerful paradigms to address these demands by providing a standardized way to handle asynchronous data flows with backpressure, composability, and declarative APIs.

This section introduces reactive streams, explains Java's built-in `Flow` API introduced in Java 9, explores how reactive programming complements or replaces NIO, and demonstrates practical reactive IO examples using standard Java and popular libraries like Reactor and RxJava.

13.3.1 What is Reactive Programming?

Reactive programming is a declarative programming paradigm oriented around data streams and the propagation of change. Instead of writing imperative code that explicitly manages threads and callbacks, reactive programming allows you to compose asynchronous, event-driven data flows that react to new data, errors, or completion signals.

13.3.2 Core Concepts

- **Publisher:** A source that emits a stream of data asynchronously.
- **Subscriber:** A consumer that processes data emitted by the publisher.
- **Backpressure:** A mechanism by which the subscriber can control the rate at which data is produced, preventing overload.
- **Operators:** Functions that transform, filter, combine, or otherwise manipulate streams.

Reactive programming helps write **non-blocking**, **scalable**, and **resilient** IO-bound applications that can efficiently handle high concurrency without thread exhaustion.

13.3.3 The Java `Flow` API Reactive Streams in Java SE 9

Java 9 introduced the `java.util.concurrent.Flow` API as a standard, minimal reactive streams framework embedded in the JDK. It was inspired by the Reactive Streams specification.

The `Flow` API consists of four core interfaces:

- **`Flow.Publisher<T>`:** Produces data asynchronously.
- **`Flow.Subscriber<T>`:** Consumes data asynchronously.
- **`Flow.Subscription`:** Represents a one-to-one lifecycle between a Publisher and Subscriber and controls data flow.
- **`Flow.Processor<T,R>`:** A processing stage that acts as both Subscriber and Publisher.

13.3.4 Example: Simple Publisher and Subscriber Using `Flow`

Full runnable code:

```
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;

public class SimpleFlowExample {
    public static void main(String[] args) throws Exception {
```

```
        SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

        Flow.Subscriber<String> subscriber = new Flow.Subscriber<>() {
            private Flow.Subscription subscription;
            @Override
            public void onSubscribe(Flow.Subscription subscription) {
                this.subscription = subscription;
                subscription.request(1); // request one item initially
            }
            @Override
            public void onNext(String item) {
                System.out.println("Received: " + item);
                subscription.request(1); // request next item
            }
            @Override
            public void onError(Throwable throwable) {
                throwable.printStackTrace();
            }
            @Override
            public void onComplete() {
                System.out.println("Done");
            }
        };

        publisher.subscribe(subscriber);

        publisher.submit("Hello");
        publisher.submit("Reactive Streams");
        publisher.close();

        Thread.sleep(100); // wait for completion
    }
}
```

This example demonstrates:

- Creating a `SubmissionPublisher` which is a built-in Publisher.
- Defining a Subscriber that requests and processes items one-by-one.
- Backpressure is managed by controlling request amounts.

13.3.5 Reactive Streams and NIO: Complement or Replacement?

Complementary Roles

- **NIO (Non-blocking IO)** provides a low-level, event-driven, scalable way to handle multiple IO channels without blocking threads.
- **Reactive Streams/Programming** builds on top of such non-blocking mechanisms but provides a higher-level abstraction focused on asynchronous data flow, composition, and backpressure.

Reactive streams do not replace NIO at the OS or channel level but often **wrap or build on top of NIO** to provide easier-to-use and composable APIs for asynchronous IO.

When to Use Reactive Programming vs NIO

Use Case	Reactive Streams	NIO
Complex asynchronous pipelines	Ideal — supports rich operators	Low-level, requires manual management
Backpressure support	Built-in	Needs manual implementation
Composability and transformations	Extensive via operators	Limited, requires custom code
Integration with frameworks	Widely used (Spring WebFlux, Reactor, RxJava)	Used internally by many libraries
Performance critical low-level IO	Can delegate to NIO underneath	Direct low-level control

13.3.6 Practical Reactive IO Examples

Example 1: Reactive File Reading Using Reactor

Reactor is a popular reactive library built on Project Reactor.

Full runnable code:

```
import reactor.core.publisher.Flux;
import reactor.core.scheduler.Schedulers;

import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class ReactorFileRead {
    public static void main(String[] args) throws Exception {
        Flux<String> lines = Flux.using(
                () -> Files.lines(Paths.get("example.txt")),
                Flux::fromStream,
                Stream::close
        );

        lines.subscribeOn(Schedulers.boundedElastic())  // IO thread pool
             .subscribe(
                 line -> System.out.println("Read line: " + line),
                 Throwable::printStackTrace,
                 () -> System.out.println("Read complete")
             );

        Thread.sleep(1000); // keep main thread alive
    }
}
```

- Uses `Flux` to model the stream of lines from a file.

- Performs file IO on a bounded elastic scheduler designed for blocking tasks.
- Provides asynchronous, non-blocking consumption of file lines.

13.3.7 Example 2: Reactive HTTP Client with Java 11s `HttpClient`

Java 11 introduced a new `HttpClient` that supports reactive-style asynchronous calls returning `CompletableFuture`.

Full runnable code:

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class ReactiveHttpExample {
    public static void main(String[] args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
                .uri(new URI("https://jsonplaceholder.typicode.com/posts"))
                .build();

        client.sendAsync(request, HttpResponse.BodyHandlers.ofLines())
              .thenAccept(response -> {
                  response.body().forEach(System.out::println);
                  System.out.println("Response fully consumed");
              })
              .join();
    }
}
```

This demonstrates a reactive-style HTTP client that processes streamed response lines asynchronously using Java's built-in `Flow` API.

13.3.8 Summary and Benefits of Reactive Streams in Modern IO

Reactive streams provide a powerful abstraction for asynchronous, event-driven IO with explicit backpressure and composability. The `Flow` API introduced in Java 9 brought a standard reactive streams foundation into the JDK, while libraries like Reactor and RxJava offer rich ecosystems and operators for practical applications.

Compared to traditional Java NIO, reactive streams simplify asynchronous IO by:

- Eliminating complex callback hell.
- Providing clear, composable operators for transformations.
- Handling backpressure natively.
- Enabling declarative pipelines for IO operations like file reading, HTTP calls, or message

processing.

Reactive programming does not replace NIO but builds on top of it, making asynchronous IO more accessible and maintainable.

13.4 Upcoming Improvements and Alternatives

Java's IO and NIO (New IO) APIs have been foundational to its success in building scalable, performant applications that handle file systems, networks, and asynchronous data streams. As the computing landscape evolves—with cloud-native architectures, microservices, and ultra-high concurrency becoming mainstream—the Java ecosystem is actively exploring and shaping the future of IO and NIO to meet these demands.

This section offers a forward-looking overview of upcoming improvements in the OpenJDK, community-driven enhancements, performance-focused proposals, and alternative approaches outside the JDK. Finally, it provides practical guidance for developers to prepare for these future shifts.

13.4.1 Upcoming Improvements in OpenJDK IO/NIO

Project Loom: Virtual Threads and Their IO Impact

While officially still in preview or incubation phases as of recent Java releases, **Project Loom** is set to fundamentally transform Java concurrency and IO by introducing **virtual threads**—lightweight user-mode threads that can scale to millions without the overhead of OS threads.

- Loom enables traditional **blocking IO calls** to become highly scalable by having the JVM schedule virtual threads efficiently.
- This can simplify IO programming by largely **replacing complex asynchronous NIO models** for many use cases.
- Future Java versions will likely offer deeper integration of virtual threads with NIO channels and asynchronous APIs, improving usability and performance.

Enhanced Asynchronous File IO and Network IO APIs

There are ongoing discussions about expanding and refining asynchronous file and network IO APIs, making them easier to use and more performant:

- Improved `AsynchronousFileChannel` and `AsynchronousSocketChannel` APIs that better integrate with virtual threads and reactive paradigms.
- Extended support for **direct memory access**, zero-copy transfers, and lower latency network operations.

- More flexible and powerful completion handlers, futures, and promises tailored for modern concurrency models.

Better Native Interoperability and OS Integration

The OpenJDK community is exploring ways to improve native IO performance and interoperability with underlying operating systems:

- Leveraging newer OS features like io_uring on Linux for more efficient asynchronous IO.
- Providing native bindings or enhancements to better utilize platform-specific optimizations.
- Improving file system event watching and file attribute access through enhanced APIs.

13.4.2 Community-Driven Enhancements and Performance Proposals

Project Panama and Foreign Memory Access API

While primarily focused on native interoperability, **Project Panama** indirectly influences IO performance by enabling safer, more efficient access to off-heap memory and native IO buffers. This can:

- Reduce GC pressure in IO-heavy applications.
- Allow zero-copy and direct IO operations with less overhead.
- Facilitate integration with high-performance native IO libraries.

Enhanced Buffer and Memory Management

There is ongoing work to improve **buffer allocation strategies** and **memory management** in NIO:

- Smarter pooling of direct and heap buffers to reduce GC overhead.
- Better diagnostics and tuning options for buffer usage.
- Potential future APIs for explicit buffer lifecycle management.

Reactive Streams and Project Reactor Evolution

Reactive streams libraries like Reactor and RxJava continue to push the envelope on reactive IO performance and expressiveness, often pioneering patterns and optimizations that influence Java's native APIs.

- Community proposals aim to bridge reactive APIs more seamlessly with standard Java NIO channels and virtual threads.
- Enhancements to backpressure handling and resource management are evolving rapidly.

Alternative Approaches: High-Performance Third-Party IO Libraries

While the JDK continues to evolve, many high-performance applications rely on **third-party IO frameworks** that offer advanced features today:

13.4.3 Netty

Netty is a widely-used asynchronous event-driven network application framework that provides:

- Highly optimized, scalable IO handling built atop Java NIO.
- Customizable pipeline architecture with rich protocol support.
- Efficient buffer management and zero-copy features.
- Seamless integration with reactive frameworks and Loom virtual threads (in recent versions).

Netty remains a de facto standard for building scalable network servers and clients with complex protocols, often outperforming plain Java NIO usage.

13.4.4 Other Notable Libraries

- **Vert.x:** A polyglot event-driven toolkit that offers reactive IO and clustering support.
- **Akka Streams:** A toolkit for building reactive streaming applications on the JVM.
- **Apache MINA:** An IO framework similar to Netty, focusing on ease of use and extensibility.

These frameworks often provide better abstractions, built-in performance optimizations, and community-tested patterns that Java's standard libraries are gradually incorporating.

13.4.5 Preparing for Future IO and NIO Shifts: Practical Developer Guidance

Embrace Virtual Threads Early

- Experiment with **Project Loom's early builds** or preview features to understand virtual threads' programming model.
- Prototype IO-bound services with blocking IO code on virtual threads to evaluate scalability and simplify codebases.

Adopt Reactive Programming Practices

- Learn and apply **reactive streams** and reactive libraries such as Reactor or RxJava.
- Understand backpressure, asynchronous data flows, and how they integrate with existing

IO APIs.
- This prepares you to leverage evolving Java APIs and third-party reactive enhancements.

Stay Informed About OpenJDK Enhancements

- Track OpenJDK JEPs (JDK Enhancement Proposals) related to IO, such as:
  - Loom (Virtual Threads)
  - Panama (Foreign Memory & Native IO)
  - IO API enhancements and async improvements
- Participate in community discussions and testing programs to influence the evolution.

Focus on Efficient Buffer and Memory Usage

- Profile and optimize your applications for buffer allocation patterns.
- Use direct buffers judiciously and consider buffer pooling or off-heap memory where appropriate.
- Prepare to adopt new APIs and tools that offer explicit control over buffer lifecycle.

Continue Leveraging High-Performance Libraries

- Don't wait for standard APIs alone; adopt battle-tested libraries like Netty for production systems requiring maximum throughput and flexibility.
- Monitor how these libraries evolve to integrate new Java platform features like virtual threads and Panama.

13.4.6 Conclusion

The future of Java IO and NIO promises exciting advancements driven by OpenJDK projects like **Loom** and **Panama**, alongside active community contributions focused on performance, usability, and native integration. Virtual threads will simplify concurrency models while maintaining scalability, and reactive streams will enhance asynchronous data processing.

At the same time, mature third-party frameworks like Netty will continue to push performance boundaries and provide advanced abstractions. Developers who stay current with emerging APIs, embrace reactive programming, and experiment with virtual threads will be well positioned to leverage the next generation of Java IO capabilities—building applications that are more scalable, maintainable, and performant in the cloud-native era.