Chapter 21: Java I/O and NIO

Introduction

Efficient data input and output operations are crucial in any programming language, and Java provides a robust set of APIs to handle them. Java I/O (Input/Output) and NIO (New Input/Output) are the two primary frameworks used to perform file, stream, and buffer operations. This chapter dives deep into both classic Java I/O and the newer, more performant Java NIO package introduced in JDK 1.4, which offers advanced features such as non-blocking I/O, memory-mapped files, channels, and selectors.

21.1 Java I/O (java.io Package)

21.1.1 Streams in Java

Java I/O uses streams to read and write data:

- **Byte Streams** – For handling raw binary data (classes under `InputStream` and `OutputStream`).
- **Character Streams** – For handling textual data (classes under `Reader` and `Writer`).

Common Byte Stream Classes

Class	Description
`FileInputStream`	Reads raw bytes from a file
`FileOutputStream`	Writes raw bytes to a file
`BufferedInputStream` / `BufferedOutputStream`	Wraps streams for efficient buffered I/O
`DataInputStream` / `DataOutputStream`	Reads/writes Java primitives in a machine-independent way

Common Character Stream Classes

Class	Description
`FileReader` / `FileWriter`	Reads/writes characters from/to a file
`BufferedReader` / `BufferedWriter`	Buffers character streams
`PrintWriter`	Conveniently writes formatted text

21.1.2 File Class

The `java.io.File` class represents a file or directory path in an abstract manner.

```
File file = new File("example.txt");
if (file.exists()) {
```

```
    System.out.println("File exists at: " + file.getAbsolutePath());
}
```

21.1.3 Serialization

Serialization allows saving the state of an object.

```
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data.ser"));
out.writeObject(someObject);
out.close();
```

21.2 Java NIO (java.nio Package)

Java NIO offers a more flexible and scalable I/O framework using buffers and channels.

21.2.1 Key Concepts in NIO

- **Buffers** – Containers for data of a specific primitive type.
- **Channels** – Bi-directional data transfer between buffers and I/O devices.
- **Selectors** – Handle multiple channels using a single thread (non-blocking I/O).
- **Paths and Files** – Introduced in Java 7 (`java.nio.file.*`) to replace `File`.

21.2.2 Buffer Classes

Buffers are used in NIO to hold data:

- `ByteBuffer`, `CharBuffer`, `IntBuffer`, etc.

Example: Using ByteBuffer

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
buffer.put((byte) 123);
buffer.flip(); // prepare for reading
byte b = buffer.get();
```

21.2.3 Channels

Channels represent open connections to I/O entities such as files or sockets. Common channels include:

- `FileChannel`
- `SocketChannel`
- `DatagramChannel`
- `ServerSocketChannel`

Reading File Using FileChannel

```
RandomAccessFile file = new RandomAccessFile("data.txt", "r");
FileChannel channel = file.getChannel();
ByteBuffer buffer = ByteBuffer.allocate(1024);
channel.read(buffer);
```

21.2.4 Selectors

Selectors are used for non-blocking I/O to monitor multiple channels using a single thread.

Selector Usage Example

```
Selector selector = Selector.open();
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
serverChannel.register(selector, SelectionKey.OP_ACCEPT);

while (true) {
    selector.select(); // blocks until an event
    Set<SelectionKey> keys = selector.selectedKeys();
    // Iterate and handle I/O events
}
```

21.2.5 Path, Paths, and Files (Java 7+)

The `java.nio.file` package improves file handling over `java.io.File`.

Example: Reading a File

```
Path path = Paths.get("example.txt");
List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);
```

21.3 Comparison: Java I/O vs NIO

Feature	Java I/O	Java NIO
Data Handling	Stream-based	Buffer-based
Blocking Mode	Always blocking	Non-blocking supported
Performance	Slower for large files or concurrent I/O	Faster, scalable for large data
File Access	Limited with `File` class	Advanced operations with `Path`, `Files`, etc.
Thread Usage	One thread per stream	One thread for multiple channels via selectors

21.4 Advanced NIO: Memory-Mapped Files

Memory-mapped files allow reading large files by mapping them into memory.

```
FileChannel channel = FileChannel.open(path, StandardOpenOption.READ);
MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0,
channel.size());
```

21.5 Java NIO.2 Enhancements (Java 7+)

- **WatchService API**: Monitors file system events (like file creation, modification).
- **Symbolic Links**: Better support.
- **Improved Exception Handling**: e.g., `AccessDeniedException`, `NoSuchFileException`.

Example: Watching File Changes

```
WatchService watcher = FileSystems.getDefault().newWatchService();
Path dir = Paths.get("/some/dir");
dir.register(watcher, StandardWatchEventKinds.ENTRY_CREATE);

WatchKey key = watcher.take();
for (WatchEvent<?> event : key.pollEvents()) {
    System.out.println("New file: " + event.context());
}
```

Summary

Java I/O and NIO are powerful libraries for handling input and output operations, but they cater to different use cases. While I/O is simpler and stream-based, NIO is designed for high performance, scalability, and flexibility using buffers and channels. Understanding when to use I/O versus NIO is crucial for developing efficient Java applications. NIO.2 further improves file operations with modern file APIs and event watching capabilities.
