THE EXPERT'S VOICE® IN JAVA

Java I/O, NIO and NIO.2

Jeff Friesen

Apress®

Java I/O, NIO and NIO.2

Jeff Friesen

Apress®

Java I/O, NIO and NIO.2

ISBN-13 (pbk): 978-1-4842-1566-1

ISBN-13 (electronic): 978-1-4842-1565-4

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewers: Vinay Kumar and Wallace Jackson
Editorial Board: Steve Anglin, Louise Corrigan, James T. DeWolf, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Steve Weiss
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com/9781484215661. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

To my parents.

Contents at a Glance

Contents

About the Author

Jeff Friesen is a freelance tutor and software developer with an emphasis on Java. In addition to authoring *Learn Java for Android Development* and co-authoring *Android Recipes*, Jeff has written numerous articles on Java and other technologies for JavaWorld (`JavaWorld.com`), InformIT (`InformIT.com`), `Java.net` and DevSource (`DevSource.com`). Jeff can be contacted via his website at `TutorTutor.ca`.

About the Technical Reviewers

Vinay Kumar is a Technology Evangelist. He has extensive experience of 8+ years in designing and implementing large scale projects in Enterprise Technologies in various consulting and system Integration Companies. His passion helped him achieve certifications in Oracle ADF, Webcenter Portal and Java/JEE. Experience and in-depth knowledge has helped him evolve into a focused domain expert and a well-known technical blogger. He loves to spend his time in mentoring and writing technical blogs, publishing white papers and maintaining a dedicated education channel at YouTube for the ADF/ Webcenter. He has experience in Java, JEE and various open stack technologies as well. Vinay has been contributing to the Java/Oracle ADF/Webcenter community by publishing 300+ technical articles at his personal blog www.techartifact.com. He was awarded an Oracle ACE in June 2014. You can follow him at @vinaykuma201 or in.linkedin.com/in/vinaykumar2.

Wallace Jackson has been writing for leading multimedia publications about his work in new media content development since the advent of Multimedia Producer Magazine nearly two decades ago. He has authored a half-dozen Android book titles for Apress, including four titles in the popular Pro Android series. Wallace received his undergraduate degree in Business Economics from the University of California at Los Angeles (UCLA) and a graduate degree in MIS Design and Implementation from the University of Southern California (USC). He is currently the CEO of Mind Taffy Design, a new media content production and digital campaign design and development agency.

Acknowledgments

I have many people to thank for assisting me in the development of this book. I especially thank Steve Anglin for asking me to write it and Mark Powers for guiding me through the writing process.

Introduction

Input/output (I/O) is not a sexy subject, but it's an important part of non-trivial applications. This book introduces you to most of Java's I/O capabilities as of Java 8 update 51.

Chapter 1 presents a broad overview of I/O in terms of Java's classic I/O, New I/O (NIO), and NIO.2 categories. You learn what each category offers in terms of its capabilities, and you also learn about concepts such as paths and Direct Memory Access.

Chapters 2 through 5 cover classic I/O APIs. You learn about the `File` and `RandomAccessFile` classes along with streams (including object serialization and externalization) and writers/readers.

Chapters 6 through 11 focus on NIO. You explore buffers, channels, selectors, regular expressions, charsets, and formatters. (Formatters were not introduced with the other NIO types in Java 1.4 because they depend on the variable arguments capability that was introduced in Java 5.)

NIO is missing several features, which were subsequently provided by NIO.2. Chapters 12 through 14 cover NIO.2's improved file system interface, asynchronous I/O, and the completion of socket channel functionality.

Each chapter ends with assorted exercises that are designed to help you master its content. Along with long answers and true/false questions, you are often confronted with programming exercises. Appendix A provides the answers and solutions.

Appendix B provides a tutorial on sockets and network interfaces. Although not directly related to classic I/O, NIO, and NIO.2, they leverage I/O capabilities and are mentioned elsewhere in this book.

> **Note** I briefly use Java 8's lambda expression and method reference language features and also use Java 8's Streams API in some examples, but don't provide a tutorial on them. You'll need to look elsewhere for that knowledge.

Thanks for purchasing this book. I hope you find it helpful in understanding classic I/O, NIO, and NIO.2.

—Jeff Friesen (September 2015)

> **Note** You can download this book's source code by pointing your web browser to www.apress.com/9781484215661 and clicking the Source Code tab followed by the Download Now link.

Part

Getting Started with I/O

Chapter 1

I/O Basics and APIs

Input and output (I/O) facilities are fundamental parts of operating systems along with computer languages and their libraries. All but trivial computer programs perform some kind of input and/or output operations.

Java has always supported I/O. Its initial suite of I/O APIs and related architecture are known as classic I/O. Because modern operating systems feature newer I/O paradigms, which classic I/O doesn't support, new I/O (NIO) was introduced as part of JDK 1.4 to support them. Lack of time prevented some planned NIO features from being included in this release, which led to these other NIO features being deferred to JDK 5 and JDK 7.

This chapter introduces you to classic I/O, NIO, and more NIO (NIO.2). You learn about the basic I/O features they address. Also, you receive an overview of their APIs. Subsequent chapters dig deeper into these APIs.

Classic I/O

JDK 1.0 introduced rudimentary I/O facilities for accessing the file system (to create a directory, remove a file, or perform another task), accessing file content randomly (as opposed to sequentially), and streaming byte-oriented data between sources and destinations in a sequential manner.

File System Access and the File Class

A *file system* is an operating system component that manages data storage and subsequent retrieval. Operating systems on which a Java virtual machine (JVM) runs support at least one file system. For example, Unix or

Linux combines all *mounted* (attached and prepared) disks into one virtual file system. In contrast, Windows associates a separate file system with each active disk drive.

A file system stores data in *files*, which are stored in *directories*. Its file and directory objects are accessed by specifying *paths*, which are compact maps that locate and identify file system objects. Paths are either absolute or relative:

- An *absolute path* is a path relative to the file system's *root directory*. It's expressed as the root directory symbol followed by a delimited hierarchy of directory names that ends in the target directory or file name.
- A *relative path* is a path relative to some other directory. It's expressed similarly to an absolute path but without the initial root directory symbol. In contrast, it's often prefixed with one or more delimited "`..`" character sequences, where each sequence refers to a parent directory.

Paths are specified differently depending on the operating system. For example, Unix, Linux, and Unix-like operating systems identify the root directory and delimit path components with a forward slash (/), whereas Windows uses a backslash (\) for these purposes. Consider two examples:

```
/users/username/bin
\users\username\bin
```

Each absolute path accesses the `bin` subdirectory of the `username` subdirectory of the `users` subdirectory of the root directory. The path on the first line accesses `bin` in a Unix/Linux context, whereas the path on the second line accesses this subdirectory in a Windows context.

Windows and similar operating systems can manage multiple file systems. Each file system is identified with a drive specifier such as "`C:`". When specifying a path without a drive specifier, the path is relative to the current file system. Otherwise, it is relative to the specified file system:

```
\users\username\bin
C:\users\username\bin
```

The first line accesses the path relative to the current file system, whereas the second line accesses the path relative to the `C:` file system.

An instance of the `java.io.File` class abstracts a file or directory path. This instance provides access to the file system to perform tasks on this path such as removing the underlying file or directory. The following example demonstrates this class:

```
new File("temp").mkdir();
```

The example constructs a `File` object initialized to the file system object `temp`. It then calls `mkdir()` on this `File` object to make a new directory named `temp`.

Chapter 2 explores the `File` class.

Accessing File Content via RandomAccessFile

File content can be accessed sequentially or randomly. Random access can speed up searching and sorting capabilities. An instance of the `java.io.RandomAccessFile` class provides random access to a file. This capability is demonstrated in the following example:

```
RandomAccessFile raf = new RandomAccessFile("employees.dat", "r");
int empIndex = 10;
raf.seek(empIndex * EMP_REC_LEN);
// Read contents of employee record.
```

In this example, file `employees.dat`, which is divided into fixed-length employee records where each record is `EMP_REC_LEN` bytes long, is being accessed. The employee record at index `10` (the first record is located at index `0`) is being sought. This task is accomplished by *seeking* (setting the file pointer) to the byte location of this record's first byte, which is located at the index multiplied by the record length. The record is then accessed.

Chapter 3 explores the `RandomAccessFile` class.

Streaming Data via Stream Classes

Classic I/O includes streams for performing I/O operations. A *stream* is an ordered sequence of bytes of arbitrary length. Bytes flow over an *output stream* from an application to a destination and flow over an *input stream* from a source to an application. Figure 1-1 illustrates these flows.

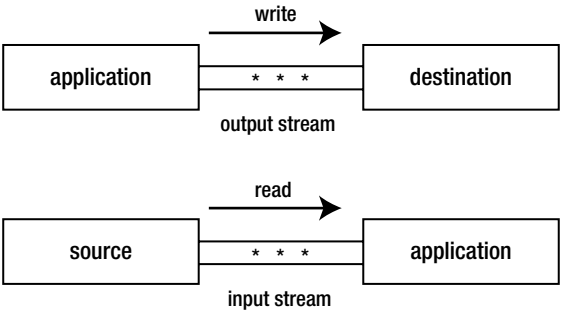

Figure 1-1. *Conceptualizing output and input streams as flows of bytes*

Java provides classes in the `java.io` package that identify various stream destinations for writing; for example, byte arrays and files. Java also provides classes in this package that identify various stream sources for reading. Examples include files and thread pipes.

For example, you would use `FileInputStream` to open an existing file and connect an input stream to it. You would then invoke various `read()` methods to read bytes from the file over the input stream. Lastly, you would invoke `close()` to close the stream and file. Consider the following example:

```
FileInputStream fis = null;
try
{
   fis = new FileInputStream("image.jpg");
   // Read bytes from file.
   int _byte;
   while ((_byte = fis.read()) != -1) // -1 signifies EOF
      ; // Process _byte in some way.
}
catch (IOException ioe)
{
   // Handle exception.
}
finally
{
   if (fis != null)
      try
      {
         fis.close();
      }
}
```

This example demonstrates the traditional way to open a file and create an input stream for reading bytes from the file. It then goes on to read the file's contents. An exception handler takes care of any thrown exceptions, which are represented by instances of the `java.io.IOException` class.

Whether or not an exception is thrown, the input stream and underlying file must be closed. This action takes place in the `try` statement's `finally` block. Because of the verbosity in closing the file, you can alternatively use JDK 7's `try`-with-resources statement to automatically close it, as follows:

```
try (FileInputStream fis = new FileInputStream("image.jpg"))
{
   // Read bytes from file.
   int _byte;
   while ((_byte = fis.read()) != -1) // -1 signifies EOF
      ; // Process _byte in some way.
}
catch (IOException ioe)
{
   // Handle exception.
}
```

I demonstrate both the traditional and `try`-with-resources approaches to closing files throughout subsequent chapters.

Some stream classes are used to filter other streams. For example, to improve performance, `BufferedInputStream` reads a block of bytes from another stream and returns bytes from its buffer until the buffer is empty, in which case it reads another block. Consider the following example:

```
try (FileInputStream fis = new FileInputStream("image.jpg");
     BufferedInputStream bis = new BufferedInputStream(fis))
{
   // Read bytes from file.
   int _byte;
   while ((_byte = bis.read()) != -1) // -1 signifies EOF
      ; // Process _byte in some way.
}
catch (IOException ioe)
{
   // Handle exception.
}
```

A file input stream that reads from the `image.jpg` file is created. This stream is passed to a buffered input stream constructor. Subsequent reads are performed on the buffered input stream, which calls file input stream `read()` methods when appropriate.

Chapter 4 explores the stream classes.

Stream Classes and Standard I/O

Many operating systems support *standard I/O*, which is preconnected input and output data streams between a computer program and its environment when it begins execution. The preconnected streams are known as *standard input*, *standard output*, and *standard error*.

Standard input defaults to reading its input from the keyboard. Also, standard output and standard error default to writing their output to the screen. However, these streams can be redirected to read input from a different source and write output to a different destination (such as a file).

JDK 1.0 introduced support for standard I/O by adding the `in`, `out`, and `err` objects of type `InputStream` and `PrintStream` to the `java.lang.System` class. You specify method calls on these objects to access standard input, standard output, and standard error, as follows:

```
int ch = System.in.read(); // Read single character from standard input.
System.out.println("Hello"); // Write string to standard output.
System.err.println("I/O error: " +
                   ioe.getMessage()); // Write string to standard error.
```

As well as exploring `InputStream` and `PrintStream`, Chapter 4 also revisits standard I/O to show you how to programmatically redirect these streams.

JDK 1.1 and the Writer/Reader Classes

JDK 1.0's I/O capabilities are suitable for streaming bytes, but cannot properly stream characters because they don't account for *character encodings*. JDK 1.1 overcame this problem by introducing writer/reader classes that take character encodings into account. For example, the `java.io` package includes `FileWriter` and `FileReader` classes for writing and reading character streams.

Chapter 5 explores various writer and reader classes.

NIO

Modern operating systems offer sophisticated I/O services (such as readiness selection) for improving I/O performance and simplifying I/O. Java Specification Request (JSR) 51 (`www.jcp.org/en/jsr/detail?id=51`) was created to address these capabilities.

JSR 51's description indicates that it provides APIs for scalable I/O, fast buffered binary and character I/O, regular expressions, and charset conversion. Collectively, these APIs are known as NIO. JDK 1.4 implemented NIO in terms of the following APIs:

- Buffers
- Channels
- Selectors
- Regular expressions
- Charsets

The regular expression and charset APIs were provided to simplify common I/O-related tasks.

Buffers

Buffers are the foundation for NIO operations. Essentially, NIO is all about moving data into and out of buffers.

A process such as the JVM performs I/O by asking the operating system to drain a buffer's contents to storage via a write operation. Similarly, it asks the operating system to fill a buffer with data read from a storage device.

Consider a read operation involving a disk drive. The operating system issues a command to the disk controller to read a block of bytes from a disk into an operating system buffer. Once this operation completes, the operating system copies the buffer contents to the buffer specified by the process when it issued a `read()` operation. Check out Figure 1-2.

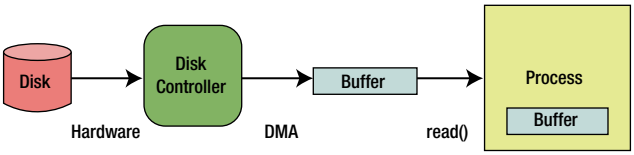

***Figure 1-2.** Filling a buffer at the operating system level*

In Figure 1-2, a process has issued a `read()` call to the operating system. In turn, the operating system has requested to the disk controller to read a block of bytes from the disk. The disk controller (also known as a DMA controller) reads these bytes directly into an operating system buffer via *Direct Memory Access (DMA)*, a feature of computer systems that allows

certain hardware subsystems to access main system (RAM) memory independently of the central processing unit (CPU). The operating system then copies these bytes to the process's buffer.

Copying bytes from the operating system buffer to the process buffer isn't very efficient. It would be more performant to have the DMA controller copy directly to the process buffer, but there are two problems with this approach:

- The DMA controller typically cannot communicate directly with the *user space* in which the JVM process runs. Instead, it communicates with the operating system's *kernel space*.
- Block-oriented devices such as a DMA controller work with fixed-size data blocks. In contrast, the JVM process might request a size of data that isn't a multiple of the block size or that is misaligned.

Because of these problems, the operating system acts as an intermediary, tearing apart and recombining data as it switches between the JVM process and the DMA controller.

The data assembly/disassembly tasks can be made more efficient by letting the JVM process pass a list of buffer addresses to the operating system in a single system call. The operating system then fills or drains these buffers in sequence, scattering data to multiple buffers during a read operation or gathering data from several buffers during a write operation. This *scatter/gather* activity reduces the number of (potentially expensive) system calls that the JVM process must make and lets the operating system optimize data handling because it knows the total amount of buffer space. Furthermore, when multiple processors or cores are available, the operating system may allow buffers to be filled or drained simultaneously.

JDK 1.4's `java.nio.Buffer` class abstracts the concept of a JVM process buffer. It serves as the superclass for `java.nio.ByteBuffer` and other buffer classes. Because I/O is fundamentally byte-oriented, only `ByteBuffer` instances can be used with channels (which are discussed shortly). Most of the other `Buffer` subclasses are conveniences for working with multibyte data (such as characters or integers).

Chapter 6 explores the `Buffer` class and its children.

Channels

Forcing a CPU to perform I/O tasks and wait for I/O completions (such a CPU is said to be *I/O bound*) is wasteful of this resource. Performance can be improved by offloading these tasks to DMA controllers so that the processor can get on with other work.

A *channel* serves as a conduit for communicating (via the operating system) with a DMA controller to efficiently drain byte buffers to or fill byte buffers from a disk. JDK 1.4's `java.nio.channels.Channel` interface, its subinterfaces, and various classes implement the channel architecture.

One of these classes is called `java.nio.channels.FileChannel`, and it abstracts a channel for reading, writing, mapping, and manipulating a file. One interesting feature of `FileChannel` is its support for file locking, upon which sophisticated applications such as database management systems rely.

File locking lets a process prevent or limit access to a file while the process is accessing the file. Although file locking can be applied to an entire file, it is often narrowed to a smaller region. A lock ranges from a starting byte offset in the file and continues for a specific number of bytes.

Another interesting `FileChannel` feature is *memory-mapped file I/O* via the `map()` method. `map()` returns a `java.nio.MappedByteBuffer` whose content is a memory-mapped region of a file. File content is accessed via memory accesses; buffer copies and read-write system calls are eliminated.

You can obtain a channel by calling the `java.nio.channels.Channels` class's methods or the methods in classic I/O classes such as `RandomAccessFile`.

Chapter 7 explores `Channel`, `Channels`, and more.

Selectors

I/O is classified as block-oriented or stream-oriented. Reading from or writing to a file is an example of block-oriented I/O. In contrast, reading from the keyboard or writing to a network connection is an example of stream-oriented I/O.

Stream I/O is often slower than block I/O. Furthermore, input tends to be intermittent. For example, the user might pause while entering a stream of characters or momentary slowness in a network connection causes a playing video to proceed in a jerky fashion.

Many operating systems allow streams to be configured to operate in *nonblocking mode* in which a thread continually checks for available input without blocking when no input is available. The thread can handle incoming data or perform other tasks until data arrives.

This "polling for available input" activity can be wasteful, especially when the thread needs to monitor many input streams (such as in a web server context). Modern operating systems can perform this checking efficiently, which is known as *readiness selection*, and which is often built on top of nonblocking mode. The operating system monitors a collection of streams and returns an indication to the thread of which streams are ready to

perform I/O. As a result, a single thread can multiplex many active streams via common code and makes it possible, in a web server context, to manage a huge number of network connections.

JDK 1.4 supports readiness selection by providing *selectors*, which are instances of the `java.nio.channels.Selector` class that can examine one or more channels and determine which channels are ready for reading or writing. This way a single thread can manage multiple channels (and, therefore, multiple network connections) efficiently. Being able to use fewer threads is advantageous where thread creation and thread context switching is expensive in terms of performance and/or memory use. See Figure 1-3.

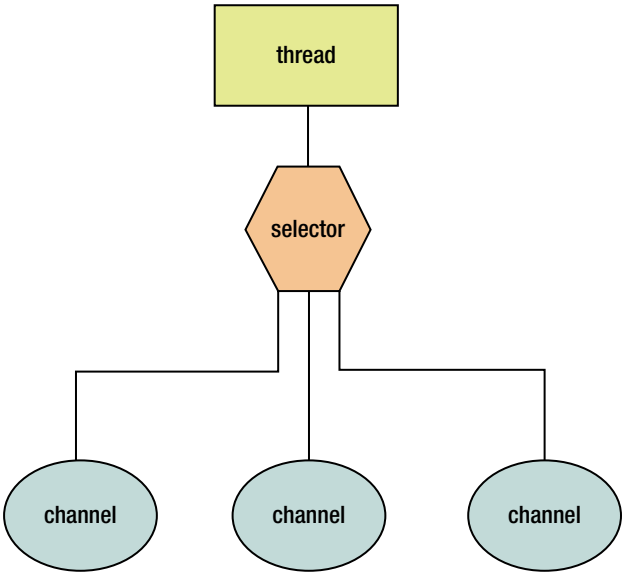

***Figure 1-3.** A thread manages three channels via a selector*

Chapter 8 explores `Selector` and its related types.

Regular Expressions

Regular expressions were introduced as part of NIO. Although you might wonder about the rationale for doing this (what have regular expressions got to do with I/O?), regular expressions are commonly used to scan textual data that is read from a file or other source. The need to perform these scans as quickly as possible mandated their inclusion. JDK 1.4 supports regular expressions via the `java.util.regex` package and its `Pattern` and `Matcher` classes.

Chapter 9 explores the `Pattern` and `Matcher` classes.