SAMPLE CHAPTER

Netty IN ACTION

Norman Maurer
Marvin Allen Wolfthal

FOREWORD BY Trustin Lee

MANNING

Netty in Action

by Norman Maurer
Marvin Allen Wolfthal

Chapter 1

brief contents

1 Netty—asynchronous and event-driven

This chapter covers

- Networking in Java
- Introducing Netty
- Netty's core components

Suppose you're just starting on a new mission-critical application for a large, important company. In the first meeting you learn that the system must scale up to 150,000 concurrent users with no loss of performance. All eyes are on you. What do you say?

If you can say with confidence, "Sure, no problem," then hats off to you. But most of us would probably take a more cautious position, like: "Sounds doable." Then, as soon as we could get to a computer, we'd search for "high performance Java networking."

If you run this search today, among the first results you'll see this:

> ***Netty: Home***
>
> netty.io/
>
> Netty is an asynchronous event-driven **network** application framework for rapid development of maintainable **high performance** protocol servers & clients.

If you discovered Netty this way, as many have, your next steps were probably to browse the site, download the code, peruse the Javadocs and a few blogs, and start hacking. If you already had solid network programming experience, you probably made good progress; otherwise, perhaps not.

Why? High-performance systems like the one in our example require more than first-class coding skills; they demand expertise in several complex areas: networking, multithreading, and concurrency. Netty captures this domain knowledge in a form that can be used even by networking neophytes. But up to now, the lack of a comprehensive guide has made the learning process far more difficult than need be—hence this book.

Our primary goal in writing it has been to make Netty accessible to the broadest possible range of developers. This includes many who have innovative content or services to offer but neither the time nor inclination to become networking specialists. If this applies to you, we believe you'll be pleasantly surprised at how quickly you'll be ready to create your first Netty application. At the other end of the spectrum, we aim to support advanced practitioners who are seeking tools for creating their own network protocols.

Netty does indeed provide an extremely rich networking toolkit, and we'll spend most of our time exploring its capabilities. But Netty is ultimately a *framework,* and its architectural approach and design principles are every bit as important as its technical content. Accordingly, we'll be talking about points such as

- Separation of concerns (decoupling business and network logic)
- Modularity and reusability
- Testability as a first-order requirement

In this first chapter, we'll begin with background on high-performance networking, particularly its implementation in the Java Development Kit (JDK). With this context in place, we'll introduce Netty, its core concepts, and building blocks. By the end of the chapter, you'll be ready to tackle your first Netty-based client and server.

1.1 Networking in Java

Developers who started out in the early days of networking spent a lot of time learning the intricacies of the C language socket libraries and dealing with their quirks on different operating systems. The earliest versions of Java (1995–2002) introduced enough of an object-oriented façade to hide some of the thornier details, but creating a complex client/server protocol still required a lot of boilerplate code (and a fair amount of peeking under the hood to get it all working smoothly).

Those first Java APIs (`java.net`) supported only the so-called blocking functions provided by the native system socket libraries. The following listing shows an unadorned example of server code using these calls.

Listing 1.1 Blocking I/O example

```
ServerSocket serverSocket = new ServerSocket(portNumber);
Socket clientSocket = serverSocket.accept();
BufferedReader in = new BufferedReader(
    new InputStreamReader(clientSocket.getInputStream()));
PrintWriter out =
    new PrintWriter(clientSocket.getOutputStream(), true);
String request, response;
while ((request = in.readLine()) != null) {
    if ("Done".equals(request) (
        break;
    }
    response = processRequest(request);
    out.println(response);
}
```

A new ServerSocket listens for connection requests on the specified port.

1 accept() call blocks until a connection is established.

2 Stream objects are derived from those of the Socket.

Processing loop begins. 3

If the client has sent "Done", the processing loop is exited.

4 The request is passed to the server's processing method.

The server's response is sent to the client.

The processing loop continues.

The previous listing implements one of the basic `Socket` API patterns. Here are the most important points:

- `accept()` blocks until a connection is established on the `ServerSocket` 1, then returns a new `Socket` for communication between the client and the server. The `ServerSocket` then resumes listening for incoming connections.
- A `BufferedReader` and a `PrintWriter` are derived from the `Socket`'s input and output streams 2. The former reads text from a character input stream, the latter prints formatted representations of objects to a text output stream.
- `readLine()` blocks until a string terminated by a linefeed or carriage return is read in 3.
- The client's request is processed 4.

This code will handle only one connection at a time. To manage multiple, concurrent clients, you need to allocate a new `Thread` for each new client `Socket`, as shown in figure 1.1.

Let's consider the implications of such an approach. First, at any point many threads could be dormant, just waiting for input or output data to appear on the line. This is

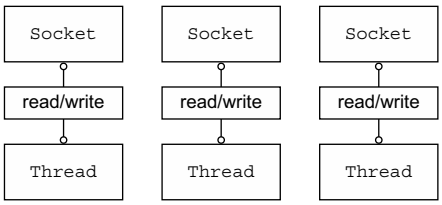


Figure 1.1 Multiple connections using blocking I/O

likely to be a waste of resources. Second, each thread requires an allocation of stack memory whose default size ranges from 64 KB to 1 MB, depending on the OS. Third, even if a Java virtual machine (JVM) can physically support a very large number of threads, the overhead of context-switching will begin to be troublesome long before that limit is reached, say by the time you reach 10,000 connections.

While this approach to concurrency might be acceptable for a small-to-moderate number of clients, the resources needed to support 100,000 or more simultaneous connections make it far from ideal. Fortunately, there is an alternative.

1.1.1 Java NIO

In addition to the blocking system calls underlying the code in listing 1.1, the native socket libraries have long included *non-blocking* calls, which provide considerably more control over the utilization of network resources.

- Using `setsockopt()` you can configure `sockets` so that read/write calls will return immediately if there is no data; that is, if a blocking call would have blocked.[1]
- You can register a set of non-blocking sockets using the system's event notification API[2] to determine whether any of them have data ready for reading or writing.

Java support for non-blocking I/O was introduced in 2002, with the JDK 1.4 package `java.nio`.

> **New or non-blocking?**
>
> NIO was originally an acronym for New Input/Output, but the Java API has been around long enough that it is no longer new. Most users now think of NIO as signifying non-blocking I/O, whereas blocking I/O is OIO or old input/output. You may also encounter references to plain I/O.

1.1.2 Selectors

Figure 1.2 shows a non-blocking design that virtually eliminates the drawbacks described in the previous section.

The `class java.nio.channels.Selector` is the linchpin of Java's non-blocking I/O implementation. It uses the event notification API to indicate which, among a set of non-blocking `sockets`, are ready for I/O. Because any read or write operation can be

---

[1] W. Richard Stevens, "4.3BSD returned EWOULDBLOCK if an operation on a non-blocking descriptor could not complete without blocking," *Advanced Programming in the UNIX Environment* (Addison-Wesley, 1992), p. 364.

[2] Also referred to as I/O multiplexing, this interface has evolved over the years from the original `select()` and `poll()` calls to more performant implementations. See Sangjin Han's "Scalable Event Multiplexing: epoll vs. kqueue" article, www.eecs.berkeley.edu/~sangjin/2012/12/21/epoll-vs-kqueue.html.

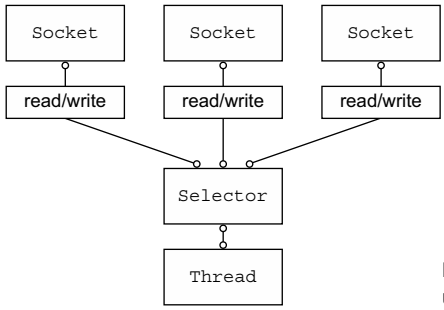


Figure 1.2 Non-blocking I/O using `Selector`

checked at any time for its completion status, a *single* thread, as shown in figure 1.2, can handle *multiple* concurrent connections.

Overall, this model provides much better resource management than the blocking I/O model:

- Many connections can be handled with fewer threads, and thus with far less overhead due to memory management and context-switching.
- Threads can be retargeted to other tasks when there is no I/O to handle.

Although many applications have been built using the Java NIO API directly, doing so correctly and safely is far from trivial. In particular, processing and dispatching I/O reliably and efficiently under heavy load is a cumbersome and error-prone task best left to a high-performance networking expert—Netty.

1.2 Introducing Netty

Not so long ago the scenario we presented at the outset—supporting thousands upon thousands of concurrent clients—would have been judged impossible. Today, as system users we take this capability for granted, and as developers we expect the bar to move even higher. We know there will always be demands for greater throughput and scalability—to be delivered at lower cost.

Don't underestimate the importance of that last point. We've learned from long and painful experience that the direct use of low-level APIs exposes complexity and introduces a critical dependency on skills that tend to be in short supply. Hence, a fundamental concept of object orientation: hide the complexity of underlying implementations behind simpler abstractions.

This principle has stimulated the development of numerous frameworks that encapsulate solutions to common programming tasks, many of them germane to distributed systems development. It's probably safe to assert that all professional Java developers are familiar with at least one of these.[3] For many of us they have

[3] Spring is probably the best known and is actually an entire ecosystem of application frameworks addressing object creation, batch processing, database programming, and so on.

become indispensable, enabling us to meet both our technical requirements and our schedules.

In the networking domain, Netty is the preeminent framework for Java.[4] Harnessing the power of Java's advanced APIs behind an easy-to-use API, Netty leaves you free to focus on what really interests you—the unique value of your application.

Before we begin our first close look at Netty, please examine the key features summarized in table 1.1. Some are technical, and others are more architectural or philosophical. We'll revisit them more than once in the course of this book.

Table 1.1 Netty feature summary

Category	Netty features
Design	Unified API for multiple transport types, both blocking and non-blocking. Simple but powerful threading model. True connectionless datagram socket support. Chaining of logic components to support reuse.
Ease of use	Extensive Javadoc and large example set. No required dependencies beyond JDK 1.6+. (Some optional features may require Java 1.7+ and/or additional dependencies.)
Performance	Better throughput and lower latency than core Java APIs. Reduced resource consumption thanks to pooling and reuse. Minimal memory copying.
Robustness	No `OutOfMemoryError` due to slow, fast, or overloaded connection. Eliminates unfair read/write ratio typical of NIO applications in high-speed networks.
Security	Complete SSL/TLS and StartTLS support. Usable in restricted environments such as Applet or OSGI.
Community-driven	Release early and often.

1.2.1 Who uses Netty?

Netty has a vibrant and growing user community that includes large companies such as Apple, Twitter, Facebook, Google, Square, and Instagram, as well as popular open source projects such as Infinispan, HornetQ, Vert.x, Apache Cassandra, and Elasticsearch, all of which have employed its powerful network abstractions in their core code.[5] Among startups, Firebase and Urban Airship are using Netty, the former for long-lived HTTP connections and the latter for all kinds of push notifications.

Whenever you use Twitter, you are using Finagle,[6] their Netty-based framework for inter-system communication. Facebook uses Netty in Nifty, their Apache Thrift service.

---

[4] Netty was awarded the Duke's Choice Award in 2011. See www.java.net/dukeschoice/2011.

[5] For a full list of known adopters see http://netty.io/wiki/adopters.html.

[6] For information on Finagle see https://twitter.github.io/finagle/.

Scalability and performance are critical concerns for both companies, and both are regular contributors to Netty.[7]

In turn, Netty has benefited from these projects, enhancing both its scope and flexibility through implementations of protocols such as FTP, SMTP, HTTP, and WebSocket, as well as others, both binary and text-based.

1.2.2 Asynchronous and event-driven

We'll be using the word asynchronous a great deal, so this is a good time to clarify the context. Asynchronous, that is, *un-synchronized*, events are certainly familiar. Consider email: you may or may not get a response to a message you have sent, or you may receive an unexpected message even while sending one. Asynchronous events can also have an *ordered* relationship. You generally get an answer to a question only *after* you have asked it, and you may be able to do something else while you are waiting for it.

In everyday life, asynchrony just happens, so you may not think about it much. But getting a computer program to work the same way presents some very special problems. In essence, a system that is both asynchronous *and* event-driven exhibits a particular and, to us, extremely valuable kind of behavior: it can respond to events occurring at any time and in any order.

This capability is critical for achieving the highest levels of *scalability*, defined as "the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth."[8]

What is the connection between asynchrony and scalability?

- Non-blocking network calls free us from having to wait for the completion of an operation. Fully asynchronous I/O builds on this feature and carries it a step further: an asynchronous method returns immediately and notifies the user when it is complete, directly or at a later time.
- Selectors allow us to monitor many connections for events with many fewer threads.

Putting these elements together, with non-blocking I/O we can handle very large numbers of events much more rapidly and economically than would be possible with blocking I/O. From the point of view of networking, this is key to the kind of systems we want to build, and as you'll see, it is also key to Netty's design from the ground up.

In the next section we'll take a first look at Netty's core components. For now, think of them as domain objects rather than concrete Java classes. Over time, we'll see how they collaborate to provide notification about events that occur on the network and make them available for processing.

---

[7] Chapters 15 and 16 present case studies describing how some of the companies mentioned here use Netty to solve real-world problems.

[8] André B. Bondi, "Characteristics of scalability and their impact on performance," *Proceedings of the second international workshop on Software and performance—WOSP '00* (2000), p. 195.

1.3 Netty's core components

In this section we'll discuss Netty's primary building blocks:

- `Channels`
- Callbacks
- `Futures`
- Events and handlers

These building blocks represent different types of constructs: resources, logic, and notifications. Your applications will use them to access the network and the data that flows through it.

For each component, we'll provide a basic definition and, where appropriate, a simple code example that illustrates its use.

1.3.1 Channels

A `Channel` is a basic construct of Java NIO. It represents

> an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct I/O operations, for example reading or writing.[9]

For now, think of a `Channel` as a vehicle for incoming (inbound) and outgoing (outbound) data. As such, it can be open or closed, connected or disconnected.

1.3.2 Callbacks

A *callback* is simply a method, a reference to which has been provided to another method. This enables the latter to call the former at an appropriate time. Callbacks are used in a broad range of programming situations and represent one of the most common ways to notify an interested party that an operation has completed.

Netty uses callbacks internally when handling events; when a callback is triggered the event can be handled by an implementation of `interface ChannelHandler`. The next listing shows an example: when a new connection has been established the `ChannelHandler` callback `channelActive()` will be called and will print a message.

Listing 1.2 `ChannelHandler` triggered by a callback

```
public class ConnectHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelActive(ChannelHandlerContext ctx)
        throws Exception {
        System.out.println(
            "Client " + ctx.channel().remoteAddress() + " connected");
    }
}
```

channelActive(ChannelHandlerContext) is called when a new connection is established.

[9] Java Platform, Standard Edition 8 API Specification, java.nio.channels, Interface Channel, http://docs.oracle.com/javase/8/docs/api/java/nio/channels/package-summary.html.

1.3.3 Futures

A `Future` provides another way to notify an application when an operation has completed. This object acts as a placeholder for the result of an asynchronous operation; it will complete at some point in the future and provide access to the result.

The JDK ships with `interface java.util.concurrent.Future`, but the provided implementations allow you only to check manually whether the operation has completed or to block until it does. This is quite cumbersome, so Netty provides its own implementation, `ChannelFuture`, for use when an asynchronous operation is executed.

`ChannelFuture` provides additional methods that allow us to register one or more `ChannelFutureListener` instances. The listener's callback method, `operationComplete()`, is called when the operation has completed. The listener can then determine whether the operation completed successfully or with an error. If the latter, we can retrieve the `Throwable` that was produced. In short, the notification mechanism provided by the `ChannelFutureListener` eliminates the need for manually checking operation completion.

Each of Netty's outbound I/O operations returns a `ChannelFuture`; that is, none of them block. As we said earlier, Netty is asynchronous and event-driven from the ground up.

Listing 1.3 shows that a `ChannelFuture` is returned as part of an I/O operation. Here, `connect()` will return directly without blocking and the call will complete in the background. When this will happen may depend on several factors, but this concern is abstracted away from the code. Because the thread is not blocked waiting for the operation to complete, it can do other work in the meantime, thus using resources more efficiently.

Listing 1.3 Asynchronous connect

```
Channel channel = ...;
// Does not block
ChannelFuture future = channel.connect(
    new InetSocketAddress("192.168.0.1", 25));
```

Asynchronous connection to a remote peer

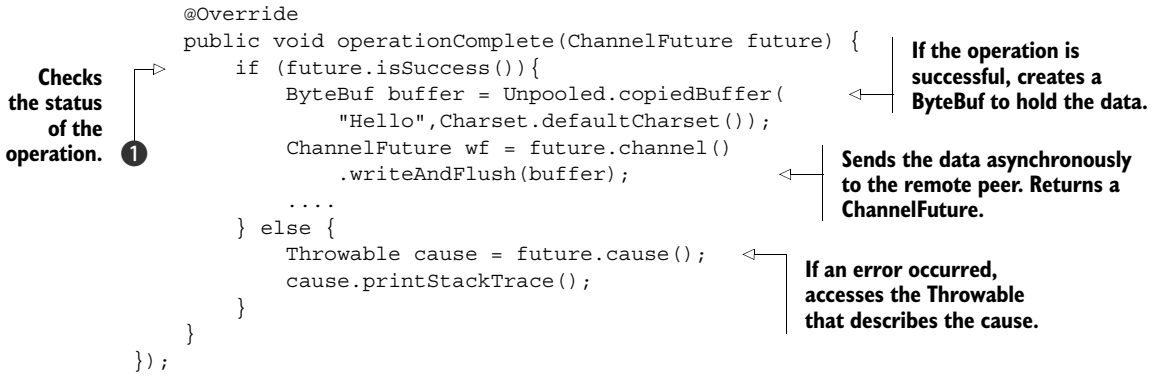
Listing 1.4 shows how to utilize the `ChannelFutureListener`. First you connect to a remote peer. Then you register a new `ChannelFutureListener` with the `ChannelFuture` returned by the `connect()` call. When the listener is notified that the connection is established, you check the status ❶. If the operation is successful, you write data to the `Channel`. Otherwise you retrieve the `Throwable` from the `ChannelFuture`.

Listing 1.4 Callback in action

```
Channel channel = ...;
// Does not block
ChannelFuture future = channel.connect(
    new InetSocketAddress("192.168.0.1", 25));
future.addListener(new ChannelFutureListener() {
```

Connects asynchronously to a remote peer.

Registers a ChannelFutureListener to be notified once the operation completes.




```
@Override
public void operationComplete(ChannelFuture future) {
    if (future.isSuccess()){
        ByteBuf buffer = Unpooled.copiedBuffer(
            "Hello",Charset.defaultCharset());
        ChannelFuture wf = future.channel()
            .writeAndFlush(buffer);
        ....
    } else {
        Throwable cause = future.cause();
        cause.printStackTrace();
    }
}
});
```

Note that error handling is entirely up to you, subject, of course, to any constraints imposed by the specific error at hand. For example, in case of a connection failure, you could try to reconnect or establish a connection to another remote peer.

If you're thinking that a `ChannelFutureListener` is a more elaborate version of a callback, you're correct. In fact, callbacks and `Futures` are complementary mechanisms; in combination they make up one of the key building blocks of Netty itself.

1.3.4 Events and handlers

Netty uses distinct events to notify us about changes of state or the status of operations. This allows us to trigger the appropriate action based on the event that has occurred. Such actions might include

- Logging
- Data transformation
- Flow-control
- Application logic

Netty is a networking framework, so events are categorized by their relevance to inbound or outbound data flow. Events that may be triggered by inbound data or an associated change of state include

- Active or inactive connections
- Data reads
- User events
- Error events

An outbound event is the result of an operation that will trigger an action in the future, which may be

- Opening or closing a connection to a remote peer
- Writing or flushing data to a socket

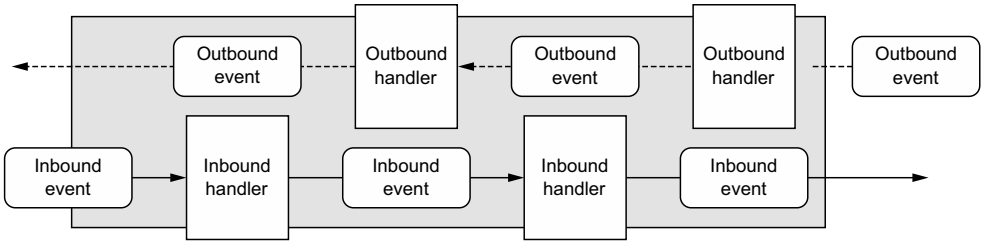


Figure 1.3 Inbound and outbound events flowing through a chain of `ChannelHandlers`

Every event can be dispatched to a user-implemented method of a handler class. This is a good example of an event-driven paradigm translating directly into application building blocks. Figure 1.3 shows how an event can be handled by a chain of such event handlers.

Netty's `ChannelHandler` provides the basic abstraction for handlers like the ones shown in figure 1.3. We'll have a lot more to say about `ChannelHandler` in due course, but for now you can think of each handler instance as a kind of callback to be executed in response to a specific event.

Netty provides an extensive set of predefined handlers that you can use out of the box, including handlers for protocols such as HTTP and SSL/TLS. Internally, `ChannelHandlers` use events and futures themselves, making them consumers of the same abstractions your applications will employ.

1.3.5 Putting it all together

In this chapter you've been introduced to Netty's approach to high-performance networking and to some of the primary components of its implementation. Let's assemble a big-picture view of what we've discussed.

FUTURES, CALLBACKS, AND HANDLERS

Netty's asynchronous programming model is built on the concepts of `Futures` and callbacks, with the dispatching of events to handler methods happening at a deeper level. Taken together, these elements provide a processing environment that allows the logic of your application to evolve independently of any concerns with network operations. This is a key goal of Netty's design approach.

Intercepting operations and transforming inbound or outbound data on the fly requires only that you provide callbacks or utilize the `Futures` that are returned by operations. This makes chaining operations easy and efficient and promotes the writing of reusable, generic code.

SELECTORS, EVENTS, AND EVENT LOOPS

Netty abstracts the `Selector` away from the application by firing events, eliminating all the handwritten dispatch code that would otherwise be required. Under the covers, an `EventLoop` is assigned to each `Channel` to handle all of the events, including

- Registration of interesting events
- Dispatching events to `ChannelHandlers`
- Scheduling further actions

The `EventLoop` itself is driven by only one thread that handles all of the I/O events for one `Channel` and does not change during the lifetime of the `EventLoop`. This simple and powerful design eliminates any concern you might have about synchronization in your `ChannelHandlers`, so you can focus on providing the right logic to be executed when there is interesting data to process. As we'll see when we explore Netty's threading model in detail, the API is simple and compact.

1.4 Summary

In this chapter, we looked at the background of the Netty framework, including the evolution of the Java networking API, the distinctions between blocking and nonblocking network operations, and the advantages of asynchronous I/O for high-volume, high-performance networking.

We then moved on to an overview of Netty's features, design, and benefits. These include the mechanisms underlying Netty's asynchronous model, including callbacks, `Futures`, and their use in combination. We also touched on how events are generated and how they can be intercepted and handled.

Going forward, we'll explore in much greater depth how this rich collection of tools can be utilized to meet the specific needs of your applications.

In the next chapter, we'll delve into the basics of Netty's API and programming model, and you'll write your first client and server.

JAVA

Netty IN ACTION

Maurer • Wolfthal

Netty is a Java-based networking framework that manages complex networking, multithreading, and concurrency for your applications. And Netty hides the boilerplate and low-level code, keeping your business logic separate and easier to reuse. With Netty, you get an easy-to-use API, leaving you free to focus on what's unique to your application.

Netty in Action introduces the Netty framework and shows you how to incorporate it into your Java network applications. You will discover how to write highly scalable applications without getting into low-level APIs. The book teaches you to think in an asynchronous way as you work through its many hands-on examples and helps you master the best practices of building large-scale network apps.

What's Inside

- Netty from the ground up
- Asynchronous, event-driven programming
- Implementing services using different protocols
- Covers Netty 4.x

This book assumes readers are comfortable with Java and basic network architecture.

Norman Maurer is a senior software engineer at Apple and a core developer of Netty. **Marvin Wolfthal** is a Dell Services consultant who has implemented mission-critical enterprise systems using Netty.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/netty-in-action

"The first-ever book on Netty ... shows how to build a high-performance, low-latency network application."
—From the Foreword by Trustin Lee, Founder of Netty

"High-performance Java network stacks—covered from concepts to best practices."
—Christian Bach
Grid Trading Platform

"The most comprehensive content for getting the most out of Netty."
—Jürgen Hoffmann, Red Hat

"An excellent overview of the Netty framework. Highly recommended to anyone doing performance-sensitive network I/O work in Java."
—Yestin Johnson, Impact Radius

ISBN 13: 978-1-61729-147-0
ISBN 10: 1-61729-147-1
55499
9 781617 291470

MANNING $54.99 / Can $63.99 [INCLUDING eBOOK]