1 Basic NIO (New Input/Output)

This chapter covers

- The New I/O system
- Doing I/O with channels and buffers
- File locking

The New I/O (NIO) API introduced in JDK 1.4 provides a completely new model of low-level I/O. Unlike the original I/O libraries in the `java.io` package, which were strongly stream-oriented, the New I/O API in the `java.nio` package is *block-oriented*. This means that I/O operations, wherever possible, are performed on large blocks of data in a single step, rather than on one byte or character at a time.

The New I/O API libraries are elegant and well designed, but their very nature represents a trade-off: some simplicity has been sacrificed for potentially enormous gains in speed. One of the major sources of speed improvement is the introduction of *direct buffers*. Where possible, data in these buffers is not copied to and from intermediate Java buffers; instead, system-level operations are performed on them directly. Although the implementation necessarily differs from platform to platform, these direct buffers can potentially permit Java programs to have I/O performance at or near that of programs written in C or C++.

The New I/O API also offers a platform-independent form of *nonblocking I/O*. This simplifies multithreaded I/O programming and can enable programs to efficiently handle a large number of connections to data sources and sinks.

The New I/O API model coexists peacefully with the original I/O libraries from the `java.io` package. In fact, to a substantial degree, the original I/O libraries have been rewritten to take advantage of the New I/O API.

Application programmers will not be forced to rewrite any of their code—existing applications written against the original APIs will continue to work as before. However, you might consider using some of the new features of the New I/O API to speed up any performance bottlenecks you find in your programs. Mixing old- and new-style I/O code is not trivial, but it is possible to do cleanly and effectively.

This book divides its NIO coverage into two chapters—chapter 1, "Basic NIO," and chapter 2, "Advanced NIO." Chapter 1 covers channels and buffers, as well as file locking. These should give you a good understanding of the basic classes used throughout the NIO system. Chapter 2 introduces you to the more advanced features, such as multiplexed I/O; these make use of the ideas presented in this chapter.

1.1 Doing I/O with channels and buffers

Channels and buffers represent the two basic abstractions within the New I/O API. Channels correspond roughly to input and output streams: they are sources and sinks for sequential data. However, whereas input and output streams deal most directly with single bytes, channels read and write data in chunks. Additionally, a channel can be bidirectional, in which case it corresponds to *both* an input stream and an output stream.

The chunks of data that are written to and read from channels are contained in objects called *buffers*. A buffer is an array of data enclosed in an abstraction that makes reading from, and writing to, channels easy and convenient. Buffers are often large, reflecting the fact that the I/O paradigm used in the New I/O API is oriented toward transferring large amounts of data quickly.

Most of the input and output streams in the original I/O libraries have been re-implemented to use channels as their underlying mechanism. This means that when you do old-style I/O programming using these stream classes, you're using channels without realizing it. Since programming with streams is conceptually simpler than programming with channels, you can continue to use streams if you find that your program is fast enough. However, channels provide the opportunity for great speed improvements, and some applications are actually easier to write using channels.

In this section, we'll learn how channels and buffers work, and how they differ from streams.

1.1.1 Getting a channel from a stream

As mentioned previously, many of the streams in the `java.io` package have been re-implemented using channels. It's easy to get the underlying channel that implements a stream, using the `getChannel()` method:

```
FileInputStream fin = new FileInputStream( infile );

FileChannel inc = fin.getChannel();
```

If you examine the documentation for the original `java.io.*` classes, you'll see that a number of the classes have been augmented with a `getChannel()` method:

- `java.io.FileInputStream`
- `java.io.FileOutputStream`
- `java.io.RandomAccessFile`
- `java.net.Socket`
- `java.net.ServerSocket`
- `java.net.DatagramSocket`
- `java.net.MulticastSocket`
- `java.net.SocketInputStream` (private)
- `java.net.SocketOutputStream` (private)

You'll notice that `InputStream` and `OutputStream` do not have `getChannel()` methods. This is because streams *in general* do not have to be implemented in terms of an underlying channel object. Streams that are directly associated with operating

system features like files and sockets generally are implemented as channels, while pure-Java streams such as `ByteArrayOutputStream` and `FilterInputStream` are not.

1.1.2 Creating a buffer revision

Before you can do any kind of I/O on a channel, you need to have a buffer to do it with. A buffer is an object that contains an array of data, and allows that data to be used for reading from, and writing to, channels.

Creating a buffer is easy. Here's how you create a `ByteBuffer`:

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

This method takes a single argument—the size of the underlying array. This value is called the buffer's *capacity*. Once a buffer is created, the capacity never changes. The best size for a buffer depends on the application. A larger buffer can allow for faster throughput, but takes up more memory, while a smaller one may degrade performance slightly, but uses less memory.

You'll notice that we didn't use a traditional constructor here. This is true in general: buffers are either allocated using the static `allocate()` method, or created from an existing byte array using `wrap()`. They are never constructed directly by the user.

You'll also notice that we've created a `ByteBuffer`. The `java.nio` package also contains `IntBuffer`, `ShortBuffer`, `FloatBuffer`, and so on. There are, in fact, buffer types for each of Java's primitive types. There is a class called `Buffer`, but it is abstract—you can't create one. (`Buffer` is the abstract superclass of all the buffer classes.) A buffer is always a buffer *of something*. In the following sections, we'll use this `ByteBuffer` to illustrate how to do basic channel I/O. In section 1.2.6 we'll learn how to use the other types of buffers.

NOTE Since the `ByteBuffer` is by far the most common, and most important, of the buffer classes, we will assume that any buffer we are talking about is a `ByteBuffer` unless otherwise specified.

1.1.3 Reading from a channel

Now that we've seen how to create a buffer, we'll see how we can read from a channel into a buffer. In many ways, reading from a channel into a buffer is like reading from an InputStream into an array, using one of the bulk-read methods in the old `java.io` package.

The old `read()` method looked like this:

```
public int read( byte[] b, int off, int len );
```

This variant on the `InputStream.read()` method allowed you to read a number of bytes into an array all at once. In a sense, this approach to using streams is the precursor to the channel-oriented method of the New I/O API.

Here's the method we use in the new API:

```
public int read( ByteBuffer dst );
```

You'll note that there's only a single argument to this method. This is because the three arguments from the old-style `read()` call, as well as a number of other things, are all wrapped up inside the `ByteBuffer` object.

You'll also note that this new `read()` method returns an integer, just like the old one. The meaning of this value hasn't changed: it's the number of bytes that were successfully read. In both cases, this value is limited, because the `read()` method will only read as many bytes as can fit in the available space. In the old method, the available space was `len-off`; in the new method, the available space is equal to `buffer.remaining()`. (More about this in section 1.2.3.)

Note that if you read from a channel that is only open for writing, a `NonReadableChannelException` will be thrown.

1.1.4 Writing to a channel

Now that we've read some data from a channel into a buffer, we can write that data out to another channel. This is done—surprise!—via the `write()` method of a channel. And, as with reading, writing a buffer is similar to doing a *bulk-write* from the old `java.io` classes. Here is the old `write()` method:

```
public void write( byte[] b, int off, int len );
```

Again, the three arguments to the old-style write are replaced by a single argument, which is a buffer, in the new `write()` method:

```
public int write( ByteBuffer src );
```

In this new method, you'll see an important difference that you don't see with the `read()` methods: the new `write()` method returns an `int`. The old `write()` call was guaranteed to write all the data or throw an exception. There were no valid conditions under which it would write only *part* of the data and return. This is not the case with the new `write()` method. It returns the number of bytes that were written.

And as with reading, if you write to a channel that is only open for reading, a `NonWritableChannelException` will be thrown.

1.1.5 Reading and writing together

The CopyFile program (see listing 1.1) illustrates the entire process of copying all the data from an input channel to an output channel.

Watch out for a couple of new methods—`flip()` and `clear()`. These methods are used any time a buffer is both written to and read from—which is almost all of the time. After reading from a channel into a buffer, you call `buffer.flip()` to prepare the buffer for being written to another channel. Likewise, once you've finished writing the contents of a buffer to one channel, you call `buffer.clear()` to prepare it for being read into again. More about this in section 1.2.4.

Make sure not to confuse reading from a buffer with reading from a channel: reading from a channel means reading data *from* the channel, and putting it *into* the buffer. Likewise, writing data to a channel means getting data *from* a buffer, and writing it *to* a channel. See section 1.2.2 for more details.

Listing 1.1 CopyFile.java

(see \Chapter1\CopyFile.java)

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

public class CopyFile
{
  static public void main( String args[] ) throws Exception {
    String infile = args[0], outfile = args[1];
    FileInputStream fin = new FileInputStream( infile );
    FileOutputStream fout = new FileOutputStream( outfile );

    FileChannel inc = fin.getChannel();
    FileChannel outc = fout.getChannel();

    ByteBuffer buffer = ByteBuffer.allocate( 1024 );

    while (true) {
      int ret = inc.read( buffer );
      if (ret==-1) // nothing left to read
        break;
      buffer.flip();
      outc.write( buffer );
      buffer.clear(); // Make room for the next read
    }
  }
}
```

A full understanding of this program—including an understanding of the `flip()` and `clear()` methods—requires that we learn more about buffers. The next section will describe how buffers work, and how they are used in practice.

1.2 Understanding buffers

Under the original I/O API, the `read()` and `write()` methods of the stream classes took primitive Java types—ints, floats, and so on, as well as arrays of ints, floats, and so on—as arguments. The management of these variables and buffers was up to the programmer.

In the New I/O API, these primitive types are never written directly to channels. Buffers are always used as the intermediaries. Buffers can also handle many of the tasks that used to have to be done by hand—keeping track of how much data has been read, making sure there's enough room in an array for the data to be read into, and so on. And buffers themselves have an I/O interface, because data must be put into and taken out of buffers.

This section will go over the details of how buffers store data and how they are used to transfer data to and from channels.

1.2.1 Creating buffers

As mentioned in section 1.1.2, buffers are never created using constructors. There are two ways of making a `ByteBuffer`: via the `allocate()` methods, and via the `wrap()` methods.

`allocate()` creates a fresh `ByteBuffer` and allocates the memory required to store the data. `allocateDirect()` does the same thing, but it attempts to allocate the required data area as direct memory. (See section 1.2.8 for more about direct buffers.)

The two `wrap()` methods create a new buffer by wrapping an existing array—or a portion of an existing array—in a `Buffer` object. Note that that this doesn't make a *copy* of the data—the data in the buffer and the data in the array are the *same data*. Any modifications to the buffer will show up in the array, and vice versa.

1.2.2 get() and put()

Generally, buffers are used to transfer data from one channel to another. The `read()` method of one channel puts data into a buffer, and the `write()` method of the other channel takes the data out of the buffer. However, buffers also have methods that can be used to fill and drain them "by hand." These are used when you want to put particular pieces of data into a buffer, or to extract the data and use it for something. These methods are called `get()` and `put()`.

It can be confusing to consider the buffer `get()` and `put()` methods along with the channel `read()` and `write()` methods, because they are backwards: when data is *read from* a channel, it is *written to* a buffer. Likewise, when data is *written to* a channel, it is *read from* a buffer. You read from a buffer using the buffer's `get()` methods, and you write to a buffer using the buffer's `put()` methods.

There are two kinds of `get()` and `put()` methods: *relative* and *absolute*. Absolute methods take an `index` parameter, which lets you choose the position in the underlying array at which you want to read or write. In contrast, relative methods do not need an `index` parameter—they use the next value or values in the array after the last one that was used. Relative methods are more commonly used, since they can be used to fill or drain a buffer sequentially.

There are five basic `put()` methods. The methods listed here are for `ByteBuffer`, but each of the `Buffer` classes has these methods. Of course, the arguments to the corresponding methods of `DoubleBuffer` are double-based, rather than byte-based, but otherwise they are the same.

- `put( byte b )`—Put a byte into this buffer
- `put( byte src[] )`—Put the bytes from an array into this buffer
- `put( byte src[], int offset, int length )`—Put a portion of the bytes from an array into this buffer
- `put( ByteBuffer src )`—Copy the contents of another buffer into this buffer
- `put( int index, byte b )`—Put a byte at array offset index (starting from zero)

Of these five methods, the first four are relative, and the last one is absolute.

There are four `get()` methods:

- `get()`—Get a single byte from this buffer
- `get( byte array[] )`—Fill an array of bytes with bytes from this buffer
- `get( byte array[], int offset, int length )`—Fill a portion of an array of bytes with bytes from this buffer
- `get( int index )`—Get the byte at array offset index (starting from zero)

Of these four methods, the first three are relative, and the last one is absolute. Note that there is no `get( ByteBuffer )` method. You can accomplish the same thing with `put( ByteBuffer )`.

In addition to these methods, `ByteBuffer` also contains a set of methods for reading and writing other primitive Java types. In each case, a call to one of these methods can be considered equivalent to calling the single-byte `get()` and `put()`

methods one or more times, with the bytes involved making up the value of the primitive type. More on this in section 1.2.7.

1.2.3 Buffer state values

In the previous sections, we saw how to read from and write to a buffer, but we never really found out what was going on inside the buffer. If you'll recall, the inner loop of the CopyFile program listed in section 1.1.5 was, schematically, something like this:

```
inc.read( buffer );
buffer.flip();
outc.write( buffer );
buffer.clear();
```

What's noteworthy about this is that our code doesn't seem to have to keep track of how many bytes were read and written each time. This is something the buffer does for us automatically.

Buffers take care of such things using a number of *buffer state values*. These are values that reflect the current state of the buffer as it is used for various reading and writing tasks. They keep track of how many bytes have been read or written, how many more can be read, how much room there is to read more, and so on. These are summarized in table 1.1 and are explained in further detail in the following sections.

Table 1.1 The state of each buffer is represented by three values. These values change as the buffer is read from, or written to, indicating progress through the buffer. In this way, a buffer keeps track of the reading or writing process.

State value name	What it is
position	The index into the underlying array of the next read (or write)
limit	The index into the underlying array of the first element that should not be read (or written)
capacity	The size of the underlying array

Buffer position

The buffer position specifies the next entry in the array that will be used for reading or writing:

- If the buffer is being written to (which means that it is being used for a channel read), the buffer position points to the location where the next byte will be stored.

- If the buffer is being read from (which means that it is being used for a channel write), the buffer position points to the next byte to be read.

In both cases, each time a byte is read or written, the value of the buffer position increases by the length of the item written. The position cannot become greater than the value of the buffer limit. If the code tries to execute a read or write that would make the position greater than the limit, a `java.nio.BufferUnderflowException` or `java.nio.BufferOverflowException`, respectively, is thrown.

Buffer limit

The buffer limit is the amount of data in the array. It defines the first array slot that should *not* be used for reading and writing. It is different from the capacity: the capacity of an array specifies how much data *could* be put in it—that is, how much could potentially fit. The limit specifies how much has actually been put in the array.

If the buffer is being written to, the limit specifies the array element after the last array element that can accept a value. In this case, the limit is generally set to be equal to the capacity of the buffer, so that the entirety of the underlying array will be used.

If the buffer is being read from, the limit specifies the array element after the last array element that can be read. The buffer limit might be equal to the buffer capacity, which means that the buffer was filled with data before reading started. The buffer limit might also be less than the capacity, which means the buffer was only partially filled when reading started.

Buffer capacity

The buffer capacity is equal to the size of the underlying array. Even if the array is only partially filled with data, the capacity refers to the *entire array*, including both the used and unused portions. The capacity of a buffer never changes.

NOTE Each buffer has a method called `remaining()`, which returns the number of slots left that can be read or written. This value is equal to `limit() - position()`.

1.2.4 flip() and clear()

Buffers are commonly used to read data from one channel and then to write that same data out to another channel. In this case, the buffer alternates between being written to and being read from. The `flip()` and `clear()` methods are called between these reads and writes, in order to prepare the buffer for each new phase in the

process. The following sequence describes the process in detail.

At the beginning, the buffer is brand new. Its limit is set to its capacity, and its position is set to 0 (as shown in figure 1.1).

In figure 1.1, the underlying array has a length of 8. The position is set to 0, while the limit and capacity values are set to 8. The limit *looks* like it is too large, since, technically, it points past the end of the usable area of the array. But if you'll recall, the definition of the limit is that it is the first slot that *shouldn't* be written to.

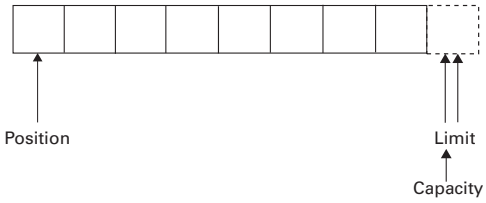

Figure 1.1 When the buffer is initialized, its position is set to 0, and its limit and capacity are set to the length of the array.

The `read()` method of the source channel is then called, and it places some data in the buffer. This data may or may not fill the buffer. The limit is still set to the capacity, while the position has advanced (see figure 1.2).

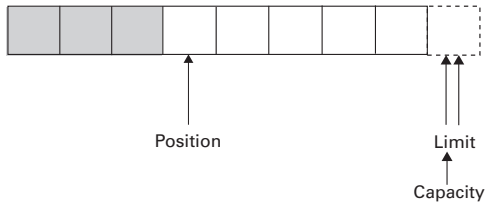

Figure 1.2 After writing some data, the position has advanced, while the limit and capacity are unchanged.

Some more data is read from the channel and placed into the buffer. The buffer position advances further (see figure 1.3).

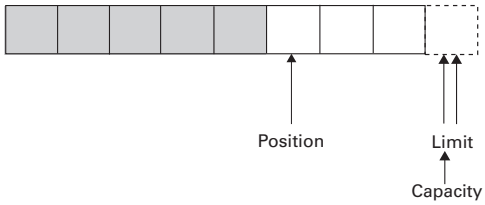

Figure 1.3 After writing more data, the position has advanced further.

The writing phase is now over. `buffer.flip()` is called to prepare the buffer to have its data read (see figure 1.4). (You can think of the `flip()` method as flipping a switch between reading and writing modes. Buffers don't actually have reading and writing modes—you can mix `read()` and `write()` calls freely. However, it is very common to use a buffer in the way we are using it here—you do some reading, flip the buffer, and do some writing.)

In order to prepare for reading, the value of limit must be changed. Before the call to `flip()`, the buffer was being used as an empty area into which data could be put; the limit value specified the end of this empty area. Now that `flip()` has been called, the buffer is being used as a source of data, and the limit value now specifies

the end of this valid data. This limit value is equal to the value that position had before `flip()` was called.

Next, the buffer is passed to the `write()` method of the destination channel, which in turn reads some data from the buffer (see figure 1.5).

The reading process continues until the position reaches the limit, at which point there is no more data in the buffer (see figure 1.6).

The reading phase is now over. At this point, the `clear()` method is called (see figure 1.7).

The position is set to 0, while the limit is set to the capacity, leaving as large a space as possible for use in the next writing phase.

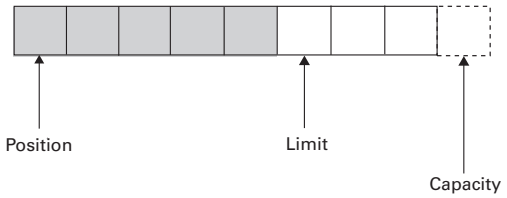

Figure 1.4 After calling `flip()`, the limit is set to the old value of position, and the position is set to 0.

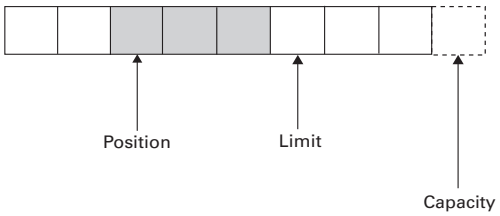

Figure 1.5 The reading process begins—as bytes are read, the position advances.

1.2.5 slice() and subbuffers

The `slice()` method allows you to create a *subbuffer* of a given buffer. A subbuffer is just another buffer that happens to share its data with a portion of the data in the buffer it was created from. It is, nevertheless, a separate buffer with its own position, limit, and capacity. The subbuffer does not have to start at the first element of the original buffer.

When `slice()` is called, it takes the current position and limit values and uses them to define the new subbuffer. The capacity and limit of the subbuffer are set to be the limit of the original buffer, and the first element of the subbuffer

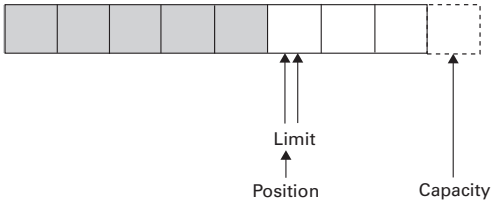

Figure 1.6 All of the data has been read, making position=limit.

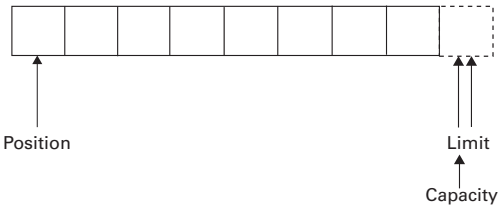

Figure 1.7 After `clear()` is called, position is set to 0 and limit is set to capacity.

corresponds to the element at value *position* within the original buffer (see figure 1.8).

In figure 1.8, the subbuffer corresponds to the second through fifth elements of the original buffer, inclusive. This corresponds to the following code:

```
ByteBuffer original = ByteBuffer.allocate( 8 );
original.position( 2 );
original.limit( 6 );
ByteBuffer slice = original.slice();
```

The individual data elements pointed to by the two buffers are in fact the same data. Thus, any change to the shared data in one buffer will be immediately reflected in the other.

1.2.6 Buffers of other types

`ByteBuffer`s are the most basic form of buffer, and it is used throughout the New I/O API. However, it is possible to have buffers of other types. In fact, there is a type of buffer for each primitive Java type. Each of these types is a subclass of `Buffer`.

A buffer of a non-byte type stores values of that type, the way that a `ByteBuffer` stores bytes. Each buffer type has five `put()` methods and four `get()` methods, just like a `ByteBuffer` (see section 1.2.2), except that these methods work with their particular type rather than on bytes.

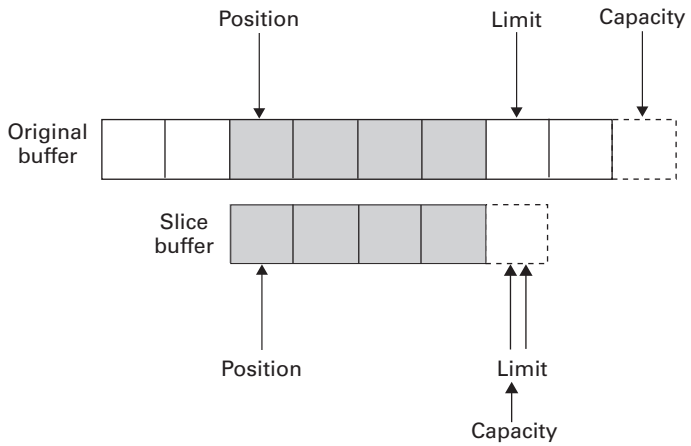

Figure 1.8 A slice buffer shares a subsequence of the original buffer. It has its own position, limit, and capacity values and does not have to start at the same position as the original buffer.

Underlying each typed buffer is a `ByteBuffer` that contains the raw bytes from which the values are built. The float values and the byte values are merely different views onto the same stream of bytes, as shown in figure 1.9.

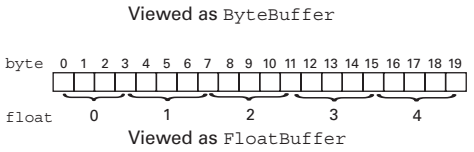

Figure 1.9 The same underlying data can be viewed as a `ByteBuffer` and as a `FloatBuffer`.

Creating a typed buffer is easy. For example, to create a `FloatBuffer`, you call the `asFloatBuffer()` method of `ByteBuffer`:

```
ByteBuffer buffer = ByteBuffer.allocate( size );
FloatBuffer floatBuffer = buffer.asFloatBuffer();
```

Since you have access to both `buffer` and `floatBuffer`, you can access this data as bytes or as floats. Note that you have *two* buffers here, each with its own position, limit, and capacity values.

Suppose, for example, you wanted to read a series of floating-point values from a channel: you could read from the channel into the `ByteBuffer`, and then read the floats from the `FloatBuffer`. Since these two buffers point to the same data, the floating-point values in the `FloatBuffer` are made up of the bytes in the `ByteBuffer`.

```
float floatArray[] = new float[floatArraySize];

FileInputStream fin = new FileInputStream( file );
FileChannel fch = fin.getChannel();

ByteBuffer buffer = ByteBuffer.allocate( floatArray.length*4 );
FloatBuffer floatBuffer = buffer.asFloatBuffer();

fch.read( buffer );

for (int i=0; i<floatArray.length; ++i) {
  floatArray[i] = floatBuffer.get();
  System.out.print( floatArray[i]+" " );
}
```

It's important to remember that the position and limit values of the two buffers are *independent* of each other. This means, for example, that although the `FloatBuffer` might be exhausted by the reading process, the `ByteBuffer` is still ready to read from the beginning—its position value is still 0.

1.2.7 Reading and writing other types from a ByteBuffer

There is another way to read floating-point values from a stream of bytes. `ByteBuffer` has a number of convenience methods that allow you to read values of other

types—floats, shorts, and so on—directly from a `ByteBuffer`. This is particularly useful if you want to read a set of mixed-type values from a buffer.

Figure 1.10 illustrates a series of mixed-type values packed into a single `ByteBuffer`.

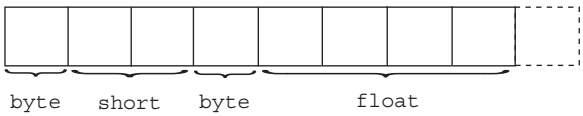

Figure 1.10 A series of mixed-type values packed into a single `ByteBuffer`

The code that reads this series of values is as follows:

```
FileInputStream fin = new FileInputStream( filename );
FileChannel fch = fin.getChannel();
ByteBuffer bb = ByteBuffer.allocate( 32 );
fch.read( bb );
bb.flip();
byte b0 = bb.get();
short s0 = bb.getShort();
byte b1 = bb.get();
float f0 = bb.getFloat();
```

The choice of whether to use typed buffers, such as `FloatBuffer`, or the typed accessor methods, such as `ByteBuffer.getFloat()` and `ByteBuffer.putFloat()`, depends on the homogeneity of the data involved. A `FloatBuffer` consists entirely of floats, and so is good for reading banks of uninterrupted floating-point data. A `ByteBuffer`, on the other hand, might be ideal for reading file headers that contain data of different types.

WARNING The default byte order of a `ByteBuffer` is big-endian*, but this can be changed using the `ByteBuffer`'s `order( ByteOrder )` method. The `order()` method can be used to find out the `ByteBuffer`'s current byte order. You can find out the platform's native byte order with the `ByteOrder.nativeOrder()` static method.

*The terms *big-endian* and *little-endian*, borrowed from Jonathan Swift, refer to two different methods for ordering bytes within a multi-byte value. The big-endian methods puts the most significant byte first and the least significant byte last; thus, the 32-bit hexadecimal value AABBCCDD is stored with the AA byte first and the DD byte last. In contrast, the little-endian method would store the DD byte first and the AA byte last.

1.2.8 Direct buffers

Direct buffers are buffers whose underlying data arrays are allocated in such a way that I/O operations can be performed considerably faster. Typically, data that crosses the boundary between the Java Virtual Machine (JVM) and the underlying operating system has to be copied to or from a Java array to an array within the JVM before it can be passed to the operating system (see figure 1.11).

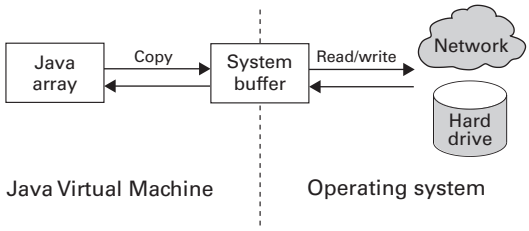

Figure 1.11 When writing from an array or nondirect `Buffer`, data must be copied to an intermediate buffer before it can be written to disk. Depending on the operation, even more copying steps may be required.

Direct buffers, however, allocate their data directly in the runtime environment memory (see figure 1.12).

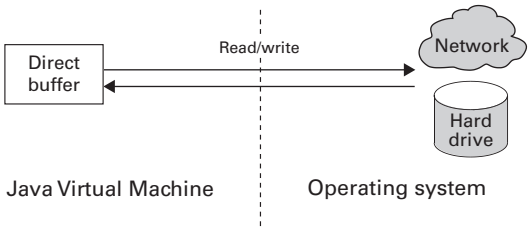

Figure 1.12 Direct buffers use system buffers for their underlying storage. In some operating systems, this means that no copying is necessary for reading and writing data—the data is transferred directly to and from the disk.

Although the actual implementation of direct buffers differs from platform to platform, it is expected that any reasonable implementation will take pains to reduce copying of data in direct buffers. These buffers should reside as close to the operating system as possible, to reduce the number of copying steps as much as possible.

You might be tempted to allocate all buffers as direct buffers, but this would be a bad idea. Direct buffers should only be used for buffers that will actually benefit from the speed increase. Direct buffers generally cost more to allocate, and may require more system resources during their lifetimes. Again, this depends on the implementation.

NOTE Direct buffers do not contain *faster memory*. They simply contain memory that can be accessed directly by the runtime system and/or the underlying operating system, so that data that is passed in and out of the JVM will not have to be copied, thus saving time.

Allocating a direct buffer is trivial. Instead of calling

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

you call

```
ByteBuffer buffer = ByteBuffer.allocateDirect( 1024 );
```

This is the only way you can create a direct buffer. Note that you clearly can't use the `wrap()` methods, because they explicitly construct a buffer using an array that is within the JVM—a Java array. Similarly, the `array()` method, which returns the underlying Java array of a `Buffer`, if any, will not work for a direct buffer, since there is no underlying Java array.

To demonstrate the speed advantage of direct buffers, let's try modifying the CopyFile program from section 1.1.5 (listing 1.1) to use a direct buffer instead of a regular buffer. The modification is shown in listing 1.2.

Listing 1.2 from FastCopyFile.java

(see \Chapter1\FastCopyFile.java)

```
FileChannel inc = fin.getChannel();
FileChannel outc = fout.getChannel();

ByteBuffer buffer = ByteBuffer.allocateDirect( 1024 );

while (true) {
  int ret = inc.read( buffer );
  if (ret==-1)
    break;
  buffer.flip();
  outc.write( buffer );
  buffer.clear();
}
```

On the system used to test the code, copying a 16-MB file with CopyFile took nine seconds, while copying it with FastCopyFile took about 4.5 seconds—a 50% savings!

1.2.9 Example: TCP/IP forwarding

Let's take a look at buffers in action in a program that does TCP/IP forwarding. This is a perfect application for buffers, because it's mostly about transferring data, rather than processing it.

TCP/IP forwarding

Forwarder is a simple TCP/IP forwarding program. It forwards TCP/IP connections coming into the forwarding machine to any of a number of destination machines. Data that is sent from the source machine to the forwarding machine is *forwarded* to the destination machine. Response data from the destination machine is sent back to the corresponding source machine (see figure 1.13).

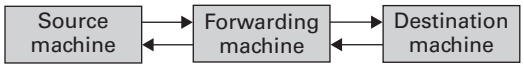

Figure 1.13 Packages from the source machine are forwarded to the destination machine by the forwarding machine. Response data from the destination machine is likewise forwarded, sent back to the source machine.

The idea here is to simulate a connection that does direction from the source machine to the destination machine. The forwarding machine acts as an intermediary. It forwards the data between the machines and tries not to interfere otherwise, much as an Internet router does (see figure 1.14).

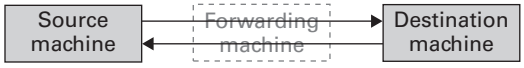

Figure 1.14 The forwarding machine seeks to simulate a *direct connection* between the source and destination machines. It forwards the data but does not modify it in any way.

It's important to understand that this is a TCP/IP forwarder, not an IP forwarder. That is, the forwarding is happening at the level of TCP virtual circuits, rather than at the IP packet level. This means that the forwarder cannot act as *transparently* as an Internet router. For example, data arriving at the destination machine is marked as having come from the forwarding machine, not from the source machine. Some connections will have a problem with being forwarded in this way, but many will work fine.

One use for a program like this is as a simple *firewall*. The forwarding machine is set up as the only machine that is accessible from the Internet. A set of other machines are connected to the firewall; these are said to be "behind" the firewall because external machines can only reach them by crossing through the forwarder firewall (see figure 1.15). The forwarder can completely control which connections will be accepted and which destination machines they will be forwarded to.

Configuring the forwarder

A *configuration file* is used to define what ports will accept connections, what destination machines they will be forwarded to, and which source machines they will be accepted from. The configuration file looks like this:

```
100     panix.com             80
110     panix.com             23
5555    www.w3c.org           80
```

The configuration file is defined more completely in the notes following listing 1.3.

To use this program, you must specify the name of the configuration file on the command line, as follows:

```
java Forwarder forwarder.cfg
```

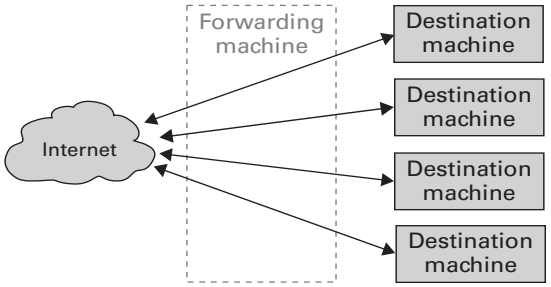

Figure 1.15 The forwarder can be used as a firewall, protecting machines hidden behind the forwarding machine from direct Internet access. Connections to the destination machines must be explicitly allowed by the forwarder.

Source code

Let's take a look at the source in listing 1.3.

Listing 1.3 Forwarder.java

(See \Chapter1\Forwarder.java)

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;
import java.util.regex.*;

public class Forwarder
{
  static private final int bufferSize = 0x4000; //BUFFER_SIZE

  // Defines the format of the config file lines
  private static final int LOCAL_PORT_POS = 0;
  private static final int HOSTNAME_POS = 1;
  private static final int PORT_POS = 2;
  private static final int SOURCES_POS = 3;
```

```
public Forwarder() {
}

public void readConfig( String configFilename )
    throws IOException {
  FileReader fr = new FileReader( configFilename );
  LineNumberReader lnr = new LineNumberReader( fr );
  while (true) {
    String line = lnr.readLine();
    if (line==null)
      break;
    Pattern pattern = Pattern.compile( "\\s+" );
    String strings[] = pattern.split( line );
    if (strings.length < SOURCES_POS) {
      System.err.println( "Config file syntax error at "+
                          configFilename+":"+
                          lnr.getLineNumber() );
      System.exit( 1 );  // Any syntax error and we quit
    }

    // First, the local forwarding port
    int forwardingPort = Integer.parseInt( strings[LOCAL_PORT_POS] );

    // Then the destination address
    InetAddress destAddress =
      InetAddress.getByName( strings[HOSTNAME_POS] );

    // Then the destination port
    int destPort = Integer.parseInt( strings[PORT_POS] );

    // Finally, zero or more permitted sources
    InetAddress sources[] = new InetAddress[strings.length-SOURCES_POS];
    for (int i=SOURCES_POS; i<strings.length; ++i) {
      sources[i-SOURCES_POS] = InetAddress.getByName( strings[i] );
    }
    AddressSet allowedSources = new AddressSet( sources );

    addForward( forwardingPort, destAddress,
                destPort, allowedSources );
  }
}

public void addForward( int forwardingPort,
                        InetAddress destAddress,
                        int destPort,
                        AddressSet allowedSources ) {
  InetSocketAddress destSocketAddress =
    new InetSocketAddress( destAddress, destPort );
  ForwarderListenerThread flt =
    new ForwarderListenerThread( forwardingPort,
                                 destSocketAddress,
                                 allowedSources );
```

1 **Parse the configuration file**

2 **Use regular expressions to parse**

3 **Add a forward for each config line**

4 **Set up a forward**

```
    flt.start();
  }

  class ForwarderListenerThread extends Thread
  {
    private int forwardingPort;
    private SocketAddress destAddress;
    private AddressSet allowedSources;
    private HashSet forwardsConnections = new HashSet();

    // Used to prevent shutdown while listening is
    // in progress.
    private Object connectionsLock = new Object();

    // Have we already shut down?
    private boolean shutdown = false;
    private ServerSocketChannel ssc;

    public ForwarderListenerThread( int forwardingPort,
                                    SocketAddress destAddress,
                                    AddressSet allowedSources ) {
      this.forwardingPort = forwardingPort;
      this.destAddress = destAddress;
      this.allowedSources = allowedSources;
    }

    public void run() {
      try {
        ssc = ServerSocketChannel.open();
        ssc.configureBlocking( true );
        ServerSocket ss = ssc.socket();
        byte anyIP[] = { 0, 0, 0, 0 };
        InetAddress forwardingHost =
          InetAddress.getByAddress( anyIP );
        InetSocketAddress isa =
          new InetSocketAddress( forwardingHost, forwardingPort );
        ss.bind( isa );
        synchronized( connectionsLock ) {
          System.out.println( "Listening on "+isa );
          while (true) {
            SocketChannel source = ssc.accept();
            InetAddress connectingAddress =
              source.socket().getInetAddress();
            // Check to see if incoming connection is from
            // an approved host
            if (allowedSources.contains( connectingAddress )) {
              SocketChannel dest = SocketChannel.open();
              source.configureBlocking( true );
              dest.configureBlocking( true );
              dest.connect( destAddress );
              ForwarderThread forwards =
                new ForwarderThread( this, source, dest );
              ForwarderThread backwards =
```

5 **Handle a forward**

6 **Listen on the forwarding port**

Accept an incoming connection

7 **Set up forwarders for the new connection**

```
            new ForwarderThread( this, dest, source );
          forwards.start();
          backwards.start();
          forwardsConnections.add( forwards );
        } else {
          System.out.println( "Connection from "+
                              connectingAddress+
                              " refused" );
          try {
            source.close();
          } catch( IOException ie ) {
            System.err.println( "Problem disconnecting "+
                                "rejected connection from "+
                                connectingAddress );
            ie.printStackTrace();
          }
        }
      }
    }
  } catch( AsynchronousCloseException ace ) {
    System.err.println( "Closed forward "+this );
    // We don't call shutdown here, because this
    // exception is triggered by the close() call
    // inside shutdown -- this exception is a *result*
    // of shutting down, not an instigation to do so.
  } catch( IOException ie ) {
    System.err.println( "Exception forwarding "+this+": "+ie );
    ie.printStackTrace();
    shutdown();
  }
}

synchronized public void shutdown() {
  if (shutdown)
    return;

  try {
    System.out.println( "Closing "+ssc );
    ssc.close();
  } catch( IOException ie ) {
    System.err.println( "Error closing "+ssc );
    ie.printStackTrace();
  }

  synchronized( connectionsLock ) {
    for (Iterator it = forwardsConnections.iterator();
         it.hasNext();) {
      ForwarderThread ft = (ForwarderThread)it.next();
      System.out.println( "Closing "+ft );
      ft.shutdown();
    }
  }
```

7 **Set up forwarders for the new connection**

● **Shut down this ForwarderListenerThread**

● **Close the ServerSocketChannel**

8 **Close all Forwarder-Threads for this port**

```
    shutdown = true;
  }

  public void remove( ForwarderThread ft ) {
    if (forwardsConnections.contains( ft ))
      forwardsConnections.remove( ft );
  }

  public String toString() {
    return forwardingPort+"-->"+destAddress;
  }
}

static class ForwarderThread extends Thread {
  private ForwarderListenerThread flt;
  private String description;
  private SocketChannel from;
  private SocketChannel to;
  private boolean shutdown = false;

  public ForwarderThread( ForwarderListenerThread flt,
                          SocketChannel from, SocketChannel to ) {
    this.flt = flt;
    this.from = from;
    this.to = to;

    Socket fromSocket = from.socket();
    Socket toSocket = to.socket();
    description =
      fromSocket.getInetAddress()+":"+fromSocket.getPort()+
      "-->"+
      toSocket.getInetAddress()+":"+toSocket.getPort();
  }

  public void run() {
    try {
      ByteBuffer buffer = ByteBuffer.allocateDirect( bufferSize );
      while (true) {
        from.read( buffer );
        if (buffer.position()==0) {
          System.out.println( "Closing on zero read: "+this );
          break;
        }
        System.out.println( this+" read "+buffer.position() );
        buffer.flip();
        while (buffer.remaining()>0) {
          int r = to.write( buffer );
          System.out.println( this+" wrote "+r+", remaining "+
                              buffer.remaining() );
        }
        buffer.clear();
      }
      shutdown();
```

9 Remove a dead ForwarderThread

10 Handle a forwarded connection

11 Copy bytes from one end of the connection to the other

```
    } catch( AsynchronousCloseException ace ) {
      System.err.println( "Closed forward "+this+": "+ace );
      shutdown();
    } catch( IOException ie ) {
      System.err.println( "Exception forwarding "+this+": "+ie );
      ie.printStackTrace();
      shutdown();
    }
  }

  public void shutdown() {
    if (shutdown)
      return;

    try {
      from.close();
    } catch( IOException ie ) {
      System.err.println( "Error closing from of "+this );
      ie.printStackTrace();
    }

    try {
      to.close();
    } catch( IOException ie ) {
      System.err.println( "Error closing to of "+this );
      ie.printStackTrace();
    }

    shutdown = true;
    flt.remove( this );
    System.err.println( "Closed forward "+this );
  }

  public String toString() {
    return description;
  }
}

static class AddressSet {
  private Set addresses = new HashSet();

  public AddressSet( InetAddress ias[] ) {
    for (int i=0; i<ias.length; ++i) {
      System.out.println( "as "+ias[i] );
      addresses.add( ias[i] );
    }
    System.out.println( "address set size "+addresses.size() );
  }

  public boolean contains( InetAddress ia ) {
    if (addresses.size()==0)
      return true;
    return addresses.contains( ia );
  }
```

● **Shut down this ForwarderThread**

⓬ **Close both ends of the forward**

⓬ **Close both ends of the forward**

● **Remove this ForwarderThread from the parent ForwarderListeningThread**

⓭ **Represent a set of hosts**

```
  }

  static public void main( String args[] ) throws IOException {
    String configFilename = args[0];

    Forwarder forwarder = new Forwarder();
    forwarder.readConfig( configFilename );
  }
}
```

1. The configuration file specifies each local port that will be forwarded. For each local port, it defines the remote hostname and port that the local port will be forwarded to.

2. Here, we use the regular expression facility in the `java.util.regex` package. The `Pattern` object represents a string pattern to look for—in this case, we are looking for any white space. The `split()` method searches the string for every occurrence of the pattern and divides the string in those locations, producing a set of smaller strings. The white space is not included in the smaller strings.

3. A forward is created for each line in the configuration file. As an example, the following line forwards port 5555 on the local machine to the web server at www.w3c.org:

```
5555    www.w3c.org          80
```

This results in the configuration shown in figure 1.16.

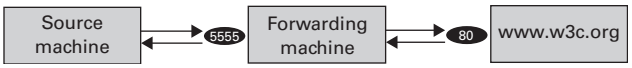

Figure 1.16 The forwarder is configured to forward traffic on local port 5555 to port 80 at remote machine, www.w3c.org.

Optionally, you can specify a list of hosts at the end of the line. Only hosts from this list can connect to the forwarding port; all others will be rejected:

```
5555    www.w3c.org          80    192.168.0.1 127.0.0.1
```

Here, `192.168.0.1` is an address on our local network; `127.0.0.1` is the loopback address, referring to the same machine on which the forwarder is running. If you do not specify any hosts at the end of the line, all hosts will be accepted.

4. `addForward()` sets up a forward from a local port to a remote host and port. A `ForwardListenerThread` is created for each forward; this thread runs in the background, listening on the specified local port for an incoming connection. If this incoming connection is allowed, a pair of `ForwarderThreads` will be created to handle the connection.

5. A `ForwarderListenerThread` handles a single forwarded port. Each port that is forwarded by the forwarder has its own `ForwarderListenerThread`. This object is responsible for listening on the local port and accepting incoming connections to that port. Each time a connection comes in, the `ForwarderListenerThread` creates a pair of `ForwarderThread`s to handle the connection—one for each direction of the communication. When a `ForwarderListenerThread` is shut down, it shuts down all the `ForwarderThread`s that it has created.
6. The `ForwarderListenerThread` listens on the address `0.0.0.0`, which is the any address. This means that an incoming source connection can be made against any IP address assigned to this machine.
7. Each connection is handled by a pair of `ForwarderThread` objects—one for the forward direction and one for the backward direction. The forward direction goes from the source machine to the destination machine, while the backward direction goes from the destination machine to the source machine.
8. This method can be called either by this class or by another class. Shutting down a `ForwarderListenerThread` means closing the socket used for accepting new connections, but it also means removing all existing connections that are currently being forwarded through this port.
9. Calling this method removes the `ForwarderThread` from the set of currently open threads. This method is called by a `ForwarderThread` when it terminates on its own. This happens when either the source or destination hosts close the connection. Generally, both `ForwarderThread`s in a pair are shut down at the same time.
10. Each forwarded connection is handled by a pair of `ForwarderThread`s, one for each direction of the forwarded connection.
11. Inside the `ForwarderThread`, data is copied from one end of the forwarded connection to the other using a `ByteBuffer`. In each step, `from.read()` is called to read as much data as possible into the `ByteBuffer`. This amount is limited by the amount of available data and by the size of the `ByteBuffer`. The `ByteBuffer` is passed to `to.write()`, which writes all of the data to the outgoing connection, either in one step or in a series of steps inside the `while()` loop. (Multiple steps might be required since this is a network connection, and limits on the size of network buffers can cause a partial write. In contrast, our CopyFile program in section 1.1.5 (listing 1.1) can safely assume that every write will block until all the data is written, or an exception will be thrown.) Note that `flip()` is used after the `read()`, and `clear()` is used after the `write()`, as described previously in section 1.2.4.

    The advantage of using a `ByteBuffer` over a simple byte array is that a `ByteBuffer` keeps track of how much has been read or written during each phase of the transfer. It respects the capacities of the underlying arrays, taking care never to try

to read more than can fit, or to write more than there is to write. In short, it takes care of the housekeeping that we normally have to take care of ourselves, and since the `Buffer` code has been extensively tested, it's more likely to work the first time.

⓬ `shutdown()` handles shutting down a `ForwarderThread`. Both the from and to `SocketChannel`s are closed, for good measure. Generally, when one of a pair of `ForwarderThread`s is shut down, the other shuts down automatically, as well, because the `SocketChannel`s have both had their `close()` methods called.

⓭ An `AddressSet` represents a set of hosts. In this program, it is used to specify the set of hosts from which a source connection will be accepted. An empty `AddressSet` accepts all hosts.

1.2.10 Doing I/O with channels and buffers

The channel-and-buffer approach to doing I/O is different than the stream-oriented approach that has been the mainstay of Java programming since the beginning of Java.

The stream-oriented approach provides flexibility and convenience. All streams have, more or less, the same interface. Creating a subclass of an `InputStream` or `OutputStream` can be as simple as overriding a single method—`read()` or `write()`, respectively. Filters allow for arbitrary transformations on the data passing through a stream without complicating the situation for the source of the data, or for its destination. Streams count on the fact that all data is, at the lowest level, built from chains of bytes.

The channel-and-buffer approach has a different focus. Channels and buffers deal in *bulk data*. Here, *bulk* means that the data is dealt with in large pieces, and *data*, as opposed to actual Java *bytes*, means that the low-level data isn't manifest. This approach encourages the use of behind-the-scenes trickery to greatly speed up data transfers. And that's really the whole point of the bulk data approach. In particular, raw data does not have to be stored in actual Java `byte` arrays; it can be stored in low-level memory buffers. Here, low-level can simply mean that the buffers are allocated from the user-space heap within the JVM process, or, in some cases, the buffers can be system-level buffers that are shared between kernel and user processes using memory-paging techniques. In any case, allowing the implementation to use special buffering methods allows for optimizations that would not be possible if the data was stored in regular Java arrays.

However, this bulk data approach also means that the data must truly be hidden behind a complete abstraction barrier. Since every implementation can implement the buffers differently, the data cannot be *manifest*—it cannot be accessible directly as data. It can only be accessed through methods—`get()` and `put()`—which transfer

data between the hidden behind-the-scenes implementation and a manifest Java variable or array. (It's true that non-direct buffers can reveal their underlying Java arrays via the `array()` method, but since this method is only available for non-direct buffers, it should be considered as something of an optional feature, at least as far as the core metaphor is concerned.)

Thus, the channel-and-buffer abstraction represents a *broadening* of the hidden portion of the I/O process. More computation is put behind the abstraction barrier so that more of it can be optimized on a per-platform basis. This means, potentially, more work for the implementers, but the payoff is clear: Java programmers now have access to an I/O system that can provide as much throughput as the underlying operating system and JVM will allow; at the same time, the system has a safe and complete abstraction barrier—any code written against the NIO library is going to be portable, at least to the extent that underlying implementations are correct. Custom JVMs, built to take advantage of special operating system features—or even hardware features—are conceivable; the most demanding I/O systems can now conceivably be written in Java, given the right system support.

Of course, the NIO libraries are compatible with the old I/O libraries. In fact, they are more than compatible—the old libraries have been re-engineered to use the NIO abstractions in places where this is appropriate. Let's face it—channels and buffers are cool, but they don't have the same brilliant elegance as the stream metaphor. However, you can freely mix stream-oriented and buffer-oriented I/O in the same program.

As we continue through this chapter (and the next), keep in mind that many of these I/O innovations are intended to integrate the Java I/O systems more fully with common operating system features. Next, we'll take a look at file locking.

1.3 The File Locking facility

File locking makes it possible to lock entire files, or regions of files, in order to prevent other threads or processes from accessing those files. If the underlying operating system has native file-locking capabilities, then the Java implementation will use them. As a result, the behavior of the File Locking facility is platform-dependent. However, it is possible to use the facility in such a way that the behavior will be the same on all platforms.

1.3.1 Types of locks

It's important to understand that file locking doesn't necessarily prevent the file—or the portion thereof—from being accessed. A lock that does so is called a *mandatory* lock. There are also *advisory* locks, which do *not* prevent the region from being

accessed. Instead, advisory locks prevent other locks from being acquired on the same region. These terms are not Java-specific.

It might seem that only mandatory locks would be useful, but in fact advisory locks serve nearly the same purpose. If all programs that access a particular file agree to acquire a lock on a region of a file before changing it, then advisory locks are enough, because only one such lock can be had at a given time. Using this kind of lock is rather like using the `synchronized` keyword in Java: synchronizing on an object means acquiring an advisory lock on that object. Such a lock doesn't prevent other threads from modifying the fields of the object; it only prevents other threads from acquiring locks. (This is the definition of *advisory lock*.)

The File Locking facility also supports two varieties of lock called *exclusive* and *shared*. Exclusive locks are like the locks provided by the `synchronized` keyword: they prevent other threads from acquiring a similar lock. Shared locks, on the other hand, do not. Multiple shared locks can be acquired at the same time, but they prevent exclusive locks from being acquired.

TIP It is a common practice to use shared locks for reading from a file region, because multiple threads can safely read the same region without interfering with each other. Likewise, it is common practice to use exclusive locks for writing to a file region, because multiple threads *cannot* safely write to the same region without interfering with each other.

The fact that shared locks prevent the acquisition of exclusive locks fits in with this: while threads are reading from a region, no other threads can write to it.

The exclusive versus shared distinction is completely separate from the mandatory versus advisory distinction—they are orthogonal distinctions. Thus, a mandatory lock can be exclusive or shared, and so can an advisory lock.

1.3.2 Using locks

File locks cover a contiguous region of a file. This region can be the entire file, as shown in figure 1.17.

Or it can be a portion of a file, as shown in figure 1.18.

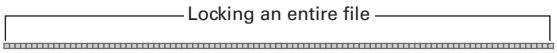

Figure 1.17 A lock can be acquired over the entire region of a file.

Figure 1.18 A lock can be acquired over a portion of a file. This portion must be contiguous.

It's possible for multiple regions of a file to be locked. If they are exclusive locks, they must not overlap; attempting to create overlapping exclusive locks within the same JVM will throw an `OverlappingFileLockException` (see figure 1.19).

Shared locks can overlap with other shared locks, but not with exclusive locks. Attempting to create an exclusive lock and a shared lock that overlap throws an `OverlappingFileLockException` (see figure 1.20).

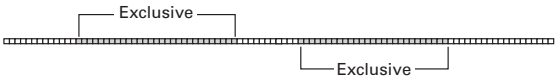

Figure 1.19 Multiple exclusive locks can be acquired within the same file, but they must not overlap.

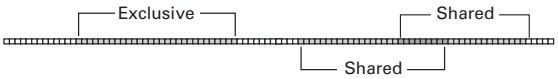

Figure 1.20 Shared locks can overlap with other shared locks, but they cannot overlap with exclusive locks.

Some operating systems do not support shared locks. In such cases, it is not an error to request a shared lock, but the lock that is returned will be an exclusive lock. The type of a lock can be determined using its `isShared()` method.

1.3.3 Acquiring locks

To create a lock, you need a `FileChannel` object. Once you have one, there are two ways to acquire a lock: the `lock()` methods will block until the lock is acquired, and the `tryLock()` methods will return `null` if the lock cannot immediately be acquired. Otherwise, the `lock()` and `tryLock()` methods are identical. For simplicity, only the `lock()` methods will be discussed.

The `lock()` method takes three arguments: a starting position (measured in bytes from the start of the file), a length (measured in bytes), and a `boolean` telling whether the lock should be shared (`true`) or exclusive (`false`):

```
FileOutputStream fout = new FileOutputStream( "abc.txt" );
FileChannel fc = fout.getChannel();
FileLock fl = fc.lock( 20, 20, false );
```

There is also an argument-free convenience method that locks an entire file. The following two code fragments are equivalent:

```
FileOutputStream fout = new FileOutputStream( "abc.txt" );
FileChannel fc = fout.getChannel();
FileLock fl = fc.lock();
```

```
FileOutputStream fout = new FileOutputStream( "abc.txt" );
FileChannel fc = fout.getChannel();
FileLock fl = fc.lock( 0L, Long.MAX_VALUE, false );
```

Locks can be released in two ways: via the `release()` method, or by closing the `FileChannel` associated with the file. Once a lock is released, exclusive locks that overlap the region can be safely acquired.

1.3.4 Portability issues

Locks created in one process may or may not interfere with locks created in another process—this depends on the implementation and the nature of the locking facility of the underlying operating system. In some cases, two processes can lock the same region of the same file; in other cases, the thread attempting to acquire the second lock will block until the first lock is released. If the operating system does, in fact, support inter-process file locking, then locks created in a Java process can interact with locks created by a program in some other language; operating system–based locks are generally language-neutral.

Although there are a number of variables that depend heavily on the underlying implementation and on the underlying operating system, it is possible to use the File Locking facility in such a way that it is portable across all platforms. The following rules will ensure portability:

- Use only exclusive locks
- Treat all locks as *advisory*—assume that acquiring a lock on a region of a file does not prevent that region from being accessed in any way
- Assume locks only affect other threads within the same process

Although these restrictions prevent the use of many of the features of the File Locking API, they should be sufficient for many purposes. Locks used under this discipline are not unlike the locks used by the `synchronized` keyword—exclusive, advisory locks that only work inside a single Java process. Such locks have proven sufficient for most purposes and can be used to build more sophisticated locking mechanisms.

1.3.5 Example: a simple database

This section describes a program called SimpleDatabase. SimpleDatabase is a tiny API for storing fixed-length blocks of data, called *slots*, indexed by number in a flat file (see figure 1.21).

A `SimpleDatabase` has a `get()` method and a `put()` method, each one taking a slot number as an argument:

```
public void get( int slot, byte data[] )
public void put( int slot, byte data[] )
```

The idea is that multiple `SimpleDatabase` objects will be in use at the same time in separate JVMs. The only concurrency guarantee provided by `SimpleDatabase` is that each `get()` and `put()` operation be *atomic*. That is, each time a slot is written, it is written *completely* before any other thread or process can read or write it. It is possible that one thread or process will immediately overwrite the data that was just written by another thread or process, but at no time will a slot contain data from one data block and data from another data block at the same time.

To test this, the SimpleDatabase program contains an inner class called `SimpleDatabaseTester`, which reads and writes data blocks very quickly to the slots of a `SimpleDatabase`. Each data block consists of a single byte repeated over and over, which makes it very easy to tell if a slot has been corrupted by an incomplete write. After each read, the `SimpleDatabaseTester` checks to make sure there has been no data corruption.

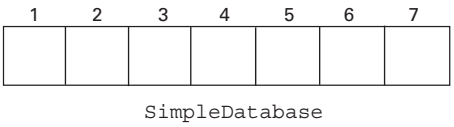

Figure 1.21 A `SimpleDatabase` is a flat file containing a set of fixed-length data slots.

You can specify the number of slots in the database, and the size of each slot, by changing the constants in the source code, which is presented in listing 1.4.

Listing 1.4 SimpleDatabase.java

(See \Chapter1\SimpleDatabase.java)

```
import java.io.*;
import java.nio.channels.*;
import java.util.*;

public class SimpleDatabase
{
  static public final int NUMSLOTS = 64;
  static public final int SLOTSIZE = 1024;
  private RandomAccessFile raf;
  private FileChannel fc;
```

```
public SimpleDatabase( String filename ) throws IOException {
  File file = new File( filename );
  boolean exists = file.exists();
  raf = new RandomAccessFile( file, "rw" );
  fc = raf.getChannel();

  if (!exists) {
    byte b[] = new byte[SLOTSIZE];
    for (int i=0; i<NUMSLOTS; ++i)
      put( i, b );
  }
}

private FileLock getLock( int slot, boolean shared )
  throws IOException {
  long position = slot*SLOTSIZE;
  long size = SLOTSIZE;

  FileLock lock = fc.lock( position, size, shared );
  return lock;
}

public void put( int slot, byte data[] ) throws IOException {
  try {
    if (data.length != slotSize)
      throw new IllegalArgumentException( "Data wrong size: "+
                                          data.length );
    FileLock fl = getLock( slot, false );
    synchronized( raf ) {
      raf.seek( slot*slotSize );
      raf.write( data, 0, slotSize/2 );
      Thread.yield();
      raf.seek( slot*slotSize+(slotSize/2) );
      raf.write( data, slotSize/2, slotSize/2 );
      raf.getFD().sync();
      Thread.yield();
    }
  } finally {
    fl.release();
  }
}

public void get( int slot, byte data[] ) throws IOException {
  try {
    if (data.length != slotSize)
      throw new IllegalArgumentException( "Data wrong size: "+
                                          data.length );
    FileLock fl = getLock( slot, true );
    synchronized( raf ) {
      raf.seek( slot*slotSize );
      raf.read( data );
    }
  } finally {
```

Open the file for read/write and get the file's Channel

1 If the file is new, clean it out

2 Create a lock for the given slot

3 Lock a slot and write data to it

4 Lock a slot and read data from it

```
    fl.release();
  }
}

public void close() throws IOException {
  raf.close();
}

static public class SimpleDatabaseTester
{
  private Random rand = new Random();
  private final int NUMSLOTS = SimpleDatabase.NUMSLOTS;
  private final int SLOTSIZE = SimpleDatabase.SLOTSIZE;
  private SimpleDatabase sd;

  public SimpleDatabaseTester( SimpleDatabase sd ) {
    this.sd = sd;
  }

  public void test() throws IOException {
    byte buffer[] = new byte[SLOTSIZE];
    int numOps = 0;
    while (true) {
      if (rand.nextInt( 100 )<50) {
        int slot = rand.nextInt( NUMSLOTS );
        generateConstantBuffer( buffer );
        sd.put( slot, buffer );
      } else {
        int slot = rand.nextInt( NUMSLOTS );
        sd.get( slot, buffer );
        confirmConstantBuffer( slot, buffer );
      }
      if (((++numOps)%50)==0) {
        System.out.println( numOps+" operations" );
      }
    }
  }

  private void generateConstantBuffer( byte buffer[] ) {
    byte b = (byte)rand.nextInt( 256 );
    for (int i=0; i<buffer.length; ++i)
      buffer[i] = b;
  }

  private void confirmConstantBuffer( int slot, byte buffer[] ) {
    int b = buffer[0];
    for (int i=1; i<buffer.length; ++i) {
      if (b != buffer[i]) {
        throw new RuntimeException( "Corrupted slot "+slot );
      }
    }
  }
}
```

Test program

5 **Flip a coin and either read or write data**

Fill a buffer with a single, repeated byte

Check a buffer to see if it is filled with a single, repeated byte

```
  static public void main( String args[] ) throws IOException {
    if (args.length != 1) {
      System.err.println( "Usage: java SimpleDatabase <filename>" );
      System.exit( 1 );
    }
    String filename = args[0];

    SimpleDatabase sd = new SimpleDatabase( filename );
    SimpleDatabaseTester sdt = new SimpleDatabaseTester( sd );
    sdt.test();
  }
}
```

1. If the file doesn't exist, it is created as a `RandomAccessFile`. If the file is read from before it is written to, it may or may not contain garbage, depending on the underlying operating system implementation. To ensure that the file contains homogeneous slots, we write the contents of a newly allocated array, which must contains zeros, to each slot in the file.

2. `getLock()` is used by both `get()` and `put()` to lock a slot before reading from it or writing to it. Each slot is locked independently. A lock can be shared or exclusive; this is specified by the second parameter. Once a lock is acquired using this call, the caller is responsible for releasing it.

3. The first line of this section acquires an exclusive lock by calling `getLock()`. The last line of this section releases the lock. In between, the data is written to the slot. You'll notice that the data is written in two pieces, and there is a call to `Thread.yield()` between these writes. This is to ensure that there is some chance that other threads and processes will get to run during this write.

   If you comment-out the first and last lines, you remove the protection this program has provided against data corruption—it becomes possible for two different threads or processes to write to the same slot at the same time. However, unless the data files are huge, the chances of this happening are very low. (We still need the locking. Saying a piece of software has a "very low chance" of data corruption is simply not sufficient.)

   In any case, writing the data in two steps, with a call to `Thread.yield()` between them, greatly increases the chances of file corruption—which allows you to experiment with this program and make a useful comparison between the safe and unsafe modes of operation.

   Note that we release the lock in a `finally` block—this is because we want to make sure that we release the lock *no matter what*. Regardless of any kind of exception that might be thrown inside the `try` block, we'll still release the lock.

4. The `get()` method gets a shared lock instead of an exclusive lock because it is reading, not writing, the data. It is safe for multiple threads or processes to read the same slot at the same time.
5. Flip a coin using `Random.nextInt()`. On the basis of that coin flip, either read a slot or write a slot. In both cases, pick the slot randomly.

1.4 Summary

The New I/O API likely is not a replacement for the stream-based `java.io` package. The stream metaphor is extremely elegant and flexible, and efficient enough for many purposes. The NIO package does, however, provide a lower level at which to write I/O code, for situations where efficiency is important. In the current release, the old system has been cleanly re-implemented on top of the new API, where appropriate.

The New I/O API provides a completely new paradigm for doing efficient I/O. By taking over the control not only of data sources and sinks (`Channels`), but also of buffers that hold the data (`Buffers`), the new API can provide tremendous speed improvements while also providing more powerful features, including simple data-type conversion, nonblocking I/O, multiplexed I/O, and integrated support for character set encodings (`Charsets`). These topics are discussed in depth in chapter 2, "Advanced NIO."

2 Advanced NIO (New Input/Output)

This chapter covers

- Reading and writing data with MappedByteBuffers
- Optimizing network communication using nonblocking I/O
- Encoding and decoding with Charsets
- Discovering and using NetworkInterfaces

This chapter covers the more advanced features of the NIO (New Input/Output) system, such as multiplexed I/O and nonblocking I/O. All of these features make use of the channel and buffer objects from the `java.nio` package that are described in detail in chapter 1, "Basic NIO." It is not necessary to read chapter 1 in its entirety before attempting this chapter; however, a perusal of sections 1.1 and 1.2 of that chapter will provide the basics required to understand the topics discussed here.

The features discussed in this chapter rely on the new concepts and classes of the NIO system; they are some of the main features that motivated the development of the NIO system. In this chapter, you'll find out about techniques that can go a long way toward making your Java application ready for heavy, real-world use.

2.1 Reading and writing with MappedByteBuffers

A `MappedByteBuffer` allows you to map a portion of a file into a memory buffer. The contents of the file are presented to the program as `Buffer`, which can be read from and written to like any other buffer. Changes made to the buffer are automatically propagated to the file by the underlying implementation of `MappedByteBuffer`.

File mapping provides an I/O *metaphor* that's entirely different from the kinds traditionally used to read and write files. Instead of treating the file as a kind of random-access stream, the file is treated like a gigantic array. This can make certain operations much easier to perform, but the main advantage is *speed*—data is retained in buffers at the operating system (OS) level, rather than being copied into user-space memory for each `read()` and out of user-space memory for each `write()`.

Of course, reading an entire file into memory would serve the same purpose, but, in general, files can be too large to read into memory. Some applications take the trouble to read in only those portions that are needed, but this is more complicated. `MappedByteBuffer`s have the same interface as regular `ByteBuffer`s, and so are easy to use.

A `MappedByteBuffer` uses *loading-on-demand*—that is, it loads into memory only the data that is being accessed. Thus, it is possible to create a 100-MB `MappedByteBuffer` that maps to a 100-MB file, and change an arbitrary byte in the file by using the `put()` method of the `MappedByteBuffer`. This will only cause the small portion around the changed byte to be loaded into memory. The blocks that aren't loaded don't take up any memory.

2.1.1 Advantages of MappedByteBuffers

The primary advantage of a memory-mapped file is *speed*. The underlying implementation is operating system–dependent, but under some implementations, a mapped byte buffer gives you direct access to the operating system–level buffers.

This means that it can be possible for you to access your file *without any copying*—the fastest access possible.

Because memory-mapped files are tied so directly to operating system details, the design of the operating system can influence the way that the mapped buffers operate. Generally speaking, for the sake of efficiency, file I/O is performed in units called *blocks*, rather than on a byte-by-byte basis. The size of the blocks varies but is usually in the range of a few kilobytes.

Memory that is demand-loaded is therefore loaded in blocks. The `MappedByteBuffer` will only load those blocks containing a byte that has been read from, or written to. The operating system can also remove a block from memory—this generally happens only if the block hasn't been used in a while. In either case, the operating system maintains a set of blocks in memory that reflect a subset of the file data on disk.

Figure 2.1 shows what a `MappedByteBuffer` might look like after it has been in use for a while. In the `MappedByteBuffer`, the gray blocks currently reside in memory, while the clear blocks do not.

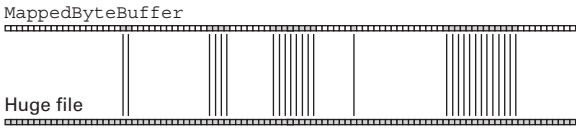

Figure 2.1 A `MappedByteBuffer` only loads the blocks that have been accessed. The gray blocks within the `MappedByteBuffer` have been loaded from the file, while the clear ones have not. The clear blocks do not take up any memory at all, which allows very large files to be handled without running out of RAM.

Under some operating systems, files are always read by mapping their data into blocks of memory, and these blocks are shared by all processes that access that file. An implementation of the `MappedByteBuffer` can take advantage of this by using these low-level blocks directly. One side effect of this technique is that changes made to the contents of the `MappedByteBuffer` are seen instantly by other programs that are reading the file. Likewise, changes made to the file by other programs are seen instantly within the content of the `MappedByteBuffer`.

WARNING The propagation of changes between different programs accessing the same file is operating system–dependent. Changes made at one point in one program may or may not be seen at another point in another program. The only consistency guarantee provided by the New I/O API is that multiple instances of `MappedByteBuffer` in the same program will be consistent with each other.

2.1.2 Disadvantages of MappedByteBuffers

A `MappedByteBuffer` is not the most usual way to access a file—streams are used more often for day-to-day file I/O. However, memory-mapped I/O has become common on modern operating systems because it fits in so well with the underlying paging system, and it is used by the OS itself, as well as by high-performance applications like database engines.

Reading and writing data via a `MappedByteBuffer` takes some getting used to. Generally, when we modify an array, we know that the modified data won't be saved until we actually write it to a file. This is not the case for `MappedByteBuffers`—the changes are made *directly to the file itself* (but see the discussion of *copy-on-write* semantics in annotation 1 of listing 2.1). This means we must take great care when modifying the data in a `MappedByteBuffer` in order to avoid leaving the file in a corrupted state. We're used to modifying data structures such as arrays or buffers, and *then* writing them out when our modifications are done. With a `MappedByteBuffer`, each tiny change is instantly saved to the filesystem.

2.1.3 Using MappedByteBuffers

A few utility methods can help you make better use of `MappedByteBuffers`:

- `force()`—After calling this method, any changes made to any portion of this `MappedByteBuffer` will be flushed out to the underlying storage device, usually a disk. Generally, the operating system does this automatically, but not immediately, after any write. Instead of letting the operating system take care of this at its own discretion, you can call `force()` after every `write()` to ensure that the data is in fact safely stored on the hard drive, but this is much slower.
- `load()`—As mentioned previously, reading or writing any byte of a block will cause that block to be loaded into memory. You can also cause an explicit load of all the blocks in a `MappedByteBuffer` by calling `load()`.
- `isLoaded()`—Calling this method allows you to find out whether all the data in the `MappedByteBuffer` has been loaded into RAM. This is only a hint, not a guarantee, because the underlying operating system is free to page data in and out of virtual memory at any time.

WARNING Calling `force()` on a `MappedByteBuffer` that corresponds to a remote storage device does *not* guarantee that all changes that have been made to the data have been flushed to that storage device. The consistency guarantee only applies to local storage devices.

2.1.4 Example: checksumming

FastChecksum (see listing 2.1) illustrates the technique of reading a file using a `MappedByteBuffer`. This program maps a portion of a file into memory and does a crude checksum on it (really, just a sum of all the bytes). Taking checksums of files is a quick way of comparing them. If the checksums differ, then the files differ; if the checksums are the same, then the files are probably, but not necessarily, the same.

Listing 2.1 FastChecksum.java

(see \Chapter2\FastChecksum.java)

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

public class FastChecksum
{
  static public void main( String args[] ) throws Exception {
    if (args.length != 3) {
      System.err.println( "Usage: java FastChecksum "+
                          "<filename> <start pos> <# bytes>" );
      System.exit( 1 );
    }
    String filename = args[0];
    int start = Integer.parseInt( args[1] );
    int length = Integer.parseInt( args[2] );

    long fileLength = new File( filename ).length();

    if (length < start) {
      throw new IllegalArgumentException( "length < start" );
    }

    if (length < 0) {
      throw new IllegalArgumentException( "length < 0" );
    }

    if (start+length > fileLength) {
      throw new IllegalArgumentException( "start+length > fileLength" );
    }

    FileInputStream fin = new FileInputStream( filename );
    FileChannel finc = fin.getChannel();
    MappedByteBuffer mbb =
      finc.map( FileChannel.MapMode.READ_ONLY, start, length );

    long sum = 0;
    for (int i=0; i<length; ++i) {
      sum += mbb.get( i );
    }
```

1 **Map the requested portion of the file into memory**

Sum the bytes in this portion

```
    fin.close();

    System.out.println( "Sum: "+sum );
  }
}
```

1 `FileChannel.MapMode` is a typesafe enumeration for the different modes of access allowed by the `map()` method. There are three modes available:

- `READ_ONLY`—The buffer can be read from, but not written to
- `READ_WRITE`—The buffer can be read from and written to
- `PRIVATE`—The buffer can be read from and written to, but writing is done using *copy-on-write* semantics. This means that any changes that are made to the buffer are made to a private copy; they are *never* propagated to the underlying file. Generally, in order to save memory, this private copy is made on demand, and is generally done in small pieces, rather than all at once

This program should run faster than its counterpart that uses traditional I/O. However, if the command-line arguments specify that the entire file should be checksummed, then the program will in fact have to load the entire file into memory.

WARNING There is no unmap method. If there were, the memory released by such a method could be reallocated to another `MappedByteBuffer`, at which point there would be two such buffers, both of which pointed to the same file data. This would introduce the possibility of data corruption.

As a result, the mapping only becomes invalidated when the `MappedByteBuffer` object is garbage-collected.

2.2 Nonblocking I/O

One of the biggest complaints about the original `java.io` classes is that they don't support nonblocking I/O. The New I/O API has included full support, and it's one of the most important of the new features.

Nonblocking I/O is a method of carrying out read and write operations (as well as other, less-common operations) without *blocking* on the method calls that carry them out. To block on a method call means to be forced to wait until it is finished before returning. Thus, a nonblocking I/O operation is one that doesn't need to wait until the operation is finished before it returns. This can greatly decrease the overhead of managing many I/O connections, especially in a client/server environment.

This section will describe the new nonblocking I/O features in the New I/O API. We'll concentrate on using this feature to write client/server programs that can handle a great number of connections efficiently. Specifically, we'll look at two different implementations of a very simple chat system—one using *polling* and one using *multiplexing*. First, though, we'll explore some naive implementations—both single- and multithreaded. We'll be looking closely at the *server* side of this chat system. The client is rather trivial, and doesn't enlighten the discussions in this chapter—it can be downloaded from the book's web site.

2.2.1 The multithreaded approach

In the early days of Java, it was very easy to write a chat server, but it was hard to write a chat server that could handle many connections at once. This is because it was very common to create a thread for every connection. This meant that each client had its own thread on the server, so a server with many clients had many threads.

These early versions of Java could not handle a tremendous number of threads. There was a certain overhead for each thread, and even if few of the clients were chatting, the server would grind to a halt. These days, JVMs can handle more and more threads each year. But in many implementations, it's still the case that threading carries an undesirable overhead. It's hard enough dealing with a lot of I/O without having to deal with a lot of threads at the same time.

To really understand the multithreaded approach, and why it failed, it's important to understand why it was so desirable. Here's a hypothetical version of the server-side pseudocode that handled a single client:

```
while (true) {
  ChatMessage cm = client.rcvMessage();
  for (client2 in clients) {
    client2.sendMessage( cm );
  }
}
```

This code fragment reads a single message from a single client and then sends that message out to all the other clients in the system. Simple, right? That's the beauty of the multithreaded model: each client has its own thread, and inside that thread is a very simple read/write loop.

The call to `client.rcvMessage()` is a *blocking* call. This means that the call waits until a message comes in from the client. If the client is sending lots of data, this wait might be short. But sometimes a user gets up for a few minutes, or even goes to lunch. This call can take minutes, or hours, or more, before it returns. This is *another* reason why having one thread per client is so desirable—no matter how

long this thread blocks on this call, the other clients are being handled by the other threads. No one client can bring the system to a halt.

This is the essence of blocking I/O—whatever operation you're doing, you can't be doing something else in the same thread. Whatever this thread is doing for one client, it's not doing anything for any other clients. Having one thread per client takes care of this problem, but it means having too many threads.

2.2.2 The really bad single-threaded approach

Given the problems of using too many threads, you might have briefly considered the option of using a single thread. Here's one way you can do this, expressed as pseudocode:

```
while (true) {
  for (client in clients) {
    ChatMessage cm = client.rcvMessage();
    for (client2 in clients) {
      client2.sendMessage( cm );
    }
  }
}
```

The idea here is to deal with each client in turn—you get a message from the first client and send it out to everyone. Then you get a message from the second client and send it out to everyone. And it all works inside a single thread!

Unfortunately, this is even worse. If the first client doesn't send any data for a while, the second client never gets any attention, even if it's sending lots of data. Even if the first client only delays a tiny bit, that delay is still time wasted—it's time spent waiting and doing *nothing else*. That's the worst thing you can have in a server.

There are refinements of this scheme—splitting the clients across, say, twenty threads, or sorting the clients based on how much time they spend not sending data, but all of these waste time, and all of them will eventually get bogged down.

There must be a better way.

2.2.3 Polling

One better way to do the I/O is through polling. Polling means checking each connection to see if it has any data. If it does, you deal with it, and if not, you don't waste any time waiting for it to have some. You poll all of the clients, deal with any data that has come in, and then wait a tenth of a second. Then you start the process over.

Polling isn't perfect, mainly because of that tenth of a second. That might not seem like a long time, but a high-powered server has to provide response times far smaller than that. The delay can be shortened to a twentieth of a second, or a fiftieth, or even less, but a smaller delay between polling rounds means greater

overhead. You're spending so much time checking the clients that you don't have as much time to actually deal with the input as it comes in.

Polling, in fact, doesn't require the New I/O API because it's always been possible in Java, even using only the `java.io` package. However, the New I/O API has direct support for nonblocking I/O, which makes polling cleaner.

Since the nonblocking I/O method of polling uses the New I/O API, it therefore uses channels rather than streams. The following code does a *blocking* read from a network socket:

```
ByteBuffer buffer = ByteBuffer.allocate( bufferSize );
Socket newSocket = ss.accept();
SocketChannel sch = socket.getChannel();
buffer.clear();
sch.read( buffer );
```

According to the documentation, a socket channel that is in blocking mode will block on the call to `read()`. Specifically, it will wait until *at least one byte* has come in. The read might get lots of data, or it might only get one byte, but `buffer` will *never* be empty after this call.

Here's the nonblocking version:

```
ByteBuffer buffer = ByteBuffer.allocate( bufferSize );
Socket newSocket = ss.accept();
newSocket.configureBlocking( false );
SocketChannel sch = socket.getChannel();
buffer.clear();
sch.read( buffer );
```

This call to `read()` will return an empty buffer if there is nothing to read, because of the call to `configureBlocking()`, which sets the channel to be in nonblocking mode.

The server inner loop at the start of section 2.2.2 contained a call to another method called `rcvMessage()`. Using polling, this method can be rewritten to optionally return `null` when there's no input:

```
while (true) {
  for (client in clients) {
    ChatMessage cm = client2.rcvMessage();
    if (cm != null) {
      for (client2 in clients) {
        client2.sendMessage( cm );
      }
    }
  }
}
```

In this case, we only send the message out if we actually got one, and it all runs inside a single thread!

In the next section, we'll take a close look at a chat server that uses polling.

2.2.4 Example: a polling chat server

In this section, we'll take a look at a complete chat server program that uses polling to carry out all of its I/O operations in a single thread. Not only does it use polling to read data from clients, but it uses polling to actually accept new client connections. It has to do this; otherwise, the main thread would be blocked by the call to `ServerSocket.accept()`.

The central loop of the PollingChatServer program can be represented by the following pseudocode:

```
while (true) {
  if (there are new connections to the server socket) {
    add those connections to the list of active sockets
  }

  for each socket in (the list of active sockets) {
    if (there is data coming into that socket) {
      echo that data back to all clients
    }
  }

  if (any sockets have been closed) {
    remove those sockets from the active list
  }
}
```

To start the server, you must select a port for it to listen on. 5555 is a good one to try:

```
java PollingChatServer 5555
```

To connect to the `PollingChatServer`, you can use the client applet contained in `ChatClient.java` and `ChatClientApplet.java`, using the HTML template contained in client.html. This can be done using the following command:

```
appletviewer ChatClientApplet
```

`client.html` must be modified to use the same port number that the server is listening on. Here are the complete contents of client.html:

```
<applet code="ChatClientApplet.class" width=500 height=300>
<param name="host" value="hostname">
<param name="port" value="5555">
</applet>
```

PollingChatServer itself is shown in listing 2.2.

Listing 2.2 PollingChatServer.java

(see \Chapter2\PollingChatServer.java)

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;

public class PollingChatServer implements Runnable
{
  static private final int sleepTime = 100; //SLEEP_TIME
  private int port;
  private Vector sockets = new Vector();
  private Set closedSockets = new HashSet();

  public PollingChatServer( int port ) {
    this.port = port;
    Thread t = new Thread( this, "PollingChatServer" );
    t.start();
  }

  public void run() {
    try {
      ServerSocketChannel ssc = ServerSocketChannel.open();
      ssc.configureBlocking( false );
      ServerSocket ss = ssc.socket();
      InetSocketAddress isa = new InetSocketAddress( port );
      ss.bind( isa );

      ByteBuffer buffer = ByteBuffer.allocate( 4096 );

      System.out.println( "Listening on port "+port );

      while (true) {
        SocketChannel sc = ssc.accept();

        if (sc != null) {
          Socket newSocket = sc.socket();
          System.out.println( "Connection from "+newSocket );
          newSocket.getChannel().configureBlocking( false );
          sockets.addElement( newSocket );
        }

        for (Enumeration e = sockets.elements();
             e.hasMoreElements();) {
          Socket socket = null;
          try {
```

1 **Open a nonblocking server socket**

2 **A new connection comes in on the server socket**

```
          socket = (Socket)e.nextElement();
          SocketChannel sch = socket.getChannel();
          buffer.clear();
          sch.read( buffer );
          if (buffer.position() > 0) {
            buffer.flip();
            System.out.println( "Read "+buffer.limit()+
                                " bytes from "+sch.socket() );
            sendToAll( buffer );
          }
        } catch( IOException ie ) {
          closedSockets.add( socket );
        }
      }

      removeClosedSockets();

      try {
        Thread.sleep( sleepTime );
      } catch( InterruptedException ie ) {}
    }
  } catch( IOException ie ) {
    ie.printStackTrace();
  }
}

private void sendToAll( ByteBuffer bb ) {
  for (Enumeration e=sockets.elements();
       e.hasMoreElements();) {
    Socket socket = null;
    try {
      socket = (Socket)e.nextElement();
      SocketChannel sc = socket.getChannel();
      bb.rewind();
      while (bb.remaining()>0) {
        sc.write( bb );
      }
    } catch( IOException ie ) {
      closedSockets.add( socket );
    }
  }
}

private void removeClosedSockets() {
  for (Iterator it=closedSockets.iterator(); it.hasNext();) {
    Socket socket = (Socket)it.next();
    sockets.remove( socket );
    System.out.println( "Removed "+socket );
  }
  closedSockets.clear();
}
```

3 **Data is read on one of the client sockets**

4 **Data is written to each of the client sockets**

Dead sockets are removed in a separate phase 5

```
  static public void main( String args[] ) throws Exception {
    int port = Integer.parseInt( args[0] );
    new PollingChatServer( port );
  }
}
```

1. Instead of using the normal method of calling `new ServerSocket( port )`, we create a `ServerSocketChannel` and put it in nonblocking mode using `configureBlocking()`. We call the `ServerSocketChannel`'s `socket()` method to get access to the `ServerSocket` object. We bind the server socket to the user-specified port using the `ServerSocket`'s `bind()` method.

   The `ServerSocket` object and the `ServerSocketChannel` should be thought of as two objects referring to the same underlying operating system resource. More precisely, the `ServerSocket` is built *on top* of the `ServerSocketChannel`.

2. Here, we *poll* for any incoming connections to our server socket. Since it has been placed in nonblocking mode, a call to `accept()` will return `null` if there are no incoming connections. If there is an incoming connection, we place it in `sockets`, the list of currently active sockets.

3. We check each open socket to see if any data is coming in. The sockets are all in nonblocking mode; this means that if a socket has no data coming in to it, we'll get an empty buffer back. If we get a buffer that isn't empty, we send that data to all of the clients using the `sendToAll()` method.

4. `sendToAll()` sends the contents of a buffer to each of the currently open sockets. It calls `rewind()` before each send to make sure the buffer's position value is pointing to the beginning of the buffer, so that everything gets written.

5. Any time a `write()` to a socket fails, we take that socket to be closed, and add it to a list of dead sockets called `closedSockets`. The `removeClosedSockets()` method takes all of the closed sockets and removes them from the main socket list, `sockets`.

   This is done as a separate step because removing the dead sockets while we are iterating through them is dangerous—the data structures can get confused, causing some sockets to be skipped during the reading or writing process.

All things considered, polling isn't a bad method of doing lots of I/O in a single thread. But the fact is that it does waste some CPU cycles, and it does cause small but perceptible delays in throughput. To get the tightest response possible, you need to use *multiplexed nonblocking I/O*. This will be discussed in the next section.

2.2.5 Multiplexing with select()

Polling isn't really the best way to do multiplexed I/O, although, for a long time, it was the only way to do it in Java. JDK 1.4 brings us a better solution—the *selector*.

The Selector

Multiplexing centers around the use of an object called a `Selector`. This object watches a set of channels of various kinds and alerts you when one of them has received some input. Instead of having to check each of the channels periodically, like you do when you're polling, you can call a single method, `Selector.select()`, which will block until one or more of the channels is ready. You then deal with whatever information has come in, and then you call `select()` again. You can see how simple the main loop is in the following schematic listing:

```
while (true) {
  selector.select();

  // deal with new input ...
}
```

The advantage of using selectors is tremendous. If you'll recall from section 2.2.2, the naive approach was to wait for input from one of the clients, which meant we were ignoring all of the other clients while we were waiting. The beauty of `select()` is that you can wait for input from *all* of the clients at once.

The `select()` method treats server sockets and regular sockets in the same way. Server sockets receive new connections, while regular sockets receive bytes. The `select()` call treats them both as a kind of *I/O event*.

Note that `select()` only works with *selectable channels*—that is, `Channels` that implement the `SelectableChannel` interface. Both `Socket` and `ServerSocket` implement `SelectableChannel`. In the base implementation, only five classes are selectable: `DatagramChannel`, `Pipe.SinkChannel`, `Pipe.SourceChannel`, `ServerSocketChannel`, and `SocketChannel`.

`select()` is very efficient. The Java version of `select()`, in fact, seems to be based on the Unix system call of the same name. The purpose of `select()` is to make available to the user the fundamentally asynchronous, multiplexed nature of I/O that you find at the operating system level. While the `select()` approach is slightly more complicated to understand than the elegant stream approach, the difficulties are justified by the increase in speed and flexibility.

Listening and reading with select()

In this section, we will re-implement the chat server of section 2.2.4 using `select()`. As before, our server will consist of a server socket that is listening for connections, and a set of client sockets, each of which is connected to a chat client at the other end. And, as in the polling version, we'll do everything in a single thread. Our new implementation, MultiplexingChatServer, will be much more efficient than PollingChatServer because it won't spend any time polling, or waiting between polls.

The first thing we need to do is create a `Selector`. This object is the center of the process: it is the object that alerts us to the presence of incoming input.

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

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

The static `open()` call creates a new `Selector` object. Now that we have a selector, we can create some channels to register with it. These channels will be the channels that the selector watches for I/O activity.

Since this is a standard chat server, we'll create a server socket, just like we did in the PollingChatServer.

```
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking( false );
ServerSocket ss = ssc.socket();
InetSocketAddress isa = new InetSocketAddress( port );
ss.bind( isa );
```

Note that we create the server socket by creating a `ServerSocketChannel` first, and then calling its `socket()` method to get access to the `ServerSocket` object. We do this mainly because we need to configure it as a nonblocking socket by calling the `configureBlocking()` method of its channel.

NOTE A `ServerSocket` will have a `ServerSocketChannel` if and only if the channel was created first using `ServerSocketChannel.open()`.

Normally, the first thing we'd do after creating a server socket would be to call its `accept()` method to listen for an incoming connection. Because we're multiplexing, we're not going to do that. Instead, we register the server socket by calling the `register()` method of the `ServerSocketChannel` object:

```
ssc.register( selector, SelectionKey.OP_ACCEPT );
```

The second argument to this method describes the set of *I/O operations* we would like to listen for. Although they are called *operations*, they really should be thought of as *events*. There are four kinds of I/O events that a `Selector` can listen for:

- `OP_READ`—This event is triggered when it becomes possible for the channel to have data read from it. In a networking context, if even a single byte comes in from the remote side of a connection, this event will be triggered. This event is valid for regular sockets, but not for server sockets.
- `OP_WRITE`—This event is triggered when it becomes possible to write to a channel. In a system without much I/O load, a socket will always be ready to

have data written to it. However, socket buffers are finite, and a heavily loaded server will sometimes have a bottleneck at the outgoing socket buffers. In this case, it is possible that a simple write to a socket will block until the buffers have enough room for the data to be written. This event is valid for regular sockets, but not for server sockets.

- `OP_CONNECT`—This event is triggered when a regular socket is ready to complete its connection to a remote server. This event is valid for regular sockets, but not for server sockets.
- `OP_ACCEPT`—This event is triggered when one or more incoming connections have arrived at the server socket. This event is valid for server sockets, but not for regular sockets.

NOTE You can find out what selection operations are valid for a particular `SelectableChannel` by calling its `validOps()` method.

Table 2.1 shows which operations are valid with each kind of socket.

Table 2.1 Valid operations for each kind of socket. Each operation is only valid for one of the two kinds of socket.

Operation	Socket	ServerSocket
`OP_READ`	✔	
`OP_WRITE`	✔	
`OP_CONNECT`		✔
`OP_ACCEPT`		✔

These operations can be *or'ed* together, allowing you to specify more than one operation at a time. For example, to register a regular socket for reading and writing, you could use the following line of code:

```
sc.register( selector,
  SelectionKey.OP_READ  |
  SelectionKey.OP_WRITE );
```

Although we've been ignoring it so far, the call to `register()` returns an object called a `SelectionKey`. This object represents the registration of the channel and has a number of purposes. You can unregister a channel by calling the `SelectionKey`'s `cancel()` method. Also, if a channel has been registered for a number of different

events, and one is triggered, you can use the `SelectionKey` to find out which ones were triggered. You can get the selection key like this:

```
SelectionKey sk = sc.register( selector, ops );
```

Now that we've registered our server socket with our selector, we're ready to start waiting for incoming connections. In our main loop, we call `select()`:

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

We are now waiting for any of the registered channels to have some event come in. At some point, one or more clients will try to connect, and when this happens, `select()` will return. The value it returns is the number of events that have been triggered.

The selected set

When one or more events have been triggered on one or more channels, the selection keys corresponding to these channels are put into a set called the *selected set*. You can get access to this set by calling the selector's `selectedKeys()` method.

```
Set skeys = selector.selectedKeys();
```

Using this set, you can iterate through the selection keys and process each one:

```
Iterator it = skeys.iterator();
while (it.hasNext()) {
  SelectionKey rsk = (SelectionKey)it.next();
  // process selection key
}
```

In our chat server, we're listening for two kinds of events: incoming connections to the server socket, and incoming data to the regular sockets. Each selection key represents a channel that has had an event, so our first task is to find out what kind of channel it was, and what kind of event it received.

```
SelectionKey rsk = (SelectionKey)it.next();
int rskOps = rsk.readyOps();
// The following line checks to see if the 'SelectionKey.OP_ACCEPT'
// bit is set within 'rskOps'
if ((rskOps & SelectionKey.OP_ACCEPT) == SelectionKey.OP_ACCEPT) {
  // it's an incoming connection; add it to the list of connections
} else {
  // it's data arriving at a regular socket; process it
}
```

In the preceding code, we check the selection key to find out what kind of event (operation) was triggered. If it was an *accept* operation, then we know that this must be the server socket accepting a new connection. Otherwise, we know that it must be regular data arriving at a regular socket.

Note that the selector doesn't carry out the operation in question—it only tells us that the operation is ready to be carried out. If it's an accept operation, we need to perform the actual accept:

```
Socket socket = serverSocket.accept();
```

If, instead, we've just had some data coming in to a `SocketChannel`, we need to read the data and process it:

```
buffer.clear();
// The selection key contains a reference to its SocketChannel
SocketChannel ch = (SocketChannel)rsk.channel();
ch.read( buffer );
buffer.flip();
sendToAll( buffer );
```

If we've just gotten a new connection, we also have to add the new channel to the selector, so that the selector is listening for incoming data on it. We'll register this `SocketChannel` just like we registered the `ServerSocketChannel`, only we'll register it for `OP_READ` rather than for `OP_ACCEPT`.

```
SocketChannel sc = socket.getChannel();
sc.configureBlocking( false );
sc.register( selector, SelectionKey.OP_READ );
```

NOTE The arrival of data isn't the only thing that will trigger an `OP_READ` event. This event will also be triggered if the connection is closed or if there is an error of some kind. This is true of all four types of events. The rationale behind this is that if you are waiting for data to arrive, you want to know when it arrives, but you also want to know when there is a fatal error preventing its arrival.

Once we've dealt with incoming data or new incoming connections, we're nearly done with our inner loop. Before we can go back to processing events, we have to make sure that we tell the selector that we've just processed a channel. We do this by removing the channel's `SelectionKey` from the selected set:

```
selector.selectedKeys().remove( selectionKey );
```

NOTE Once a `SelectionKey` has been added to a selector's selected set, it must be removed explicitly. If you do not remove it from the selected set, it will still be in the selected set the next time an event comes in, making it seem like the event has been triggered again. Failure to remove a `SelectionKey` from the selected set can result in a bug in which an event seems to be triggered repeatedly.

The big view

Figure 2.2 gives an overview of the flow of events in the chat server's main thread:

The steps in figure 2.2 are described in more detail here:

1. A new `ServerSocketChannel` is created. This will be used to listen for new connections.
2. The `ServerSocketChannel` is registered with the `Selector`. We will be able to receive notice of I/O events for the `ServerSocketChannel` by calling this `Selector`'s `select()` method. The call to `register()` returns a `SelectionKey`.
3. `select()` waits until one or more registered channels have I/O events.
4. `selectedKeys()` is used to find out which channels have I/O events. Specifically, it returns a `Set` containing `SelectionKeys`. Each `SelectionKey` can be used to get access to the underlying `Channel`.
5. If a new connection has come in, `ServerSocketChannel.accept()` is called to get the new `SocketChannel`.
6. The new `SocketChannel` is registered with the `Selector`, just as the `ServerSocketChannel` was. Whereas the `ServerSocketChannel` is being watched for new incoming connections, the new `SocketChannel` is being watched for incoming chat data.
7. Incoming chat data is processed. In our simple chat system, this means it is sent out to all other connections.

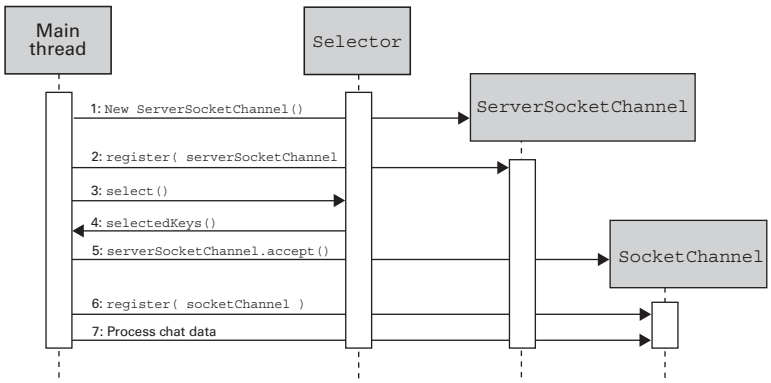

Figure 2.2 The flow of events in the main thread of the chat server

The source

In the preceding sections, we've gone through the essential elements of our chat server's inner loop. Listing 2.3 shows the complete program.

Listing 2.3 MultiplexingChatServer.java

(see \Chapter2\MultiplexingChatServer.java)

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;

public class MultiplexingChatServer implements Runnable
{
  private int port;
  private Vector sockets = new Vector();
  private Set closedSockets = new HashSet();

  public MultiplexingChatServer( int port ) {
    this.port = port;
    Thread t = new Thread( this, "MultiplexingChatServer" );
    t.start();
  }

  public void run() {
    try {
      ServerSocketChannel ssc = ServerSocketChannel.open();
      ssc.configureBlocking( false );
      ServerSocket ss = ssc.socket();
      InetSocketAddress isa = new InetSocketAddress( port );
      ss.bind( isa );

      Selector selector = Selector.open();
      ssc.register( selector, SelectionKey.OP_ACCEPT );
      System.out.println( "Listening on port "+port );

      ByteBuffer buffer = ByteBuffer.allocate( 4096 );

      while (true) {
        int numKeys = selector.select();
        if (numKeys>0) {
          Set skeys = selector.selectedKeys();
          Iterator it = skeys.iterator();
          while (it.hasNext()) {
            SelectionKey rsk = (SelectionKey)it.next();
            int rskOps = rsk.readyOps();
            if ((rskOps & SelectionKey.OP_ACCEPT) ==
                SelectionKey.OP_ACCEPT) {
              Socket socket = ss.accept();
              System.out.println( "Connection from "+socket );
              sockets.addElement( socket );
```

Prepare a nonblocking ServerSocket

Add the ServerSocket to the selector

Wait for an I/O event

Deal with a new incoming connection

```
            SocketChannel sc = socket.getChannel();
            sc.configureBlocking( false );
            sc.register( selector, SelectionKey.OP_READ );
            selector.selectedKeys().remove( rsk );
          } else if ((rskOps & SelectionKey.OP_READ) ==
                     SelectionKey.OP_READ) {
            SocketChannel ch = (SocketChannel)rsk.channel();
            selector.selectedKeys().remove( rsk );
            buffer.clear();
            ch.read( buffer );
            buffer.flip();
            System.out.println( "Read "+buffer.limit()+
                                " bytes from "+ch.socket() );
            if (buffer.limit()==0) {
              System.out.println( "closing on 0 read" );
              rsk.cancel();
              Socket socket = ch.socket();
              close( socket );
            } else {
              sendToAll( buffer );
            }
          }
        }

        removeClosedSockets();
      }
    }
  } catch( IOException ie ) {
    ie.printStackTrace();
  }
}

// This method is identical to
// PollingChatServer.sendToAll()
private void sendToAll( ByteBuffer bb ) {
  for (Enumeration e=sockets.elements();
       e.hasMoreElements();) {
    Socket socket = null;
    try {
      socket = (Socket)e.nextElement();
      SocketChannel sc = socket.getChannel();
      bb.rewind();
      while (bb.remaining()>0) {
        sc.write( bb );
      }
    } catch( IOException ie ) {
      closedSockets.add( socket );
    }
  }
}
```

Deal with the arrival of chat

Send a buffer of data to all clients

```
  private void close( Socket socket ) {
    closedSockets.add( socket );
  }

  // This method is identical to
  // PollingChatServer.removeClosedSockets()
  private void removeClosedSockets() {
    for (Iterator it=closedSockets.iterator(); it.hasNext();) {
      Socket socket = (Socket)it.next();
      sockets.remove( socket );
      System.out.println( "Removed "+socket );
    }
    closedSockets.clear();
  }

  static public void main( String args[] ) throws Exception {
    int port = Integer.parseInt( args[0] );
    new MultiplexingChatServer( port );
  }
}
```

Put the socket on the to-close list

Remove dead sockets from active list

Although the MultiplexingChatServer doesn't cover *every* valid use of selectors, it covers the ones you'll use in most of your servers—accepting new connections and processing the data coming from them.

Even though this technique can be done in a single thread, you'll probably want to use multiple threads in practice, to take advantage of any parallelism within the underlying operating system or hardware. Additionally, any time-consuming data processing should probably be moved to other threads, if only so that thread priorities can be used to fine-tune the amount of time spent on I/O and the amount of time spent on data processing.

2.3 Encoding and decoding with Charsets

Since its release, Java has used Unicode characters throughout. However, because most operating systems are not fully Unicode-compliant, most Java programs are still effectively using 8-bit characters. Although a great deal of support for transitioning between 8-bit ASCII and 16-bit Unicode has been available within the Java API, it is generally circumvented.

`Charsets` integrate Unicode characters with the New I/O API by providing methods for converting `ByteBuffers` to `CharBuffers`, and back again. Precisely defined, a `Charset` is a particular mapping between bytes and Unicode characters. There are many different ways of performing this mapping—some optimized for space, others optimized for completeness of encoding. The `Charset` facility within the New I/O API allows multiple mappings to coexist peacefully, since each `Charset`

object can represent a different mapping, and each one can encode and decode characters separately.

2.3.1 Decoding and encoding

Converting bytes to chars is called decoding. This might seem backwards, but it makes sense. Before Unicode adoption, a char was considered to be nothing more than a byte with an interpretation attached. Java chars, however, are their own entities, and should not be considered equivalent to any particular byte-encoding. In particular, a char can be encoded by any number of bytes. As a result, there is no definitive mapping that turns a char into one or more bytes. Instead, there are multiple mappings. Each one can be represented by a different `Charset` object.

A `Charset` is created using the `Charset.forName()` method. The single argument to the method is a string providing the name of the `Charset`.

```
String charsetName = "ISO-8859-1";
Charset charset = Charset.forName( charsetName );
```

While the available `Charsets` differ from system to system, the ones listed in Table 2.2 are available in any Java installation.

Table 2.2 These `Charsets` are available in every Java installation. Other `Charsets` may also be available.

Name	Definition
`US-ASCII`	Traditional 7-bit ASCII
`ISO-8859-1`	ISO Latin alphabet 1; also known as ISO-LATIN-1
`UTF-8`	8-bit UCS Translation Format
`UTF-16BE`	16-bit UCS Transformation Format (big-endian)
`UTF-16LE`	16-bit UCS Transformation Format (little-endian)
`UTF-16`	16-bit UCS Transformation Format (byte order determined by optional byte-order mark)

2.3.2 Finding available Charsets

You can find out what `Charsets` are available on your system by using ListCharsets (see listing 2.4). It lists each `Charset`, followed by a sublist of the *aliases* of that `Charset`. An alias is another name for the same `Charset`.

Listing 2.4 ListCharsets.java

(see \Chapter2\ListCharsets.java)

```
import java.util.*;
import java.nio.charset.*;

public class ListCharsets
{
  static public void main( String args[] ) throws Exception {
    SortedMap charsets = Charset.availableCharsets();
    Set names = charsets.keySet();
    for (Iterator e=names.iterator(); e.hasNext();) {
      String name = (String)e.next();
      Charset charset = (Charset)charsets.get( name );
      System.out.println( charset );
      Set aliases = charset.aliases();
      for (Iterator ee=aliases.iterator(); ee.hasNext();) {
        System.out.println( "    "+ee.next() );
      }
    }
  }
}
```

Here is some sample output from this program, showing the variety of `Charsets` and their aliases:

```
ISO-8859-1
    latin1
    ISO8859-1
    IBM819
US-ASCII
    us
    ISO_646.irv:1991
UTF-16
    utf_16
UTF-16BE
    iso-10646-ucs-2
    utf_16be
UTF-16LE
    utf_16le
UTF-8
    UTF8
windows-1252
    Cp1252
```

This is only a partial listing. You'll notice that most `Charsets` have a large number of aliases.

2.3.3 Using encoders and decoders

Once you have acquired a `Charset` object, you can use it to convert a `ByteBuffer` to a `CharBuffer`, and vice versa. To do this, you must first create a `CharsetEncoder` or a `CharsetDecoder` using the `newEncoder()` and `newDecoder()` methods. When you have an instance of a `CharsetEncoder` or `CharsetDecoder`, you can carry out a conversion using either the `encode()` or `decode()` method, respectively.

The following code fragment demonstrates the process of converting a `ByteBuffer` to a `CharBuffer` using a `Charset`:

```
Charset charset = Charset.forName( charsetName );
CharsetDecoder decoder = charset.newDecoder();
CharBuffer charBuffer = decoder.decode( byteBuffer );
```

Likewise, you can convert a `CharBuffer` to a `ByteBuffer` as follows:

```
Charset charset = Charset.forName( charsetName );
CharsetEncoder encoder = charset.newEncoder();
ByteBuffer byteBuffer = encoder.encode( charBuffer );
```

The decoding process involves reading bytes, one at a time, from the input `ByteBuffer`. Many `Charset`s deal with characters that use more than one byte per character; in these cases, multiple bytes have to be read to produce a single character. Each time the decoder reads enough bytes to produce a character, this character is written to the `CharBuffer`. The decoder makes sure not to write more characters than can fit in the `CharBuffer`, or to read more bytes than are available in the `ByteBuffer`. In the preceding example, the `CharBuffer` is created by the `CharsetDecoder`, so it won't overrun; however, there are variants of `decode()` and `encode()` that take a destination buffer as an argument, and these buffers might not be large enough.

The encoding process can also generate more than one byte per character. It reads characters, one at a time, from the `CharBuffer`. For each character, it writes one or more bytes to the `ByteBuffer`. The capacities of the buffers are likewise respected.

The preceding description is accurate for most `CharsetDecoder`s, but it is by no means a requirement—a `CharsetDecoder` could, if it wanted to, process the characters in a strange order, or copy data into other buffers. However, it must preserve the ordering of the characters and properly maintain the buffers' limit and position values.

You can combine the previous code fragments to convert a piece of text from one encoding to another by converting it first to a `CharBuffer` using one `Charset`, and then converting it back to a `ByteBuffer` using another `Charset`.

```
Charset charsetA = Charset.forName( charsetName );
CharsetDecoder decoderA = charsetA.newDecoder();
Charset charsetB = Charset.forName( charsetName );
CharsetEncoder encoderB = charsetB.newEncoder();

CharBuffer charBuffer = decoderA.decode( byteBuffer );
ByteBuffer newByteBuffer = encoderB.encode( charBuffer );
```

The TranslateCharset program (see listing 2.5) can be used to convert a file from one encoding to another. You specify the source file and the destination file, along with the names of the old and new `Charset`s, on the command line. For example, the following command-line command would use TranslateCharset to convert the plaintext file old.txt to be the 16-bit Unicode file new.txt:

```
java TranslateCharset old.txt ISO-8859-1 new.txt UTF-16BE
```

Listing 2.5 TranslateCharset.java

(see \Chapter2\TranslateCharset.java)

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

public class TranslateCharset
{
  static public void main( String args[] ) throws Exception {
    if (args.length != 4) {
      System.err.println(
        "Usage: java TranslateCharset <infile> <incharset> "+
        "<outfile> <outcharset>" );
      System.exit( 1 );
    }

    String inFilename = args[0];
    String inFileCharsetName = args[1];
    String outFilename = args[2];
    String outFileCharsetName = args[3];

    File infile = new File( inFilename );
    File outfile = new File( outFilename );

    RandomAccessFile inraf =
      new RandomAccessFile( infile, "r" );
    RandomAccessFile outraf =
      new RandomAccessFile( outfile, "rw" );

    FileChannel finc = inraf.getChannel();
    FileChannel foutc = outraf.getChannel();

    MappedByteBuffer inmbb =
      finc.map( FileChannel.MapMode.READ_ONLY, 0, (int)infile.length() );
```

```
    Charset inCharset = Charset.forName( inFileCharsetName );
    Charset outCharset = Charset.forName( outFileCharsetName );

    CharsetDecoder inDecoder = inCharset.newDecoder();
    CharsetEncoder outEncoder = outCharset.newEncoder();

    CharBuffer cb = inDecoder.decode( inmbb );
    ByteBuffer outbb = outEncoder.encode( cb );

    foutc.write( outbb );

    inraf.close();
    outraf.close();
  }
}
```

Create the encoder and decoder

Convert from one encoding to another

For most applications, you won't have to worry about `Charsets`. Many of the simple text-processing facilities in Java (such as `System.out.println()`) will take care of the details for you, using reasonable defaults. However, if you intend to create an application that uses text in a substantial way, `Charsets` are an elegant and efficient way to deal with Unicode encodings.

2.4 Network interfaces

The new `NetworkInterface` object provides access to the operating system–level objects that represent the interfaces on which network communications can happen. Some high-end servers come equipped with multiple network cards, and there are several reasons why this might be done. For example, having multiple network cards can allow a machine to engage in network communications at a much higher rate. To the extent that network I/O is bound by the network interface cards, multiple cards can parallelize communications, increasing throughput.

Having multiple interface cards can also be a physical strategy for securing a machine. For example, access to certain services might be restricted to connections coming through a particular interface that only certain users have access to. A server could have one network interface card for connections originating within the same office, and another network interface card for connections coming from the wider Internet. Multiple cards also provide redundancy in the face of hardware failure.

Network interfaces don't necessarily have to be physical devices. Most machines have a *loopback interface* that allows the machine to connect to itself. While this does not require special hardware to implement, the interface nevertheless has status as a network interface.

Each `NetworkInterface` object represents one of the network interfaces on your machine. Each one of these objects can be used to gather information about that interface. You can also get a list of `InetAddress` objects from an interface, which allows you to create a `ServerSocket` that only listens on a particular address.

We'll be seeing example output from some programs and program fragments in this section. The output was generated by running the programs on two different systems—Windows 95 and a Linux/GNU system.

2.4.1 When to use a network interface

Generally, you don't need to think about network interfaces. In most configurations, a machine has a *default* interface that is used for all IP communications. This means that socket programming can be done without reference to the particular interface that is being used. If you don't know what a network interface is, then you probably don't need to use one.

However, as mentioned previously, there are times when you need to access a particular interface. Or, to put it another way, there are times when you need to override the default network interface. With the arrival of JDK 1.4, you can now do this using the `NetworkInterface` class.

2.4.2 Getting a list of NetworkInterfaces

`NetworkInterface` provides a static method called `getNetworkInterfaces()`. This method provides an `Enumeration` of the `NetworkInterface` objects available.

Here is an example of its use:

```
Enumeration interfaces = NetworkInterface.getNetworkInterfaces();
while (interfaces.hasMoreElements()) {
  NetworkInterface ni = (NetworkInterface)interfaces.nextElement();
  // ...
}
```

This fragment iterates through all of the network interfaces on the system.

2.4.3 Reporting on NetworkInterfaces

You can pass a `NetworkInterface` to `System.out.println` to see what information is available on your system (see listing 2.6).

Listing 2.6 from ListNetworkInterfaces.java

(see \Chapter2\ListNetworkInterfaces.java)

```
Enumeration interfaces = NetworkInterface.getNetworkInterfaces();
while (interfaces.hasMoreElements()) {
  NetworkInterface ni = (NetworkInterface)interfaces.nextElement();
  System.out.println( ni );
}
```

Under Linux, this results in the following output:

```
name:ppp0 (ppp0) index: 115 addresses:
/111.222.33.44;

name:eth0 (eth0) index: 2 addresses:
/192.168.0.1;

name:lo (lo) index: 1 addresses:
/127.0.0.1;
```

Under Windows, we get the following:

```
name:lo0 (localhost) index: 2 addresses:
/127.0.0.1;

name:lan0 (3Com EtherLink III ISA (3C509/3C509b) in ISA mode) index: 1
   addresses:
/192.168.0.2;
```

Each paragraph represents a different network interface. There are a number of pieces of information in each entry, as shown in figure 2.3.

- `interface name`—The short name of the network interface. On Unix systems, this string is used by the `ifconfig` command, which can be used to configure the system to use a particular IP address for a particular network interface.
- `interface full name`—A longer, more descriptive name for the network interface. This string is probably not used in any official capacity, but rather exists to provide a more user-friendly description of the interface. A program for selecting a network interface might display this string in the user interface.

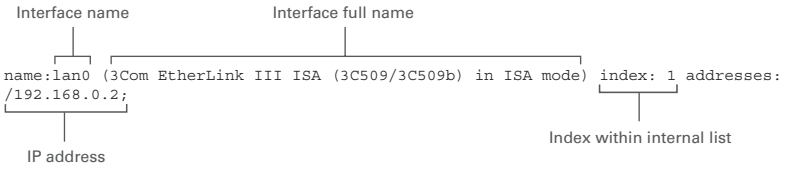

Figure 2.3 These are the parts of a `NetworkInterface` report. This report is generated by the `toString()` method of the `NetworkInterface` object.

- `index`—This is the index of this interface within an internal table.
- `IP address`—A list of all the Internet addresses bound to this interface.

2.4.4 Getting a list of InetAddresses

Although the output in section 2.4.3 provides a list of addresses, they are inconveniently imbedded within a block of text. However, it's easy to get a list of `InetAddress` objects belonging to a given interface by using the `getInetAddresses()` method.

```
NetworkInterface ni = ...;
Enumeration e = ni.getInetAddresses();
while (e.hasMoreElements()) {
  InetAddress ia = (InetAddress)e.nextElement();
  System.out.println( ia );
}
```

Most network interfaces have a single address; for the ones that don't, the preceding code will list the addresses.

2.4.5 Getting a NetworkInterface by InetAddress

Given that there may be several network interfaces on a given system, how do you pick one to use? One way is to pick the interface based on an address that is bound to it. `NetworkInterface` provides a static method called `getByInetAddress()`, which allows you to get the `NetworkInterface` object corresponding to a particular `InetAddress` object.

ReportByAddress (see listing 2.7) looks up a network interface based on an address provided on the command line. It then prints out some information about this interface.

Listing 2.7 ReportByAddress.java

(see \Chapter2\ReportByAddress.java)

```
import java.net.*;

public class ReportByAddress
{
  static public void main( String args[] ) throws Exception {
    InetAddress ia = InetAddress.getByName( args[0] );
    NetworkInterface ni = NetworkInterface.getByInetAddress( ia );
    System.out.println( ni );
  }
}
```

Here's an example of this program in action under Windows:

```
java ReportByAddress 192.168.0.2

name:lan0 (3Com EtherLink III ISA (3C509/3C509b) in ISA mode) index: 1
  addresses:
/192.168.0.2;
```

Note that the address used to find this interface is listed in the `addresses:` section of the output.

2.4.6 Getting a NetworkInterface by name

You can also access a `NetworkInterface` given only its short device name, such as `lan0` or `ppp0`. This is done by using the `getByName()` method.

ReportByName (see listing 2.8) looks up a network interface based on a name provided on the command line. It then prints out some information about this interface.

Listing 2.8 ReportByName.java

(see \Chapter2\ReportByName.java)

```
import java.net.*;

public class ReportByName
{
  static public void main( String args[] ) throws Exception {
    NetworkInterface ni = NetworkInterface.getByName( args[0] );
    System.out.println( ni );
  }
}
```

Here's the output of this program under Linux:

```
java ReportByName eth0

name:eth0 (eth0) index: 2 addresses:
/192.168.0.1;
```

This program can be used to find the IP address corresponding to a particular interface name.

2.4.7 Listening on a particular address

As was mentioned at the beginning of this section, one practical reason why you'd need to use `NetworkInterface` objects is to create server sockets that only listen on a particular address. This can be useful if you are hosting multiple Internet addresses on the same machine.

In order to understand how this works, we need to understand a little bit more about what it means to listen on a socket.

Listening on the default address

If you've written socket code in Java before, you know that you don't need to think about network interfaces to listen on a socket. This is because most machines are configured to have a default address to listen on, if none is specified.

This default address is generally `0.0.0.0`, which isn't really an address at all. It's more like a placeholder that says, "listening on this address means listening on all addresses at once." This means that if a connection comes in for the specified port, on any interface, then that is considered a connection.

To illustrate this, we'll try out some code. The following piece of code does the following:

- Listens for a connection on a particular port number
- Gets one incoming connection on that port
- Prints out information about that connection

```
int port = 5555;
ServerSocket ss = new ServerSocket( port );
Socket s = ss.accept();
System.out.println( s );
```

If you execute this program, it won't quit right away. It will sit there waiting for an incoming connection. By using the `netstat` command under Linux, we can see that the program is waiting on a connection. The output in figure 2.4 shows that something is listening on `0.0.0.0:5555`, which means it's listening on port 5555, on all addresses.

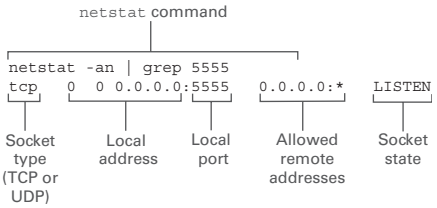

Figure 2.4
Output from the `netstat` command. `netstat` lists all open sockets, along with information about them. This listing shows the information for a socket listening for new connections on port 5555.

Getting the first address of an interface

Before we get into the mechanics of listening on particular interfaces, we need to define a helper method called `getAddress()`.

A `NetworkInterface` object can have multiple addresses bound to it. However, it is often the case that each `NetworkInterface` in the system has only one address bound to it. For convenience's sake, we're going to assume that is the case.

The `NetworkInterface` class does *not* include a method called getInetAddress or getSingleInetAddress. It only provides `getInetAddresses()`, which returns an `Enumeration`. To simplify matters, we've created a helper function that grabs the first address in the `Enumeration` and returns it. It is also possible for a `NetworkInterface` to have *no* addresses; in this case, an empty `Enumeration` is returned.

```
static private InetAddress getAddress( NetworkInterface ni ) {
  Enumeration e = ni.getInetAddresses();
  if (!e.hasMoreElements())
    return null;
  InetAddress ia = (InetAddress)e.nextElement();
  return ia;
}
```

It is common to assume that the first address for an interface is sufficient; more sophisticated programs can provide a configuration interface to specify a different address.

Listening on an address

Listening for a connection on a particular address is a lot like accepting letters only if they are addressed to you personally. Listening on all addresses, then, is like accepting any letter that shows up at your door.

As mentioned previously, being particular about what address you listen on can let you multiplex the connections that are coming in. This is analogous to having roommates, each of whom receives letters at the same address. When the mail comes in, it is sorted by recipient before it is handed over.

NOTE Technically, a program listens on an *address*, not an *interface*. However, if an interface has a single address bound to it, then it is reasonable to say that a program listening on that *address* is also listening on that *interface*.

Now we're going to find out how to listen for connections on a particular interface. The Accept program will do this for us (see listing 2.9). Accept listens on an interface and listens for a single incoming connection. It won't do anything with this incoming connection—Accept just *listens*. We can examine the state of this listening process by running `netstat` at the same time. `netstat` shows the states of all ports that are being listened on—we can use it to get direct feedback about what ports, and what network interfaces, are being listened on.

Listing 2.9 Accept.java

(see \Chapter2\Accept.java)

```
import java.net.*;
import java.util.*;

public class Accept
{
  static private InetAddress getAddress( NetworkInterface ni ) {
    Enumeration e = ni.getInetAddresses();
    InetAddress ia = (InetAddress)e.nextElement();
    return ia;
  }

  static public void main( String args[] ) throws Exception {
    int port = Integer.parseInt( args[0] );
    String interf = args.length > 1 ? args[1] : null;

    if (interf != null) {
      NetworkInterface ni = NetworkInterface.getByName( interf );
      InetAddress ia = getAddress( ni );
      ServerSocket ss = new ServerSocket( port, 20, ia );
      System.out.println( "Listening" );
      Socket s = ss.accept();
      System.out.println( s );
    } else {
      ServerSocket ss = new ServerSocket( port );
      System.out.println( "Listening" );
      Socket s = ss.accept();
      System.out.println( s );
    }
  }
}
```

Return the first address of a network interface

The first argument to this program is the number of the port to listen on, and is required. The second argument is the name of the interface to listen on, and is optional. If it is omitted, then the program listens on `0.0.0.0`; that is, it listens on all interfaces at once.

Let's now take a look at the results of running this program. In each case, we'll run the program and then use the Unix command `netstat` to find out what's going on.

First, we'll run Accept and listen on all sockets:

```
java Accept 5555
```

Note that we're using port 5555. This choice is arbitrary—you can use any port on your system that you have permission to use, and which isn't already being used by

something else. Generally, a nonprivileged user has permission to use any port greater than 1023.

Now that we've run Accept, it is listening on port 5555. We can use `netstat` to verify this. The exact Unix command we are using is `netstat -an | grep 5555`, which can be translated as "list all network sockets in use, but only report those mentioning '5555'."

```
netstat -an | grep 5555
tcp    0  0 0.0.0.0:5555    0.0.0.0:*       LISTEN
```

The output tells us that someone is listening on port 5555 on address `0.0.0.0`, that is, on every address. This is just like our earlier example.

We can verify that this is working by trying to connect to this address. The simplest way to do this is with the telnet program, which is available under most operating systems, including Windows and Linux. Under Unix and most versions of Microsoft Windows, you can simply telnet at the command line:

```
telnet localhost 5555
```

Since there is a program listening on port 5555, this connection request will succeed, and you'll see something like this:

```
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
Connection closed by foreign host.
```

If there hadn't been a program listening on port 5555, then you would have seen something like this:

```
telnet localhost 5555

Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connection refused
```

Now, we'll try listening on a particular interface. We learned earlier that our Linux system had three interfaces: `ppp0`, `eth0`, and `lo`. Let's run Accept again and try listening on `eth0`. Then we'll run `netstat` again:

```
java Accept 5555 eth0

netstat -an | grep 5555
tcp    0  0 192.168.0.1:5555  0.0.0.0:*       LISTEN
```

Note the change. Again, we see that there is a program listening on port 5555. However, this time it's not listening on the default address, `0.0.0.0`, but rather on `192.168.0.1`. This is the local IP address assigned to our Linux box on its LAN, so it makes sense that we'd see that address here.

If you use telnet to verify this, you'll find that the behavior is slightly different. In our previous example, we used telnet to connect to `localhost:5555`. `localhost` is actually an alias for the address `127.0.0.1`, which is an address that generally refers to the local machine.

If you try to connect to this address now, you'll find that it doesn't work. That's because our listening program is only listening on `192.168.0.1`. Previously, we were listening on all addresses, so `localhost` was valid, but now we are only listening on a single address, and if you want to connect, you have to use that address. If you use telnet to connect to `192.168.0.1`, you'll find that it works.

We can try listening on the `lo0` interface:

```
java Accept 5555 lo0
```

```
netstat -an | grep 5555
tcp   0  0 127.0.0.1:5555    0.0.0.0:*       LISTEN
```

You can see that the program is now listening on `127.0.0.1`, also known as `localhost`. Telnetting to `192.168.0.1` fails, as we see here:

```
telnet 192.168.0.1 5555
Trying 192.168.0.1...
telnet: Unable to connect to remote host: Connection refused
```

Meanwhile, telnetting to `127.0.0.1` succeeds, as we see here:

```
telnet 127.0.0.1 5555
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Connection closed by foreign host.
```

Just to verify that we can do it, let's try running several servers at once on different interfaces. This is what you would do if you were configuring a machine to serve multiple web sites by using a different network interface card for each site.

Before we try this experiment, we'll do the control test. We'll create a server listening on *all* interfaces, and then try to run another server listening on all interfaces. The first one works fine, but the second one throws an exception:

```
java Accept 5555
Exception in thread "main" java.net.BindException: Address already in use
      at java.net.PlainSocketImpl.socketBind(Native Method)
      at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:322)
      at java.net.ServerSocket.bind(ServerSocket.java:311)
      at java.net.ServerSocket.bind(ServerSocket.java:269)
      at java.net.ServerSocket.<init>(ServerSocket.java:185)
      at java.net.ServerSocket.<init>(ServerSocket.java:97)
      at Accept.main(Accept.java:23)
```

This makes sense—you can't have two programs listening at the same time on the same port and address.

Now, let's see what happens when we try listening on two different addresses. First, in one window we use Accept to listen on one interface:

```
java Accept 5555 eth0
Listening
```

Then we run another listener on another interface:

```
java Accept 5555 lo
Listening
```

Since no exception was thrown, we can conclude that both programs are listening happily and have not come into conflict with each other. But, to be sure, let's check out `netstat` to see what's really going on:

```
netstat -an | grep 5555
tcp    0  0 127.0.0.1:5555      0.0.0.0:*       LISTEN
tcp    0  0 192.168.0.1:5555    0.0.0.0:*       LISTEN
```

As you can see, we've got both servers listening at the same time, each one on a different interface.

2.5 Summary

The New I/O API strives to provide Java APIs for a number of operating system features that have become standard in modern operating systems, such as file locking, nonblocking I/O, and multiplexed I/O. In these cases, the Java API provides a portion that is portable across all Java installations, and a portion that may not work in all installations but that can make use of the more advanced features found in some operating systems.

The advantages provided by these features are a large part of what motivated the NIO in the first place. Efficiency is a primary motivation, because while Java is an excellent language for developing server-side programs, it has never been able to compete with languages like C and C++ in terms of raw speed. The new features described in this chapter—and the operating system features they expose—go a long way toward remedying this.