# Table of Contents

**Chapter 12. Non-Blocking I/O522..... 1**
12.1. An Example Client303689..... 2
12.2. An Example Server303689..... 6
12.3. Buffers303689..... 13
12.4. Channels303689..... 33
12.5. Readiness Selection303689..... 39


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

# Chapter 12. Non-Blocking I/O

Compared to CPUs and memory or even disks, networks are slow. A high-end modern PC is capable of moving data between the CPU and main memory at speeds of around six gigabytes per second. It can move data to and from disk at the much slower but still respectable speed of about 150 megabytes per second.[1] By contrast, the theoretical maximum on today's fastest local area networks tops out at 120 megabytes per second, though most LANs only support speeds ten to a hundred times slower than that. And the speed across the public Internet is generally at least an order of magnitude smaller than what you see across a LAN. My faster than average SDSL line promises 96 kilobytes per second, but normally delivers only about two-thirds of that. And as I type this, my router has died and I've been reduced to a dialup connection whose bandwidth is less than six kilobytes per second. CPUs, disks, and networks are all speeding up over time. These numbers are all substantially higher than I could have reported in the first couple of editions of this book. Nonetheless, CPUs and disks are likely to remain several orders of magnitude faster than networks for the foreseeable future. The last thing you want to do in these circumstances is make the blazingly fast CPU wait for the (relatively) molasses-slow network.



[1] These are rough, theoretical maximum numbers. Nonetheless, it's worth pointing out that I'm using megabyte to mean 1,024*1,024 bytes and gigabyte to mean 1,024 megabytes. Manufacturers often round the size of a gigabyte down to 1,000 megabytes and the size of a megabyte down to 1,000,000 bytes to make their numbers sound more impressive. Furthermore, networking speeds are often referred to in kilo/mega/giga *bits* per second rather than bytes per second. Here I'm reporting all numbers in bytes so I can compare hard drive, memory, and network bandwidths.

The traditional Java solution for allowing the CPU to race ahead of the network is a combination of buffering and multithreading. Multiple threads can generate data for several different connections at once and store that data in buffers until the network is actually ready to send it; this approach works well for fairly simple servers and clients without extreme performance needs. However, the overhead of spawning multiple threads and switching between them is nontrivial. For instance, each thread requires about one extra megabyte of RAM. On a large server that may be processing thousands of requests a second, it's better not to assign a thread to each connection, even if threads for subsequent requests can be reused, as discussed in Chapter 5. The overhead of thread management severely degrades system performance. It's faster if one thread can take responsibility for multiple connections, pick one that's ready to receive data, fill it with as much data as that connection can manage as quickly as possible, then move on to the next ready connection.


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

To really work well, this approach needs to be supported by the underlying operating system. Fortunately, pretty much every modern operating system you're likely to be using as a high-volume server supports such non-blocking I/O. However, it might not be well-supported on some client systems of interest, such as PDAs, cell phones, and the like (i.e., J2ME environments). Indeed, the `java.nio` package that provides this support is not part of any current or planned J2ME profiles. However, the whole new I/O API is designed for and only really matters on servers, which is why I haven't done more than allude to it until we began talking about servers. Client and even peer-to-peer systems rarely need to process so many simultaneous connections that multithreaded, stream-based I/O becomes a noticeable bottleneck. There are some exceptions—a web spider such as Google that downloads millions of pages simultaneously could certainly use the performance boost the new I/O APIs provide—but for most clients the new I/O API is overkill, and not worth the extra complexity it entails.

## 12.1. An Example Client

Although the new I/O APIs aren't specifically designed for clients, they do work for them. I'm going to begin with a client program using the new I/O APIs because it's a little simpler. In particular, many clients can be implemented with one connection at a time, so I can introduce channels and buffers before talking about selectors and non-blocking I/O.

To demonstrate the basics, I'll implement a simple client for the character generator protocol defined in RFC 864. This protocol is designed for testing clients. The server listens for connections on port 19. When a client connects, the server sends a continuous sequence of characters until the client disconnects. Any input from the client is ignored. The RFC does not specify which character sequence to send, but recommends that the server use a recognizable pattern. One common pattern is rotating, 72-character carriage return/linefeed delimited lines of the 95 ASCII printing characters, like this:

```
!"#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
```

I picked this protocol for the examples in this chapter because both the protocol for transmitting the data and the algorithm to generate the data are simple enough that they won't obscure the I/O. However, chargen can transmit a lot of data over a relatively few

connections and quickly saturate a network connection. It's thus a good candidate for the new I/O APIs.

When implementing a client that takes advantage of Java 1.4's new I/O APIs, begin by invoking the static factory method `SocketChannel.open()` to create a new `java.nio.channels.SocketChannel` object. The argument to this method is a `java.net.SocketAddress` object indicating the host and port to connect to. For example, this fragment connects the channel to rama.poly.edu on port 19:

```
SocketAddress rama = new InetSocketAddress("rama.poly.edu", 19);
SocketChannel client = SocketChannel .open(rama);
```

The channel is opened in blocking mode, so the next line of code won't execute until the connection is established. If the connection can't be established, an `IOException` is thrown.

If this were a traditional client, you'd now ask for the socket's input and/or output streams. However, it's not. With a channel you write directly to the channel itself. Rather than writing byte arrays, you write `ByteBuffer` objects. We've got a pretty good idea that the lines of text are 74 ASCII characters long (72 printable characters followed by a carriage return/ linefeed pair) so we'll create a `ByteBuffer` that has a 74-byte capacity using the static `allocate( )` method:

```
ByteBuffer buffer= ByteBuffer.allocate(74);
```

Pass this `ByteBuffer` object to the channel's `read( )` method. The channel fills this buffer with the data it reads from the socket. It returns the number of bytes it successfully read and stored in the buffer:

```
int bytesRead = client.read(buffer);
```

By default, this will read at least one byte or return -1 to indicate the end of the data, exactly as an `InputStream` does. It will often read more bytes if more bytes are available to be read. Shortly you'll see how to put this client in non-blocking mode where it will return 0 immediately if no bytes are available, but for the moment this code blocks just like an `InputStream`. As you could probably guess, this method can also throw an `IOException` if anything goes wrong with the read.

Assuming there is some data in the buffer—that is, n > 0—this data can be copied to `System.out`. There are ways to extract a byte array from a `ByteBuffer` that can then be written on a traditional `OutputStream` such as `System.out`. However, it's more informative to stick with a pure, channel-based solution. Such a solution requires wrapping


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

the `OutputStream System.out` in a channel using the `Channels` utility class, specifically, its `newChannel( )` method:

```
WritableByteChannel output = Channels.newChannel(System.out);
```

You can then write the data that was read onto this output channel connected to `System.out`. However, before you do that you have to *flip* the buffer so that the output channel starts from the beginning of the data that was read rather than the end:

```
buffer.flip( );
output.write(buffer);
```

You don't have to tell the output channel how many bytes to write. Buffers keep track of how many bytes they contain. However, in general, the output channel is not guaranteed to write all the bytes in the buffer. In this specific case, though, it's a blocking channel and it will either do so or throw an `IOException`.

You shouldn't create a new buffer for each read and write. That would kill the performance. Instead, reuse the existing buffer. You'll need to clear the buffer before reading into it again:

```
buffer.clear( );
```

This is a little different than flipping. Flipping leaves the data in the buffer intact, but prepares it for writing rather than reading. Clearing resets the buffer to a pristine state.[2]

[2] Actually that's a tad simplistic. The old data is still present. It's not overwritten, but it will be overwritten with new data read from the source as soon as possible.

Example 12-1 puts this together into a complete client. Because chargen is by design an endless protocol, you'll need to kill the program using Ctrl-C.

**Example 12-1. A channel-based chargen client**

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

public class ChargenClient {

  public static int DEFAULT_PORT = 19;

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


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        if (args.length == 0) {
          System.out.println("Usage: java ChargenClient host [port]");
          return;
        }

        int port;
        try {
          port = Integer.parseInt(args[1]);
        }
        catch (Exception ex) {
          port = DEFAULT_PORT;
        }

        try {
          SocketAddress address = new InetSocketAddress(args[0], port);
          SocketChannel client = SocketChannel.open(address);

          ByteBuffer buffer = ByteBuffer.allocate(74);
          WritableByteChannel out = Channels.newChannel(System.out);

          while (client.read(buffer) != -1) {
            buffer.flip( );
            out.write(buffer);
            buffer.clear( );
          }
        }
        catch (IOException ex) {
            ex.printStackTrace( );
        }

      }

    }
```

Here's the output from a sample run:

```
$ java ChargenClient rama.poly.edu
 !"#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefg
!"#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'( )*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
...
```

So far, this is just an alternate vision of a program that could have easily been written using streams. The really new feature comes if you want the client to do something besides copying all input to output. You can run this connection in either blocking or non-blocking mode in which `read( )` returns immediately even if no data is available. This allows the program to do something else before it attempts to read. It doesn't have to wait for a slow network connection. To change the blocking mode, pass `true` (block) or `false` (don't block) to the `configureBlocking( )` method. Let's make this connection non-blocking:


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
client.configureBlocking(false);
```

In non-blocking mode, `read()` may return 0 because it doesn't read anything. Therefore the loop needs to be a little different:

```
while (true) {
  // Put whatever code here you want to run every pass through the loop
  // whether anything is read or not
  int n = client.read(buffer);
  if (n > 0) {
     buffer.flip( );
     out.write(buffer);
     buffer.clear( );
  }
  else if (n == -1) {
    // This shouldn't happen unless the server is misbehaving.
    break;
  }
}
```

There's not a lot of call for this in a one-connection client like this one. Perhaps you could check to see if the user has done something to cancel input, for example. However, as you'll see in the next section, when a program is processing multiple connections, this enables code to run very quickly on the fast connections and more slowly on the slow ones. Each connection gets to run at its own speed without being held up behind the slowest driver on the one-lane road.

## 12.2. An Example Server

Clients are well and good, but channels and buffers are really intended for server systems that need to process many simultaneous connections efficiently. Handling servers requires a third new piece in addition to the buffers and channels used for the client. Specifically, you need selectors that allow the server to find all the connections that are ready to receive output or send input.

To demonstrate the basics, I'll implement a simple server for the character generator protocol. When implementing a server that takes advantage of Java 1.4's new I/O APIs, begin by calling the static factory method `ServerSocketChannel.open()` method to create a new `ServerSocketChannel` object:

```
ServerSocketChannel serverChannel = ServerSocketChannel .open( );
```

Initially this channel is not actually listening on any port. To bind it to a port, retrieve its `ServerSocket` peer object with the `socket( )` method and then use the `bind( )`

method on that peer. For example, this code fragment binds the channel to a server socket on port 19:

```
ServerSocket ss = serverChannel.socket( );
ss.bind(new InetSocketAddress(19));
```

As with regular server sockets, binding to port 19 requires you to be root on Unix (including Linux and Mac OS X). Nonroot users can only bind to ports 1024 and higher.

The server socket channel is now listening for incoming connections on port 19. To accept one, call the `ServerSocketChannel` `accept()` method, which returns a `SocketChannel` object:

```
SocketChannel clientChannel = serverChannel.accept( );
```

On the server side, you'll definitely want to make the client channel non-blocking to allow the server to process multiple simultaneous connections:

```
clientChannel.configureBlocking(false);
```

You may also want to make the `ServerSocketChannel` non-blocking. By default, this `accept( )` method blocks until there's an incoming connection, like the `accept( )` method of `ServerSocket`. To change this, simply call `configureBlocking(false)` before calling `accept( )`:

```
serverChannel.configureBlocking(false);
```

A non-blocking `accept( )` returns null almost immediately if there are no incoming connections. Be sure to check for that or you'll get a nasty `NullPointerException` when trying to use the socket.

There are now two open channels: a server channel and a client channel. Both need to be processed. Both can run indefinitely. Furthermore, processing the server channel will create more open client channels. In the traditional approach, you assign each connection a thread, and the number of threads climbs rapidly as clients connect. Instead, in the new I/O API, you create a `Selector` that enables the program to iterate over all the connections that are ready to be processed. To construct a new `Selector`, just call the static `Selector.open ( )` factory method:

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

Next you need to register each channel with the selector that monitors it using the channel's `register()` method. When registering, specify the operation you're interested in using a

named constant from the `SelectionKey` class. For the server socket, the only operation of interest is OP_ACCEPT; that is, is the server socket channel ready to accept a new connection?

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

For the client channels, you want to know something a little different, specifically, whether they're ready to have data written onto them. For this, use the OP_WRITE key:

```
SelectionKey key = clientChannel.register(selector, SelectionKey.OP_WRITE);
```

Both `register( )` methods return a `SelectionKey` object. However, we're only going to need to use that key for the client channels, because there can be more than one of them. Each `SelectionKey` has an attachment of arbitrary `Object` type. This is normally used to hold an object that indicates the current state of the connection. In this case, we can store the buffer that the channel writes onto the network. Once the buffer is fully drained, we'll refill it. Fill an array with the data that will be copied into each buffer. Rather than writing to the end of the buffer, then rewinding to the beginning of the buffer and writing again, it's easier just to start with two sequential copies of the data so every line is available as a contiguous sequence in the array:

```
byte[] rotation = new byte[95*2];
for (byte i = ' '; i <= '~'; i++) {
  rotation[i-' '] = i;
  rotation[i+95-' '] = i;
}
```

Because this array will only be read from after it's been initialized, you can reuse it for multiple channels. However, each channel will get its own buffer filled with the contents of this array. We'll stuff the buffer with the first 72 bytes of the rotation array, then add a carriage return/linefeed pair to break the line. Then we'll flip the buffer so it's ready for draining, and attach it to the channel's key:

```
ByteBuffer buffer = ByteBuffer.allocate(74);
buffer.put(rotation, 0, 72);
buffer.put((byte) '\r');
buffer.put((byte) '\n');
buffer.flip( );
key2.attach(buffer);
```

To check whether anything is ready to be acted on, call the selector's `select( )` method. For a long-running server, this normally goes in an infinite loop:

```
while (true) {
  selector.select ( );
  // process selected keys...
}
```

Assuming the selector does find a ready channel, its `selectedKeys( )` method returns a `java.util.Set` containing one `SelectionKey` object for each ready channel. Otherwise it returns an empty set. In either case, you can loop through this with a `java.util.Iterator`:

```
Set readyKeys = selector.selectedKeys( );
Iterator iterator = readyKeys.iterator( );
while (iterator.hasNext( )) {
  SelectionKey key = (SelectionKey) (iterator.next( ));
  // Remove key from set so we don't process it twice
  iterator.remove( );
  // operate on the channel...
}
```

Removing the key from the set tells the selector that we've dealt with it, and the `Selector` doesn't need to keep giving it back to us every time we call `select( )`. The `Selector` will add the channel back into the ready set when `select( )` is called again if the channel becomes ready again. It's really important to remove the key from the ready set here, though.

If the ready channel is the server channel, the program accepts a new socket channel and adds it to the selector. If the ready channel is a socket channel, the program writes as much of the buffer as it can onto the channel. If no channels are ready, the selector waits for one. One thread, the main thread, processes multiple simultaneous connections.

In this case, it's easy to tell whether a client or a server channel has been selected because the server channel will only be ready for accepting and the client channels will only be ready for writing. Both of these are I/O operations, and both can throw `IOException`s for a variety of reasons, so you'll want to wrap this all in a `try` block.

```
try {
  if (key.isAcceptable( )) {
    ServerSocketChannel server = (ServerSocketChannel )
                                          key.channel( );
    SocketChannel  connection = server.accept( );
    connection.configureBlocking(false);
    connection.register(selector,
                        SelectionKey.OP_WRITE);
    // set up the buffer for the client...
  }
  else if (key.isWritable( )) {
    SocketChannel client = (SocketChannel ) key.channel( );
    // write data to client...
  }
}
```

Writing the data onto the channel is easy. Retrieve the key's attachment, cast it to `ByteBuffer`, and call `hasRemaining()` to check whether there's any unwritten data left


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

in the buffer. If there is, write it. Otherwise, refill the buffer with the next line of data from the `rotation` array and write that.

```
ByteBuffer buffer = (ByteBuffer) key.attachment( );
if (!buffer.hasRemaining( )) {
  // Refill the buffer with the next line
  // Figure out where the last line started
  buffer.rewind( );
  int first = buffer.get( );
  // Increment to the next character
  buffer.rewind( );
  int position = first - ' ' + 1;
  buffer.put(rotation, position, 72);
  buffer.put((byte) '\r');
  buffer.put((byte) '\n');
  buffer.flip( );
}
client.write(buffer);
```

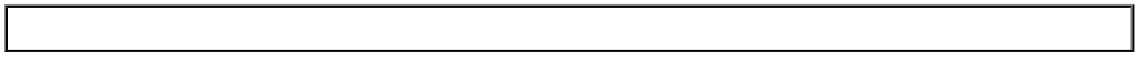
The algorithm that figures out where to grab the next line of data relies on the characters being stored in the `rotation` array in ASCII order. It should be familiar to anyone who learned C from Kernighan and Ritchie, but for the rest of us it needs a little explanation. `buffer.get()` reads the first byte of data from the buffer. From this number we subtract the space character (32) because that's the first character in the `rotation` array. This tells us which index in the array the buffer currently starts at. We add 1 to find the start of the next line and refill the buffer.

In the chargen protocol, the server never closes the connection. It waits for the client to break the socket. When this happens, an exception will be thrown. Cancel the key and close the corresponding channel:

```
catch (IOException ex) {
  key.cancel( );
  try {
    // You can still get the channel from the key after cancelling the key.
    key.channel( ).close( );
  }
  catch (IOException cex) {
  }
}
```

Example 12-2 puts this all together in a complete chargen server that processes multiple connections efficiently in a single thread.

**Example 12-2. A non-blocking chargen server**



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

public class ChargenServer {

  public static int DEFAULT_PORT = 19;

  public static void main(String[] args) {

    int port;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
      port = DEFAULT_PORT;
    }
    System.out.println("Listening for connections on port " + port);

    byte[] rotation = new byte[95*2];
    for (byte i = ' '; i <= '~'; i++) {
        rotation[i-' '] = i;
        rotation[i+95-' '] = i;
    }

    ServerSocketChannel serverChannel;
    Selector selector;
    try {
      serverChannel = ServerSocketChannel.open( );
      ServerSocket ss = serverChannel.socket( );
      InetSocketAddress address = new InetSocketAddress(port);
      ss.bind(address);
      serverChannel.configureBlocking(false);
      selector = Selector.open( );
      serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    }
    catch (IOException ex) {
      ex.printStackTrace( );
      return;
    }

    while (true) {

      try {
        selector.select( );
      }
      catch (IOException ex) {
        ex.printStackTrace( );
        break;
      }

      Set readyKeys = selector.selectedKeys( );
      Iterator iterator = readyKeys.iterator( );
      while (iterator.hasNext( )) {

        SelectionKey key = (SelectionKey) iterator.next( );
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
            iterator.remove( );
            try {
              if (key.isAcceptable( )) {
                ServerSocketChannel server = (ServerSocketChannel) key.channel( );
                SocketChannel client = server.accept( );
                System.out.println("Accepted connection from " + client);
                client.configureBlocking(false);
                SelectionKey key2 = client.register(selector, SelectionKey.
                                                                    OP_WRITE);
                ByteBuffer buffer = ByteBuffer.allocate(74);
                buffer.put(rotation, 0, 72);
                buffer.put((byte) '\r');
                buffer.put((byte) '\n');
                buffer.flip( );
                key2.attach(buffer);
              }
              else if (key.isWritable( )) {
                SocketChannel client = (SocketChannel) key.channel( );
                ByteBuffer buffer = (ByteBuffer) key.attachment( );
                if (!buffer.hasRemaining( )) {
                  // Refill the buffer with the next line
                  buffer.rewind( );
                  // Get the old first character
                  int first = buffer.get( );
                  // Get ready to change the data in the buffer
                  buffer.rewind( );
                  // Find the new first characters position in rotation
                  int position = first - ' ' + 1;
                  // copy the data from rotation into the buffer
                  buffer.put(rotation, position, 72);
                  // Store a line break at the end of the buffer
                  buffer.put((byte) '\r');
                  buffer.put((byte) '\n');
                  // Prepare the buffer for writing
                  buffer.flip( );
                }
                client.write(buffer);
              }
            }
            catch (IOException ex) {
              key.cancel( );
              try {
                key.channel( ).close( );
              }
              catch (IOException cex) {}
            }

          }

        }

      }

    }
```

This example only uses one thread. There are situations where you might still want to use multiple threads, especially if different operations have different priorities. For instance, you might want to accept new connections in one high priority thread and service existing


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

connections in a lower priority thread. However, you're no longer required to have a 1:1 ratio between threads and connections, which dramatically improves the scalability of servers written in Java.

It may also be important to use multiple threads for maximum performance. Multiple threads allow the server to take advantage of multiple CPUs. Even with a single CPU, it's often a good idea to separate the accepting thread from the processing threads. The thread pools discussed in Chapter 5 are still relevant even with the new I/O model. The thread that accepts the connections can add the connections it's accepted into the queue for processing by the threads in the pool. This is still faster than doing the same thing without selectors because `select()` ensures you're never wasting any time on connections that aren't ready to receive data. On the other hand, the synchronization issues here are quite tricky, so don't attempt this solution until profiling proves there is a bottleneck.

## 12.3. Buffers

In Chapter 4, I recommended that you always buffer your streams. Almost nothing has a greater impact on the performance of network programs than a big enough buffer. In the new I/O model, however, you're no longer given the choice. All I/O is buffered. Indeed the buffers are fundamental parts of the API. Instead of writing data onto output streams and reading data from input streams, you read and write data from buffers. Buffers may appear to be just an array of bytes as in buffered streams. However, native implementations can connect them directly to hardware or memory or use other, very efficient implementations.

From a programming perspective, the key difference between streams and channels is that streams are byte-based while channels are block-based. A stream is designed to provide one byte after the other, in order. Arrays of bytes can be passed for performance. However, the basic notion is to pass data one byte at a time. By contrast, a channel passes blocks of data around in buffers. Before bytes can be read from or written to a channel, the bytes have to be stored in a buffer, and the data is written or read one buffer at a time.

The second key difference between streams and channels/buffers is that channels and buffers tend to support both reading and writing on the same object. This isn't always true. For instance, a channel that points to a file on a CD-ROM can be read but not written. A channel connected to a socket that has shutdown input could be written but not read. If you try to write to a read-only channel or read from a write-only channel, an `UnsupportedOperationException` will be thrown. However, more often that not network programs can read from and write to the same channels.

Without worrying too much about the underlying details (which can vary hugely from one implementation to the next, mostly a result of being tuned very closely to the host operating system and hardware), you can think of a buffer as a fixed-size list of elements of a particular, normally primitive data type, like an array. However, it's not necessarily an array behind the scenes. Sometimes it is; sometimes it isn't. There are specific subclasses of `Buffer` for all of Java's primitive data types except boolean: `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer`, and `DoubleBuffer`. The methods in each subclass have appropriately typed return values and argument lists. For example, the `DoubleBuffer` class has methods to put and get doubles. The `IntBuffer` class has methods to put and get ints. The common `Buffer` superclass only provides methods that don't need to know the type of the data the buffer contains. (The lack of primitive-aware generics really hurts here.) Network programs use `ByteBuffer` almost exclusively, although occasionally one program might use a view that overlays the `ByteBuffer` with one of the other types.

Besides its list of data, each buffer tracks four key pieces of information. All buffers have the same methods to set and get these values, regardless of the buffer's type:

*position*
: The next location in the buffer that will be read from or written to. Like most indexes in Java, this starts counting at 0 and has a maximum value one less than the size of the buffer. It can be set or gotten with these two methods:

```
public final int    position( )
public final Buffer position(int newPosition)
```

*capacity*
: The maximum number of elements the buffer can hold. This is set when the buffer is created and cannot be changed thereafter. It can be read with this method:

```
public final int  capacity( )
```

*limit*
: The last location in the buffer that can hold data. You cannot write or read past this point without changing the limit, even if the buffer has more capacity. It is set and gotten with these two methods:

```
public final int    limit( )
public final Buffer limit(int newLimit)
```

*mark*
: A client-specified index in the buffer. It is set at the current position by invoking the `mark()` method. The current position is set to the marked position by invoking `reset( )`:

```
public final Buffer mark( )
public final Buffer reset( )
```

No, I can't explain why these methods (and several similar methods in the `java.nio` packages) don't follow the standard Java `get`*`Foo`*`()`/`set`*`Foo`*`( )` naming convention. Blame it on the smoke-filled chat rooms in the Java Community Process.

Unlike reading from an `InputStream`, reading from a buffer does not actually change the buffer's data in any way. It's possible to set the position either forwards or backwards so you can start reading from a particular place in the buffer. Similarly, a program can adjust the limit to control the end of the data that will be read. Only the capacity is fixed.

The common `Buffer` superclass also provides a few other methods that operate by reference to these common properties.

The clear( ) method "empties" the buffer by setting the position to zero and the limit to the capacity. This allows the buffer to be completely refilled:

```
public final Buffer clear( )
```

However, the `clear( )` method does not remove the old data from the buffer. It's still present and could be read using absolute get methods or changing the limit and position again.

The `rewind( )` method sets the position to zero, but does not change the limit:

```
public final Buffer rewind( )
```

This allows the buffer to be reread.

The `flip( )` method sets the limit to the current position and the position to zero:

```
public final Buffer flip( )
```

It is called when you want to drain a buffer you've just filled.

Finally, there are two methods that return information about the buffer but don't change it. The `remaining()` method returns the number of elements in the buffer between the current position and the limit. The `hasRemaining( )` method returns true if the number of remaining elements is greater than zero:

```
public final int     remaining( )
public final boolean hasRemaining( )
```

## 12.3.1. Creating Buffers

The buffer class hierarchy is based on inheritance but not really on polymorphism, at least not at the top level. You normally need to know whether you're dealing with an `IntBuffer` or a `ByteBuffer` or a `CharBuffer` or something else. You write code to one of these subclasses, not to the common `Buffer` superclass. However, at the level of `IntBuffer`/`ByteBuffer`/`CharBuffer`, etc., the classes are polymorphic. These classes are abstract too, and you use a factory method to retrieve an implementation-specific subclass such as `java.nio.HeapByteBuffer`. However, you only treat the actual object as an instance of its superclass, `ByteBuffer` in this case.

Each typed buffer class has several factory methods that create implementation-specific subclasses of that type in various ways. Empty buffers are normally created by *allocate* methods. Buffers that are prefilled with data are created by *wrap* methods. The allocate methods are often useful for input while the wrap methods are normally used for output.

### 12.3.1.1. Allocation

The basic `allocate( )` method simply returns a new, empty buffer with a specified fixed capacity. For example, these lines create byte and int buffers, each with a size of 100:

```
ByteBuffer buffer1 = ByteBuffer.allocate(100);
IntBuffer  buffer2 = IntBuffer.allocate(100);
```

The cursor is positioned at the beginning of the buffer; that is, the position is 0. A buffer created by `allocate( )` will be implemented on top of a Java array, which can be accessed by the `array( )` and `arrayOffset( )` methods. For example, you could read a large chunk of data into a buffer using a channel and then retrieve the array from the buffer to pass to other methods:

```
byte[] data1 = buffer1.array( );
int[]  data2 = buffer2.array( );
```

The `array( )` method does expose the buffer's private data, so use it with caution. Changes to the backing array are reflected in the buffer and vice versa. The normal pattern here is to fill the buffer with data, retrieve its backing array, and then operate on the array. This isn't a problem as long as you don't write to the buffer after you've started working with the array.


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

#### 12.3.1.2. Direct allocation

The `ByteBuffer` class (but not the other buffer classes) has an additional `allocateDirect()` method that may not create a backing array for the buffer. The VM may implement a directly allocated `ByteBuffer` using direct memory access to the buffer on an Ethernet card, kernel memory, or something else. It's not required, but it's allowed, and this can improve performance for I/O operations. From an API perspective, the `allocateDirect( )` is used exactly like `allocate( )`:

```
ByteBuffer buffer1 = ByteBuffer.allocateDirect(100);
```

Invoking `array( )` and `arrayOffset()` on a direct buffer will throw an `UnsupportedOperationException`. Direct buffers may be faster on some virtual machines, especially if the buffer is large (roughly a megabyte or more). However, direct buffers are more expensive to create than indirect buffers, so they should only be allocated when the buffer is expected to be around for awhile. The details are highly VM-dependent. As is generally true for most performance advice, you probably shouldn't even consider using direct buffers until measurements prove performance is an issue.

#### 12.3.1.3. Wrapping

If you already have an array of data that you want to output, you'll normally wrap a buffer around it, rather than allocating a new buffer and copying its components into the buffer one at a time. For example:

```
byte[] data = "Some data".getBytes("UTF-8");
ByteBuffer buffer1 = ByteBuffer.wrap(data);
char[] text = "Some text".toCharArray( );
CharBuffer buffer2 = CharBuffer.wrap(text);
```

Here, the buffer contains a reference to the array, which serves as its backing array. Buffers created by wrapping are never direct. Again, changes to the array are reflected in the buffer and vice versa, so don't wrap the array until you're finished with it.

### 12.3.2. Filling and Draining

Buffers are designed for sequential access. Besides its list of data, each buffer has a cursor indicating its current position. The cursor is an int that counts from zero to the number of elements in the buffer; the cursor is incremented by one when an element is read from or written to the buffer. It can also be positioned manually. For example, suppose you want to


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

reverse the characters in a string. There are at least a dozen different ways to do this, including using string buffers,[3] `char[]` arrays, linked lists, and more. However, if we were to do it with a `CharBuffer`, we might begin by filling a buffer with the data from the string:

[3] By the way, a `StringBuffer` is not a buffer in the sense of this section. Aside from the very generic notion of buffering, it has nothing in common with the classes being discussed here.

```
String s = "Some text";
CharBuffer buffer = CharBuffer.wrap(s);
```

We can only fill the buffer up to its capacity. If we tried to fill it past its initially set capacity, the `put( )` method would throw a `BufferOverflowException`. Similarly, if we now tried to `get( )` from the buffer, there'd be a `BufferOverflowException`. Before we can read the data out again, we need to flip the buffer:

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

This repositions the cursor at the start of the buffer. We can drain it into a new string:

```
String result = "";
while (buffer.hasRemaining( )) {
  result+= buffer.get( );
}
```

Buffer classes also have *absolute* methods that fill and drain at specific positions within the buffer without updating the cursor. For example, `ByteBuffer` has these two:

```
public abstract byte       get(int index)
public abstract ByteBuffer put(int index, byte b)
```

These both throw `IndexOutOfBoundsException` if you try to access a position past the limit of the buffer. For example, using absolute methods, you could reverse a string into a buffer like this:

```
String s = "Some text";
CharBuffer buffer = CharBuffer.allocate(s.length( ));
for (int i = 0; i < s.length( ); i++) {
  buffer.put(s.length( ) - i - 1, s.charAt(i));
}
```

## 12.3.3. Bulk Methods

Even with buffers it's often faster to work with blocks of data rather than filling and draining one element at a time. The different buffer classes have bulk methods that fill and drain an array of their element type.

For example, `ByteBuffer` has `put( )` and `get( )` methods that fill and drain a `ByteBuffer` from a preexisting byte array or subarray:

```
public ByteBuffer get(byte[] dst, int offset, int length)
public ByteBuffer get(byte[] dst)
public ByteBuffer put(byte[] array, int offset, int length)
public ByteBuffer put(byte[] array)
```

These put methods insert the data from the specified array or subarray, beginning at the current position. The get methods read the data into the argument array or subarray beginning at the current position. Both put and get increment the position by the length of the array or subarray. The put methods throw a `BufferOverflowException` if the buffer does not have sufficient space for the array or subarray. The get methods throw a `BufferUnderflowException` if the buffer does not have enough data remaining to fill the array or subarrray. These are runtime exceptions.

## 12.3.4. Data Conversion

All data in Java ultimately resolves to bytes. Any primitive data type—`int`, `double`, `float`, etc.—can be written as bytes. Any sequence of bytes of the right length can be interpreted as a primitive datum. For example, any sequence of four bytes corresponds to an `int` or a `float` (actually both, depending on how you want to read it). A sequence of eight bytes corresponds to a `long` or a `double`. The `ByteBuffer` class (and only the `ByteBuffer` class) provides relative and absolute put methods that fill a buffer with the bytes corresponding to an argument of primitive type (except boolean) and relative and absolute get methods that read the appropriate number of bytes to form a new primitive datum:

```
public abstract char       getChar( )
public abstract ByteBuffer putChar(char value)
public abstract char       getChar(int index)
public abstract ByteBuffer putChar(int index, char value)
public abstract short      getShort( )
public abstract ByteBuffer putShort(short value)
public abstract short      getShort(int index)
public abstract ByteBuffer putShort(int index, short value)
public abstract int        getInt( )
public abstract ByteBuffer putInt(int value)
public abstract int        getInt(int index)
public abstract ByteBuffer putInt(int index, int value)
public abstract long       getLong( )
public abstract ByteBuffer putLong(long value)
public abstract long       getLong(int index)
public abstract ByteBuffer putLong(int index, long value)
public abstract float      getFloat( )
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public abstract ByteBuffer putFloat(float value)
public abstract float      getFloat(int index)
public abstract ByteBuffer putFloat(int index, float value)
public abstract double     getDouble( )
public abstract ByteBuffer putDouble(double value)
public abstract double     getDouble(int index)
public abstract ByteBuffer putDouble(int index, double value)
```

In the world of new I/O, these methods do the job performed by `DataOutputStream` and `DataInputStream` in traditional I/O. These methods do have an additional ability not present in `DataOutputStream` and `DataInputStream`. You can choose whether to interpret the byte sequences as big-endian or little-endian ints, floats, doubles, etc. By default, all values are read and written as big-endian; that is, most significant byte first. The two `order ( )` methods inspect and set the buffer's byte order using the named constants in the `ByteOrder` class. For example, you can change the buffer to little-endian interpretation like so:

```
if (buffer.order( ).equals(ByteOrder.BIG_ENDIAN)) {
  buffer.order(ByteOrder.LITLLE_ENDIAN);
}
```

Suppose instead of a chargen protocol, you want to test the network by generating binary data. This test can highlight problems that aren't apparent in the ASCII chargen protocol, such as an old gateway configured to strip off the high order bit of every byte, throw away every $2^{30}$ byte, or put into diagnostic mode by an unexpected sequence of control characters. These are not theoretical problems. I've seen variations on all of these at one time or another.

You could test the network for such problems by sending out every possible `int`. This would, after about 4.2 billion iterations, test every possible four-byte sequence. On the receiving end, you could easily test whether the data received is expected with a simple numeric comparison. If any problems are found, it is easy to tell exactly where they occurred. In other words, this protocol (call it `Intgen`) behaves like this:

1. The client connects to the server.
2. The server immediately begins sending four-byte, big-endian integers, starting with 0 and incrementing by 1 each time. The server will eventually wrap around into the negative numbers.
3. The server runs indefinitely. The client closes the connection when it's had enough.

The server would store the current int in a 4-byte long direct `ByteBuffer`. One buffer would be attached to each channel. When the channel becomes available for writing, the buffer is drained onto the channel. Then the buffer is rewound and the content of the buffer is read with `getInt( )`. The program then clears the buffer, increments the previous value by one, and fills the buffer with the new value using `putInt()`. Finally, it flips the buffer so it will be ready to be drained the next time the channel becomes writable. Example 12-3 demonstrates.

**Example 12-3. Intgen server**

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

public class IntgenServer {

  public static int DEFAULT_PORT = 1919;

  public static void main(String[] args) {

    int port;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
      port = DEFAULT_PORT;
    }
    System.out.println("Listening for connections on port " + port);

    ServerSocketChannel serverChannel;
    Selector selector;
    try {
      serverChannel = ServerSocketChannel.open( );
      ServerSocket ss = serverChannel.socket( );
      InetSocketAddress address = new InetSocketAddress(port);
      ss.bind(address);
      serverChannel.configureBlocking(false);
      selector = Selector.open( );
      serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    }
    catch (IOException ex) {
      ex.printStackTrace( );
      return;
    }

    while (true) {

      try {
        selector.select( );
      }
      catch (IOException ex) {
        ex.printStackTrace( );
        break;
      }

      Set readyKeys = selector.selectedKeys( );
      Iterator iterator = readyKeys.iterator( );
      while (iterator.hasNext( )) {

        SelectionKey key = (SelectionKey) iterator.next( );
        iterator.remove( );
        try {
          if (key.isAcceptable( )) {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
                ServerSocketChannel server = (ServerSocketChannel ) key.channel( );
                SocketChannel client = server.accept( );
                System.out.println("Accepted connection from " + client);
                client.configureBlocking(false);
                SelectionKey key2 = client.register(selector, SelectionKey.
                                                                    OP_WRITE);
                ByteBuffer output = ByteBuffer.allocate(4);
                output.putInt(0);
                output.flip( );
                key2.attach(output);
              }
              else if (key.isWritable( )) {
                SocketChannel client = (SocketChannel) key.channel( );
                ByteBuffer output = (ByteBuffer) key.attachment( );
                if (! output.hasRemaining( )) {
                  output.rewind( );
                  int value = output.getInt( );
                  output.clear( );
                  output.putInt(value+1);
                  output.flip( );
                }
                client.write(output);
              }
            }
            catch (IOException ex) {
              key.cancel( );
              try {
                key.channel( ).close( );
              }
              catch (IOException cex) {}
            }

          }

        }

      }

    }
```

## 12.3.5. View Buffers

If you know the `ByteBuffer` read from a `SocketChannel` contains nothing but elements of one particular primitive data type, it may be worthwhile to create a *view buffer*. This is a new `Buffer` object of appropriate type such as `DoubleBuffer`, `IntBuffer`, etc., which draws its data from an underlying `ByteBuffer` beginning with the current position. Changes to the view buffer are reflected in the underlying buffer and vice versa. However, each buffer has its own independent limit, capacity, mark, and position. View buffers are created with one of these six methods in `ByteBuffer`:


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public abstract ShortBuffer  asShortBuffer( )
public abstract CharBuffer   asCharBuffer( )
public abstract IntBuffer    asIntBuffer( )
public abstract LongBuffer   asLongBuffer( )
public abstract FloatBuffer  asFloatBuffer( )
public abstract DoubleBuffer asDoubleBuffer( )
```

For example, consider a client for the `Intgen` protocol. This protocol is only going to read ints, so it may be helpful to use an `IntBuffer` rather than a `ByteBuffer`. Example 12-4 demonstrates. For variety, this client is synchronous and blocking, but it still uses channels and buffers.

**Example 12-4. Intgen client**

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

public class IntgenClient {

  public static int DEFAULT_PORT = 1919;

  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println("Usage: java IntgenClient host [port]");
      return;
    }

    int port;
    try {
      port = Integer.parseInt(args[1]);
    }
    catch (Exception ex) {
      port = DEFAULT_PORT;
    }

    try {
      SocketAddress address = new InetSocketAddress(args[0], port);
      SocketChannel client  = SocketChannel.open(address);
      ByteBuffer    buffer  = ByteBuffer.allocate(4);
      IntBuffer     view    = buffer.asIntBuffer( );

      for (int expected = 0; ; expected++) {
        client.read(buffer);
        int actual = view.get( );
        buffer.clear( );
        view.rewind( );

        if (actual != expected) {
          System.err.println("Expected " + expected + "; was " + actual);
          break;
```

```
            }
            System.out.println(actual);
          }
        }
        catch (IOException ex) {
            ex.printStackTrace( );
        }

      }

    }
```

There's one thing to note here. Although you can fill and drain the buffers using the methods of the `IntBuffer` class exclusively, data must be read from and written to the channel using the original `ByteBuffer` of which the `IntBuffer` is a view. The `SocketChannel` class only has methods to read and write `ByteBuffer`s. It cannot read or write any other kind of buffer. This also means you need to clear the `ByteBuffer` on each pass through the loop or the buffer will fill up and the program will halt. The positions and limits of the two buffers are independent and must be considered separately. Finally, if you're working in non-blocking mode, be careful that all the data in the underlying `ByteBuffer` is drained before reading or writing from the overlaying view buffer. Non-blocking mode provides no guarantee that the buffer will still be aligned on an `int`/`double`/`char`/etc. boundary following a drain. It's completely possible for a non-blocking channel to write half the bytes of an int or a double. When using non-blocking I/O, be sure to check for this problem before putting more data in the view buffer.

## 12.3.6. Compacting Buffers

Most writable buffers support a `compact( )` method:

```
public abstract ByteBuffer   compact( )
public abstract IntBuffer    compact( )
public abstract ShortBuffer  compact( )
public abstract FloatBuffer  compact( )
public abstract CharBuffer   compact( )
public abstract DoubleBuffer compact( )
```

(If it weren't for invocation chaining, these six methods could have been replaced by one method in the common `Buffer` superclass.) Compacting shifts any remaining data in the buffer to the start of the buffer, freeing up more space for elements. Any data that was in those positions will be overwritten. The buffer's position is set to the end of the data so it's ready for writing more data.


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Compacting is an especially useful operation when you're *copying*—reading from one channel and writing the data to another using non-blocking I/O. You can read some data into a buffer, write the buffer out again, then compact the data so all the data that wasn't written is at the beginning of the buffer, and the position is at the end of the data remaining in the buffer, ready to receive more data. This allows the reads and writes to be interspersed more or less at random with only one buffer. Several reads can take place in a row, or several writes follow consecutively. If the network is ready for immediate output but not input (or vice versa), the program can take advantage of that. This technique can be used to implement an echo server as shown in Example 12-5. The echo protocol simply responds to the client with whatever data the client sent. Like chargen, it's useful for network testing. Also like chargen, echo relies on the client to close the connection. Unlike chargen, however, an echo server must both read and write from the connection.

**Example 12-5. Echo server**

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

public class EchoServer {

  public static int DEFAULT_PORT = 7;

  public static void main(String[] args) {

    int port;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
      port = DEFAULT_PORT;
    }
    System.out.println("Listening for connections on port " + port);


    ServerSocketChannel serverChannel;
    Selector selector;
    try {
      serverChannel = ServerSocketChannel.open( );
      ServerSocket ss = serverChannel.socket( );
      InetSocketAddress address = new InetSocketAddress(port);
      ss.bind(address);
      serverChannel.configureBlocking(false);
      selector = Selector.open( );
      serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    }
    catch (IOException ex) {
```

```
            ex.printStackTrace( );
            return;
          }

          while (true) {

            try {
              selector.select( );
            }
            catch (IOException ex) {
              ex.printStackTrace( );
              break;
            }

            Set readyKeys = selector.selectedKeys( );
            Iterator iterator = readyKeys.iterator( );
            while (iterator.hasNext( )) {

              SelectionKey key = (SelectionKey) iterator.next( );
              iterator.remove( );
              try {
                if (key.isAcceptable( )) {
                  ServerSocketChannel server = (ServerSocketChannel ) key.channel( );
                  SocketChannel client = server.accept( );
                  System.out.println("Accepted connection from " + client);
                  client.configureBlocking(false);
                  SelectionKey clientKey = client.register(
                    selector, SelectionKey.OP_WRITE | SelectionKey.OP_READ);
                  ByteBuffer buffer = ByteBuffer.allocate(100);
                  clientKey.attach(buffer);
                }
                if (key.isReadable( )) {
                  SocketChannel client = (SocketChannel) key.channel( );
                  ByteBuffer output = (ByteBuffer) key.attachment( );
                  client.read(output);
                }
                if (key.isWritable( )) {
                  SocketChannel client = (SocketChannel) key.channel( );
                  ByteBuffer output = (ByteBuffer) key.attachment( );
                  output.flip( );
                  client.write(output);
                  output.compact( );
                }
              }
              catch (IOException ex) {
                key.cancel( );
                try {
                  key.channel( ).close( );
                }
                catch (IOException cex) {}
              }

            }

          }

        }

      }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

One thing I noticed while writing and debugging this program: the buffer size makes a big difference, although perhaps not in the way you might think. A big buffer can hide a lot of bugs. If the buffer is large enough to hold complete test cases without being flipped or drained, it's very easy to not notice that the buffer isn't being flipped or compacted at the right times because the test cases never actually need to do that. Before releasing your program, try turning the buffer size down to something significantly lower than the input you're expecting. In this case, I tested with a buffer size of 10. This test degrades performance, so you shouldn't ship with such a ridiculously small buffer, but you absolutely should test your code with small buffers to make sure it behaves properly when the buffer fills up.

## 12.3.7. Duplicating Buffers

It's often desirable to make a copy of a buffer to deliver the same information to two or more channels. The `duplicate()` methods in each of the six typed buffer classes do this:

```
public abstract ByteBuffer   duplicate( )
public abstract IntBuffer    duplicate( )
public abstract ShortBuffer  duplicate( )
public abstract FloatBuffer  duplicate( )
public abstract CharBuffer   duplicate( )
public abstract DoubleBuffer duplicate( )
```

The return values are not clones. The duplicated buffers share the same data, including the same backing array if the buffer is indirect. Changes to the data in one buffer are reflected in the other buffer. Thus, you should mostly use this method when you're only going to read from the buffers. Otherwise, it can be tricky to keep track of where the data is being modified.

The original and duplicated buffers do have independent marks, limits, and positions even though they share the same data. One buffer can be ahead of or behind the other buffer.

Duplication is useful when you want to transmit the same data over multiple channels, roughly in parallel. You can make duplicates of the main buffer for each channel and allow each channel to run at its own speed. For example, recall the single file HTTP server in Example 10-6. Reimplemented with channels and buffers as shown in Example 12-6, `NonblockingSingleFileHTTPServer`, the single file to serve is stored in one constant, read-only buffer. Every time a client connects, the program makes a duplicate of this buffer just for that channel, which is stored as the channel's attachment. Without duplicates, one client has to wait till the other finishes so the original buffer can be rewound. Duplicates enable simultaneous buffer reuse.

**Example 12-6. A non-blocking HTTP server that chunks out the same file**

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

public class NonblockingSingleFileHTTPServer {

  private ByteBuffer contentBuffer;
  private int port = 80;

  public NonblockingSingleFileHTTPServer(
   ByteBuffer data, String encoding, String MIMEType, int port)
   throws UnsupportedEncodingException {

    this.port = port;
    String header = "HTTP/1.0 200 OK\r\n"
     + "Server: OneFile 2.0\r\n"
     + "Content-length: " + data.limit( ) + "\r\n"
     + "Content-type: " + MIMEType + "\r\n\r\n";
    byte[] headerData = header.getBytes("ASCII");

    ByteBuffer buffer = ByteBuffer.allocate(
     data.limit( ) + headerData.length);
    buffer.put(headerData);
    buffer.put(data);
    buffer.flip( );
    this.contentBuffer = buffer;

  }

  public void run( ) throws IOException {

    ServerSocketChannel serverChannel = ServerSocketChannel.open( );
    ServerSocket  serverSocket = serverChannel.socket( );
    Selector selector = Selector.open( );
    InetSocketAddress localPort = new InetSocketAddress(port);
    serverSocket.bind(localPort);
    serverChannel.configureBlocking(false);
    serverChannel.register(selector, SelectionKey.OP_ACCEPT);

    while (true) {

      selector.select( );
      Iterator keys = selector.selectedKeys( ).iterator( );
      while (keys.hasNext( )) {
        SelectionKey key = (SelectionKey) keys.next( );
        keys.remove( );
        try {
          if (key.isAcceptable( )) {
            ServerSocketChannel server = (ServerSocketChannel) key.channel( );
            SocketChannel channel = server.accept( );
            channel.configureBlocking(false);
            SelectionKey newKey = channel.register(selector,
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
                                                    SelectionKey.OP_READ);
            }
            else if (key.isWritable( )) {
              SocketChannel channel = (SocketChannel) key.channel( );
              ByteBuffer buffer = (ByteBuffer) key.attachment( );
              if (buffer.hasRemaining( )) {
                 channel.write(buffer);
              }
              else {  // we're done
                 channel.close( );
              }
            }
            else if (key.isReadable( )) {
              // Don't bother trying to parse the HTTP header.
              // Just read something.
              SocketChannel channel = (SocketChannel) key.channel( );
              ByteBuffer buffer = ByteBuffer.allocate(4096);
              channel.read(buffer);
              // switch channel to write-only mode
              key.interestOps(SelectionKey.OP_WRITE);
              key.attach(contentBuffer.duplicate( ));
            }
          }
          catch (IOException ex) {
            key.cancel( );
            try {
              key.channel( ).close( );
            }
            catch (IOException cex) {}
          }
        }
      }
    }

    public static void main(String[] args) {

      if (args.length == 0) {
        System.out.println(
          "Usage: java NonblockingSingleFileHTTPServer file port encoding");
        return;
      }

      try {
        String contentType = "text/plain";
        if (args[0].endsWith(".html") || args[0].endsWith(".htm")) {
          contentType = "text/html";
        }

        FileInputStream fin = new FileInputStream(args[0]);
        FileChannel in = fin.getChannel( );
        ByteBuffer input = in.map(FileChannel.MapMode.READ_ONLY, 0, in.size( ));

        // set the port to listen on
        int port;
        try {
          port = Integer.parseInt(args[1]);
          if (port < 1 || port > 65535) port = 80;
        }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
            catch (Exception ex) {
              port = 80;
            }

            String encoding = "ASCII";
            if (args.length > 2) encoding = args[2];

            NonblockingSingleFileHTTPServer server
             = new NonblockingSingleFileHTTPServer(
               input, encoding, contentType, port);
            server.run( );

          }
          catch (Exception ex) {
            ex.printStackTrace( );
            System.err.println(ex);
          }

        }

      }
```

The constructors set up the data to be sent along with an HTTP header that includes information about content length and content encoding. The header and the body of the response are stored in a single `ByteBuffer` so that they can be blasted to clients very quickly. However, although all clients receive the same content, they may not receive it at the same time. Different parallel clients will be at different locations in the file. This is why we duplicate the buffer, so each channel has its own buffer. The overhead is small because all channels do share the same content. They just have different indexes into that content.

All incoming connections are handled by a single `Selector` in the `run( )` method. The initial setup here is very similar to the earlier chargen server. The `run( )` method opens a `ServerSocketChannel` and binds it to the specified port. Then it creates the `Selector` and registers it with the `ServerSocketChannel`. When a `SocketChannel` is accepted, the same `Selector` object is registered with it. Initially it's registered for reading because the HTTP protocol requires the client to send a request before the server responds.

The response to a read is simplistic. The program reads as many bytes of input as it can up to 4K. Then it resets the interest operations for the channel to writability. (A more complete server would actually attempt to parse the HTTP header request here and choose the file to send based on that information.) Next, the content buffer is duplicated and attached to the channel.

The next time the program passes through the `while` loop, this channel should be ready to receive data (or if not the next time, the time after that; the asynchronous nature of the connection means we won't see it until it's ready). At this point, we get the buffer out of the attachment, and write as much of the buffer as we can onto the channel. It's no big deal if

we don't write it all this time. We'll just pick up where we left off the next pass through the loop. The buffer keeps track of its own position. Although many incoming clients might result in the creation of many buffer objects, the real overhead is minimal because they'll all share the same underlying data.

The `main( )` method just reads parameters from the command line. The name of the file to be served is read from the first command-line argument. If no file is specified or the file cannot be opened, an error message is printed and the program exits. Assuming the file can be read, its contents are mapped into the `ByteBuffer` array `input`. (To be perfectly honest, this is complete overkill for the small to medium size files you're most likely to be serving here, and probably would be slower than using an `InputStream` that reads into a byte array, but I wanted to show you file mapping at least once.) A reasonable guess is made about the content type of the file, and that guess is stored in the `contentType` variable. Next, the port number is read from the second command-line argument. If no port is specified, or if the second argument is not an integer from 0 to 65,535, port 80 is used. The encoding is read from the third command-line argument if present. Otherwise, ASCII is assumed. Then these values are used to construct a `NonblockingSingleFileHTTPServer` object and start it running.

## 12.3.8. Slicing Buffers

*Slicing* a buffer is a slight variant of duplicating. Slicing also creates a new buffer that shares the same data with the old buffer. However, the slice's initial position is the current position of the original buffer. That is, the slice is like a subsequence of the original buffer that only contains the elements from the current position to the limit. Rewinding the slice only moves it back to the position of the original buffer when the slice was created. The slice can't see anything in the original buffer before that point. Again, there are separate `slice( )` methods in each of the six typed buffer classes:

```
public abstract ByteBuffer   slice( )
public abstract IntBuffer    slice( )
public abstract ShortBuffer  slice( )
public abstract FloatBuffer  slice( )
public abstract CharBuffer   slice( )
public abstract DoubleBuffer slice( )
```

This is useful when you have a long buffer of data that is easily divided into multiple parts such as a protocol header followed by the data. You can read out the header then slice the buffer and pass the new buffer containing only the data to a separate method or class.


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

## 12.3.9. Marking and Resetting

Like input streams, buffers can be marked and reset if you want to reread some data. Unlike input streams, this can be done to all buffers, not just some of them. For a change, the relevant methods are declared once in the `Buffer` superclass and inherited by all the various subclasses:

```
public final Buffer mark( )
public final Buffer reset( )
```

The `reset( )` method throws an `InvalidMarkException`, a runtime exception, if the mark is not set.

The mark is unset if the limit is set to a point below the mark.

## 12.3.10. Object Methods

The buffer classes all provide the usual `equals( )`, `hashCode( )`, and `toString( )` methods. They also implement `Comparable`, and therefore provide `compareTo( )` methods. However, buffers are not `Serializable` or `Cloneable`.

Two buffers are considered to be equal if:

- They have the same type (e.g., a `ByteBuffer` is never equal to an `IntBuffer` but may be equal to another `ByteBuffer`).
- They have the same number of elements remaining in the buffer.
- The remaining elements at the same relative positions are equal to each other.

Note that equality does not consider the buffers' elements that precede the cursor, the buffers' capacity, limits, or marks. For example, this code fragment would print true even though the first buffer is twice the size of the second:

```
CharBuffer buffer1 = CharBuffer.wrap("12345678");
CharBuffer buffer2 = CharBuffer.wrap("5678");
buffer1.get( );
buffer1.get( );
buffer1.get( );
buffer1.get( );
System.out.println(buffer1.equals(buffer2));
```

The `hashCode( )` method is implemented in accordance with the contract for equality. That is, two equal buffers will have equal hash codes and two unequal buffers are very unlikely


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

to have equal hash codes. However, because the buffer's hash code changes every time an element is added to or removed from the buffer, buffers do not make good hash table keys.

Comparison is implemented by comparing the remaining elements in each buffer, one by one. If all the corresponding elements are equal, the buffers are equal. Otherwise, the result is the outcome of comparing the first pair of unequal elements. If one buffer runs out of elements before an unequal element is found and the other buffer still has elements, the shorter buffer is considered to be less than the longer buffer.

The `toString( )` method returns strings that look something like this:

```
java.nio.HeapByteBuffer[pos=0 lim=62 cap=62]
```

These are primarily useful for debugging. The notable exception is `CharBuffer`, which returns a string containing the remaining chars in the buffer.

# 12.4. Channels

Channels move blocks of data into and out of buffers to and from various I/O sources such as files, sockets, datagrams, and so forth. The channel class hierarchy is rather convoluted, with multiple interfaces and many optional operations. However, for purposes of network programming there are only three really important channel classes, `SocketChannel`, `ServerSocketChannel`, and `DatagramChannel`; and for the TCP connections we've talked about so far you only need the first two.

## 12.4.1. SocketChannel

The `SocketChannel` class reads from and writes to TCP sockets. The data must be encoded in `ByteBuffer` objects for reading and writing. Each `SocketChannel` is associated with a peer `Socket` object that can be used for advanced configuration, but this requirement can be ignored for applications where the default options are fine.

### 12.4.1.1. Connecting

The `SocketChannel` class does not have any public constructors. Instead, you create a new `SocketChannel` object using one of the two static `open( )` methods:


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public static SocketChannel open(SocketAddress remote) throws IOException
public static SocketChannel open( ) throws IOException
```

The first variant makes the connection. This method blocks; that is, the method will not return until the connection is made or an exception is thrown. For example:

```
SocketAddress address = new InetSocketAddress("www.cafeaulait.org",  80);
SocketChannel channel = SocketChannel.open(address);
```

The noargs version does not immediately connect. It creates an initially unconnected socket that must be connected later using the `connect( )` method. For example:

```
SocketChannel channel = SocketChannel.open( );
SocketAddress address = new InetSocketAddress("www.cafeaulait.org",  80);
channel.connect(address);
```

You might choose this more roundabout approach in order to configure various options on the channel and/or the socket before connecting. Specifically, use this approach if you want to open the channel without blocking:

```
SocketChannel channel = SocketChannel.open( );
SocketAddress address = new InetSocketAddress("www.cafeaulait.org",  80);
channel.configureBlocking(false);
channel.connect( );
```

With a non-blocking channel, the `connect( )` method returns immediately, even before the connection is established. The program can do other things while it waits for the operating system to finish the connection. However, before it can actually use the connection, the program must call `finishConnect()`:

```
public abstract boolean finishConnect( ) throws IOException
```

(This is only necessary in non-blocking mode. For a blocking channel this method returns true immediately.) If the connection is now ready for use, `finishConnect( )` returns `true`. If the connection has not been established yet, `finishConnect( )` returns `false`. Finally, if the connection could not be established, for instance because the network is down, this method throws an exception.

If the program wants to check whether the connection is complete, it can call these two methods:

```
public abstract boolean isConnected( )
public abstract boolean isConnectionPending( )
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The `isConnected( )` method returns `true` if the connection is open. The `isConnectionPending( )` method returns `true` if the connection is still being set up but is not yet open.

#### 12.4.1.2. Reading

To read from a `SocketChannel`, first create a `ByteBuffer` the channel can store data in. Then pass it to the `read( )` method:

```
public abstract int read(ByteBuffer dst) throws IOException
```

The channel fills the buffer with as much data as it can, then returns the number of bytes it put there. If it encounters the end of stream, it returns -1. If the channel is blocking, this method will read at least one byte or return -1 or throw an exception. If the channel is non-blocking, however, this method may return 0.

Because the data is stored into the buffer at the cursor position, which is updated automatically as more data is added, you can keep passing the same buffer to the `read( )` method until the buffer is filled. For example, this loop will read until the buffer is filled or the end of stream is detected:

```
while (buffer.hasRemaining( ) && channel.read(buffer) != -1) ;
```

It is sometimes useful to be able to fill several buffers from one source. This is called a *scatter*. These two methods accept an array of `ByteBuffer` objects as arguments and fill each one in turn:

```
public final long read(ByteBuffer[] dsts)throws IOException
public final long read(ByteBuffer[] dsts, int offset, int length)
                                                    throws IOException
```

To fill these, just loop while the last buffer in the list has space remaining. For example:

```
ByteBuffer[] buffers = new ByteBuffer[2];
buffers[0] = ByteBuffer.allocate(1000);
buffers[1] = ByteBuffer.allocate(1000);
while (buffers[1].hasRemaining( ) && channel.read(buffers) != -1) ;
```

#### 12.4.1.3. Writing

Socket channels have both read and write methods. In general, they are full duplex. In order to write, simply fill a `ByteBuffer`, flip it, and pass it to one of the write methods, which


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

drains it while copying the data onto the output—pretty much the reverse of the reading process.

The basic `write( )` method takes a single buffer as an argument:

```
public abstract int write(ByteBuffer src) throws IOException
```

As with reads (and unlike `OutputStream`s), this method is not guaranteed to write the complete contents of the buffer if the channel is non-blocking. Again, however, the cursor-based nature of buffers enables you to easily call this method again and again until the buffer is fully drained and the data has been completely written:

```
while (buffer.hasRemaining( ) && channel.write(buffer) != -1) ;
```

It is often useful to be able to write data from several buffers onto one socket. This is called a *gather*. For example, you might want to store the HTTP header in one buffer and the HTTP body in another buffer. The implementation might even fill the two buffers simultaneously using two threads or overlapped I/O. These two methods accept an array of `ByteBuffer` objects as arguments, and drain each one in turn:

```
public final long write(ByteBuffer[] dsts)throws IOException
public final long write(ByteBuffer[] dsts, int offset, int length)
                                                    throws IOException
```

The first variant drains all the buffers. The second method drains `length` buffers, starting with the one at `offset`.

#### 12.4.1.4. Closing

Just as with regular sockets, you should close a channel when you're done with it to free up the port and any other resources it may be using:

```
public void close( ) throws IOException
```

Closing an already closed channel has no effect. Attempting to write data to or read data from a closed channel throws an exception. If you're uncertain whether a channel has been closed, check with `isOpen( )`:

```
public boolean isOpen( )
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Naturally, this returns `false` if the channel is closed, `true` if it's open. (`close( )` and `isOpen( )` are the only two methods declared in the `Channel` interface and shared by all channel classes.)

## 12.4.2. ServerSocketChannel

The `ServerSocketChannel` class has one purpose: to accept incoming connections. You cannot read from, write to, or connect a `ServerSocketChannel`. The only operation it supports is accepting a new incoming connection. The class itself only declares four methods, of which `accept()` is the most important. `ServerSocketChannel` also inherits several methods from its superclasses, mostly related to registering with a `Selector` for notification of incoming connections. And finally, like all channels it has a `close( )` method that's used to shut down the server socket.

### 12.4.2.1. Creating server socket channels

The static factory method `ServerSocketChannel.open()` creates a new `ServerSocketChannel` object. However, the name is a little deceptive. This method does not actually open a new server socket. Instead, it just creates the object. Before you can use it, you need to use the `socket( )` method to get the corresponding peer `ServerSocket`. At this point, you can configure any server options you like, such as the receive buffer size or the socket timeout, using the various setter methods in `ServerSocket`. Then connect this `ServerSocket` to a `SocketAddress` for the port you want to bind to. For example, this code fragment opens a `ServerSocketChannel` on port 80:

```
try {
  ServerSocketChannel server= ServerSocketChannel.open( );
  ServerSocket socket = serverChannel.socket( );
  SocketAddress address = new InetSocketAddress(80);
  socket.bind(address);
}
catch (IOException ex) {
  System.err.println("Could not bind to port 80 because " + ex.getMessage( ));
}
```

A factory method is used here rather than a constructor so that different virtual machines can provide different implementations of this class, more closely tuned to the local hardware and OS. However, this factory is not user-configurable. The `open( )` method always returns an instance of the same class when running in the same virtual machine.

#### 12.4.2.2. Accepting connections

Once you've opened and bound a `ServerSocketChannel` object, the `accept()` method can listen for incoming connections:

```
public abstract SocketChannel accept( ) throws IOException
```

`accept( )` can operate in either blocking or non-blocking mode. In blocking mode, the `accept( )` method waits for an incoming connection. It then accepts that connection and returns a `SocketChannel` object connected to the remote client. The thread cannot do anything until a connection is made. This strategy might be appropriate for simple servers that can respond to each request immediately. Blocking mode is the default.

A `ServerSocketChannel` can also operate in non-blocking mode. In this case, the `accept( )` method returns null if there are no incoming connections. Non-blocking mode is more appropriate for servers that need to do a lot of work for each connection and thus may want to process multiple requests in parallel. Non-blocking mode is normally used in conjunction with a `Selector`. To make a `ServerSocketChannel` non-blocking, pass `false` to its `configureBlocking()` method.

The `accept( )` method is declared to throw an `IOException` if anything goes wrong. There are several subclasses of `IOException` that indicate more detailed problems, as well as a couple of runtime exceptions:

`ClosedChannelException`
: You cannot reopen a `ServerSocketChannel` after closing it.

`AsynchronousCloseException`
: Another thread closed this `ServerSocketChannel` while `accept( )` was executing.

`ClosedByInterruptException`
: Another thread interrupted this thread while a blocking `ServerSocketChannel` was waiting.

`NotYetBoundException`
: You called `open( )` but did not bind the `ServerSocketChannel`'s peer `ServerSocket` to an address before calling `accept( )`. This is a runtime exception, not an `IOException`.

`SecurityException`
: The security manager refused to allow this application to bind to the requested port.

### 12.4.3. The Channels Class

`Channels` is a simple utility class for wrapping channels around traditional I/O-based streams, readers, and writers, and vice versa. It's useful when you want to use the new I/O model in one part of a program for performance, but still interoperate with legacy APIs that


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

expect streams. It has methods that convert from streams to channels and methods that convert from channels to streams, readers, and writers:

```
public static InputStream newInputStream(ReadableByteChannel ch)
public static OutputStream newOutputStream(WritableByteChannel ch)
public static ReadableByteChannel newChannel(InputStream in)
public static WritableByteChannel newChannel(OutputStream out)
public static Reader newReader (ReadableByteChannel channel,
  CharsetDecoder dec, int minBufferCap)
public static Reader newReader (ReadableByteChannel ch, String encoding)
public static Writer newWriter (WritableByteChannel ch, String encoding)
```

The `SocketChannel` class discussed in this chapter implements both the `ReadableByteChannel` and `WritableByteChannel` interfaces seen in these signatures. `ServerSocketChannel` implements neither of these because you can't read from or write to it.

For example, all current XML APIs use streams, files, readers, and other traditional I/O APIs to read the XML document. If you're writing an HTTP server designed to process SOAP requests, you may want to read the HTTP request bodies using channels and parse the XML using SAX for performance. In this case, you'd need to convert these channels into streams before passing them to `XMLReader`'s `parse()` method:

```
SocketChannel channel = server.accept( );
processHTTPHeader(channel);
XMLReader parser = XMLReaderFactory.createXMLReader( );
parser.setContentHandler(someContentHandlerObject);
InputStream in = Channels.newInputStream(channel);
parser.parse(in);
```

## 12.5. Readiness Selection

For network programming, the second part of the new I/O APIs is readiness selection, the ability to choose a socket that will not block when read or written. This is primarily of interest to servers, although clients running multiple simultaneous connections with several windows open—such as a web spider or a browser—can take advantage of it as well.

In order to perform readiness selection, different channels are registered with a `Selector` object. Each channel is assigned a `SelectionKey`. The program can then ask the `Selector` object for the set of keys to the channels that are ready to perform the operation you want to perform without blocking.

## 12.5.1. The Selector Class

The only constructor in `Selector` is protected. Normally, a new selector is created by invoking the static factory method `Selector.open( )`:

```
public static Selector open( ) throws IOException
```

The next step is to add channels to the selector. There are no methods in the `Selector` class to add a channel. The `register( )` method is declared in the `SelectableChannel` class. Not all channels are selectable—in particular, `FileChannel`s aren't selectable—but all network channels are. Thus, the channel is registered with a selector by passing the selector to one of the channel's register methods:

```
public final SelectionKey register(Selector sel, int ops)
  throws ClosedChannelException
public final SelectionKey register(Selector sel, int ops, Object att)
  throws ClosedChannelException
```

This approach feels backwards to me, but it's not hard to use. The first argument is the selector the channel is registering with. The second argument is a named constant from the `SelectionKey` class identifying the operation the channel is registering for. The `SelectionKey` class defines four named bit constants used to select the type of the operation:

- `SelectionKey.OP_ACCEPT`
- `SelectionKey.OP_CONNECT`
- `SelectionKey.OP_READ`
- `SelectionKey.OP_WRITE`

These are bit-flag int constants (1, 2, 4, etc.). Therefore, if a channel needs to register for multiple operations in the same selector (e.g., for both reading and writing on a socket), combine the constants with the bitwise or operator (`|`) when registering:

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

The optional third argument is an attachment for the key. This object is often used to store state for the connection. For example, if you were implementing a web server, you might attach a `FileInputStream` or `FileChannel` connected to the local file the server streams to the client.

After the different channels have been registered with the selector, you can query the selector at any time to find out which channels are ready to be processed. Channels may be ready for


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

some operations and not others. For instance, a channel could be ready for reading but not writing.

There are three methods that select the ready channels. They differ in how long they wait to find a ready channel. The first, `selectNow( )`, performs a non-blocking select. It returns immediately if no connections are ready to be processed now:

```
public abstract int selectNow( ) throws IOException
```

The other two select methods are blocking:

```
public abstract int select( ) throws IOException
public abstract int select(long timeout) throws IOException
```

The first method waits until at least one registered channel is ready to be processed before returning. The second waits no longer than `timeout` milliseconds for a channel to be ready before returning 0. These methods are useful if your program doesn't have anything to do when no channels are ready to be processed.

When you know the channels are ready to be processed, retrieve the ready channels using `selectedKeys()`:

```
public abstract Set selectedKeys( )
```

The return value is just a standard `java.util.Set`. Each item in the set is a `SelectionKey` object. You can iterate through it in the usual way, casting each item to `SelectionKey` in turn. You'll also want to remove the key from the iterator to tell the `Selector` that you've handled it. Otherwise, the `Selector` will keep telling you about it on future passes through the loop.

Finally, when you're ready to shut the server down or when you no longer need the `Selector`, you should close it:

```
public abstract void close( ) throws IOException
```

This step releases any resources associated with the selector. More importantly, it cancels all keys registered with the selector and interrupts up any threads blocked by one of this selector's select methods.

## 12.5.2. The SelectionKey Class

`SelectionKey` objects serve as pointers to channels. They can also hold an object attachment, which is how you normally store the state for the connection on that channel.

`SelectionKey` objects are returned by the `register( )` method when registering a channel with a `Selector`. However, you don't usually need to retain this reference. The `selectedKeys( )` method returns the same objects again inside a `Set`. A single channel can be registered with multiple selectors.

When retrieving a `SelectionKey` from the set of selected keys, you often first test what that key is ready to do. There are four possibilities:

```
public final boolean isAcceptable( )
public final boolean isConnectable( )
public final boolean isReadable( )
public final boolean isWritable( )
```

This test isn't always necessary. In some cases, the `Selector` is only testing for one possibility and will only return keys to do that one thing. But if the `Selector` does test for multiple readiness states, you'll want to test which one kicked the channel into the ready state before operating on it. It's also possible that a channel is ready to do more than one thing.

Once you know what the channel associated with the key is ready to do, retrieve the channel with the `channel( )` method:

```
public abstract SelectableChannel channel( )
```

If you've stored an object in the `SelectionKey` to hold state information, you can retrieve it with the `attachment( )` method:

```
public final Object attachment( )
```

Finally, when you're finished with a connection, deregister its `SelectionKey` object so the `Selector` doesn't waste any resources querying it for readiness. I don't know that this is absolutely essential in all cases, but it doesn't hurt. You do this by invoking the key's `cancel()` method:

```
public abstract void cancel( )
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

However, this step is only necessary if you haven't closed the channel. Closing a channel automatically deregisters all keys for that channel in all selectors. Similarly, closing a selector invalidates all keys in that selector.