Getting started with new I/O (NIO)

Greg Travis (mito@panix.com)
Java programmer and technology writer

09 July 2003

The new input/output (NIO) library, introduced with JDK 1.4, provides high-speed, block-oriented I/O in standard Java code. This hands-on tutorial covers the NIO library in great detail, from the high-level concepts to under-the-hood programming detail. You'll learn about crucial I/O elements like buffers and channels, and examine how standard I/O works in the updated library. You'll also learn about things you can do only with NIO, such as asynchronous I/O and direct buffers.

Before you start

About this tutorial

The new input/output (NIO) library was introduced with JDK 1.4. Picking up where original I/O leaves off, NIO provides high-speed, block-oriented I/O in standard Java code. By defining classes to hold data, and by processing that data in blocks, NIO takes advantage of low-level optimizations in a way that the original I/O package could not, without using native code.

In this tutorial, we'll cover almost every aspect of the NIO library, from the high-level conceptual stuff to under-the-hood programming detail. In addition to learning about crucial I/O elements like buffers and channels, you'll have the opportunity to see how standard I/O works in the updated library. You'll also learn about things you can *only* do with NIO, such as asynchronous I/O and direct buffers.

Throughout the tutorial, we'll work with code samples that illustrate different aspects of the NIO library. Almost every code sample is part of an extended Java program, which you'll find in Resources. As you are working through the exercises, you're encouraged to download, compile, and run these programs on your own system. The code will also come in handy when you're done with the tutorial, providing a starting point for your NIO programming efforts.

This tutorial is intended for any programmer who wants to learn more about the JDK 1.4 NIO library. To get the most from the discussion you should understand basic Java programming concepts such as classes, inheritance, and using packages. Some familiarity with the original I/O library (from the `java.io.*` package) will also be helpful.

While this tutorial does require a working vocabulary and conceptual understanding of the Java language, it does not require a lot of actual programming experience. In addition to explaining thoroughly all the concepts relevant to the tutorial, I've kept the code examples fairly small and simple. The goal is to provide an easy entry point for learning about NIO, even for those who don't have much Java programming experience.

How to run the code

The source code archive (available in Resources) contains all of the programs used in this tutorial. Each program consists of a single Java file. Each file is identified by name and easily related to the programming concept it illustrates.

Some of the programs in the tutorial require command-line arguments to run. To run a program from the command line, simply go to your nearest command-line prompt. Under Windows, the command-line prompt is the "Command" or "command.com" program. Under UNIX, any shell will do.

You will need to have JDK 1.4 installed and in your path to complete the exercises in the tutorial. See Resources if you need help installing and configuring JDK 1.4.

Input/output: A conceptual overview

Intro to I/O

I/O -- or input/output -- refers to the interface between a computer and the rest of the world, or between a single program and the rest of the computer. It is such a crucial element of any computer system that the bulk of any I/O is actually built into the operating system. Individual programs generally have most of their work done for them.

In Java programming, I/O has until recently been carried out using a *stream* metaphor. All I/O is viewed as the movement of single bytes, one at a time, through an object called a `Stream`. Stream I/O is used for contacting the outside world. It is also used internally, for turning objects into bytes and then back into objects.

NIO has the same role and purpose as original I/O, but it uses a different metaphor -- *block I/O*. As you will learn in this tutorial, block I/O can be a lot more efficient than stream I/O.

Why NIO?

NIO was created to allow Java programmers to implement high-speed I/O without having to write custom native code. NIO moves the most time-consuming I/O activities (namely, filling and draining buffers) back into the operating system, thus allowing for a great increase in speed.

Streams versus blocks

The most important distinction between the original I/O library (found in `java.io.*`) and NIO has to do with how data is packaged and transmitted. As previously mentioned, original I/O deals with data in streams, whereas NIO deals with data in blocks.

A *stream-oriented* I/O system deals with data one byte at a time. An input stream produces one byte of data, and an output stream consumes one byte of data. It is very easy to create filters for streamed data. It is also relatively simply to chain several filters together so that each one does its part in what amounts to a single, sophisticated processing mechanism. On the flip side, stream-oriented I/O is often rather slow.

A *block-oriented* I/O system deals with data in blocks. Each operation produces or consumes a block of data in one step. Processing data by the block can be much faster than processing it by the (streamed) byte. But block-oriented I/O lacks some of the elegance and simplicity of stream-oriented I/O.

Integrated I/O

The original I/O package and NIO have been well integrated in JDK 1.4. `java.io.*` has been reimplemented using NIO as its base, so it can now take advantage of some features of NIO. For example, some of the classes in the `java.io.*` package contain methods to read and write data in blocks, which leads to faster processing even in more stream-oriented systems.

It is also possible to use the NIO library to implement standard I/O functions. For example, you could easily use block I/O to move data one byte at a time. But as you will see, NIO also offers many advantages that are not available from the original I/O package.

Channels and buffers

Channels and buffers overview

`Channel` s and `Buffer` s are the central objects in NIO, and are used for just about every I/O operation.

Channels are analogous to streams in the original I/O package. All data that goes anywhere (or comes from anywhere) must pass through a `Channel` object. A `Buffer` is essentially a container object. All data that is sent to a channel must first be placed in a buffer; likewise, any data that is read from a channel is read into a buffer.

In this section, you will learn about working with channels and buffers in NIO.

What is a buffer?

A `Buffer` is an object, which holds some data, that is to be written to or that has just been read from. The addition of the `Buffer` object in NIO marks one of the most significant differences between the new library and original I/O. In stream-oriented I/O, you wrote data directly to, and read data directly from, `Stream` objects.

In the NIO library, all data is handled with buffers. When data is read, it is read directly into a buffer. When data is written, it is written into a buffer. Anytime you access data in NIO, you are pulling it out of the buffer.

A buffer is essentially an array. Generally, it is an array of bytes, but other kinds of arrays can be used. But a buffer is more than *just* an array. A buffer provides structured access to data and also keeps track of the system's read/write processes.

Kinds of buffers

The most commonly used kind of buffer is the `ByteBuffer`. A `ByteBuffer` allows get/set operations (that is, the getting and setting of bytes) on its underlying byte array.

`ByteBuffer` is not the only type of buffer in NIO. In fact, there is a buffer type for each of the primitive Java types:

- `ByteBuffer`
- `CharBuffer`
- `ShortBuffer`
- `IntBuffer`
- `LongBuffer`
- `FloatBuffer`
- `DoubleBuffer`

Each of the `Buffer` classes is an instance of the `Buffer` interface. With the exception of `ByteBuffer`, each one has the exact same operations, differing only in the type of data it deals with. Because `ByteBuffer` is used for most standard I/O operations it has all of the shared buffer operations as well as some that are unique.

You may want to take a moment now to run the UseFloatBuffer.java, which contains an example of typed buffers in action.

What is a channel?

A `Channel` is an object from which you can read data and to which you can write data. Comparing NIO with original I/O, a channel is like a stream.

As previously mentioned, all data is handled through `Buffer` objects. You never write a byte directly to a channel; instead you write to a buffer containing one or more bytes. Likewise, you don't read a byte directly from a channel; you read from a channel into a buffer, and then get the bytes from the buffer.

Kinds of channels

Channels differ from streams in that they are bi-directional. Whereas streams only go in one direction (a stream must be a subclass of either `InputStream` or `OutputStream`), a `Channel` can be opened for reading, for writing, or for both.

Because they are bi-directional, channels better reflect the reality of the underlying operating system than streams do. In the UNIX model in particular, the underlying operating system channels are bi-directional.

From theory to practice: Reading and writing in NIO

NIO overview

Reading and writing are the fundamental processes of I/O. Reading from a channel is simple: we simply create a buffer and then ask a channel to read data into it. Writing is also fairly simply: we create a buffer, fill it with data, and then ask a channel to write from it.

In this section, we'll learn a little bit about reading and writing data in Java programs. We'll go over the main components of NIO (buffers, channels, and some related methods) and see how they interact for reading and writing. In the sections that follow, we will look at each of these components and interactions in more detail.

Reading from a file

For our first exercise, we'll read some data from a file. If we were using original I/O, we would simply create a `FileInputStream` and read from that. In NIO, however, things work a little differently: we first get a `Channel` object from the `FileInputStream`, and then use that channel to read the data.

Any time you perform a read operation in an NIO system, you are reading from a channel, but you don't read *directly* from a channel. Since all data ultimately resides in the buffer, you read from a channel into a buffer.

So reading from a file involves three steps: (1) getting the `Channel` from `FileInputStream`; (2) creating the `Buffer`; and (3) reading from the `Channel` into the `Buffer`.

Now, let's see how this works.

Three easy steps

Our first step is to get a channel. We get the channel from the `FileInputStream`:

```
FileInputStream fin = new FileInputStream( "readandshow.txt" );
FileChannel fc = fin.getChannel();
```

The next step is to create a buffer:

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

And, finally, we need to read from the channel into the buffer, as shown here:

```
fc.read( buffer );
```

You'll notice that we didn't need to tell the channel *how much* to read into the buffer. Each buffer has a sophisticated internal accounting system that keeps track of how much data has been read and how much room there is for more data. We'll talk more about the buffer accounting system in the Buffer internals overview.

Writing to a file

Writing to a file in NIO is similar to reading from one. We start by getting a channel from a `FileOutputStream`:

```
FileOutputStream fout = new FileOutputStream( "writesomebytes.txt" );
FileChannel fc = fout.getChannel();
```

Our next step is to create a buffer and put some data in it -- in this case, the data will be taken from an array called `message` which contains the ASCII bytes for the string "Some bytes." (The `buffer.flip()` and `buffer.put()` calls will be explained later in the tutorial.)

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

for (int i=0; i<message.length; ++i) {
     buffer.put( message[i] );
}
buffer.flip();
```

Our final step is to write to the buffer:

```
fc.write( buffer );
```

Notice that once again we did not need to tell the channel how much data we wanted to write. The buffer's internal accounting system keeps track of how much data it contains and how much is left to be written.

Reading and writing together

Next we'll see what happens when we combine reading and writing. We'll base this exercise on a simple program called CopyFile.java, which copies all the data from one file to another one. CopyFile.java carries out three basic operations: it first creates a `Buffer`, then reads data from the source file into this buffer, and then writes the buffer to the destination file. The program repeats -- read, write, read, write -- until the source file is exhausted.

The CopyFile program will let you see how we check the status of an operation, as well as how we use the `clear()` and `flip()` methods to reset the buffer and prepare it to have newly read data written to another channel.

Running the CopyFile example

Because the buffer tracks its own data, the inner loop of the CopyFile program is very simple, as shown below:

```
fcin.read( buffer );
fcout.write( buffer );
```

The first line reads data into the buffer from the input channel, `fcin`, and the second line writes the data to the output channel, `fcout`.

Checking the status

Our next step is to check to see when we're done copying. We're done when there's no more data, and we can tell this when the `read()` method returns `-1`, as shown below:

```
int r = fcin.read( buffer );

if (r==-1) {
     break;
}
```

Resetting the buffer

And, finally, we call the `clear()` method before we read into a buffer from the input channel. Likewise, we call the `flip()` method before we write a buffer to the output channel, as shown below:

```
buffer.clear();
int r = fcin.read( buffer );

if (r==-1) {
     break;
}

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

The `clear()` method resets the buffer, making it ready to have data read into it. The `flip()` method prepares the buffer to have the newly-read data written to another channel.

Buffer internals

Buffer internals overview

In this section, we'll look at two important components of buffers in NIO: state variables and accessor methods.

State variables are key to the "internal accounting system" mentioned in the previous section. With each read/write operation, the buffer's state changes. By recording and tracking those changes, a buffer is able to internally manage its own resources.

When you read data from a channel, the data is placed in a buffer. In some cases, you can write this buffer directly to another channel, but often, you'll want to look at the data itself. This is accomplished using the *accessor method* `get()`. Likewise, when you want to put raw data in a buffer, you use the accessor method `put()`.

In this section, you'll learn about state variables and accessor methods in NIO. Each component will be described, and then you'll have the opportunity to see it in action. While NIO's internal accounting system might seem complicated at first, you'll quickly see that most of the real work is done for you. The bookkeeping you're probably accustomed to coding by hand -- using byte arrays and index variables -- is handled internally in NIO.

State variables

Three values can be used to specify the state of a buffer at any given moment in time:

- `position`
- `limit`
- `capacity`

Together, these three variables track the state of the buffer and the data it contains. We'll examine each one in detail, and also see how they fit into a typical read/write (input/output) process. For the sake of the example, we'll assume that we are copying data from an input channel to an output channel.

Position

You will recall that a buffer is really just a glorified array. When you read from a channel, you put the data that you read into an underlying array. The `position` variable keeps track of how much data you have written. More precisely, it specifies into which array element the next byte will go. Thus, if you've read three bytes from a channel into a buffer, that buffer's `position` will be set to 3, referring to the fourth element of the array.

Likewise, when you are writing to a channel, you get the data from a buffer. The `position` value keeps track of how much you have gotten from the buffer. More precisely, it specifies from which array element the next byte will come. Thus, if you've written 5 bytes to a channel from a buffer, that buffer's `position` will be set to 5, referring to the sixth element of the array.

Limit

The `limit` variable specifies how much data there is left to get (in the case of writing from a buffer into a channel), or how much room there is left to put data into (in the case of reading from a channel into a buffer).

The `position` is always less than, or equal to, the `limit`.

Capacity

The `capacity` of a buffer specifies the maximum amount of data that can be stored therein. In effect, it specifies the size of the underlying array -- or, at least, the amount of the underlying array that we are permitted to use.

The `limit` can never be larger than the `capacity`.

Observing the variables

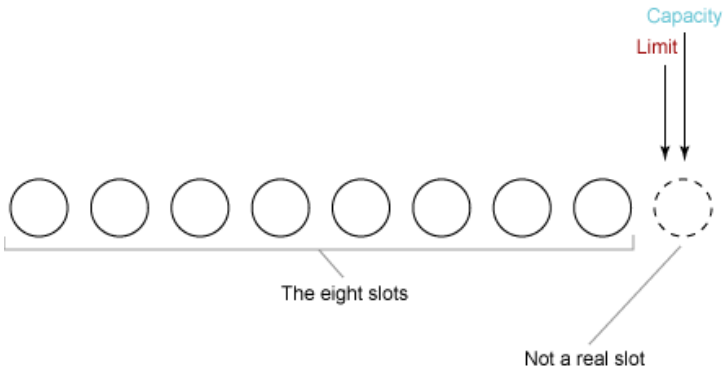
We'll start with a newly created buffer. For the sake of the example, let's assume that our buffer has a total `capacity` of eight bytes. The `Buffer` 's state is shown here:

React operations

The eight slots

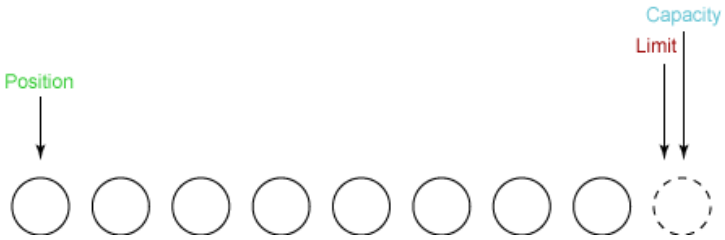
Recall that the `limit` can never be larger than the `capacity`, and in this case both values are set to 8. We show this by pointing them off the end of the array (which is where slot 8 would be if there were a slot 8):

React operations



The `position` is set to 0. If we read some data into the buffer, the next byte read will go into slot 0. If we write from the buffer, the next byte taken from the buffer will be taken from slot 0. The `position` setting is shown here:

React operations

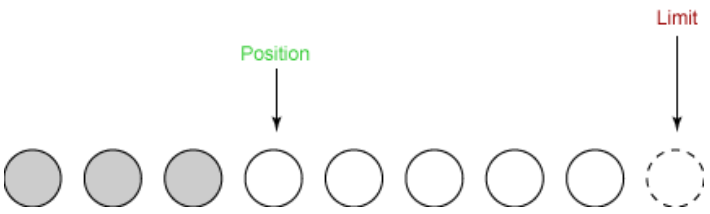


Because the `capacity` is not going to change, we can omit it from the discussion that follows.

The first read

Now we are ready to begin read/write operations on our newly created buffer. We start by reading some data from our input channel into the buffer. The first read gets three bytes. These are put into the array starting at the `position`, which was set to 0. After this read, the position is increased to 3, as shown here:

React operations

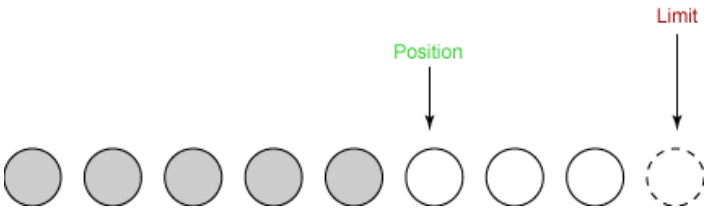


The `limit` is unchanged.

The second read

For our second read, we read two more bytes from the input channel into our buffer. The two bytes are stored at the location pointed to by `position`; `position` is thus increased by two:

React operations



The `limit` is unchanged.

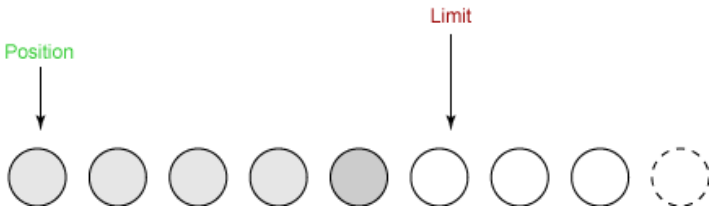
The flip

Now we are ready to write our data to an output channel. Before we can do this, we must call the `flip()` method. This method does two crucial things:

1. It sets the `limit` to the current `position`.
2. It sets the `position` to 0.

The figure on the section shows our buffer before the flip. Here is the buffer after the flip:

React operations

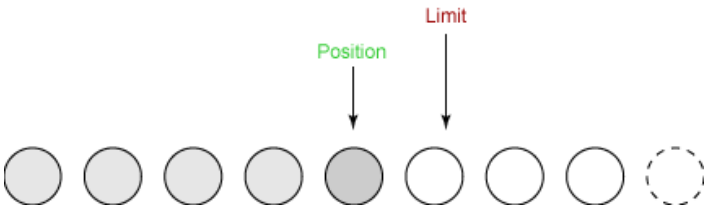


We are now ready to begin writing data to a channel from the buffer. The `position` has been set to 0, which means the next byte we get will be the first one. And the `limit` has been set to the old `position`, which means that it just includes all the bytes we read before, and no more.

The first write

In our first write, we take four bytes from the buffer and write them to our output channel. This advances the `position` to 4, and leaves the `limit` unchanged, as shown here:

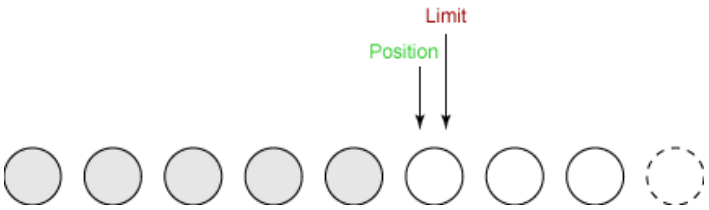
React operations



The second write

We only have one byte left to write. The `limit` was set to 5 when we did our `flip()`, and the `position` cannot go past the `limit`. So the last write takes one byte from our buffer and writes it to the output channel. This advances the `position` to 5, and leaves the `limit` unchanged, as shown here:

React operations



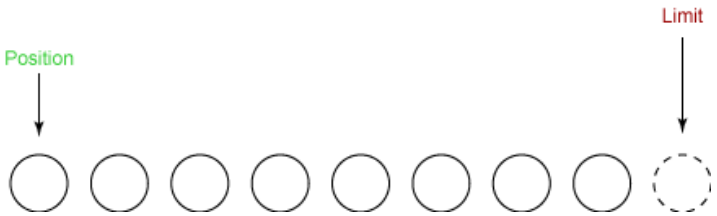
The clear

Our final step is to call the buffer's `clear()` method. This method resets the buffer in preparation for receiving more bytes. `Clear` does two crucial things:

1. It sets the `limit` to match the `capacity`.
2. It sets the `position` to 0.

This figure shows the state of the buffer after `clear()` has been called:

React operations



The buffer is now ready to receive fresh data.

Accessor methods

So far, we've only used buffers to move data from one channel to another. Frequently, however, your program will need to deal directly with the data. For example, you might want to save user data to disk. In this case, you'll have to put that data directly into a buffer, and then write the buffer to disk using a channel.

Or, you might want to read user data back in from disk. In this case, you would read the data into a buffer from a channel, and then examine the data in the buffer.

We'll close this section with a detailed look at accessing data directly in the buffer, using the `get()` and `put()` methods for the `ByteBuffer` class.

The get() methods

In the `ByteBuffer` class, there are four `get()` methods:

1. `byte get();`
2. `ByteBuffer get( byte dst[] );`
3. `ByteBuffer get( byte dst[], int offset, int length );`
4. `byte get( int index );`

The first method gets a single byte. The second and third methods read a group of bytes into an array. The fourth method gets the byte from a particular position in the buffer. The methods that return a `ByteBuffer` simply return the `this` value on which they are called.

Additionally, we say that the first three `get()` methods are relative, while the last one is absolute. *Relative* means that the `limit` and `position` values are respected by the `get()` operation -- specifically, the byte is read from the current `position`, and the `position` is incremented after the `get`. An *absolute* method, on the other hand, ignores the `limit` and `position` values, and does not affect them. In effect, it bypasses the buffer's accounting methods entirely.

The methods shown above correspond to the `ByteBuffer` class. The other classes have equivalent `get()` methods that are identical except that rather than dealing with bytes, they deal with the type appropriate for that buffer class.

The put() methods

In the `ByteBuffer` class, there are five `put()` methods:

1. `ByteBuffer put( byte b );`
2. `ByteBuffer put( byte src[] );`
3. `ByteBuffer put( byte src[], int offset, int length );`
4. `ByteBuffer put( ByteBuffer src );`
5. `ByteBuffer put( int index, byte b );`

The first method `put` s a single byte. The second and third methods write a group of bytes from an array. The fourth method copies data from the given source `ByteBuffer` into this `ByteBuffer`. The fifth method puts the byte into the buffer at a particular `position`. The methods that return a `ByteBuffer` simply return the `this` value on which they are called.

As with the `get()` methods, we characterize the `put()` methods as being *relative* or *absolute*. The first four methods are relative, while the fifth one is absolute.

The methods shown above correspond to the `ByteBuffer` class. The other classes have equivalent `put()` methods that are identical except that rather than dealing with bytes, they deal with the type appropriate for that buffer class.

Typed get() and put() methods

In addition to the `get()` and `put()` methods described previously, `ByteBuffer` also has extra methods for reading and writing values of different types, as follows:

- `getByte()`
- `getChar()`
- `getShort()`

- `getInt()`
- `getLong()`
- `getFloat()`
- `getDouble()`
- `putByte()`
- `putChar()`
- `putShort()`
- `putInt()`
- `putLong()`
- `putFloat()`
- `putDouble()`

Each of these methods, in fact, comes in two varieties -- one relative and one absolute. They are useful for reading formatted binary data, such as the header of an image file.

You can see these methods in action in the example program TypesInByteBuffer.java.

The buffer at work: An inner loop

The following inner loop summarizes the process of using a buffer to copy data from an input channel to an output channel.

```
while (true) {
     buffer.clear();
     int r = fcin.read( buffer );

     if (r==-1) {
       break;
     }

     buffer.flip();
     fcout.write( buffer );
}
```

The `read()` and `write()` calls are greatly simplified because the buffer takes care of many of the details. The `clear()` and `flip()` methods are used to switch the buffer between reading and writing.

More about buffers

Buffers overview

Thus far, you have learned most of what you need to know about buffers to use them on a day-to-day basis. Our examples haven't strayed much beyond the kind of standard read/write procedures you could just as easily implement in original I/O as in NIO.

In this section, we'll get into some of the more complex aspects of working with buffers, such as buffer allocation, wrapping, and slicing. We'll also talk about some of the new features NIO brings to the Java platform. You'll learn how to create different types of buffers to meet different goals, such as *read-only* buffers, which protect data from modification, and *direct* buffers, which map directly onto the underlying OS buffers. We'll close the section with an introduction to creating memory-mapped files in NIO.

Buffer allocation and wrapping

Before you can read or write, you must have a buffer. To create a buffer, you must *allocate* it. We allocate a buffer using the static method of `allocate()`:

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

The `allocate()` method allocates an underlying array of the specified size and wraps it in a buffer object -- in this case a `ByteBuffer`.

You can also turn an existing array into a buffer, as shown here:

```
byte array[] = new byte[1024];
ByteBuffer buffer = ByteBuffer.wrap( array );
```

In this case, you've used the `wrap()` method to wrap a buffer around an array. You must be very careful about performing this type of operation. Once you've done it, the underlying data can be accessed through the buffer as well as directly.

Buffer slicing

The `slice()` method creates a kind of *sub-buffer* from an existing buffer. That is, it creates a new buffer that shares its data with a portion of the original buffer.

This is best explained with an example. Let's start by creating a `ByteBuffer` of length 10:

```
ByteBuffer buffer = ByteBuffer.allocate( 10 );
```

We fill this buffer with data, putting the number *n* in slot *n*:

```
for (int i=0; i<buffer.capacity(); ++i) {
     buffer.put( (byte)i );
}
```

Now we'll *slice* the buffer to create a sub-buffer that covers slots 3 through 6. In a sense, the sub-buffer is like a *window* onto the original buffer.

You specify the start and end of the window by setting the `position` and `limit` values, and then call the `Buffer's slice()` method:

```
buffer.position( 3 );
buffer.limit( 7 );
ByteBuffer slice = buffer.slice();
```

`slice` is a sub-buffer of `buffer`. However, `slice` and `buffer` share the same underlying data array, as we'll see in the next section.

Buffer slicing and data sharing

We've created a sub-buffer of our original buffer, and we know that the two buffers and the sub-buffers share the same underlying data array. Let's see what this means.

We run through the sub-buffer, and alter each element by multiplying it by 11. This changes, for example, a 5 into a 55.

```
for (int i=0; i<slice.capacity(); ++i) {
     byte b = slice.get( i );
     b *= 11;
     slice.put( i, b );
}
```

Finally, let's take a look at the contents of the original buffer:

```
buffer.position( 0 );
buffer.limit( buffer.capacity() );

while (buffer.remaining()> 0) {
     System.out.println( buffer.get() );
}
```

The result shows that only the elements in the window of the sub-buffer were changed:

```
$ java SliceBuffer
0
1
2
33
44
55
66
7
8
9
```

Slice buffers are excellent for facilitating abstraction. You can write your functions to process an entire buffer, and if you find you want to apply that process to a sub-buffer, you can just take a slice of the main buffer and pass that to your function. This is easier than writing your functions to take additional parameters specifying what portion of the buffer should be acted upon.

Read-only buffers

Read-only buffers are very simple -- you can read them, but you can't write to them. You can turn any regular buffer into a read-only buffer by calling its `asReadOnlyBuffer()` method, which returns a new buffer that is identical to the first (and shares data with it), but is read-only.

Read-only buffers are useful for protecting data. When you pass a buffer to a method of some object, you really have no way of knowing if that method is going to try to modify the data in the buffer. Creating a read-only buffer *guarantees* that the buffer won't be modified.

You cannot convert a read-only buffer to a writable buffer.

Direct and indirect buffers

Another useful kind of `ByteBuffer` is the direct buffer. A *direct buffer* is one whose memory is allocated in a special way to increase I/O speed.

Actually, the exact definition of a direct buffer is implementation-dependent. Sun's documentation has this to say about direct buffers:

Given a direct byte buffer, the Java virtual machine will make a best effort to perform native I/O operations directly upon it. That is, it will attempt to avoid copying the buffer's content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system's native I/O operations.

You can see direct buffers in action in the example program FastCopyFile.java, which is a version of CopyFile.java that uses direct buffers for increased speed.

You can also create a direct buffer using memory-mapped files.

Memory-mapped file I/O

Memory-mapped file I/O is a method for reading and writing file data that can be a great deal faster than regular stream- or channel-based I/O.

Memory-mapped file I/O is accomplished by causing the data in a file to magically appear as the contents of a memory array. At first, this sounds like it simply means reading the entire file into memory, but in fact it does not. In general, only the parts of the file that you actually read or write are brought, or *mapped*, into memory.

Memory-mapping isn't really magical, or all that uncommon. Modern operating systems generally implement filesystems by mapping portions of a file into portions of memory, doing so on demand. The Java memory-mapping system simply provides access to this facility if it is available in the underlying operating system.

Although they are fairly simple to create, writing to memory-mapped files can be dangerous. By the simple act of changing a single element of an array, you are directly modifying the file on disk. There is no separation between modifying the data and saving it to a disk.

Mapping a file into memory

The easiest way to learn about memory mapping is by example. In the example below, we want to map a `FileChannel` (all or a portion of it) into memory. For this we use the `FileChannel.map()` method. The following line of code maps the first 1024 bytes of a file into memory:

```
MappedByteBuffer mbb = fc.map( FileChannel.MapMode.READ_WRITE,
     0, 1024 );
```

The `map()` method returns a `MappedByteBuffer`, which is a subclass of `ByteBuffer`. Thus, you can use the newly-mapped buffer as you would any other `ByteBuffer`, and the operating system will take care of doing the mapping for you, on demand.

Scattering and gathering

Scattering and gathering overview

Scatter/gather I/O is a method of reading and writing that uses multiple buffers, rather than a single buffer, to hold data.

A scattering read is like a regular channel read, except that it reads data into an array of buffers rather than a single buffer. Likewise, a gathering write writes data from an array of buffers rather than a single buffer.

Scatter/gather I/O is useful for dividing a data stream into separate sections, which can help implement complicated data formats.

Scatter/gather I/O

Channels can optionally implement two new interfaces: `ScatteringByteChannel` and `GatheringByteChannel`. A `ScatteringByteChannel` is a channel that has two additional read methods:

- `long read( ByteBuffer[] dsts );`
- `long read( ByteBuffer[] dsts, int offset, int length );`

These `long read()` methods are rather like the standard `read` methods, except that instead of taking a single buffer they take an array of buffers.

In a *scattering read*, the channel fills up each buffer in turn. When it fills up one buffer, it starts filling the next one. In a sense, the array of buffers is treated like one big buffer.

Applications of scatter/gather

Scatter/gather I/O is useful for dividing a piece of data into sections. For example, you might be writing a networking application that uses message objects, and each message is divided into a fixed-length header and a fixed-length body. You create one buffer that's just big enough for the header, and another buffer that's just big enough for the body. When you put these two in an array and read into them using a scattering read the header and body will be neatly divided between two buffers.

The convenience that we already get from buffers applies to buffer arrays as well. Because each buffer keeps track of how much room it has for more data, the scattering read will automatically find the first buffer with room in it. After that's filled up, it moves onto the next one.

Gathering writes

A *gathering write* is like a scattering read, only for writing. It too has methods that take an array of buffers:

- `long write( ByteBuffer[] srcs );`

- `long write( ByteBuffer[] srcs, int offset, int length );`

A gathering write is useful for forming a single data stream from a group of separate buffers. In keeping with the message example described above, you could use a gathering write to automatically assemble the components of a network message into a single data stream for transmission across a network.

You can see scattering reads and gathering writes in action in the example program UseScatterGather.java.

File locking

File locking overview

File-locking can be confusing at first. It *sounds* like it refers to preventing programs or users from accessing a particular file. In fact, file locks are just like regular Java object locks -- they are *advisory* locks. They don't prevent any kind of data access; instead, they allow different parts of a system to coordinate through the sharing and acquisition of locks.

You can lock an entire file or a portion of a file. If you acquire an exclusive lock, then no one else can acquire a lock on that same file or portion of a file. If you acquire a shared lock, then others can acquire shared locks, but not exclusive locks, on that same file or portion of a file. File locking is not always done for the purpose of protecting data. For example, you might temporarily lock a file to ensure that a particular write operation is made atomically, without interference from other programs.

Most operating systems provide filesystem locks, but they don't all do it in the same way. Some implementations provide shared locks, while others provide only exclusive locks. And some implementations do, in fact, make a locked portion of a file inaccessible, although most do not.

In this section, you'll learn how to do a simple file locking procedure in NIO, and we'll also talk about some of the ways you can ensure your locked files are as portable as they can be.

Locking a file

To acquire a lock on a portion of a file, you call the `lock()` method on an open `FileChannel`. Note that you must open the file for writing if you want to acquire an exclusive lock.

```
RandomAccessFile raf = new RandomAccessFile( "usefilelocks.txt", "rw" );
FileChannel fc = raf.getChannel();
FileLock lock = fc.lock( start, end, false );
```

After you have the lock, you can carry out any sensitive operations that you need to, and then release the lock:

```
lock.release();
```

After you have released the lock, any other programs trying to acquire the lock will have a chance to do so.

The example program, UseFileLocks.java, is meant to be run in parallel with itself. This program acquires a lock on a file, holds it for three seconds, and then releases it. If you run several instances of this program at the same time, you can see each one acquiring the lock in turn.

File locking and portability

File locking can be tricky business, especially given the fact that different operating systems implement locks differently. The following guidelines will help you keep your code as portable as possible:

- Only use exclusive locks.
- Treat all locks as advisory.

Networking and asynchronous I/O

Networking and asynchronous I/O overview

Networking is an excellent foundation for learning about asynchronous I/O, which is of course essential knowledge for anyone doing input/output procedures in the Java language. Networking in NIO isn't much different from any other operation in NIO -- it relies on channels and buffers, and you acquire the channels from the usual `InputStream`s and `OutputStream`s.

In this section we'll start with the fundamentals of asynchronous I/O -- what it is and what it is not -- and then move on to a more hands-on, procedural example.

Asynchronous I/O

Asynchronous I/O is a method for reading and writing data *without blocking*. Normally, when your code makes a `read()` call, the code blocks until there is data to be read. Likewise, a `write()` call will block until the data can be written.

Asynchronous I/O calls, on the other hand, do not block. Instead, you register your interest in a particular I/O event -- the arrival of readable data, a new socket connection, and so on -- and the system tells you when such an event occurs.

One of the advantages of asynchronous I/O is that it lets you do I/O from a great many inputs and outputs at the same time. Synchronous programs often have to resort to polling, or to the creation of many, many threads, to deal with lots of connections. With asynchronous I/O, you can listen for I/O events on an arbitrary number of channels, without polling and without extra threads.

We'll see asynchronous I/O in action by examining an example program called MultiPortEcho.java. This program is like the traditional *echo server*, which takes network connections and echoes back to them any data they might send. However, it has the added feature that it can listen on multiple ports at the same time, and deal with connections from all of those ports. And it does it all in a single thread.

Selectors

The explanation in this section corresponds to the implementation of the `go()` method in the source code for `MultiPortEcho`, so take a look at the source for a fuller picture of what is going on.

The central object in asynchronous I/O is called the `Selector`. A `Selector` is where you register your interest in various I/O events, and it is the object that tells you when those events occur.

So, the first thing we need to do is create a `Selector`:

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

Later on, we will call the `register()` method on various channel objects, in order to register our interest in I/O events happening inside those objects. The first argument to `register()` is always the `Selector`.

Opening a ServerSocketChannel

In order to receive connections, we need a `ServerSocketChannel`. In fact, we need one for each of the ports on which we are going to listen. For each of the ports, we open a `ServerSocketChannel`, as shown here:

```
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking( false );

ServerSocket ss = ssc.socket();
InetSocketAddress address = new InetSocketAddress( ports[i] );
ss.bind( address );
```

The first line creates a new `ServerSocketChannel` and the last three lines bind it to the given port. The second line sets the `ServerSocketChannel` to be *non-blocking*. We must call this method on every socket channel that we're using; otherwise asynchronous I/O won't work.

Selection keys

Our next step is to register the newly opened `ServerSocketChannels` with our `Selector`. We do this using the ServerSocketChannel.register() method, as shown below:

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

The first argument to `register()` is always the `Selector`. The second argument, `OP_ACCEPT`, here specifies that we want to listen for *accept* events -- that is, the events that occur when a new connection is made. This is the only kind of event that is appropriate for a `ServerSocketChannel`.

Note the return value of the call to `register()`. A `SelectionKey` represents this registration of this channel with this `Selector`. When a `Selector` notifies you of an incoming event, it does this by supplying the `SelectionKey` that corresponds to that event. The `SelectionKey` can also be used to de-register the channel.

The inner loop

Now that we have registered our interest in some I/O events, we enter the main loop. Just about every program that uses `Selectors` uses an inner loop much like this one:

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

Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();

while (it.hasNext()) {
     SelectionKey key = (SelectionKey)it.next();
     // ... deal with I/O event ...
}
```

First, we call the `select()` method of our `Selector`. This method blocks until at least one of the registered events occurs. When one or more events occur, the `select()` method returns the number of events that occurred.

Next, we call the `Selector` 's `selectedKeys()` method, which returns a `Set` of the `SelectionKey` objects for which events have occurred.

We process the events by iterating through the `SelectionKeys` and dealing with each one in turn. For each `SelectionKey`, you must determine what I/O event has happened and which I/O objects have been impacted by that event.

Listening for new connections

At this point in the execution of our program, we've only registered `ServerSocketChannel` s, and we have only registered them for "accept" events. To confirm this, we call the `readyOps()` method on our `SelectionKey` and check to see what kind of event has occurred:

```
if ((key.readyOps() & SelectionKey.OP_ACCEPT)
     == SelectionKey.OP_ACCEPT) {

     // Accept the new connection
     // ...
}
```

Sure enough, the `readOps()` method tells us that the event is a new connection.

Accepting a new connection

Because we know there is an incoming connection waiting on this server socket, we can safely accept it; that is, without fear that the `accept()` operation will block:

```
ServerSocketChannel ssc = (ServerSocketChannel)key.channel();
SocketChannel sc = ssc.accept();
```

Our next step is to configure the newly-connected `SocketChannel` to be non-blocking. And because the purpose of accepting this connection is to read data from the socket, we must also register the `SocketChannel` with our `Selector`, as shown below:

```
sc.configureBlocking( false );
SelectionKey newKey = sc.register( selector, SelectionKey.OP_READ );
```

Note that we've registered the `SocketChannel` for *reading* rather than *accepting* new connections, using the `OP_READ` argument to `register()`.

Removing the processed SelectionKey

Having processed the `SelectionKey`, we're almost ready to return to the main loop. But first we must remove the processed `SelectionKey` from the set of selected keys. If we do not remove the processed key, it will still be present as an activated key in the main set, which would lead us to attempt to process it again. We call the iterator's `remove()` method to remove the processed `SelectionKey`:

```
it.remove();
```

Now we're set to return to the main loop and receive incoming data (or an incoming I/O event) on one of our sockets.

Incoming I/O

When data arrives from one of the sockets, it triggers an I/O event. This causes the call to `Selector.select()`, in our main loop, to return with an I/O event or events. This time, the `SelectionKey` will be marked as an `OP_READ` event, as shown below:

```
} else if ((key.readyOps() & SelectionKey.OP_READ)
     == SelectionKey.OP_READ) {
     // Read the data
     SocketChannel sc = (SocketChannel)key.channel();
     // ...
}
```

As before, we get the channel in which the I/O event occurred and process it. In this case, because this is an echo server, we just want to read the data from the socket and send it right back. See the source code (MultiPortEcho.java) in Resources for details on this process.

Back to the main loop

Each time we return to the main loop we call the `select()` method on our `Selector`, and we get a set of `SelectionKey` s. Each key represents an I/O event. We process the events, remove the `SelectionKey` s from the selected set, and go back to the top of the main loop.

This program is a bit simplistic, since it aims only to demonstrate the techniques involved in asynchronous I/O. In a real application, you would need to deal with closed channels by removing them from the `Selector`. And you would probably want to use more than one thread. This program can get away with a single thread because it's only a demo, but in a real-world scenario it might make more sense to create a pool of threads for taking care of the time-consuming portions of I/O event processing.

Character sets

Character sets overview

According to Sun's documentation, a `Charset` is "a named mapping between sequences of sixteen-bit Unicode characters and sequences of bytes." In practice, a `Charset` lets you read and write character sequences in the most portable way possible.

The Java language is defined as being based on Unicode. In practice, however, many people write programs under the assumption that a single character is represented on disk, or in a network stream, as a single byte. This assumption works in many cases, but not all, and as computers become more Unicode-friendly, it becomes less true every day.

In this section, we'll see how to use `Charsets` to process textual data in conformance with modern text formats. The sample program we'll work with here is rather simple; nevertheless, it touches on all the crucial aspects of using `Charset`s: creating a `Charset` for a given character encoding, and using that `Charset` to decode and encode text data.

Encoders/decoders

To read and write text, we are going to use `CharsetDecoder`s and `CharsetEncoder`s, respectively. There's a good reason why these are called *encoders* and *decoders*. A *character* no longer represents a particular bit-pattern, but rather an entity within a character system. Thus, characters represented by an actual bit pattern must therefore be represented in some particular *encoding*.

A `CharsetDecoder` is used to convert the bit-by-bit representation of a string of characters into actual `char` values. Likewise, a `CharsetEncoder` is used to convert the characters back to bits.

Next, we'll take a look at a program that reads and writes data using these objects.

The right way to process text

We'll take a look now at the example program, UseCharsets.java. This program is very simple -- it reads some text from one file, and writes it to another file. But it treats the data as textual data, and reads it into a `CharBuffer` using a `CharsetDecoder`. Likewise, it writes the data back out using a `CharsetEncoder`.

We're going to assume that our characters are stored on disk in the ISO-8859-1 (Latin1) character set -- the standard extension of ASCII. Even though we must be prepared for Unicode, we also must realize that different files are stored in different formats, and ASCII is of course a very common one. In fact, every Java implementation is required to come complete with support for the following character encodings:

- US-ASCII
- ISO-8859-1
- UTF-8
- UTF-16BE
- UTF-16LE
- UTF-16

The sample program

After opening the appropriate files reading the input data into a `ByteBuffer` called `inputData`, our program must create an instance of an ISO-8859-1 (Latin1) character set:

```
Charset latin1 = Charset.forName( "ISO-8859-1" );
```

Then, we create a decoder (for reading) and encoder (for writing):

```
CharsetDecoder decoder = latin1.newDecoder();
CharsetEncoder encoder = latin1.newEncoder();
```

To decode our byte data into a set of characters, we pass our `ByteBuffer` to the `CharsetDecoder`, resulting in a `CharBuffer`:

```
CharBuffer cb = decoder.decode( inputData );
```

If we wanted to process our characters, we could do it at this point in the program. But we only want to write it back out unchanged, so there's nothing to do.

To write the data back out, we must convert it back to bytes, using the `CharsetEncoder`:

```
ByteBuffer outputData = encoder.encode( cb );
```

After the conversion is complete we can write the data out to a file.

Summary

Summary

As you've seen, there are a lot of features in the NIO library. While some of the new features -- file locking and character sets, for example -- provide new capabilities, many of the features excel in the area of optimization.

At a fundamental level, there's nothing that channels and buffers can do that we couldn't do using the old stream-oriented classes. But channels and buffers allow for the possibility of doing the same old operations *much faster* -- approaching the maximum allowed by the system, in fact.

But one of the greatest strengths of NIO is that it provides a new -- and much needed -- structuring metaphor for doing input/output in the Java language. Along with such new conceptual (and realizable) entities as buffers, channels, and asynchronous I/O comes the opportunity to rethink I/O procedures in your Java programs. In this way, NIO breathes new life into even the most familiar procedures of I/O and gives us the opportunity to do them differently, and better, than we have before.

Resources

- Download nio-src.zip, the complete source for the examples in this tutorial.
- See the SDK Documentation for more information about installing and configuring JDK 1.4.
- Sun's guide to the new I/O APIs provides a thorough introduction to NIO, including some minor elements not covered in this tutorial.
- The online API specification describes the classes and methods of NIO, in the autodoc format that you know and love.
- JSR 51 is the Java Community Process document that first specified the new features of NIO. In fact, NIO, as implemented in JDK 1.4, is a subset of the features described in this document.
- Want a thorough introduction to stream I/O (including problems, solutions, and the introduction of NIO)? You couldn't do better than Merlin Hughes's " Turning streams inside out " (*developerWorks*, July 2002).
- Of course, you could also just take the tutorial. " Introduction to Java I/O " (*developerWorks*, April 2000) covers all the basics of Java I/O prior to JDK 1.4.
- John Zukowski has written some good articles about NIO for his *Magic with Merlin* column:
  - " The ins and outs of Merlin's new I/O buffers " (*developerWorks*, March 2003) is another look at buffer basics.
  - " Character sets " (*developerWorks*, October 2002) is all about charsets (especially conversion and encoding schemes).
- Get a little deeper into NIO with Aruna Kalagnanam and Balu G's " Merlin brings nonblocking I/O to the Java platform " (*developerWorks*, March 2002).
- Greg Travis examines NIO in his book, *JDK 1.4 Tutorial* (Manning Publications, March 2002).
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.

About the author

Greg Travis

Greg Travis is a freelance Java programmer and technology writer living in New York City. Greg started his programming career in 1992, spending three years in the world of high-end PC games. In 1995, he joined EarthWeb, where he began developing new technologies with the Java programming language. Since 1997, Greg has been a consultant in a variety of Web technologies, specializing in real-time graphics and sound. His interests include algorithm optimization, programming language design, signal processing (with emphasis on music), and real-time 3D graphics. Other articles by Greg can be found on his personal Web page. He is also the author of *JDK 1.4 Tutorial*, published by Manning Publications.
For technical questions or comments about the content of this tutorial, contact Greg Travis at mito@panix.com.