Turning streams inside out, Part 2: Optimizing internal Java I/O

Replacing the byte-array and piped streams
Level: Intermediate

Merlin Hughes (merlin@merlin.org)
Cryptographer, Baltimore Technologies
September 2002

While the new Java I/O framework, `java.nio`, addresses most of the performance issues with I/O support, it does not address all the performance needs of internal application communications using byte-arrays and pipes. In this final article in a two-part series, Java cryptographer and author Merlin Hughes develops a new set of streams that complement the standard Java I/O byte-array and piped-stream classes, emphasizing performance as a design goal. Share your thoughts on this article with the author and other readers in the discussion forum on this article. (You can also click **Discuss** at the top or bottom of the article.)

In the first article in this series, you learned some different approaches to solving the problem of reading data from a source that can only write out its data. Among the potential solutions, we investigated use of the byte-array streams, the piped streams, and a custom framework that tackles the problem directly. The custom approach was clearly the most efficient solution; however, examining the other approaches was constructive in highlighting some problems with the standard Java streams. In particular, the byte-array output stream does not provide an efficient mechanism to provide read-only access to its contents, and the piped streams generally exhibit extremely poor performance.

We'll address these issues in this article by implementing alternative classes that provide the same broad capabilities, but with more of an emphasis on performance. To begin, let's briefly touch on the issue of synchronization as it relates to I/O streams.

Synchronization issues

In general, I recommend avoiding the needless use of synchronization where there is no particular need for it. Clearly, if a class will be concurrently accessed by multiple threads, then it needs to be thread safe. However, there are many cases where concurrent access makes no sense, and synchronization is a needless overhead. For example, concurrent access to a stream is inherently non-deterministic -- you cannot predict which data will be written first, or which thread will read what data -- and, as such, is rarely of any use. Imposing synchronization on all streams is thus a cost that provides no practical benefit. If a particular application requires thread safety, that can be enforced though the application's own synchronization primitives.

This is, in fact, the same choice that was made for the classes of the Collections API: by default, sets, lists, and so on are not thread safe. If an application desires a thread safe collection, it can use the `Collections` class to create a thread safe wrapper around a non-thread safe collection. Were it of any use, an application could use exactly the same mechanism to allow thread safety to be wrapped around a stream; for example, `OutputStream out = Streams.synchronizedOutputStream (byteStream)`. See the `Streams` class in the accompanying source code for an example implementation.

Consequently, I have not used synchronization to provide thread safety for classes where I do not envision them being usable by multiple concurrent threads. Before you adopt this approach in general, I recommend that you study the *Threads and Locks* chapter of the Java Language Specification (see Resources) to understand the potential pitfalls; in particular, the ordering of reads and writes is not guaranteed when synchronization is not employed, so apparently harmless concurrent access to unsynchronized read-only methods could lead to unexpected behaviour.

A better byte-array output stream

The `ByteArrayOutputStream` class is a great stream to use when you need to dump an unknown volume of data into a memory buffer. I use it frequently when I need to store some data for later re-reading. However, using the `toByteArray()` method to obtain read access to the resulting data is quite inefficient because it actually returns a copy of the internal byte array. For small volumes of data, this approach is of little concern; however, as volumes grow, it becomes needlessly inefficient. The class has to copy the data because it cannot enforce read-only access to the resulting byte array. If it returned its internal buffer, then the recipient would have, in the general case, no guarantee that the buffer was not being concurrently modified by another recipient of the same array.

The `StringBuffer` class has already solved a similar problem; it provides a writable buffer of characters, and supports efficiently returning read-only `String`s that read directly from the internal character array. Because the `StringBuffer` class controls write access to its internal array, it can copy the array only when necessary; that is, when it has exported a `String` and a caller subsequently modifies the `StringBuffer`. If no such modification occurs, then no unnecessary copying will be performed. The new I/O framework addresses this problem in a similar manner by supporting wrappers around byte-arrays that can enforce appropriate access controls.

We can use this same general mechanism to provide efficient buffering and re-reading of data for applications that need to use the standard streams API. Our example will present an alternative to the `ByteArrayOutputStream` class that can efficiently export read-only access to the internal buffer by returning a read-only `InputStream` that reads directly from the internal byte array.

Let's take a look at the code. The constructors in Listing 1 allocate an initial buffer to hold data written to this stream. This buffer will automatically expand as needed to hold more data.

Listing 1. An unsynchronized byte-array output stream

```
package org.merlin.io;

import java.io.*;

/**
 * An unsynchronized ByteArrayOutputStream alternative that efficiently
 * provides read-only access to the internal byte array with no
 * unnecessary copying.
 *
 * @author Copyright (c) 2002 Merlin Hughes <merlin@merlin.org>
 */
public class BytesOutputStream extends OutputStream {
  private static final int DEFAULT_INITIAL_BUFFER_SIZE = 8192;

  // internal buffer
  private byte[] buffer;
  private int index, capacity;
  // is the stream closed?
  private boolean closed;
  // is the buffer shared?
  private boolean shared;

  public BytesOutputStream () {
    this (DEFAULT_INITIAL_BUFFER_SIZE);
  }

  public BytesOutputStream (int initialBufferSize) {
    capacity = initialBufferSize;
    buffer = new byte[capacity];
  }
```

Listing 2 shows the writing methods. These methods expand the internal buffer, if necessary, and then copy in the new data. When expanding the internal buffer, we double its size and add the amount necessary to hold the new data; the capacity thus grows exponentially to hold any necessary volume. For efficiency, if you know the expected volume of data that you will write, you should specify a corresponding initial buffer size. The `close()` method just sets an appropriate flag.

Listing 2. The write methods

```
  public void write (int datum) throws IOException {
    if (closed) {
      throw new IOException ("Stream closed");
    } else {
      if (index >= capacity) {
        // expand the internal buffer
        capacity = capacity * 2 + 1;
        byte[] tmp = new byte[capacity];
        System.arraycopy (buffer, 0, tmp, 0, index);
        buffer = tmp;
        // the new buffer is not shared
        shared = false;
      }
      // store the byte
      buffer[index ++] = (byte) datum;
    }
  }

  public void write (byte[] data, int offset, int length)
    throws IOException {
    if (data == null) {
      throw new NullPointerException ();
    } else if ((offset < 0) || (offset + length > data.length)
        || (length < 0)) {
      throw new IndexOutOfBoundsException ();
    } else if (closed) {
      throw new IOException ("Stream closed");
    } else {
      if (index + length > capacity) {
        // expand the internal buffer
        capacity = capacity * 2 + length;
        byte[] tmp = new byte[capacity];
        System.arraycopy (buffer, 0, tmp, 0, index);
        buffer = tmp;
        // the new buffer is not shared
        shared = false;
      }
      // copy in the subarray
      System.arraycopy (data, offset, buffer, index, length);
      index += length;
    }
  }

  public void close () {
    closed = true;
  }
```

The byte-array extraction method in Listing 3 returns a copy of the internal byte-array. Because we cannot prevent the caller from writing to the resulting array, we cannot safely return a direct reference to the internal buffer.

Listing 3. Converting to a byte-array

```
  public byte[] toByteArray () {
    // return a copy of the internal buffer
    byte[] result = new byte[index];
    System.arraycopy (buffer, 0, result, 0, index);
    return result;
  }
```

When methods provide read-only access to the stored data, they can safely and efficiently use the internal byte-array directly.

Listing 4 shows two such methods. The `writeTo()` method writes the contents of this stream to an output stream; it performs the write operation directly from the internal buffer. The `toInputStream()` method returns an input stream from which the data can be efficiently read. It returns a `BytesInputStream`, which is an unsynchronized alternative to `ByteArrayInputStream`, that reads directly from our internal byte array. In this case, we also set a flag to indicate that the internal buffer is now shared with the input stream. This is important to protect against this stream being modified while the internal buffer is shared.

Listing 4. Read-only access methods

```
  public void writeTo (OutputStream out) throws IOException {
    // write the internal buffer directly
    out.write (buffer, 0, index);
  }

  public InputStream toInputStream () {
    // return a stream reading from the shared internal buffer
    shared = true;
    return new BytesInputStream (buffer, 0, index);
  }
```

The only method that can potentially overwrite shared data is the `reset()` method, shown in Listing 5, which empties this stream. Therefore, if `shared` is true and `reset()` is called, we create a new internal buffer instead of just resetting the write index.

Listing 5. Resetting the stream

```
  public void reset () throws IOException {
    if (closed) {
      throw new IOException ("Stream closed");
    } else {
      if (shared) {
        // create a new buffer if it is shared
        buffer = new byte[capacity];
        shared = false;
      }
      // reset index
      index = 0;
    }
  }
}
```

A better byte-array input stream
The `ByteArrayInputStream` class is ideal to use for providing streams-based read access to in-memory binary data. However, it has two design features that cause me occasional desire for an alternative. Firstly, the class is synchronized; this, as I have explained, is unnecessary for most applications. Secondly, it provides an implementation of the `reset()` method that, if called prior to `mark()`, ignores the initial reading offset. Neither of these are flaws; however, they're not necessarily always desirable.

The `BytesInputStream` class, shown in Listing 6, is an unsynchronized, and largely unremarkable byte-array input stream class.

Listing 6. An unsynchronized byte-array input stream

```
package org.merlin.io;

import java.io.*;

/**
 * An unsynchronized ByteArrayInputStream alternative.
 *
 * @author Copyright (c) 2002 Merlin Hughes <merlin@merlin.org>
 */
public class BytesInputStream extends InputStream {
  // buffer from which to read
  private byte[] buffer;
  private int index, limit, mark;
  // is the stream closed?
  private boolean closed;

  public BytesInputStream (byte[] data) {
    this (data, 0, data.length);
  }

  public BytesInputStream (byte[] data, int offset, int length) {
    if (data == null) {
      throw new NullPointerException ();
    } else if ((offset < 0) || (offset + length > data.length)
        || (length < 0)) {
      throw new IndexOutOfBoundsException ();
    } else {
      buffer = data;
      index = offset;
      limit = offset + length;
      mark = offset;
    }
  }

  public int read () throws IOException {
    if (closed) {
      throw new IOException ("Stream closed");
    } else if (index >= limit) {
      return -1; // EOF
    } else {
      return buffer[index ++] & 0xff;
    }
  }

  public int read (byte data[], int offset, int length)
      throws IOException {
    if (data == null) {
      throw new NullPointerException ();
    } else if ((offset < 0) || (offset + length > data.length)
        || (length < 0)) {
      throw new IndexOutOfBoundsException ();
    } else if (closed) {
      throw new IOException ("Stream closed");
    } else if (index >= limit) {
      return -1; // EOF
    } else {
      // restrict length to available data
      if (length > limit - index)
        length = limit - index;
      // copy out the subarray
      System.arraycopy (buffer, index, data, offset, length);
      index += length;
      return length;
    }
  }

  public long skip (long amount) throws IOException {
    if (closed) {
      throw new IOException ("Stream closed");
    } else if (amount <= 0) {
      return 0;
    } else {
      // restrict amount to available data
      if (amount > limit - index)
        amount = limit - index;
      index += (int) amount;
      return amount;
    }
  }

  public int available () throws IOException {
    if (closed) {
      throw new IOException ("Stream closed");
    } else {
      return limit - index;
    }
  }

  public void close () {
    closed = true;
  }

  public void mark (int readLimit) {
    mark = index;
  }

  public void reset () throws IOException {
    if (closed) {
      throw new IOException ("Stream closed");
    } else {
      // reset index
      index = mark;
    }
  }

  public boolean markSupported () {
    return true;
  }
}
```

Using the new byte-array streams
The code in Listing 7 demonstrates how to use the new byte-array streams to solve the problem addressed in the first article (reading the compressed form of some data):

Listing 7. Using the new byte-array streams

```
public static InputStream newBruteForceCompress (InputStream in)
    throws IOException {
  BytesOutputStream sink = new BytesOutputStream ();
  OutputStream out = new GZIPOutputStream (sink);
  Streams.io (in, out);
  out.close ();
  return sink.toInputStream ();
}
```

Better piped streams

The standard piped streams, while safe and reliable, leave much to be desired in terms of their performance. Several factors contribute to this performance problem:

- The internal 1024-byte buffer is inflexible for different usage scenarios; it is just too small for large volumes of data.
- Array-based operations simply call through to an inefficient byte-by-byte copy operation. This operation is itself synchronized, resulting in extremely heavy lock contention.
- If a pipe becomes empty or full and a thread is blocked on this state changing, the thread is awakened even if just a single byte is read or written. In many cases, it will use this single byte and immediately block again, resulting in little useful work being done.

The last factor is a consequence of the strict contract that the API provides. This strict contract is necessary for the streams to be used in the most general possible applications. However, it is possible to come up with a more relaxed contract for a piped stream implementation that sacrifices strictness for increased performance:

- Blocked readers and writers awaken only when the buffer's available data (in the case of a blocked reader) or free space (in the case of a writer) reaches a certain specified *hysteresis* threshold or when an abnormal event, such as pipe closure, occurs. This improves performance because threads awaken only when they can perform a reasonable amount of work.
- Only a single thread may read from the pipe, and a single thread may write to the pipe. Otherwise, the pipe cannot reliably determine when the reader or writer thread has accidentally died.

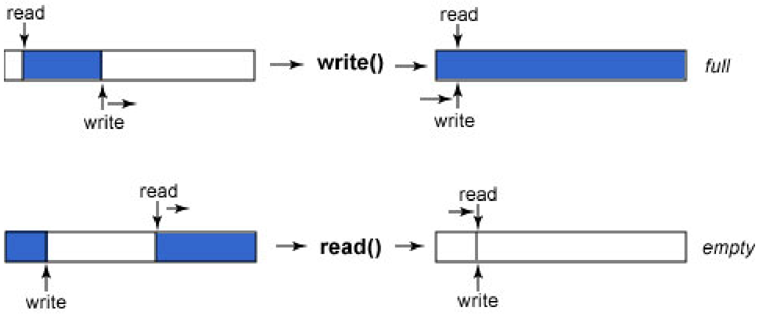
The typical application scenario of an independent reader and writer thread is perfectly served by this contract; applications that require immediate waking can use a zero hysteresis level. As we'll see later, an implementation of this contract can operate over two orders of magnitude (one hundred times) faster than the standard API streams.

We could use one of several potential APIs to expose these piped streams: we could mimic the standard classes and explicitly attach two streams; alternatively, we could expose a `Pipe` class from which we can extract the input and output streams. Instead, we'll go with a simpler approach: create a `PipeInputStream` and then extract the associated output stream.

The general operation of these streams is as follows:

- We use an internal array as a ring-buffer (see Figure 1): A read and a write index are maintained in this array; data are written to the write index and read from the read index; and the indices wrap around when they reach the end of the buffer. Neither index can pass the other. When the write index reaches the read index, the pipe is full and no more data can be written. When the read index reaches the write index, the pipe is empty and no more data can be read.
- Synchronization is used to ensure that the two co-operating threads see up-to-date values for the pipe's state. The Java Language Specification is quite lenient in its rules on the ordering of memory accesses, which prevents the use of lock-free buffering techniques.

Figure 1. A ring buffer



The code that implements these piped streams is presented in the following code listings. Listing 8 shows the constructors and variables used by the class. This is an `InputStream` from which you can extract the corresponding `OutputStream` (code shown in Listing 17). In the constructors you can specify the internal buffer size and a hysteresis level; this is the fraction of the buffer's capacity that must become used or available before a corresponding reader or writer thread will be immediately awakened.

We maintain two variables, `reader` and `writer`, which correspond to the reader and writer threads. We use these to identify when one thread dies while the other is still accessing the stream.

Listing 8. An alternative piped stream implementation

```
package org.merlin.io;

import java.io.*;

/**
 * An efficient connected stream pair for communicating between
 * the threads of an application. This provides a less-strict contract
 * than the standard piped streams, resulting in much-improved
 * performance. Also supports non-blocking operation.
 *
 * @author Copyright (c) 2002 Merlin Hughes <merlin@merlin.org>
 */
public class PipeInputStream extends InputStream {
  // default values
  private static final int DEFAULT_BUFFER_SIZE = 8192;
  private static final float DEFAULT_HYSTERESIS = 0.75f;
  private static final int DEFAULT_TIMEOUT_MS = 1000;

  // flag indicates whether method applies to reader or writer
  private static final boolean READER = false, WRITER = true;

  // internal pipe buffer
  private byte[] buffer;
  // read/write index
  private int readx, writex;
  // pipe capacity, hysteresis level
  private int capacity, level;
  // flags
  private boolean eof, closed, sleeping, nonBlocking;
  // reader/writer thread
  private Thread reader, writer;
  // pending exception
  private IOException exception;
  // deadlock-breaking timeout
  private int timeout = DEFAULT_TIMEOUT_MS;

  public PipeInputStream () {
    this (DEFAULT_BUFFER_SIZE, DEFAULT_HYSTERESIS);
  }

  public PipeInputStream (int bufferSize) {
    this (bufferSize, DEFAULT_HYSTERESIS);
  }

  // e.g., hysteresis .75 means sleeping reader/writer is not
  // immediately woken until the buffer is 75% full/empty
  public PipeInputStream (int bufferSize, float hysteresis) {
    if ((hysteresis < 0.0) || (hysteresis > 1.0))
      throw new IllegalArgumentException ("Hysteresis: " + hysteresis);
    capacity = bufferSize;
    buffer = new byte[capacity];
    level = (int) (bufferSize * hysteresis);
  }
```

The configuration methods in Listing 9 allow you to configure the stream timeout and non-blocking mode. The timeout is the number of milliseconds after which blocked threads will automatically awaken; this is necessary to break the potential deadlock that could occur if one thread died. In non-blocking mode, an `InterruptedIOException` will be thrown if a thread would block.

Listing 9. Pipe configuration

```
  public void setTimeout (int ms) {
    this.timeout = ms;
  }

  public void setNonBlocking (boolean nonBlocking) {
    this.nonBlocking = nonBlocking;
  }
```

Listing 10 shows the reading methods, which all follow a fairly standard pattern: we take a reference to the reading thread if we don't already have one, then we verify the input parameters, check that the stream is not closed or that an exception is not pending, determine how much data can be read, and finally copy the data from the internal ring buffer into the reader's buffer. The `checkedAvailable()` method, shown in Listing 12, automatically waits until some data are available or the stream is closed, before returning.

Listing 10. Reading data

```
  private byte[] one = new byte[1];

  public int read () throws IOException {
    // read 1 byte
    int amount = read (one, 0, 1);
    // return EOF / the byte
    return (amount < 0) ? -1 : one[0] & 0xff;
  }

  public synchronized int read (byte data[], int offset, int length)
      throws IOException {
    // take a reference to the reader thread
    if (reader == null)
      reader = Thread.currentThread ();
    // check parameters
    if (data == null) {
      throw new NullPointerException ();
    } else if ((offset < 0) || (offset + length > data.length)
        || (length < 0)) { // check indices
      throw new IndexOutOfBoundsException ();
    } else {
      // throw an exception if the stream is closed
      closedCheck ();
      // throw any pending exception
      exceptionCheck ();
      if (length <= 0) {
        return 0;
      } else {
        // wait for some data to become available for reading
        int available = checkedAvailable (READER);
        // return -1 on EOF
        if (available < 0)
          return -1;
        // calculate amount of contiguous data in pipe buffer
        int contiguous = capacity - (readx % capacity);
        // calculate how much we will read this time
        int amount = (length > available) ? available : length;
        if (amount > contiguous) {
          // two array copies needed if data wrap around the buffer end
          System.arraycopy (buffer, readx % capacity, data, offset,
            contiguous);
          System.arraycopy (buffer, 0, data, offset + contiguous,
            amount - contiguous);
        } else {
```

```
        // otherwise, one array copy needed
        System.arraycopy (buffer, readx % capacity, data, offset,
          amount);
      }
      // update indices with amount of data read
      processed (READER, amount);
      // return amount read
      return amount;
    }
  }
}

public synchronized long skip (long amount) throws IOException {
  // take a reference to the reader thread
  if (reader == null)
    reader = Thread.currentThread ();
  // throw an exception if the stream is closed
  closedCheck ();
  // throw any pending exception
  exceptionCheck ();
  if (amount <= 0) {
    return 0;
  } else {
    // wait for some data to become available for skipping
    int available = checkedAvailable (READER);
    // return 0 on EOF
    if (available < 0)
      return 0;
    // calculate how much we will skip this time
    if (amount > available)
      amount = available;
    // update indices with amount of data skipped
    processed (READER, (int) amount);
    // return amount skipped
    return amount;
  }
}
```

The method shown in Listing 11 is called when data are read from or written to this pipe. It updates the relevant indexes and automatically wakes a blocked thread if the pipe reaches its hysteresis level.

Listing 11. Updating indices

```
private void processed (boolean rw, int amount) {
  if (rw == READER) {
    // update read index with amount read
    readx = (readx + amount) % (capacity * 2);
  } else {
    // update write index with amount written
    writex = (writex + amount) % (capacity * 2);
  }
  // check whether a thread is sleeping and we have reached the
  // hysteresis threshold
  if (sleeping && (available (!rw) >= level)) {
    // wake sleeping thread
    notify ();
    sleeping = false;
  }
}
```

The `checkedAvailable()` method shown in Listing 12 waits until this pipe has space or data available (depending on the `rw` parameter) and then returns the amount to the caller. Internally, this also checks that the stream is not closed, the pipe is not broken, and so on.

Listing 12. Checking availability

```
public synchronized int available () throws IOException {
  // throw an exception if the stream is closed
  closedCheck ();
  // throw any pending exception
  exceptionCheck ();
  // determine how much can be read
  int amount = available (READER);
  // return 0 on EOF, otherwise the amount readable
  return (amount < 0) ? 0 : amount;
}

private int checkedAvailable (boolean rw) throws IOException {
  // always called from synchronized(this) method
  try {
    int available;
    // loop while no data can be read/written
    while ((available = available (rw)) == 0) {
      if (rw == READER) { // reader
        // throw any pending exception
        exceptionCheck ();
      } else { // writer
        // throw an exception if the stream is closed
        closedCheck ();
      }
      // throw an exception if the pipe is broken
      brokenCheck (rw);
      if (!nonBlocking) { // blocking mode
        // wake any sleeping thread
        if (sleeping)
          notify ();
        // sleep for timeout ms (in case of peer thread death)
        sleeping = true;
        wait (timeout);
        // timeout means that hysteresis may not be obeyed
      } else { // non-blocking mode
        // throw an InterruptedIOException
        throw new InterruptedIOException
          ("Pipe " + (rw ? "full" : "empty"));
      }
    }
    return available;
  } catch (InterruptedException ex) {
    // rethrow InterruptedException as InterruptedIOException
    throw new InterruptedIOException (ex.getMessage ());
  }
}

private int available (boolean rw) {
  // calculate amount of space used in pipe
  int used = (writex + capacity * 2 - readx) % (capacity * 2);
  if (rw == WRITER) { // writer
    // return amount of space available for writing
    return capacity - used;
  } else { // reader
    // return amount of data in pipe or -1 at EOF
    return (eof && (used == 0)) ? -1 : used;
  }
}
```

The methods in Listing 13 close this stream; support is provided for the reader or the writer to close the stream. Blocked threads are automatically awakened, and various other sanity checks are made.

Listing 13. Closing the stream

```
public void close () throws IOException {
  // close the read end of this pipe
  close (READER);
}

private synchronized void close (boolean rw) throws IOException {
  if (rw == READER) { // reader
    // set closed flag
    closed = true;
  } else if (!eof) { // writer
    // set eof flag
    eof = true;
    // check if data remain unread
    if (available (READER) > 0) {
      // throw an exception if the reader has already closed the pipe
      closedCheck ();
      // throw an exception if the reader thread has died
      brokenCheck (WRITER);
    }
  }
  // wake any sleeping thread
  if (sleeping) {
    notify ();
    sleeping = false;
  }
}
```

The methods shown in Listing 14 check the status of this stream. If an exception is pending, the stream is closed or the pipe is broken (that is, the reader or writer thread is no longer alive) and an exception is thrown.

Listing 14. Checking stream status

```
private void exceptionCheck () throws IOException {
  // throw any pending exception
  if (exception != null) {
    IOException ex = exception;
    exception = null;
    throw ex; // could wrap ex in a local exception
  }
}

private void closedCheck () throws IOException {
  // throw an exception if the pipe is closed
  if (closed)
    throw new IOException ("Stream closed");
}

private void brokenCheck (boolean rw) throws IOException {
  // get a reference to the peer thread
  Thread thread = (rw == WRITER) ? reader : writer;
  // throw an exception if  the peer thread has died
  if ((thread != null) && !thread.isAlive ())
    throw new IOException ("Broken pipe");
}
```

Listing 15 presents the method that is called when data are written into this pipe. It is broadly similar to the reading methods: we first take a copy of the writer thread, then check that the stream is not closed and enter a loop copying data into the pipe. As before, this approach uses the `checkedAvailable()` method, which automatically blocks until there is available capacity in the pipe.

Listing 15. Writing data

```
private synchronized void writeImpl (byte[] data, int offset, int length)
      throws IOException {
    // take a reference to the writer thread
    if (writer == null)
      writer = Thread.currentThread ();
    // throw an exception if the stream is closed
    if (eof || closed) {
      throw new IOException ("Stream closed");
    } else {
      int written = 0;
      try {
        // loop to write all the data
        do {
          // wait for space to become available for writing
          int available = checkedAvailable (WRITER);
          // calculate amount of contiguous space in pipe buffer
          int contiguous = capacity - (writex % capacity);
          // calculate how much we will write this time
          int amount = (length > available) ? available : length;
          if (amount > contiguous) {
          // two array copies needed if space wraps around the buffer end
            System.arraycopy (data, offset, buffer, writex % capacity,
              contiguous);
            System.arraycopy (data, offset + contiguous, buffer, 0,
              amount - contiguous);
          } else {
            // otherwise, one array copy needed
            System.arraycopy (data, offset, buffer, writex % capacity,
              amount);
          }
          // update indices with amount of data written
          processed (WRITER, amount);
          // update amount written by this method
          written += amount;
        } while (written < length);
        // data successfully written
      } catch (InterruptedIOException ex) {
        // write operation was interrupted; set the bytesTransferred
        // exception field to reflect the amount of data written
        ex.bytesTransferred = written;
        // rethrow exception
        throw ex;
      }
    }
  }
```

One of the features of this piped-stream implementation is that the writer can set an exception that is passed on to the reader, as shown in Listing 16.

Listing 16. Setting an exception

```
private synchronized void setException (IOException ex)
    throws IOException {
  // fail if an exception is already pending
  if (exception != null)
    throw new IOException ("Exception already set: " + exception);
  // throw an exception if the pipe is broken
  brokenCheck (WRITER);
  // take a reference to the pending exception
  this.exception = ex;
  // wake any sleeping thread
  if (sleeping) {
    notify ();
    sleeping = false;
  }
}
```

Listing 17 presents the output stream-related code for this pipe. The `getOutputStream()` method returns an `OutputStreamImpl`, which is an output stream that writes data into the internal pipe using the methods presented earlier. This class extends `OutputStreamEx`, which is an extension to the output stream class that allows an exception to be set for the reading thread.

Listing 17. The output stream

```
public OutputStreamEx getOutputStream () {
  // return an OutputStreamImpl associated with this pipe
  return new OutputStreamImpl ();
}

private class OutputStreamImpl extends OutputStreamEx {
  private byte[] one = new byte[1];

  public void write (int datum) throws IOException {
    // write one byte using internal array
    one[0] = (byte) datum;
    write (one, 0, 1);
  }

  public void write (byte[] data, int offset, int length)
      throws IOException {
    // check parameters
    if (data == null) {
      throw new NullPointerException ();
    } else if ((offset < 0) || (offset + length > data.length)
        || (length < 0)) {
      throw new IndexOutOfBoundsException ();
    } else if (length > 0) {
      // call through to writeImpl()
      PipeInputStream.this.writeImpl (data, offset, length);
    }
  }

  public void close () throws IOException {
    // close the write end of this pipe
    PipeInputStream.this.close (WRITER);
  }

  public void setException (IOException ex) throws IOException {
    // set a pending exception
    PipeInputStream.this.setException (ex);
  }
}
```

```
  // static OutputStream extension with setException() method
  public static abstract class OutputStreamEx extends OutputStream {
    public abstract void setException (IOException ex) throws IOException;
  }
}
```

Using the new piped streams

Listing 18 demonstrates how to use the new piped streams to solve the problem from the previous article. Note that any exception occurring in the writer thread can be just passed down the stream.

Listing 18. Using the new piped streams

```
public static InputStream newPipedCompress (final InputStream in)
    throws IOException {
  PipeInputStream source = new PipeInputStream ();
  final PipeInputStream.OutputStreamEx sink = source.getOutputStream ();
  new Thread () {
    public void run () {
      try {
        GZIPOutputStream gzip = new GZIPOutputStream (sink);
        Streams.io (in, gzip);
        gzip.close ();
      } catch (IOException ex) {
        try {
          sink.setException (ex);
        } catch (IOException ignored) {
        }
      }
    }
  }.start ();
  return source;
}
```

Performance results

The performance of these new streams on a 800MHz Linux box running the Java 2 SDK, version 1.4.0, is shown in the following table, along with the performance of the standard streams. The performance harness is the same as I used in the previous article:

Piped streams
    15KB: 21ms; 15MB: 20675ms
New piped streams
    15KB: 0.68ms; 15MB: 158ms
Byte-array streams
    15KB: 0.31ms; 15MB: 745ms
New byte-array streams
    15KB: 0.26ms; 15MB: 438ms

Performance differences from the last article simply reflect the varying ambient load on my machine. As you can see from these results, the new piped streams result in much better performance than the brute-force solution for large volumes of data; however, they are still about twice as slow as the engineering solution we examined. Clearly, the overhead of using multiple threads within modern Java virtual machines is not nearly as great as before.

Conclusion

We've examined two sets of alternative streams to those of the standard Java API: `BytesOutputStream` and `BytesInputStream` are unsynchronized alternatives to the byte-array streams. Because the expected use cases for these classes involves access by just a single thread, the absence of synchronization is a legitimate choice. In practise, the reduction in execution time of up to 40 percent is probably only marginally due to the elimination of synchronization; the majority of the performance improvement is due to the elimination of unnecessary copying when providing read-only access. The second example, `PipeInputStream`, is an alternative to the piped streams; it uses a relaxed contract, improved buffer size, and array-based operations to cut execution time by over 99 percent. In this case, unsynchronized code is not an option; the Java Language Specification rules out reliable execution of such code, even though it is otherwise theoretically possible to implement a pipe with minimal locking.

The byte-array and piped streams are the primary choices for streams-based intra-application communications. While the new I/O API provides some alternatives, many applications and APIs still rely on the standard streams, and the new I/O API is not necessarily much more efficient for these particular uses. Appropriately reducing the use of synchronization, effectively adopting array-based operations, and minimizing unnecessary copying have produced dramatic results here, providing much more efficient operations that directly fit into the standard streams framework. Taking these same steps in other areas of application development can often produce similar performance improvements.

Resources

- Participate in the discussion forum on this article by clicking **Discuss** at the top or bottom of the article.
- The first part of this series describes a framework that lets an application efficiently read data from a source that only supports writing data to an output stream.
- Download the source code discussed in this article. This code is freely licensed under the terms of the GNU General Public License.
- Read this hands-on introduction to Java I/O from *developerWorks*.
- Learn about the new I/O APIs in the J2SE 1.4 guide.
- The IBM Developer Kit, Java 2 Technology Edition, Version 1.3 runs the piped-stream examples noticeably faster than the Sun J2SE 1.4.0 on Merlin's system.
- The Java Language specification chapter Threads and Locks offers excellent information on this tricky subject.
- JSR 133 is planning to further clarify this aspect of the Java platform.
- Brian Goetz's *developerWorks* series *Threading lightly* provides additional insight into the practical issues that arise in concurrent programming:
  - Part 1: "Synchronization is not the enemy " (July 2001) describes how the performance impact of uncontended synchronization is not as great as is widely believed.
  - Part 2: "Reducing contention " (September 2001) describes techniques to reduce the impact of contended synchronization on program performance.
  - Part 3: "Sometimes it's best not to share" (October, 2001) examines `ThreadLocal` and offers tips for exploiting its power.
- You'll find hundreds of articles about every aspect of Java programming in the developerWorks Java technology zone.

About the author
Merlin Hughes is a cryptographer and chief technical evangelist with the Irish e-security company Baltimore Technologies, and a part time janitor and dishwasher; not to be confused with JDK 1.4. Based in New York, New York (a city so nice, they named it twice), he can be reached at merlin@merlin.org.