Bits.java

```
/*
 * Copyright (c) 2001, 2010, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Utility methods for packing/unpacking primitive values in/out of byte arrays
 * using big-endian byte ordering.
 */
class Bits {

    /*
     * Methods for unpacking primitive values from byte arrays starting at
     * given offsets.
     */

    static boolean getBoolean(byte[] b, int off) {
        return b[off] != 0;
    }

    static char getChar(byte[] b, int off) {
        return (char) ((b[off + 1] & 0xFF) +
                       (b[off] << 8));
    }

    static short getShort(byte[] b, int off) {
        return (short) ((b[off + 1] & 0xFF) +
                        (b[off] << 8));
    }

    static int getInt(byte[] b, int off) {
        return ((b[off + 3] & 0xFF)      ) +
               ((b[off + 2] & 0xFF) <<  8) +
               ((b[off + 1] & 0xFF) << 16) +
               ((b[off    ]       ) << 24);
    }

    static float getFloat(byte[] b, int off) {
```

```
        return Float.intBitsToFloat(getInt(b, off));
    }

    static long getLong(byte[] b, int off) {
        return ((b[off + 7] & 0xFFL)      ) +
               ((b[off + 6] & 0xFFL) <<  8) +
               ((b[off + 5] & 0xFFL) << 16) +
               ((b[off + 4] & 0xFFL) << 24) +
               ((b[off + 3] & 0xFFL) << 32) +
               ((b[off + 2] & 0xFFL) << 40) +
               ((b[off + 1] & 0xFFL) << 48) +
               (((long) b[off])      << 56);
    }

    static double getDouble(byte[] b, int off) {
        return Double.longBitsToDouble(getLong(b, off));
    }

    /*
     * Methods for packing primitive values into byte arrays starting at given
     * offsets.
     */

    static void putBoolean(byte[] b, int off, boolean val) {
        b[off] = (byte) (val ? 1 : 0);
    }

    static void putChar(byte[] b, int off, char val) {
        b[off + 1] = (byte) (val      );
        b[off    ] = (byte) (val >>> 8);
    }

    static void putShort(byte[] b, int off, short val) {
        b[off + 1] = (byte) (val      );
        b[off    ] = (byte) (val >>> 8);
    }

    static void putInt(byte[] b, int off, int val) {
        b[off + 3] = (byte) (val       );
        b[off + 2] = (byte) (val >>>  8);
        b[off + 1] = (byte) (val >>> 16);
        b[off    ] = (byte) (val >>> 24);
    }

    static void putFloat(byte[] b, int off, float val) {
        putInt(b, off,  Float.floatToIntBits(val));
    }

    static void putLong(byte[] b, int off, long val) {
        b[off + 7] = (byte) (val       );
        b[off + 6] = (byte) (val >>>  8);
        b[off + 5] = (byte) (val >>> 16);
        b[off + 4] = (byte) (val >>> 24);
        b[off + 3] = (byte) (val >>> 32);
        b[off + 2] = (byte) (val >>> 40);
        b[off + 1] = (byte) (val >>> 48);
        b[off    ] = (byte) (val >>> 56);
    }

    static void putDouble(byte[] b, int off, double val) {
        putLong(b, off, Double.doubleToLongBits(val));
    }
}
```

BufferedInputStream.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;
import java.util.concurrent.atomic.AtomicReferenceFieldUpdater;

/**
 * A <code>BufferedInputStream</code> adds
 * functionality to another input stream-namely,
 * the ability to buffer the input and to
 * support the <code>mark</code> and <code>reset</code>
 * methods. When  the <code>BufferedInputStream</code>
 * is created, an internal buffer array is
 * created. As bytes  from the stream are read
 * or skipped, the internal buffer is refilled
 * as necessary  from the contained input stream,
 * many bytes at a time. The <code>mark</code>
 * operation  remembers a point in the input
 * stream and the <code>reset</code> operation
 * causes all the  bytes read since the most
 * recent <code>mark</code> operation to be
 * reread before new bytes are  taken from
 * the contained input stream.
 *
 * @author  Arthur van Hoff
 * @since   JDK1.0
 */
public
class BufferedInputStream extends FilterInputStream {

    private static int DEFAULT_BUFFER_SIZE = 8192;

    /**
     * The maximum size of array to allocate.
     * Some VMs reserve some header words in an array.
     * Attempts to allocate larger arrays may result in
     * OutOfMemoryError: Requested array size exceeds VM limit
     */
```

```
private static int MAX_BUFFER_SIZE = Integer.MAX_VALUE - 8;

/**
 * The internal buffer array where the data is stored. When necessary,
 * it may be replaced by another array of
 * a different size.
 */
protected volatile byte buf[];

/**
 * Atomic updater to provide compareAndSet for buf. This is
 * necessary because closes can be asynchronous. We use nullness
 * of buf[] as primary indicator that this stream is closed. (The
 * "in" field is also nulled out on close.)
 */
private static final
    AtomicReferenceFieldUpdater<BufferedInputStream, byte[]> bufUpdater =
    AtomicReferenceFieldUpdater.newUpdater
    (BufferedInputStream.class,  byte[].class, "buf");

/**
 * The index one greater than the index of the last valid byte in
 * the buffer.
 * This value is always
 * in the range <code>0</code> through <code>buf.length</code>;
 * elements <code>buf[0]</code>  through <code>buf[count-1]
 * </code>contain buffered input data obtained
 * from the underlying  input stream.
 */
protected int count;

/**
 * The current position in the buffer. This is the index of the next
 * character to be read from the <code>buf</code> array.
 * <p>
 * This value is always in the range <code>0</code>
 * through <code>count</code>. If it is less
 * than <code>count</code>, then  <code>buf[pos]</code>
 * is the next byte to be supplied as input;
 * if it is equal to <code>count</code>, then
 * the  next <code>read</code> or <code>skip</code>
 * operation will require more bytes to be
 * read from the contained  input stream.
 *
 * @see     java.io.BufferedInputStream#buf
 */
protected int pos;

/**
 * The value of the <code>pos</code> field at the time the last
 * <code>mark</code> method was called.
 * <p>
 * This value is always
 * in the range <code>-1</code> through <code>pos</code>.
 * If there is no marked position in  the input
 * stream, this field is <code>-1</code>. If
 * there is a marked position in the input
 * stream,  then <code>buf[markpos]</code>
 * is the first byte to be supplied as input
 * after a <code>reset</code> operation. If
 * <code>markpos</code> is not <code>-1</code>,
 * then all bytes from positions <code>buf[markpos]</code>
 * through  <code>buf[pos-1]</code> must remain
```

```
     * in the buffer array (though they may be
     * moved to  another place in the buffer array,
     * with suitable adjustments to the values
     * of <code>count</code>,  <code>pos</code>,
     * and <code>markpos</code>); they may not
     * be discarded unless and until the difference
     * between <code>pos</code> and <code>markpos</code>
     * exceeds <code>marklimit</code>.
     *
     * @see     java.io.BufferedInputStream#mark(int)
     * @see     java.io.BufferedInputStream#pos
     */
    protected int markpos = -1;

    /**
     * The maximum read ahead allowed after a call to the
     * <code>mark</code> method before subsequent calls to the
     * <code>reset</code> method fail.
     * Whenever the difference between <code>pos</code>
     * and <code>markpos</code> exceeds <code>marklimit</code>,
     * then the  mark may be dropped by setting
     * <code>markpos</code> to <code>-1</code>.
     *
     * @see     java.io.BufferedInputStream#mark(int)
     * @see     java.io.BufferedInputStream#reset()
     */
    protected int marklimit;

    /**
     * Check to make sure that underlying input stream has not been
     * nulled out due to close; if not return it;
     */
    private InputStream getInIfOpen() throws IOException {
        InputStream input = in;
        if (input == null)
            throw new IOException("Stream closed");
        return input;
    }

    /**
     * Check to make sure that buffer has not been nulled out due to
     * close; if not return it;
     */
    private byte[] getBufIfOpen() throws IOException {
        byte[] buffer = buf;
        if (buffer == null)
            throw new IOException("Stream closed");
        return buffer;
    }

    /**
     * Creates a <code>BufferedInputStream</code>
     * and saves its  argument, the input stream
     * <code>in</code>, for later use. An internal
     * buffer array is created and  stored in <code>buf</code>.
     *
     * @param   in   the underlying input stream.
     */
    public BufferedInputStream(InputStream in) {
        this(in, DEFAULT_BUFFER_SIZE);
    }

    /**
```

```
     * Creates a <code>BufferedInputStream</code>
     * with the specified buffer size,
     * and saves its  argument, the input stream
     * <code>in</code>, for later use.  An internal
     * buffer array of length  <code>size</code>
     * is created and stored in <code>buf</code>.
     *
     * @param   in     the underlying input stream.
     * @param   size   the buffer size.
     * @exception IllegalArgumentException if {@code size <= 0}.
     */
    public BufferedInputStream(InputStream in, int size) {
        super(in);
        if (size <= 0) {
            throw new IllegalArgumentException("Buffer size <= 0");
        }
        buf = new byte[size];
    }

    /**
     * Fills the buffer with more data, taking into account
     * shuffling and other tricks for dealing with marks.
     * Assumes that it is being called by a synchronized method.
     * This method also assumes that all data has already been read in,
     * hence pos > count.
     */
    private void fill() throws IOException {
        byte[] buffer = getBufIfOpen();
        if (markpos < 0)
            pos = 0;            /* no mark: throw away the buffer */
        else if (pos >= buffer.length)  /* no room left in buffer */
            if (markpos > 0) {  /* can throw away early part of the buffer */
                int sz = pos - markpos;
                System.arraycopy(buffer, markpos, buffer, 0, sz);
                pos = sz;
                markpos = 0;
            } else if (buffer.length >= marklimit) {
                markpos = -1;   /* buffer got too big, invalidate mark */
                pos = 0;        /* drop buffer contents */
            } else if (buffer.length >= MAX_BUFFER_SIZE) {
                throw new OutOfMemoryError("Required array size too large");
            } else {            /* grow buffer */
                int nsz = (pos <= MAX_BUFFER_SIZE - pos) ?
                        pos * 2 : MAX_BUFFER_SIZE;
                if (nsz > marklimit)
                    nsz = marklimit;
                byte nbuf[] = new byte[nsz];
                System.arraycopy(buffer, 0, nbuf, 0, pos);
                if (!bufUpdater.compareAndSet(this, buffer, nbuf)) {
                    // Can't replace buf if there was an async close.
                    // Note: This would need to be changed if fill()
                    // is ever made accessible to multiple threads.
                    // But for now, the only way CAS can fail is via close.
                    // assert buf == null;
                    throw new IOException("Stream closed");
                }
                buffer = nbuf;
            }
        count = pos;
        int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
        if (n > 0)
            count = n + pos;
    }
```

```
/**
 * See
 * the general contract of the <code>read</code>
 * method of <code>InputStream</code>.
 *
 * @return     the next byte of data, or <code>-1</code> if the end of the
 *             stream is reached.
 * @exception  IOException  if this input stream has been closed by
 *                          invoking its {@link #close()} method,
 *                          or an I/O error occurs.
 * @see        java.io.FilterInputStream#in
 */
public synchronized int read() throws IOException {
    if (pos >= count) {
        fill();
        if (pos >= count)
            return -1;
    }
    return getBufIfOpen()[pos++] & 0xff;
}

/**
 * Read characters into a portion of an array, reading from the underlying
 * stream at most once if necessary.
 */
private int read1(byte[] b, int off, int len) throws IOException {
    int avail = count - pos;
    if (avail <= 0) {
        /* If the requested length is at least as large as the buffer, and
           if there is no mark/reset activity, do not bother to copy the
           bytes into the local buffer.  In this way buffered streams will
           cascade harmlessly. */
        if (len >= getBufIfOpen().length && markpos < 0) {
            return getInIfOpen().read(b, off, len);
        }
        fill();
        avail = count - pos;
        if (avail <= 0) return -1;
    }
    int cnt = (avail < len) ? avail : len;
    System.arraycopy(getBufIfOpen(), pos, b, off, cnt);
    pos += cnt;
    return cnt;
}

/**
 * Reads bytes from this byte-input stream into the specified byte array,
 * starting at the given offset.
 *
 * <p> This method implements the general contract of the corresponding
 * <code>{@link InputStream#read(byte[], int, int) read}</code> method of
 * the <code>{@link InputStream}</code> class.  As an additional
 * convenience, it attempts to read as many bytes as possible by repeatedly
 * invoking the <code>read</code> method of the underlying stream.  This
 * iterated <code>read</code> continues until one of the following
 * conditions becomes true: <ul>
 *
 *   <li> The specified number of bytes have been read,
 *
 *   <li> The <code>read</code> method of the underlying stream returns
 *   <code>-1</code>, indicating end-of-file, or
 *
```

```
     *   <li> The <code>available</code> method of the underlying stream
     *   returns zero, indicating that further input requests would block.
     *
     * </ul> If the first <code>read</code> on the underlying stream returns
     * <code>-1</code> to indicate end-of-file then this method returns
     * <code>-1</code>.  Otherwise this method returns the number of bytes
     * actually read.
     *
     * <p> Subclasses of this class are encouraged, but not required, to
     * attempt to read as many bytes as possible in the same fashion.
     *
     * @param      b     destination buffer.
     * @param      off   offset at which to start storing bytes.
     * @param      len   maximum number of bytes to read.
     * @return     the number of bytes read, or <code>-1</code> if the end of
     *             the stream has been reached.
     * @exception  IOException  if this input stream has been closed by
     *                          invoking its {@link #close()} method,
     *                          or an I/O error occurs.
     */
    public synchronized int read(byte b[], int off, int len)
        throws IOException
    {
        getBufIfOpen(); // Check for closed stream
        if ((off | len | (off + len) | (b.length - (off + len))) < 0) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return 0;
        }

        int n = 0;
        for (;;) {
            int nread = read1(b, off + n, len - n);
            if (nread <= 0)
                return (n == 0) ? nread : n;
            n += nread;
            if (n >= len)
                return n;
            // if not closed but no bytes available, return
            InputStream input = in;
            if (input != null && input.available() <= 0)
                return n;
        }
    }

    /**
     * See the general contract of the <code>skip</code>
     * method of <code>InputStream</code>.
     *
     * @exception  IOException  if the stream does not support seek,
     *                          or if this input stream has been closed by
     *                          invoking its {@link #close()} method, or an
     *                          I/O error occurs.
     */
    public synchronized long skip(long n) throws IOException {
        getBufIfOpen(); // Check for closed stream
        if (n <= 0) {
            return 0;
        }
        long avail = count - pos;

        if (avail <= 0) {
            // If no mark position set then don't keep in buffer
```

```
        if (markpos <0)
            return getInIfOpen().skip(n);

        // Fill in buffer to save bytes for reset
        fill();
        avail = count - pos;
        if (avail <= 0)
            return 0;
    }

    long skipped = (avail < n) ? avail : n;
    pos += skipped;
    return skipped;
}

/**
 * Returns an estimate of the number of bytes that can be read (or
 * skipped over) from this input stream without blocking by the next
 * invocation of a method for this input stream. The next invocation might be
 * the same thread or another thread.  A single read or skip of this
 * many bytes will not block, but may read or skip fewer bytes.
 * <p>
 * This method returns the sum of the number of bytes remaining to be read in
 * the buffer (<code>count - pos</code>) and the result of calling the
 * {@link java.io.FilterInputStream#in in}.available().
 *
 * @return     an estimate of the number of bytes that can be read (or skipped
 *             over) from this input stream without blocking.
 * @exception  IOException  if this input stream has been closed by
 *                          invoking its {@link #close()} method,
 *                          or an I/O error occurs.
 */
public synchronized int available() throws IOException {
    int n = count - pos;
    int avail = getInIfOpen().available();
    return n > (Integer.MAX_VALUE - avail)
                ? Integer.MAX_VALUE
                : n + avail;
}

/**
 * See the general contract of the <code>mark</code>
 * method of <code>InputStream</code>.
 *
 * @param   readlimit   the maximum limit of bytes that can be read before
 *                      the mark position becomes invalid.
 * @see     java.io.BufferedInputStream#reset()
 */
public synchronized void mark(int readlimit) {
    marklimit = readlimit;
    markpos = pos;
}

/**
 * See the general contract of the <code>reset</code>
 * method of <code>InputStream</code>.
 * <p>
 * If <code>markpos</code> is <code>-1</code>
 * (no mark has been set or the mark has been
 * invalidated), an <code>IOException</code>
 * is thrown. Otherwise, <code>pos</code> is
 * set equal to <code>markpos</code>.
 *
```

```
     * @exception  IOException  if this stream has not been marked or,
     *                  if the mark has been invalidated, or the stream
     *                  has been closed by invoking its {@link #close()}
     *                  method, or an I/O error occurs.
     * @see        java.io.BufferedInputStream#mark(int)
     */
    public synchronized void reset() throws IOException {
        getBufIfOpen(); // Cause exception if closed
        if (markpos < 0)
            throw new IOException("Resetting to invalid mark");
        pos = markpos;
    }

    /**
     * Tests if this input stream supports the <code>mark</code>
     * and <code>reset</code> methods. The <code>markSupported</code>
     * method of <code>BufferedInputStream</code> returns
     * <code>true</code>.
     *
     * @return  a <code>boolean</code> indicating if this stream type supports
     *          the <code>mark</code> and <code>reset</code> methods.
     * @see     java.io.InputStream#mark(int)
     * @see     java.io.InputStream#reset()
     */
    public boolean markSupported() {
        return true;
    }

    /**
     * Closes this input stream and releases any system resources
     * associated with the stream.
     * Once the stream has been closed, further read(), available(), reset(),
     * or skip() invocations will throw an IOException.
     * Closing a previously closed stream has no effect.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void close() throws IOException {
        byte[] buffer;
        while ( (buffer = buf) != null) {
            if (bufUpdater.compareAndSet(this, buffer, null)) {
                InputStream input = in;
                in = null;
                if (input != null)
                    input.close();
                return;
            }
            // Else retry in case a new buf was CASed in fill()
        }
    }
}
```

BufferedOutputStream.java

```
/*
 * Copyright (c) 1994, 2003, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * The class implements a buffered output stream. By setting up such
 * an output stream, an application can write bytes to the underlying
 * output stream without necessarily causing a call to the underlying
 * system for each byte written.
 *
 * @author  Arthur van Hoff
 * @since   JDK1.0
 */
public
class BufferedOutputStream extends FilterOutputStream {
    /**
     * The internal buffer where data is stored.
     */
    protected byte buf[];

    /**
     * The number of valid bytes in the buffer. This value is always
     * in the range <tt>0</tt> through <tt>buf.length</tt>; elements
     * <tt>buf[0]</tt> through <tt>buf[count-1]</tt> contain valid
     * byte data.
     */
    protected int count;

    /**
     * Creates a new buffered output stream to write data to the
     * specified underlying output stream.
     *
     * @param   out   the underlying output stream.
     */
    public BufferedOutputStream(OutputStream out) {
        this(out, 8192);
    }
```

```
/**
 * Creates a new buffered output stream to write data to the
 * specified underlying output stream with the specified buffer
 * size.
 *
 * @param   out    the underlying output stream.
 * @param   size   the buffer size.
 * @exception IllegalArgumentException if size <= 0.
 */
public BufferedOutputStream(OutputStream out, int size) {
    super(out);
    if (size <= 0) {
        throw new IllegalArgumentException("Buffer size <= 0");
    }
    buf = new byte[size];
}

/** Flush the internal buffer */
private void flushBuffer() throws IOException {
    if (count > 0) {
        out.write(buf, 0, count);
        count = 0;
    }
}

/**
 * Writes the specified byte to this buffered output stream.
 *
 * @param      b   the byte to be written.
 * @exception  IOException  if an I/O error occurs.
 */
public synchronized void write(int b) throws IOException {
    if (count >= buf.length) {
        flushBuffer();
    }
    buf[count++] = (byte)b;
}

/**
 * Writes <code>len</code> bytes from the specified byte array
 * starting at offset <code>off</code> to this buffered output stream.
 *
 * <p> Ordinarily this method stores bytes from the given array into this
 * stream's buffer, flushing the buffer to the underlying output stream as
 * needed.  If the requested length is at least as large as this stream's
 * buffer, however, then this method will flush the buffer and write the
 * bytes directly to the underlying output stream.  Thus redundant
 * <code>BufferedOutputStream</code>s will not copy data unnecessarily.
 *
 * @param      b     the data.
 * @param      off   the start offset in the data.
 * @param      len   the number of bytes to write.
 * @exception  IOException  if an I/O error occurs.
 */
public synchronized void write(byte b[], int off, int len) throws IOException {
    if (len >= buf.length) {
        /* If the request length exceeds the size of the output buffer,
           flush the output buffer and then write the data directly.
           In this way buffered streams will cascade harmlessly. */
        flushBuffer();
        out.write(b, off, len);
        return;
```

```
        }
        if (len > buf.length - count) {
            flushBuffer();
        }
        System.arraycopy(b, off, buf, count, len);
        count += len;
    }

    /**
     * Flushes this buffered output stream. This forces any buffered
     * output bytes to be written out to the underlying output stream.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#out
     */
    public synchronized void flush() throws IOException {
        flushBuffer();
        out.flush();
    }
}
```

BufferedReader.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Spliterator;
import java.util.Spliterators;
import java.util.stream.Stream;
import java.util.stream.StreamSupport;

/**
 * Reads text from a character-input stream, buffering characters so as to
 * provide for the efficient reading of characters, arrays, and lines.
 *
 * <p> The buffer size may be specified, or the default size may be used.  The
 * default is large enough for most purposes.
 *
 * <p> In general, each read request made of a Reader causes a corresponding
 * read request to be made of the underlying character or byte stream.  It is
 * therefore advisable to wrap a BufferedReader around any Reader whose read()
 * operations may be costly, such as FileReaders and InputStreamReaders.  For
 * example,
 *
 * <pre>
 * BufferedReader in
 *   = new BufferedReader(new FileReader("foo.in"));
 * </pre>
 *
 * will buffer the input from the specified file.  Without buffering, each
 * invocation of read() or readLine() could cause bytes to be read from the
 * file, converted into characters, and then returned, which can be very
 * inefficient.
 *
 * <p> Programs that use DataInputStreams for textual input can be localized by
 * replacing each DataInputStream with an appropriate BufferedReader.
```

```
 *
 * @see FileReader
 * @see InputStreamReader
 * @see java.nio.file.Files#newBufferedReader
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class BufferedReader extends Reader {

    private Reader in;

    private char cb[];
    private int nChars, nextChar;

    private static final int INVALIDATED = -2;
    private static final int UNMARKED = -1;
    private int markedChar = UNMARKED;
    private int readAheadLimit = 0; /* Valid only when markedChar > 0 */

    /** If the next character is a line feed, skip it */
    private boolean skipLF = false;

    /** The skipLF flag when the mark was set */
    private boolean markedSkipLF = false;

    private static int defaultCharBufferSize = 8192;
    private static int defaultExpectedLineLength = 80;

    /**
     * Creates a buffering character-input stream that uses an input buffer of
     * the specified size.
     *
     * @param  in   A Reader
     * @param  sz   Input-buffer size
     *
     * @exception  IllegalArgumentException  If {@code sz <= 0}
     */
    public BufferedReader(Reader in, int sz) {
        super(in);
        if (sz <= 0)
            throw new IllegalArgumentException("Buffer size <= 0");
        this.in = in;
        cb = new char[sz];
        nextChar = nChars = 0;
    }

    /**
     * Creates a buffering character-input stream that uses a default-sized
     * input buffer.
     *
     * @param  in   A Reader
     */
    public BufferedReader(Reader in) {
        this(in, defaultCharBufferSize);
    }

    /** Checks to make sure that the stream has not been closed */
    private void ensureOpen() throws IOException {
        if (in == null)
            throw new IOException("Stream closed");
    }
```

```
/**
 * Fills the input buffer, taking the mark into account if it is valid.
 */
private void fill() throws IOException {
    int dst;
    if (markedChar <= UNMARKED) {
        /* No mark */
        dst = 0;
    } else {
        /* Marked */
        int delta = nextChar - markedChar;
        if (delta >= readAheadLimit) {
            /* Gone past read-ahead limit: Invalidate mark */
            markedChar = INVALIDATED;
            readAheadLimit = 0;
            dst = 0;
        } else {
            if (readAheadLimit <= cb.length) {
                /* Shuffle in the current buffer */
                System.arraycopy(cb, markedChar, cb, 0, delta);
                markedChar = 0;
                dst = delta;
            } else {
                /* Reallocate buffer to accommodate read-ahead limit */
                char ncb[] = new char[readAheadLimit];
                System.arraycopy(cb, markedChar, ncb, 0, delta);
                cb = ncb;
                markedChar = 0;
                dst = delta;
            }
            nextChar = nChars = delta;
        }
    }

    int n;
    do {
        n = in.read(cb, dst, cb.length - dst);
    } while (n == 0);
    if (n > 0) {
        nChars = dst + n;
        nextChar = dst;
    }
}

/**
 * Reads a single character.
 *
 * @return The character read, as an integer in the range
 *         0 to 65535 (<tt>0x00-0xffff</tt>), or -1 if the
 *         end of the stream has been reached
 * @exception  IOException  If an I/O error occurs
 */
public int read() throws IOException {
    synchronized (lock) {
        ensureOpen();
        for (;;) {
            if (nextChar >= nChars) {
                fill();
                if (nextChar >= nChars)
                    return -1;
            }
            if (skipLF) {
```

```
                skipLF = false;
                if (cb[nextChar] == '\n') {
                    nextChar++;
                    continue;
                }
            }
            return cb[nextChar++];
        }
    }
}

/**
 * Reads characters into a portion of an array, reading from the underlying
 * stream if necessary.
 */
private int read1(char[] cbuf, int off, int len) throws IOException {
    if (nextChar >= nChars) {
        /* If the requested length is at least as large as the buffer, and
           if there is no mark/reset activity, and if line feeds are not
           being skipped, do not bother to copy the characters into the
           local buffer.  In this way buffered streams will cascade
           harmlessly. */
        if (len >= cb.length && markedChar <= UNMARKED && !skipLF) {
            return in.read(cbuf, off, len);
        }
        fill();
    }
    if (nextChar >= nChars) return -1;
    if (skipLF) {
        skipLF = false;
        if (cb[nextChar] == '\n') {
            nextChar++;
            if (nextChar >= nChars)
                fill();
            if (nextChar >= nChars)
                return -1;
        }
    }
    int n = Math.min(len, nChars - nextChar);
    System.arraycopy(cb, nextChar, cbuf, off, n);
    nextChar += n;
    return n;
}

/**
 * Reads characters into a portion of an array.
 *
 * <p> This method implements the general contract of the corresponding
 * <code>{@link Reader#read(char[], int, int) read}</code> method of the
 * <code>{@link Reader}</code> class.  As an additional convenience, it
 * attempts to read as many characters as possible by repeatedly invoking
 * the <code>read</code> method of the underlying stream.  This iterated
 * <code>read</code> continues until one of the following conditions becomes
 * true: <ul>
 *
 *   <li> The specified number of characters have been read,
 *
 *   <li> The <code>read</code> method of the underlying stream returns
 *   <code>-1</code>, indicating end-of-file, or
 *
 *   <li> The <code>ready</code> method of the underlying stream
 *   returns <code>false</code>, indicating that further input requests
 *   would block.
```

```
     *
     * </ul> If the first <code>read</code> on the underlying stream returns
     * <code>-1</code> to indicate end-of-file then this method returns
     * <code>-1</code>.  Otherwise this method returns the number of characters
     * actually read.
     *
     * <p> Subclasses of this class are encouraged, but not required, to
     * attempt to read as many characters as possible in the same fashion.
     *
     * <p> Ordinarily this method takes characters from this stream's character
     * buffer, filling it from the underlying stream as necessary.  If,
     * however, the buffer is empty, the mark is not valid, and the requested
     * length is at least as large as the buffer, then this method will read
     * characters directly from the underlying stream into the given array.
     * Thus redundant <code>BufferedReader</code>s will not copy data
     * unnecessarily.
     *
     * @param      cbuf  Destination buffer
     * @param      off   Offset at which to start storing characters
     * @param      len   Maximum number of characters to read
     *
     * @return     The number of characters read, or -1 if the end of the
     *             stream has been reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read(char cbuf[], int off, int len) throws IOException {
        synchronized (lock) {
            ensureOpen();
            if ((off < 0) || (off > cbuf.length) || (len < 0) ||
                ((off + len) > cbuf.length) || ((off + len) < 0)) {
                throw new IndexOutOfBoundsException();
            } else if (len == 0) {
                return 0;
            }

            int n = read1(cbuf, off, len);
            if (n <= 0) return n;
            while ((n < len) && in.ready()) {
                int n1 = read1(cbuf, off + n, len - n);
                if (n1 <= 0) break;
                n += n1;
            }
            return n;
        }
    }

    /**
     * Reads a line of text.  A line is considered to be terminated by any one
     * of a line feed ('\n'), a carriage return ('\r'), or a carriage return
     * followed immediately by a linefeed.
     *
     * @param      ignoreLF  If true, the next '\n' will be skipped
     *
     * @return     A String containing the contents of the line, not including
     *             any line-termination characters, or null if the end of the
     *             stream has been reached
     *
     * @see        java.io.LineNumberReader#readLine()
     *
     * @exception  IOException  If an I/O error occurs
     */
    String readLine(boolean ignoreLF) throws IOException {
```

```
    StringBuffer s = null;
    int startChar;

    synchronized (lock) {
        ensureOpen();
        boolean omitLF = ignoreLF || skipLF;

    bufferLoop:
        for (;;) {

            if (nextChar >= nChars)
                fill();
            if (nextChar >= nChars) { /* EOF */
                if (s != null && s.length() > 0)
                    return s.toString();
                else
                    return null;
            }
            boolean eol = false;
            char c = 0;
            int i;

            /* Skip a leftover '\n', if necessary */
            if (omitLF && (cb[nextChar] == '\n'))
                nextChar++;
            skipLF = false;
            omitLF = false;

        charLoop:
            for (i = nextChar; i < nChars; i++) {
                c = cb[i];
                if ((c == '\n') || (c == '\r')) {
                    eol = true;
                    break charLoop;
                }
            }

            startChar = nextChar;
            nextChar = i;

            if (eol) {
                String str;
                if (s == null) {
                    str = new String(cb, startChar, i - startChar);
                } else {
                    s.append(cb, startChar, i - startChar);
                    str = s.toString();
                }
                nextChar++;
                if (c == '\r') {
                    skipLF = true;
                }
                return str;
            }

            if (s == null)
                s = new StringBuffer(defaultExpectedLineLength);
            s.append(cb, startChar, i - startChar);
        }
    }
}

/**
```

```
 * Reads a line of text.  A line is considered to be terminated by any one
 * of a line feed ('\n'), a carriage return ('\r'), or a carriage return
 * followed immediately by a linefeed.
 *
 * @return     A String containing the contents of the line, not including
 *             any line-termination characters, or null if the end of the
 *             stream has been reached
 *
 * @exception  IOException  If an I/O error occurs
 *
 * @see java.nio.file.Files#readAllLines
 */
public String readLine() throws IOException {
    return readLine(false);
}

/**
 * Skips characters.
 *
 * @param  n  The number of characters to skip
 *
 * @return    The number of characters actually skipped
 *
 * @exception  IllegalArgumentException  If <code>n</code> is negative.
 * @exception  IOException  If an I/O error occurs
 */
public long skip(long n) throws IOException {
    if (n < 0L) {
        throw new IllegalArgumentException("skip value is negative");
    }
    synchronized (lock) {
        ensureOpen();
        long r = n;
        while (r > 0) {
            if (nextChar >= nChars)
                fill();
            if (nextChar >= nChars) /* EOF */
                break;
            if (skipLF) {
                skipLF = false;
                if (cb[nextChar] == '\n') {
                    nextChar++;
                }
            }
            long d = nChars - nextChar;
            if (r <= d) {
                nextChar += r;
                r = 0;
                break;
            }
            else {
                r -= d;
                nextChar = nChars;
            }
        }
        return n - r;
    }
}

/**
 * Tells whether this stream is ready to be read.  A buffered character
 * stream is ready if the buffer is not empty, or if the underlying
 * character stream is ready.
```

```
     *
     * @exception  IOException  If an I/O error occurs
     */
    public boolean ready() throws IOException {
        synchronized (lock) {
            ensureOpen();

            /*
             * If newline needs to be skipped and the next char to be read
             * is a newline character, then just skip it right away.
             */
            if (skipLF) {
                /* Note that in.ready() will return true if and only if the next
                 * read on the stream will not block.
                 */
                if (nextChar >= nChars && in.ready()) {
                    fill();
                }
                if (nextChar < nChars) {
                    if (cb[nextChar] == '\n')
                        nextChar++;
                    skipLF = false;
                }
            }
            return (nextChar < nChars) || in.ready();
        }
    }

    /**
     * Tells whether this stream supports the mark() operation, which it does.
     */
    public boolean markSupported() {
        return true;
    }

    /**
     * Marks the present position in the stream.  Subsequent calls to reset()
     * will attempt to reposition the stream to this point.
     *
     * @param readAheadLimit   Limit on the number of characters that may be
     *                         read while still preserving the mark. An attempt
     *                         to reset the stream after reading characters
     *                         up to this limit or beyond may fail.
     *                         A limit value larger than the size of the input
     *                         buffer will cause a new buffer to be allocated
     *                         whose size is no smaller than limit.
     *                         Therefore large values should be used with care.
     *
     * @exception  IllegalArgumentException  If {@code readAheadLimit < 0}
     * @exception  IOException  If an I/O error occurs
     */
    public void mark(int readAheadLimit) throws IOException {
        if (readAheadLimit < 0) {
            throw new IllegalArgumentException("Read-ahead limit < 0");
        }
        synchronized (lock) {
            ensureOpen();
            this.readAheadLimit = readAheadLimit;
            markedChar = nextChar;
            markedSkipLF = skipLF;
        }
    }
```

```
/**
 * Resets the stream to the most recent mark.
 *
 * @exception  IOException  If the stream has never been marked,
 *                          or if the mark has been invalidated
 */
public void reset() throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (markedChar < 0)
            throw new IOException((markedChar == INVALIDATED)
                                  ? "Mark invalid"
                                  : "Stream not marked");
        nextChar = markedChar;
        skipLF = markedSkipLF;
    }
}

public void close() throws IOException {
    synchronized (lock) {
        if (in == null)
            return;
        try {
            in.close();
        } finally {
            in = null;
            cb = null;
        }
    }
}

/**
 * Returns a {@code Stream}, the elements of which are lines read from
 * this {@code BufferedReader}.  The {@link Stream} is lazily populated,
 * i.e., read only occurs during the
 * <a href="../util/stream/package-summary.html#StreamOps">terminal
 * stream operation</a>.
 *
 * <p> The reader must not be operated on during the execution of the
 * terminal stream operation. Otherwise, the result of the terminal stream
 * operation is undefined.
 *
 * <p> After execution of the terminal stream operation there are no
 * guarantees that the reader will be at a specific position from which to
 * read the next character or line.
 *
 * <p> If an {@link IOException} is thrown when accessing the underlying
 * {@code BufferedReader}, it is wrapped in an {@link
 * UncheckedIOException} which will be thrown from the {@code Stream}
 * method that caused the read to take place. This method will return a
 * Stream if invoked on a BufferedReader that is closed. Any operation on
 * that stream that requires reading from the BufferedReader after it is
 * closed, will cause an UncheckedIOException to be thrown.
 *
 * @return a {@code Stream<String>} providing the lines of text
 *         described by this {@code BufferedReader}
 *
 * @since 1.8
 */
public Stream<String> lines() {
    Iterator<String> iter = new Iterator<String>() {
        String nextLine = null;
```

```
            @Override
            public boolean hasNext() {
                if (nextLine != null) {
                    return true;
                } else {
                    try {
                        nextLine = readLine();
                        return (nextLine != null);
                    } catch (IOException e) {
                        throw new UncheckedIOException(e);
                    }
                }
            }

            @Override
            public String next() {
                if (nextLine != null || hasNext()) {
                    String line = nextLine;
                    nextLine = null;
                    return line;
                } else {
                    throw new NoSuchElementException();
                }
            }
        };
        return StreamSupport.stream(Spliterators.spliteratorUnknownSize(
                iter, Spliterator.ORDERED | Spliterator.NONNULL), false);
    }
}
```

BufferedWriter.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Writes text to a character-output stream, buffering characters so as to
 * provide for the efficient writing of single characters, arrays, and strings.
 *
 * <p> The buffer size may be specified, or the default size may be accepted.
 * The default is large enough for most purposes.
 *
 * <p> A newLine() method is provided, which uses the platform's own notion of
 * line separator as defined by the system property <tt>line.separator</tt>.
 * Not all platforms use the newline character ('\n') to terminate lines.
 * Calling this method to terminate each output line is therefore preferred to
 * writing a newline character directly.
 *
 * <p> In general, a Writer sends its output immediately to the underlying
 * character or byte stream.  Unless prompt output is required, it is advisable
 * to wrap a BufferedWriter around any Writer whose write() operations may be
 * costly, such as FileWriters and OutputStreamWriters.  For example,
 *
 * <pre>
 * PrintWriter out
 *   = new PrintWriter(new BufferedWriter(new FileWriter("foo.out")));
 * </pre>
 *
 * will buffer the PrintWriter's output to the file.  Without buffering, each
 * invocation of a print() method would cause characters to be converted into
 * bytes that would then be written immediately to the file, which can be very
 * inefficient.
 *
 * @see PrintWriter
 * @see FileWriter
 * @see OutputStreamWriter
 * @see java.nio.file.Files#newBufferedWriter
```

```
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class BufferedWriter extends Writer {

    private Writer out;

    private char cb[];
    private int nChars, nextChar;

    private static int defaultCharBufferSize = 8192;

    /**
     * Line separator string.  This is the value of the line.separator
     * property at the moment that the stream was created.
     */
    private String lineSeparator;

    /**
     * Creates a buffered character-output stream that uses a default-sized
     * output buffer.
     *
     * @param  out  A Writer
     */
    public BufferedWriter(Writer out) {
        this(out, defaultCharBufferSize);
    }

    /**
     * Creates a new buffered character-output stream that uses an output
     * buffer of the given size.
     *
     * @param  out  A Writer
     * @param  sz   Output-buffer size, a positive integer
     *
     * @exception  IllegalArgumentException  If {@code sz <= 0}
     */
    public BufferedWriter(Writer out, int sz) {
        super(out);
        if (sz <= 0)
            throw new IllegalArgumentException("Buffer size <= 0");
        this.out = out;
        cb = new char[sz];
        nChars = sz;
        nextChar = 0;

        lineSeparator = java.security.AccessController.doPrivileged(
            new sun.security.action.GetPropertyAction("line.separator"));
    }

    /** Checks to make sure that the stream has not been closed */
    private void ensureOpen() throws IOException {
        if (out == null)
            throw new IOException("Stream closed");
    }

    /**
     * Flushes the output buffer to the underlying character stream, without
     * flushing the stream itself.  This method is non-private only so that it
     * may be invoked by PrintStream.
     */
```

```
void flushBuffer() throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (nextChar == 0)
            return;
        out.write(cb, 0, nextChar);
        nextChar = 0;
    }
}

/**
 * Writes a single character.
 *
 * @exception  IOException  If an I/O error occurs
 */
public void write(int c) throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (nextChar >= nChars)
            flushBuffer();
        cb[nextChar++] = (char) c;
    }
}

/**
 * Our own little min method, to avoid loading java.lang.Math if we've run
 * out of file descriptors and we're trying to print a stack trace.
 */
private int min(int a, int b) {
    if (a < b) return a;
    return b;
}

/**
 * Writes a portion of an array of characters.
 *
 * <p> Ordinarily this method stores characters from the given array into
 * this stream's buffer, flushing the buffer to the underlying stream as
 * needed.  If the requested length is at least as large as the buffer,
 * however, then this method will flush the buffer and write the characters
 * directly to the underlying stream.  Thus redundant
 * <code>BufferedWriter</code>s will not copy data unnecessarily.
 *
 * @param  cbuf  A character array
 * @param  off   Offset from which to start reading characters
 * @param  len   Number of characters to write
 *
 * @exception  IOException  If an I/O error occurs
 */
public void write(char cbuf[], int off, int len) throws IOException {
    synchronized (lock) {
        ensureOpen();
        if ((off < 0) || (off > cbuf.length) || (len < 0) ||
            ((off + len) > cbuf.length) || ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return;
        }

        if (len >= nChars) {
            /* If the request length exceeds the size of the output buffer,
               flush the buffer and then write the data directly.  In this
               way buffered streams will cascade harmlessly. */
```

```
            flushBuffer();
            out.write(cbuf, off, len);
            return;
        }

        int b = off, t = off + len;
        while (b < t) {
            int d = min(nChars - nextChar, t - b);
            System.arraycopy(cbuf, b, cb, nextChar, d);
            b += d;
            nextChar += d;
            if (nextChar >= nChars)
                flushBuffer();
        }
    }
}

/**
 * Writes a portion of a String.
 *
 * <p> If the value of the <tt>len</tt> parameter is negative then no
 * characters are written.  This is contrary to the specification of this
 * method in the {@linkplain java.io.Writer#write(java.lang.String,int,int)
 * superclass}, which requires that an {@link IndexOutOfBoundsException} be
 * thrown.
 *
 * @param  s     String to be written
 * @param  off   Offset from which to start reading characters
 * @param  len   Number of characters to be written
 *
 * @exception  IOException  If an I/O error occurs
 */
public void write(String s, int off, int len) throws IOException {
    synchronized (lock) {
        ensureOpen();

        int b = off, t = off + len;
        while (b < t) {
            int d = min(nChars - nextChar, t - b);
            s.getChars(b, b + d, cb, nextChar);
            b += d;
            nextChar += d;
            if (nextChar >= nChars)
                flushBuffer();
        }
    }
}

/**
 * Writes a line separator.  The line separator string is defined by the
 * system property <tt>line.separator</tt>, and is not necessarily a single
 * newline ('\n') character.
 *
 * @exception  IOException  If an I/O error occurs
 */
public void newLine() throws IOException {
    write(lineSeparator);
}

/**
 * Flushes the stream.
 *
 * @exception  IOException  If an I/O error occurs
```

```
     */
    public void flush() throws IOException {
        synchronized (lock) {
            flushBuffer();
            out.flush();
        }
    }

    @SuppressWarnings("try")
    public void close() throws IOException {
        synchronized (lock) {
            if (out == null) {
                return;
            }
            try (Writer w = out) {
                flushBuffer();
            } finally {
                out = null;
                cb = null;
            }
        }
    }
}
```

ByteArrayInputStream.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * A <code>ByteArrayInputStream</code> contains
 * an internal buffer that contains bytes that
 * may be read from the stream. An internal
 * counter keeps track of the next byte to
 * be supplied by the <code>read</code> method.
 * <p>
 * Closing a <tt>ByteArrayInputStream</tt> has no effect. The methods in
 * this class can be called after the stream has been closed without
 * generating an <tt>IOException</tt>.
 *
 * @author  Arthur van Hoff
 * @see     java.io.StringBufferInputStream
 * @since   JDK1.0
 */
public
class ByteArrayInputStream extends InputStream {

    /**
     * An array of bytes that was provided
     * by the creator of the stream. Elements <code>buf[0]</code>
     * through <code>buf[count-1]</code> are the
     * only bytes that can ever be read from the
     * stream;  element <code>buf[pos]</code> is
     * the next byte to be read.
     */
    protected byte buf[];

    /**
     * The index of the next character to read from the input stream buffer.
     * This value should always be nonnegative
     * and not larger than the value of <code>count</code>.
     * The next byte to be read from the input stream buffer
```

```
     * will be <code>buf[pos]</code>.
     */
    protected int pos;

    /**
     * The currently marked position in the stream.
     * ByteArrayInputStream objects are marked at position zero by
     * default when constructed.  They may be marked at another
     * position within the buffer by the <code>mark()</code> method.
     * The current buffer position is set to this point by the
     * <code>reset()</code> method.
     * <p>
     * If no mark has been set, then the value of mark is the offset
     * passed to the constructor (or 0 if the offset was not supplied).
     *
     * @since   JDK1.1
     */
    protected int mark = 0;

    /**
     * The index one greater than the last valid character in the input
     * stream buffer.
     * This value should always be nonnegative
     * and not larger than the length of <code>buf</code>.
     * It  is one greater than the position of
     * the last byte within <code>buf</code> that
     * can ever be read  from the input stream buffer.
     */
    protected int count;

    /**
     * Creates a <code>ByteArrayInputStream</code>
     * so that it  uses <code>buf</code> as its
     * buffer array.
     * The buffer array is not copied.
     * The initial value of <code>pos</code>
     * is <code>0</code> and the initial value
     * of  <code>count</code> is the length of
     * <code>buf</code>.
     *
     * @param   buf   the input buffer.
     */
    public ByteArrayInputStream(byte buf[]) {
        this.buf = buf;
        this.pos = 0;
        this.count = buf.length;
    }

    /**
     * Creates <code>ByteArrayInputStream</code>
     * that uses <code>buf</code> as its
     * buffer array. The initial value of <code>pos</code>
     * is <code>offset</code> and the initial value
     * of <code>count</code> is the minimum of <code>offset+length</code>
     * and <code>buf.length</code>.
     * The buffer array is not copied. The buffer's mark is
     * set to the specified offset.
     *
     * @param   buf      the input buffer.
     * @param   offset   the offset in the buffer of the first byte to read.
     * @param   length   the maximum number of bytes to read from the buffer.
     */
    public ByteArrayInputStream(byte buf[], int offset, int length) {
```

```
    this.buf = buf;
    this.pos = offset;
    this.count = Math.min(offset + length, buf.length);
    this.mark = offset;
}

/**
 * Reads the next byte of data from this input stream. The value
 * byte is returned as an <code>int</code> in the range
 * <code>0</code> to <code>255</code>. If no byte is available
 * because the end of the stream has been reached, the value
 * <code>-1</code> is returned.
 * <p>
 * This <code>read</code> method
 * cannot block.
 *
 * @return  the next byte of data, or <code>-1</code> if the end of the
 *          stream has been reached.
 */
public synchronized int read() {
    return (pos < count) ? (buf[pos++] & 0xff) : -1;
}

/**
 * Reads up to <code>len</code> bytes of data into an array of bytes
 * from this input stream.
 * If <code>pos</code> equals <code>count</code>,
 * then <code>-1</code> is returned to indicate
 * end of file. Otherwise, the  number <code>k</code>
 * of bytes read is equal to the smaller of
 * <code>len</code> and <code>count-pos</code>.
 * If <code>k</code> is positive, then bytes
 * <code>buf[pos]</code> through <code>buf[pos+k-1]</code>
 * are copied into <code>b[off]</code>  through
 * <code>b[off+k-1]</code> in the manner performed
 * by <code>System.arraycopy</code>. The
 * value <code>k</code> is added into <code>pos</code>
 * and <code>k</code> is returned.
 * <p>
 * This <code>read</code> method cannot block.
 *
 * @param   b     the buffer into which the data is read.
 * @param   off   the start offset in the destination array <code>b</code>
 * @param   len   the maximum number of bytes read.
 * @return  the total number of bytes read into the buffer, or
 *          <code>-1</code> if there is no more data because the end of
 *          the stream has been reached.
 * @exception  NullPointerException If <code>b</code> is <code>null</code>.
 * @exception  IndexOutOfBoundsException If <code>off</code> is negative,
 * <code>len</code> is negative, or <code>len</code> is greater than
 * <code>b.length - off</code>
 */
public synchronized int read(byte b[], int off, int len) {
    if (b == null) {
        throw new NullPointerException();
    } else if (off < 0 || len < 0 || len > b.length - off) {
        throw new IndexOutOfBoundsException();
    }

    if (pos >= count) {
        return -1;
    }
```

```
    int avail = count - pos;
    if (len > avail) {
        len = avail;
    }
    if (len <= 0) {
        return 0;
    }
    System.arraycopy(buf, pos, b, off, len);
    pos += len;
    return len;
}

/**
 * Skips <code>n</code> bytes of input from this input stream. Fewer
 * bytes might be skipped if the end of the input stream is reached.
 * The actual number <code>k</code>
 * of bytes to be skipped is equal to the smaller
 * of <code>n</code> and  <code>count-pos</code>.
 * The value <code>k</code> is added into <code>pos</code>
 * and <code>k</code> is returned.
 *
 * @param   n   the number of bytes to be skipped.
 * @return  the actual number of bytes skipped.
 */
public synchronized long skip(long n) {
    long k = count - pos;
    if (n < k) {
        k = n < 0 ? 0 : n;
    }

    pos += k;
    return k;
}

/**
 * Returns the number of remaining bytes that can be read (or skipped over)
 * from this input stream.
 * <p>
 * The value returned is <code>count - pos</code>,
 * which is the number of bytes remaining to be read from the input buffer.
 *
 * @return  the number of remaining bytes that can be read (or skipped
 *          over) from this input stream without blocking.
 */
public synchronized int available() {
    return count - pos;
}

/**
 * Tests if this <code>InputStream</code> supports mark/reset. The
 * <code>markSupported</code> method of <code>ByteArrayInputStream</code>
 * always returns <code>true</code>.
 *
 * @since   JDK1.1
 */
public boolean markSupported() {
    return true;
}

/**
 * Set the current marked position in the stream.
 * ByteArrayInputStream objects are marked at position zero by
 * default when constructed.  They may be marked at another
```

```
     * position within the buffer by this method.
     * <p>
     * If no mark has been set, then the value of the mark is the
     * offset passed to the constructor (or 0 if the offset was not
     * supplied).
     *
     * <p> Note: The <code>readAheadLimit</code> for this class
     *  has no meaning.
     *
     * @since   JDK1.1
     */
    public void mark(int readAheadLimit) {
        mark = pos;
    }

    /**
     * Resets the buffer to the marked position.  The marked position
     * is 0 unless another position was marked or an offset was specified
     * in the constructor.
     */
    public synchronized void reset() {
        pos = mark;
    }

    /**
     * Closing a <tt>ByteArrayInputStream</tt> has no effect. The methods in
     * this class can be called after the stream has been closed without
     * generating an <tt>IOException</tt>.
     */
    public void close() throws IOException {
    }

}
```

ByteArrayOutputStream.java

```java
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.util.Arrays;

/**
 * This class implements an output stream in which the data is
 * written into a byte array. The buffer automatically grows as data
 * is written to it.
 * The data can be retrieved using <code>toByteArray()</code> and
 * <code>toString()</code>.
 * <p>
 * Closing a <tt>ByteArrayOutputStream</tt> has no effect. The methods in
 * this class can be called after the stream has been closed without
 * generating an <tt>IOException</tt>.
 *
 * @author  Arthur van Hoff
 * @since   JDK1.0
 */

public class ByteArrayOutputStream extends OutputStream {

    /**
     * The buffer where data is stored.
     */
    protected byte buf[];

    /**
     * The number of valid bytes in the buffer.
     */
    protected int count;

    /**
     * Creates a new byte array output stream. The buffer capacity is
     * initially 32 bytes, though its size increases if necessary.
     */
```

```
public ByteArrayOutputStream() {
    this(32);
}

/**
 * Creates a new byte array output stream, with a buffer capacity of
 * the specified size, in bytes.
 *
 * @param   size   the initial size.
 * @exception  IllegalArgumentException if size is negative.
 */
public ByteArrayOutputStream(int size) {
    if (size < 0) {
        throw new IllegalArgumentException("Negative initial size: "
                                           + size);
    }
    buf = new byte[size];
}

/**
 * Increases the capacity if necessary to ensure that it can hold
 * at least the number of elements specified by the minimum
 * capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 * @throws OutOfMemoryError if {@code minCapacity < 0}.  This is
 * interpreted as a request for the unsatisfiably large capacity
 * {@code (long) Integer.MAX_VALUE + (minCapacity - Integer.MAX_VALUE)}.
 */
private void ensureCapacity(int minCapacity) {
    // overflow-conscious code
    if (minCapacity - buf.length > 0)
        grow(minCapacity);
}

/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = buf.length;
    int newCapacity = oldCapacity << 1;
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity < 0) {
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        newCapacity = Integer.MAX_VALUE;
    }
    buf = Arrays.copyOf(buf, newCapacity);
}

/**
 * Writes the specified byte to this byte array output stream.
 *
 * @param   b   the byte to be written.
 */
public synchronized void write(int b) {
    ensureCapacity(count + 1);
    buf[count] = (byte) b;
```

```
    count += 1;
}

/**
 * Writes <code>len</code> bytes from the specified byte array
 * starting at offset <code>off</code> to this byte array output stream.
 *
 * @param   b     the data.
 * @param   off   the start offset in the data.
 * @param   len   the number of bytes to write.
 */
public synchronized void write(byte b[], int off, int len) {
    if ((off < 0) || (off > b.length) || (len < 0) ||
        ((off + len) - b.length > 0)) {
        throw new IndexOutOfBoundsException();
    }
    ensureCapacity(count + len);
    System.arraycopy(b, off, buf, count, len);
    count += len;
}

/**
 * Writes the complete contents of this byte array output stream to
 * the specified output stream argument, as if by calling the output
 * stream's write method using <code>out.write(buf, 0, count)</code>.
 *
 * @param      out   the output stream to which to write the data.
 * @exception  IOException  if an I/O error occurs.
 */
public synchronized void writeTo(OutputStream out) throws IOException {
    out.write(buf, 0, count);
}

/**
 * Resets the <code>count</code> field of this byte array output
 * stream to zero, so that all currently accumulated output in the
 * output stream is discarded. The output stream can be used again,
 * reusing the already allocated buffer space.
 *
 * @see     java.io.ByteArrayInputStream#count
 */
public synchronized void reset() {
    count = 0;
}

/**
 * Creates a newly allocated byte array. Its size is the current
 * size of this output stream and the valid contents of the buffer
 * have been copied into it.
 *
 * @return  the current contents of this output stream, as a byte array.
 * @see     java.io.ByteArrayOutputStream#size()
 */
public synchronized byte toByteArray()[] {
    return Arrays.copyOf(buf, count);
}

/**
 * Returns the current size of the buffer.
 *
 * @return  the value of the <code>count</code> field, which is the number
 *          of valid bytes in this output stream.
 * @see     java.io.ByteArrayOutputStream#count
```

```
     */
    public synchronized int size() {
        return count;
    }

    /**
     * Converts the buffer's contents into a string decoding bytes using the
     * platform's default character set. The length of the new <tt>String</tt>
     * is a function of the character set, and hence may not be equal to the
     * size of the buffer.
     *
     * <p> This method always replaces malformed-input and unmappable-character
     * sequences with the default replacement string for the platform's
     * default character set. The {@linkplain java.nio.charset.CharsetDecoder}
     * class should be used when more control over the decoding process is
     * required.
     *
     * @return String decoded from the buffer's contents.
     * @since  JDK1.1
     */
    public synchronized String toString() {
        return new String(buf, 0, count);
    }

    /**
     * Converts the buffer's contents into a string by decoding the bytes using
     * the named {@link java.nio.charset.Charset charset}. The length of the new
     * <tt>String</tt> is a function of the charset, and hence may not be equal
     * to the length of the byte array.
     *
     * <p> This method always replaces malformed-input and unmappable-character
     * sequences with this charset's default replacement string. The {@link
     * java.nio.charset.CharsetDecoder} class should be used when more control
     * over the decoding process is required.
     *
     * @param      charsetName  the name of a supported
     *             {@link java.nio.charset.Charset charset}
     * @return     String decoded from the buffer's contents.
     * @exception  UnsupportedEncodingException
     *             If the named charset is not supported
     * @since      JDK1.1
     */
    public synchronized String toString(String charsetName)
        throws UnsupportedEncodingException
    {
        return new String(buf, 0, count, charsetName);
    }

    /**
     * Creates a newly allocated string. Its size is the current size of
     * the output stream and the valid contents of the buffer have been
     * copied into it. Each character <i>c</i> in the resulting string is
     * constructed from the corresponding element <i>b</i> in the byte
     * array such that:
     * <blockquote><pre>
     *     c == (char)(((hibyte & 0xff) << 8) | (b & 0xff))
     * </pre></blockquote>
     *
     * @deprecated This method does not properly convert bytes into characters.
     * As of JDK 1.1, the preferred way to do this is via the
     * <code>toString(String enc)</code> method, which takes an encoding-name
     * argument, or the <code>toString()</code> method, which uses the
     * platform's default character encoding.
```

```
     *
     * @param      hibyte    the high byte of each resulting Unicode character.
     * @return     the current contents of the output stream, as a string.
     * @see        java.io.ByteArrayOutputStream#size()
     * @see        java.io.ByteArrayOutputStream#toString(String)
     * @see        java.io.ByteArrayOutputStream#toString()
     */
    @Deprecated
    public synchronized String toString(int hibyte) {
        return new String(buf, hibyte, 0, count);
    }

    /**
     * Closing a <tt>ByteArrayOutputStream</tt> has no effect. The methods in
     * this class can be called after the stream has been closed without
     * generating an <tt>IOException</tt>.
     */
    public void close() throws IOException {
    }

}
```

CharArrayReader.java

```
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * This class implements a character buffer that can be used as a
 * character-input stream.
 *
 * @author      Herb Jellinek
 * @since       JDK1.1
 */
public class CharArrayReader extends Reader {
    /** The character buffer. */
    protected char buf[];

    /** The current buffer position. */
    protected int pos;

    /** The position of mark in buffer. */
    protected int markedPos = 0;

    /**
     *  The index of the end of this buffer.  There is not valid
     *  data at or beyond this index.
     */
    protected int count;

    /**
     * Creates a CharArrayReader from the specified array of chars.
     * @param buf       Input buffer (not copied)
     */
    public CharArrayReader(char buf[]) {
        this.buf = buf;
        this.pos = 0;
        this.count = buf.length;
    }
```

```
/**
 * Creates a CharArrayReader from the specified array of chars.
 *
 * <p> The resulting reader will start reading at the given
 * <tt>offset</tt>.  The total number of <tt>char</tt> values that can be
 * read from this reader will be either <tt>length</tt> or
 * <tt>buf.length-offset</tt>, whichever is smaller.
 *
 * @throws IllegalArgumentException
 *         If <tt>offset</tt> is negative or greater than
 *         <tt>buf.length</tt>, or if <tt>length</tt> is negative, or if
 *         the sum of these two values is negative.
 *
 * @param buf       Input buffer (not copied)
 * @param offset    Offset of the first char to read
 * @param length    Number of chars to read
 */
public CharArrayReader(char buf[], int offset, int length) {
    if ((offset < 0) || (offset > buf.length) || (length < 0) ||
        ((offset + length) < 0)) {
        throw new IllegalArgumentException();
    }
    this.buf = buf;
    this.pos = offset;
    this.count = Math.min(offset + length, buf.length);
    this.markedPos = offset;
}

/** Checks to make sure that the stream has not been closed */
private void ensureOpen() throws IOException {
    if (buf == null)
        throw new IOException("Stream closed");
}

/**
 * Reads a single character.
 *
 * @exception   IOException  If an I/O error occurs
 */
public int read() throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (pos >= count)
            return -1;
        else
            return buf[pos++];
    }
}

/**
 * Reads characters into a portion of an array.
 * @param b  Destination buffer
 * @param off  Offset at which to start storing characters
 * @param len   Maximum number of characters to read
 * @return  The actual number of characters read, or -1 if
 *          the end of the stream has been reached
 *
 * @exception   IOException  If an I/O error occurs
 */
public int read(char b[], int off, int len) throws IOException {
    synchronized (lock) {
        ensureOpen();
        if ((off < 0) || (off > b.length) || (len < 0) ||
```

```
            ((off + len) > b.length) || ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return 0;
        }

        if (pos >= count) {
            return -1;
        }
        if (pos + len > count) {
            len = count - pos;
        }
        if (len <= 0) {
            return 0;
        }
        System.arraycopy(buf, pos, b, off, len);
        pos += len;
        return len;
    }
}

/**
 * Skips characters.  Returns the number of characters that were skipped.
 *
 * <p>The <code>n</code> parameter may be negative, even though the
 * <code>skip</code> method of the {@link Reader} superclass throws
 * an exception in this case. If <code>n</code> is negative, then
 * this method does nothing and returns <code>0</code>.
 *
 * @param n The number of characters to skip
 * @return       The number of characters actually skipped
 * @exception  IOException If the stream is closed, or an I/O error occurs
 */
public long skip(long n) throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (pos + n > count) {
            n = count - pos;
        }
        if (n < 0) {
            return 0;
        }
        pos += n;
        return n;
    }
}

/**
 * Tells whether this stream is ready to be read.  Character-array readers
 * are always ready to be read.
 *
 * @exception  IOException  If an I/O error occurs
 */
public boolean ready() throws IOException {
    synchronized (lock) {
        ensureOpen();
        return (count - pos) > 0;
    }
}

/**
 * Tells whether this stream supports the mark() operation, which it does.
 */
```

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

    /**
     * Marks the present position in the stream.  Subsequent calls to reset()
     * will reposition the stream to this point.
     *
     * @param  readAheadLimit  Limit on the number of characters that may be
     *                         read while still preserving the mark.  Because
     *                         the stream's input comes from a character array,
     *                         there is no actual limit; hence this argument is
     *                         ignored.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void mark(int readAheadLimit) throws IOException {
        synchronized (lock) {
            ensureOpen();
            markedPos = pos;
        }
    }

    /**
     * Resets the stream to the most recent mark, or to the beginning if it has
     * never been marked.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void reset() throws IOException {
        synchronized (lock) {
            ensureOpen();
            pos = markedPos;
        }
    }

    /**
     * Closes the stream and releases any system resources associated with
     * it.  Once the stream has been closed, further read(), ready(),
     * mark(), reset(), or skip() invocations will throw an IOException.
     * Closing a previously closed stream has no effect.
     */
    public void close() {
        buf = null;
    }
}
```

CharArrayWriter.java

```
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.util.Arrays;

/**
 * This class implements a character buffer that can be used as an Writer.
 * The buffer automatically grows when data is written to the stream.  The data
 * can be retrieved using toCharArray() and toString().
 * <P>
 * Note: Invoking close() on this class has no effect, and methods
 * of this class can be called after the stream has closed
 * without generating an IOException.
 *
 * @author      Herb Jellinek
 * @since       JDK1.1
 */
public
class CharArrayWriter extends Writer {
    /**
     * The buffer where data is stored.
     */
    protected char buf[];

    /**
     * The number of chars in the buffer.
     */
    protected int count;

    /**
     * Creates a new CharArrayWriter.
     */
    public CharArrayWriter() {
        this(32);
    }
```

```
/**
 * Creates a new CharArrayWriter with the specified initial size.
 *
 * @param initialSize  an int specifying the initial buffer size.
 * @exception IllegalArgumentException if initialSize is negative
 */
public CharArrayWriter(int initialSize) {
    if (initialSize < 0) {
        throw new IllegalArgumentException("Negative initial size: "
                                           + initialSize);
    }
    buf = new char[initialSize];
}

/**
 * Writes a character to the buffer.
 */
public void write(int c) {
    synchronized (lock) {
        int newcount = count + 1;
        if (newcount > buf.length) {
            buf = Arrays.copyOf(buf, Math.max(buf.length << 1, newcount));
        }
        buf[count] = (char)c;
        count = newcount;
    }
}

/**
 * Writes characters to the buffer.
 * @param c the data to be written
 * @param off       the start offset in the data
 * @param len       the number of chars that are written
 */
public void write(char c[], int off, int len) {
    if ((off < 0) || (off > c.length) || (len < 0) ||
        ((off + len) > c.length) || ((off + len) < 0)) {
        throw new IndexOutOfBoundsException();
    } else if (len == 0) {
        return;
    }
    synchronized (lock) {
        int newcount = count + len;
        if (newcount > buf.length) {
            buf = Arrays.copyOf(buf, Math.max(buf.length << 1, newcount));
        }
        System.arraycopy(c, off, buf, count, len);
        count = newcount;
    }
}

/**
 * Write a portion of a string to the buffer.
 * @param  str  String to be written from
 * @param  off  Offset from which to start reading characters
 * @param  len  Number of characters to be written
 */
public void write(String str, int off, int len) {
    synchronized (lock) {
        int newcount = count + len;
        if (newcount > buf.length) {
            buf = Arrays.copyOf(buf, Math.max(buf.length << 1, newcount));
        }
```

```
        str.getChars(off, off + len, buf, count);
        count = newcount;
    }
}

/**
 * Writes the contents of the buffer to another character stream.
 *
 * @param out       the output stream to write to
 * @throws IOException If an I/O error occurs.
 */
public void writeTo(Writer out) throws IOException {
    synchronized (lock) {
        out.write(buf, 0, count);
    }
}

/**
 * Appends the specified character sequence to this writer.
 *
 * <p> An invocation of this method of the form <tt>out.append(csq)</tt>
 * behaves in exactly the same way as the invocation
 *
 * <pre>
 *     out.write(csq.toString()) </pre>
 *
 * <p> Depending on the specification of <tt>toString</tt> for the
 * character sequence <tt>csq</tt>, the entire sequence may not be
 * appended. For instance, invoking the <tt>toString</tt> method of a
 * character buffer will return a subsequence whose content depends upon
 * the buffer's position and limit.
 *
 * @param  csq
 *         The character sequence to append.  If <tt>csq</tt> is
 *         <tt>null</tt>, then the four characters <tt>"null"</tt> are
 *         appended to this writer.
 *
 * @return  This writer
 *
 * @since  1.5
 */
public CharArrayWriter append(CharSequence csq) {
    String s = (csq == null ? "null" : csq.toString());
    write(s, 0, s.length());
    return this;
}

/**
 * Appends a subsequence of the specified character sequence to this writer.
 *
 * <p> An invocation of this method of the form <tt>out.append(csq, start,
 * end)</tt> when <tt>csq</tt> is not <tt>null</tt>, behaves in
 * exactly the same way as the invocation
 *
 * <pre>
 *     out.write(csq.subSequence(start, end).toString()) </pre>
 *
 * @param  csq
 *         The character sequence from which a subsequence will be
 *         appended.  If <tt>csq</tt> is <tt>null</tt>, then characters
 *         will be appended as if <tt>csq</tt> contained the four
 *         characters <tt>"null"</tt>.
 *
```

```
     * @param  start
     *         The index of the first character in the subsequence
     *
     * @param  end
     *         The index of the character following the last character in the
     *         subsequence
     *
     * @return  This writer
     *
     * @throws  IndexOutOfBoundsException
     *          If <tt>start</tt> or <tt>end</tt> are negative, <tt>start</tt>
     *          is greater than <tt>end</tt>, or <tt>end</tt> is greater than
     *          <tt>csq.length()</tt>
     *
     * @since  1.5
     */
    public CharArrayWriter append(CharSequence csq, int start, int end) {
        String s = (csq == null ? "null" : csq).subSequence(start, end).toString();
        write(s, 0, s.length());
        return this;
    }

    /**
     * Appends the specified character to this writer.
     *
     * <p> An invocation of this method of the form <tt>out.append(c)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.write(c) </pre>
     *
     * @param  c
     *         The 16-bit character to append
     *
     * @return  This writer
     *
     * @since 1.5
     */
    public CharArrayWriter append(char c) {
        write(c);
        return this;
    }

    /**
     * Resets the buffer so that you can use it again without
     * throwing away the already allocated buffer.
     */
    public void reset() {
        count = 0;
    }

    /**
     * Returns a copy of the input data.
     *
     * @return an array of chars copied from the input data.
     */
    public char toCharArray()[] {
        synchronized (lock) {
            return Arrays.copyOf(buf, count);
        }
    }

    /**
```

```
     * Returns the current size of the buffer.
     *
     * @return an int representing the current size of the buffer.
     */
    public int size() {
        return count;
    }

    /**
     * Converts input data to a string.
     * @return the string.
     */
    public String toString() {
        synchronized (lock) {
            return new String(buf, 0, count);
        }
    }

    /**
     * Flush the stream.
     */
    public void flush() { }

    /**
     * Close the stream.  This method does not release the buffer, since its
     * contents might still be required. Note: Invoking this method in this class
     * will have no effect.
     */
    public void close() { }

}
```

CharConversionException.java

```
/*
 * Copyright (c) 1996, 2008, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
package java.io;

/**
 * Base class for character conversion exceptions.
 *
 * @author      Asmus Freytag
 * @since       JDK1.1
 */
public class CharConversionException
    extends java.io.IOException
{
    private static final long serialVersionUID = -8680016352018427031L;

    /**
     * This provides no detailed message.
     */
    public CharConversionException() {
    }
    /**
     * This provides a detailed message.
     *
     * @param s the detailed message associated with the exception.
     */
    public CharConversionException(String s) {
        super(s);
    }
}
```

Closeable.java

```java
/*
 * Copyright (c) 2003, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.io.IOException;

/**
 * A {@code Closeable} is a source or destination of data that can be closed.
 * The close method is invoked to release resources that the object is
 * holding (such as open files).
 *
 * @since 1.5
 */
public interface Closeable extends AutoCloseable {

    /**
     * Closes this stream and releases any system resources associated
     * with it. If the stream is already closed then invoking this
     * method has no effect.
     *
     * <p> As noted in {@link AutoCloseable#close()}, cases where the
     * close may fail require careful attention. It is strongly advised
     * to relinquish the underlying resources and to internally
     * <em>mark</em> the {@code Closeable} as closed, prior to throwing
     * the {@code IOException}.
     *
     * @throws IOException if an I/O error occurs
     */
    public void close() throws IOException;
}
```

Console.java

```
/*
 * Copyright (c) 2005, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.util.*;
import java.nio.charset.Charset;
import sun.nio.cs.StreamDecoder;
import sun.nio.cs.StreamEncoder;

/**
 * Methods to access the character-based console device, if any, associated
 * with the current Java virtual machine.
 *
 * <p> Whether a virtual machine has a console is dependent upon the
 * underlying platform and also upon the manner in which the virtual
 * machine is invoked.  If the virtual machine is started from an
 * interactive command line without redirecting the standard input and
 * output streams then its console will exist and will typically be
 * connected to the keyboard and display from which the virtual machine
 * was launched.  If the virtual machine is started automatically, for
 * example by a background job scheduler, then it will typically not
 * have a console.
 * <p>
 * If this virtual machine has a console then it is represented by a
 * unique instance of this class which can be obtained by invoking the
 * {@link java.lang.System#console()} method.  If no console device is
 * available then an invocation of that method will return <tt>null</tt>.
 * <p>
 * Read and write operations are synchronized to guarantee the atomic
 * completion of critical operations; therefore invoking methods
 * {@link #readLine()}, {@link #readPassword()}, {@link #format format()},
 * {@link #printf printf()} as well as the read, format and write operations
 * on the objects returned by {@link #reader()} and {@link #writer()} may
 * block in multithreaded scenarios.
 * <p>
 * Invoking <tt>close()</tt> on the objects returned by the {@link #reader()}
 * and the {@link #writer()} will not close the underlying stream of those
```

```
 * objects.
 * <p>
 * The console-read methods return <tt>null</tt> when the end of the
 * console input stream is reached, for example by typing control-D on
 * Unix or control-Z on Windows.  Subsequent read operations will succeed
 * if additional characters are later entered on the console's input
 * device.
 * <p>
 * Unless otherwise specified, passing a <tt>null</tt> argument to any method
 * in this class will cause a {@link NullPointerException} to be thrown.
 * <p>
 * <b>Security note:</b>
 * If an application needs to read a password or other secure data, it should
 * use {@link #readPassword()} or {@link #readPassword(String, Object...)} and
 * manually zero the returned character array after processing to minimize the
 * lifetime of sensitive data in memory.
 *
 * <blockquote><pre>{@code
 * Console cons;
 * char[] passwd;
 * if ((cons = System.console()) != null &&
 *     (passwd = cons.readPassword("[%s]", "Password:")) != null) {
 *     ...
 *     java.util.Arrays.fill(passwd, ' ');
 * }
 * }</pre></blockquote>
 *
 * @author  Xueming Shen
 * @since   1.6
 */

public final class Console implements Flushable
{
   /**
    * Retrieves the unique {@link java.io.PrintWriter PrintWriter} object
    * associated with this console.
    *
    * @return  The printwriter associated with this console
    */
    public PrintWriter writer() {
        return pw;
    }

   /**
    * Retrieves the unique {@link java.io.Reader Reader} object associated
    * with this console.
    * <p>
    * This method is intended to be used by sophisticated applications, for
    * example, a {@link java.util.Scanner} object which utilizes the rich
    * parsing/scanning functionality provided by the <tt>Scanner</tt>:
    * <blockquote><pre>
    * Console con = System.console();
    * if (con != null) {
    *     Scanner sc = new Scanner(con.reader());
    *     ...
    * }
    * </pre></blockquote>
    * <p>
    * For simple applications requiring only line-oriented reading, use
    * <tt>{@link #readLine}</tt>.
    * <p>
    * The bulk read operations {@link java.io.Reader#read(char[]) read(char[]) },
    * {@link java.io.Reader#read(char[], int, int) read(char[], int, int) } and
```

```
 * {@link java.io.Reader#read(java.nio.CharBuffer) read(java.nio.CharBuffer)}
 * on the returned object will not read in characters beyond the line
 * bound for each invocation, even if the destination buffer has space for
 * more characters. The {@code Reader}'s {@code read} methods may block if a
 * line bound has not been entered or reached on the console's input device.
 * A line bound is considered to be any one of a line feed (<tt>'\n'</tt>),
 * a carriage return (<tt>'\r'</tt>), a carriage return followed immediately
 * by a linefeed, or an end of stream.
 *
 * @return  The reader associated with this console
 */
 public Reader reader() {
     return reader;
 }

/**
 * Writes a formatted string to this console's output stream using
 * the specified format string and arguments.
 *
 * @param  fmt
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a> section
 *          of the formatter class specification.
 *
 * @return  This console
 */
 public Console format(String fmt, Object ...args) {
     formatter.format(fmt, args).flush();
     return this;
 }

/**
 * A convenience method to write a formatted string to this console's
 * output stream using the specified format string and arguments.
 *
 * <p> An invocation of this method of the form <tt>con.printf(format,
 * args)</tt> behaves in exactly the same way as the invocation of
 * <pre>con.format(format, args)</pre>.
 *
 * @param  format
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>.
 *
 * @param  args
```

```
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a> section of the
 *          formatter class specification.
 *
 * @return  This console
 */
public Console printf(String format, Object ... args) {
    return format(format, args);
}

/**
 * Provides a formatted prompt, then reads a single line of text from the
 * console.
 *
 * @param  fmt
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>.
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *
 * @throws  IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a> section
 *          of the formatter class specification.
 *
 * @throws IOError
 *         If an I/O error occurs.
 *
 * @return  A string containing the line read from the console, not
 *          including any line-termination characters, or <tt>null</tt>
 *          if an end of stream has been reached.
 */
public String readLine(String fmt, Object ... args) {
    String line = null;
    synchronized (writeLock) {
        synchronized(readLock) {
            if (fmt.length() != 0)
                pw.format(fmt, args);
            try {
```

```
                char[] ca = readline(false);
                if (ca != null)
                    line = new String(ca);
            } catch (IOException x) {
                throw new IOError(x);
            }
        }
    }
    return line;
}

/**
 * Reads a single line of text from the console.
 *
 * @throws IOError
 *         If an I/O error occurs.
 *
 * @return  A string containing the line read from the console, not
 *          including any line-termination characters, or <tt>null</tt>
 *          if an end of stream has been reached.
 */
public String readLine() {
    return readLine("");
}

/**
 * Provides a formatted prompt, then reads a password or passphrase from
 * the console with echoing disabled.
 *
 * @param  fmt
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>
 *         for the prompt text.
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *
 * @throws  IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a>
 *          section of the formatter class specification.
 *
 * @throws IOError
 *         If an I/O error occurs.
 *
 * @return  A character array containing the password or passphrase read
 *          from the console, not including any line-termination characters,
 *          or <tt>null</tt> if an end of stream has been reached.
 */
public char[] readPassword(String fmt, Object ... args) {
    char[] passwd = null;
    synchronized (writeLock) {
        synchronized(readLock) {
            try {
                echoOff = echo(false);
```

```
            } catch (IOException x) {
                throw new IOError(x);
            }
            IOError ioe = null;
            try {
                if (fmt.length() != 0)
                    pw.format(fmt, args);
                passwd = readline(true);
            } catch (IOException x) {
                ioe = new IOError(x);
            } finally {
                try {
                    echoOff = echo(true);
                } catch (IOException x) {
                    if (ioe == null)
                        ioe = new IOError(x);
                    else
                        ioe.addSuppressed(x);
                }
                if (ioe != null)
                    throw ioe;
            }
            pw.println();
        }
    }
    return passwd;
}

/**
 * Reads a password or passphrase from the console with echoing disabled
 *
 * @throws IOError
 *         If an I/O error occurs.
 *
 * @return  A character array containing the password or passphrase read
 *          from the console, not including any line-termination characters,
 *          or <tt>null</tt> if an end of stream has been reached.
 */
public char[] readPassword() {
    return readPassword("");
}

/**
 * Flushes the console and forces any buffered output to be written
 * immediately .
 */
public void flush() {
    pw.flush();
}

private Object readLock;
private Object writeLock;
private Reader reader;
private Writer out;
private PrintWriter pw;
private Formatter formatter;
private Charset cs;
private char[] rcb;
private static native String encoding();
private static native boolean echo(boolean on) throws IOException;
private static boolean echoOff;

private char[] readline(boolean zeroOut) throws IOException {
```

```
    int len = reader.read(rcb, 0, rcb.length);
    if (len < 0)
        return null;  //EOL
    if (rcb[len-1] == '\r')
        len--;        //remove CR at end;
    else if (rcb[len-1] == '\n') {
        len--;        //remove LF at end;
        if (len > 0 && rcb[len-1] == '\r')
            len--;    //remove the CR, if there is one
    }
    char[] b = new char[len];
    if (len > 0) {
        System.arraycopy(rcb, 0, b, 0, len);
        if (zeroOut) {
            Arrays.fill(rcb, 0, len, ' ');
        }
    }
    return b;
}

private char[] grow() {
    assert Thread.holdsLock(readLock);
    char[] t = new char[rcb.length * 2];
    System.arraycopy(rcb, 0, t, 0, rcb.length);
    rcb = t;
    return rcb;
}

class LineReader extends Reader {
    private Reader in;
    private char[] cb;
    private int nChars, nextChar;
    boolean leftoverLF;
    LineReader(Reader in) {
        this.in = in;
        cb = new char[1024];
        nextChar = nChars = 0;
        leftoverLF = false;
    }
    public void close () {}
    public boolean ready() throws IOException {
        //in.ready synchronizes on readLock already
        return in.ready();
    }

    public int read(char cbuf[], int offset, int length)
        throws IOException
    {
        int off = offset;
        int end = offset + length;
        if (offset < 0 || offset > cbuf.length || length < 0 ||
            end < 0 || end > cbuf.length) {
            throw new IndexOutOfBoundsException();
        }
        synchronized(readLock) {
            boolean eof = false;
            char c = 0;
            for (;;) {
                if (nextChar >= nChars) {   //fill
                    int n = 0;
                    do {
                        n = in.read(cb, 0, cb.length);
                    } while (n == 0);
```

```
    if (n > 0) {
        nChars = n;
        nextChar = 0;
        if (n < cb.length &&
            cb[n-1] != '\n' && cb[n-1] != '\r') {
            /*
             * we're in canonical mode so each "fill" should
             * come back with an eol. if there no lf or nl at
             * the end of returned bytes we reached an eof.
             */
            eof = true;
        }
    } else { /*EOF*/
        if (off - offset == 0)
            return -1;
        return off - offset;
    }
}
if (leftoverLF && cbuf == rcb && cb[nextChar] == '\n') {
    /*
     * if invoked by our readline, skip the leftover, otherwise
     * return the LF.
     */
    nextChar++;
}
leftoverLF = false;
while (nextChar < nChars) {
    c = cbuf[off++] = cb[nextChar];
    cb[nextChar++] = 0;
    if (c == '\n') {
        return off - offset;
    } else if (c == '\r') {
        if (off == end) {
            /* no space left even the next is LF, so return
             * whatever we have if the invoker is not our
             * readLine()
             */
            if (cbuf == rcb) {
                cbuf = grow();
                end = cbuf.length;
            } else {
                leftoverLF = true;
                return off - offset;
            }
        }
        if (nextChar == nChars && in.ready()) {
            /*
             * we have a CR and we reached the end of
             * the read in buffer, fill to make sure we
             * don't miss a LF, if there is one, it's possible
             * that it got cut off during last round reading
             * simply because the read in buffer was full.
             */
            nChars = in.read(cb, 0, cb.length);
            nextChar = 0;
        }
        if (nextChar < nChars && cb[nextChar] == '\n') {
            cbuf[off++] = '\n';
            nextChar++;
        }
        return off - offset;
    } else if (off == end) {
       if (cbuf == rcb) {
```

```
                            cbuf = grow();
                            end = cbuf.length;
                        } else {
                            return off - offset;
                        }
                    }
                }
                if (eof)
                    return off - offset;
            }
        }
    }
}

// Set up JavaIOAccess in SharedSecrets
static {
    try {
        // Add a shutdown hook to restore console's echo state should
        // it be necessary.
        sun.misc.SharedSecrets.getJavaLangAccess()
            .registerShutdownHook(0 /* shutdown hook invocation order */,
                false /* only register if shutdown is not in progress */,
                new Runnable() {
                    public void run() {
                        try {
                            if (echoOff) {
                                echo(true);
                            }
                        } catch (IOException x) { }
                    }
                });
    } catch (IllegalStateException e) {
        // shutdown is already in progress and console is first used
        // by a shutdown hook
    }

    sun.misc.SharedSecrets.setJavaIOAccess(new sun.misc.JavaIOAccess() {
        public Console console() {
            if (istty()) {
                if (cons == null)
                    cons = new Console();
                return cons;
            }
            return null;
        }

        public Charset charset() {
            // This method is called in sun.security.util.Password,
            // cons already exists when this method is called
            return cons.cs;
        }
    });
}
private static Console cons;
private native static boolean istty();
private Console() {
    readLock = new Object();
    writeLock = new Object();
    String csname = encoding();
    if (csname != null) {
        try {
            cs = Charset.forName(csname);
        } catch (Exception x) {}
```

```
        }
        if (cs == null)
            cs = Charset.defaultCharset();
        out = StreamEncoder.forOutputStreamWriter(
                  new FileOutputStream(FileDescriptor.out),
                  writeLock,
                  cs);
        pw = new PrintWriter(out, true) { public void close() {} };
        formatter = new Formatter(out);
        reader = new LineReader(StreamDecoder.forInputStreamReader(
                     new FileInputStream(FileDescriptor.in),
                     readLock,
                     cs));
        rcb = new char[1024];
    }
}
```

DataInput.java

```
/*
 * Copyright (c) 1995, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * The {@code DataInput} interface provides
 * for reading bytes from a binary stream and
 * reconstructing from them data in any of
 * the Java primitive types. There is also
 * a
 * facility for reconstructing a {@code String}
 * from data in
 * <a href="#modified-utf-8">modified UTF-8</a>
 * format.
 * <p>
 * It is generally true of all the reading
 * routines in this interface that if end of
 * file is reached before the desired number
 * of bytes has been read, an {@code EOFException}
 * (which is a kind of {@code IOException})
 * is thrown. If any byte cannot be read for
 * any reason other than end of file, an {@code IOException}
 * other than {@code EOFException} is
 * thrown. In particular, an {@code IOException}
 * may be thrown if the input stream has been
 * closed.
 *
 * <h3><a name="modified-utf-8">Modified UTF-8</a></h3>
 * <p>
 * Implementations of the DataInput and DataOutput interfaces represent
 * Unicode strings in a format that is a slight modification of UTF-8.
 * (For information regarding the standard UTF-8 format, see section
 * <i>3.9 Unicode Encoding Forms</i> of <i>The Unicode Standard, Version
 * 4.0</i>).
 * Note that in the following table, the most significant bit appears in the
 * far left-hand column.
 *
```

```
* <blockquote>
*   <table border="1" cellspacing="0" cellpadding="8"
*          summary="Bit values and bytes">
*     <tr>
*       <th colspan="9"><span style="font-weight:normal">
*         All characters in the range {@code '\u005Cu0001'} to
*         {@code '\u005Cu007F'} are represented by a single byte:</span></th>
*     </tr>
*     <tr>
*       <td></td>
*       <th colspan="8" id="bit_a">Bit Values</th>
*     </tr>
*     <tr>
*       <th id="byte1_a">Byte 1</th>
*       <td><center>0</center>
*       <td colspan="7"><center>bits 6-0</center>
*     </tr>
*     <tr>
*       <th colspan="9"><span style="font-weight:normal">
*         The null character {@code '\u005Cu0000'} and characters
*         in the range {@code '\u005Cu0080'} to {@code '\u005Cu07FF'} are
*         represented by a pair of bytes:</span></th>
*     </tr>
*     <tr>
*       <td></td>
*       <th colspan="8" id="bit_b">Bit Values</th>
*     </tr>
*     <tr>
*       <th id="byte1_b">Byte 1</th>
*       <td><center>1</center>
*       <td><center>1</center>
*       <td><center>0</center>
*       <td colspan="5"><center>bits 10-6</center>
*     </tr>
*     <tr>
*       <th id="byte2_a">Byte 2</th>
*       <td><center>1</center>
*       <td><center>0</center>
*       <td colspan="6"><center>bits 5-0</center>
*     </tr>
*     <tr>
*       <th colspan="9"><span style="font-weight:normal">
*         {@code char} values in the range {@code '\u005Cu0800'}
*         to {@code '\u005CuFFFF'} are represented by three bytes:</span></th>
*     </tr>
*     <tr>
*       <td></td>
*       <th colspan="8"id="bit_c">Bit Values</th>
*     </tr>
*     <tr>
*       <th id="byte1_c">Byte 1</th>
*       <td><center>1</center>
*       <td><center>1</center>
*       <td><center>1</center>
*       <td><center>0</center>
*       <td colspan="4"><center>bits 15-12</center>
*     </tr>
*     <tr>
*       <th id="byte2_b">Byte 2</th>
*       <td><center>1</center>
*       <td><center>0</center>
*       <td colspan="6"><center>bits 11-6</center>
*     </tr>
```

```
 *     <tr>
 *       <th id="byte3">Byte 3</th>
 *       <td><center>1</center>
 *       <td><center>0</center>
 *       <td colspan="6"><center>bits 5-0</center>
 *     </tr>
 *   </table>
 * </blockquote>
 * <p>
 * The differences between this format and the
 * standard UTF-8 format are the following:
 * <ul>
 * <li>The null byte {@code '\u005Cu0000'} is encoded in 2-byte format
 *     rather than 1-byte, so that the encoded strings never have
 *     embedded nulls.
 * <li>Only the 1-byte, 2-byte, and 3-byte formats are used.
 * <li><a href="../lang/Character.html#unicode">Supplementary characters</a>
 *     are represented in the form of surrogate pairs.
 * </ul>
 * @author  Frank Yellin
 * @see     java.io.DataInputStream
 * @see     java.io.DataOutput
 * @since   JDK1.0
 */
public
interface DataInput {
    /**
     * Reads some bytes from an input
     * stream and stores them into the buffer
     * array {@code b}. The number of bytes
     * read is equal
     * to the length of {@code b}.
     * <p>
     * This method blocks until one of the
     * following conditions occurs:
     * <ul>
     * <li>{@code b.length}
     * bytes of input data are available, in which
     * case a normal return is made.
     *
     * <li>End of
     * file is detected, in which case an {@code EOFException}
     * is thrown.
     *
     * <li>An I/O error occurs, in
     * which case an {@code IOException} other
     * than {@code EOFException} is thrown.
     * </ul>
     * <p>
     * If {@code b} is {@code null},
     * a {@code NullPointerException} is thrown.
     * If {@code b.length} is zero, then
     * no bytes are read. Otherwise, the first
     * byte read is stored into element {@code b[0]},
     * the next one into {@code b[1]}, and
     * so on.
     * If an exception is thrown from
     * this method, then it may be that some but
     * not all bytes of {@code b} have been
     * updated with data from the input stream.
     *
     * @param     b   the buffer into which the data is read.
     * @exception  EOFException  if this stream reaches the end before reading
```

```
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
void readFully(byte b[]) throws IOException;

/**
 *
 * Reads {@code len}
 * bytes from
 * an input stream.
 * <p>
 * This method
 * blocks until one of the following conditions
 * occurs:
 * <ul>
 * <li>{@code len} bytes
 * of input data are available, in which case
 * a normal return is made.
 *
 * <li>End of file
 * is detected, in which case an {@code EOFException}
 * is thrown.
 *
 * <li>An I/O error occurs, in
 * which case an {@code IOException} other
 * than {@code EOFException} is thrown.
 * </ul>
 * <p>
 * If {@code b} is {@code null},
 * a {@code NullPointerException} is thrown.
 * If {@code off} is negative, or {@code len}
 * is negative, or {@code off+len} is
 * greater than the length of the array {@code b},
 * then an {@code IndexOutOfBoundsException}
 * is thrown.
 * If {@code len} is zero,
 * then no bytes are read. Otherwise, the first
 * byte read is stored into element {@code b[off]},
 * the next one into {@code b[off+1]},
 * and so on. The number of bytes read is,
 * at most, equal to {@code len}.
 *
 * @param     b   the buffer into which the data is read.
 * @param off  an int specifying the offset into the data.
 * @param len  an int specifying the number of bytes to read.
 * @exception  EOFException  if this stream reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
void readFully(byte b[], int off, int len) throws IOException;

/**
 * Makes an attempt to skip over
 * {@code n} bytes
 * of data from the input
 * stream, discarding the skipped bytes. However,
 * it may skip
 * over some smaller number of
 * bytes, possibly zero. This may result from
 * any of a
 * number of conditions; reaching
 * end of file before {@code n} bytes
 * have been skipped is
```

```
 * only one possibility.
 * This method never throws an {@code EOFException}.
 * The actual
 * number of bytes skipped is returned.
 *
 * @param      n   the number of bytes to be skipped.
 * @return     the number of bytes actually skipped.
 * @exception  IOException   if an I/O error occurs.
 */
int skipBytes(int n) throws IOException;

/**
 * Reads one input byte and returns
 * {@code true} if that byte is nonzero,
 * {@code false} if that byte is zero.
 * This method is suitable for reading
 * the byte written by the {@code writeBoolean}
 * method of interface {@code DataOutput}.
 *
 * @return     the {@code boolean} value read.
 * @exception  EOFException  if this stream reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
boolean readBoolean() throws IOException;

/**
 * Reads and returns one input byte.
 * The byte is treated as a signed value in
 * the range {@code -128} through {@code 127},
 * inclusive.
 * This method is suitable for
 * reading the byte written by the {@code writeByte}
 * method of interface {@code DataOutput}.
 *
 * @return     the 8-bit value read.
 * @exception  EOFException  if this stream reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
byte readByte() throws IOException;

/**
 * Reads one input byte, zero-extends
 * it to type {@code int}, and returns
 * the result, which is therefore in the range
 * {@code 0}
 * through {@code 255}.
 * This method is suitable for reading
 * the byte written by the {@code writeByte}
 * method of interface {@code DataOutput}
 * if the argument to {@code writeByte}
 * was intended to be a value in the range
 * {@code 0} through {@code 255}.
 *
 * @return     the unsigned 8-bit value read.
 * @exception  EOFException  if this stream reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
int readUnsignedByte() throws IOException;

/**
```

```
     * Reads two input bytes and returns
     * a {@code short} value. Let {@code a}
     * be the first byte read and {@code b}
     * be the second byte. The value
     * returned
     * is:
     * <pre>{@code (short)((a << 8) | (b & 0xff))
     * }</pre>
     * This method
     * is suitable for reading the bytes written
     * by the {@code writeShort} method of
     * interface {@code DataOutput}.
     *
     * @return     the 16-bit value read.
     * @exception  EOFException  if this stream reaches the end before reading
     *               all the bytes.
     * @exception  IOException   if an I/O error occurs.
     */
    short readShort() throws IOException;

    /**
     * Reads two input bytes and returns
     * an {@code int} value in the range {@code 0}
     * through {@code 65535}. Let {@code a}
     * be the first byte read and
     * {@code b}
     * be the second byte. The value returned is:
     * <pre>{@code (((a & 0xff) << 8) | (b & 0xff))
     * }</pre>
     * This method is suitable for reading the bytes
     * written by the {@code writeShort} method
     * of interface {@code DataOutput}  if
     * the argument to {@code writeShort}
     * was intended to be a value in the range
     * {@code 0} through {@code 65535}.
     *
     * @return     the unsigned 16-bit value read.
     * @exception  EOFException  if this stream reaches the end before reading
     *               all the bytes.
     * @exception  IOException   if an I/O error occurs.
     */
    int readUnsignedShort() throws IOException;

    /**
     * Reads two input bytes and returns a {@code char} value.
     * Let {@code a}
     * be the first byte read and {@code b}
     * be the second byte. The value
     * returned is:
     * <pre>{@code (char)((a << 8) | (b & 0xff))
     * }</pre>
     * This method
     * is suitable for reading bytes written by
     * the {@code writeChar} method of interface
     * {@code DataOutput}.
     *
     * @return     the {@code char} value read.
     * @exception  EOFException  if this stream reaches the end before reading
     *               all the bytes.
     * @exception  IOException   if an I/O error occurs.
     */
    char readChar() throws IOException;
```

```
/**
 * Reads four input bytes and returns an
 * {@code int} value. Let {@code a-d}
 * be the first through fourth bytes read. The value returned is:
 * <pre>{@code
 * (((a & 0xff) << 24) | ((b & 0xff) << 16) |
 *  ((c & 0xff) <<  8) | (d & 0xff))
 * }</pre>
 * This method is suitable
 * for reading bytes written by the {@code writeInt}
 * method of interface {@code DataOutput}.
 *
 * @return     the {@code int} value read.
 * @exception  EOFException  if this stream reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
int readInt() throws IOException;

/**
 * Reads eight input bytes and returns
 * a {@code long} value. Let {@code a-h}
 * be the first through eighth bytes read.
 * The value returned is:
 * <pre>{@code
 * (((long)(a & 0xff) << 56) |
 *  ((long)(b & 0xff) << 48) |
 *  ((long)(c & 0xff) << 40) |
 *  ((long)(d & 0xff) << 32) |
 *  ((long)(e & 0xff) << 24) |
 *  ((long)(f & 0xff) << 16) |
 *  ((long)(g & 0xff) <<  8) |
 *  ((long)(h & 0xff)))
 * }</pre>
 * <p>
 * This method is suitable
 * for reading bytes written by the {@code writeLong}
 * method of interface {@code DataOutput}.
 *
 * @return     the {@code long} value read.
 * @exception  EOFException  if this stream reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
long readLong() throws IOException;

/**
 * Reads four input bytes and returns
 * a {@code float} value. It does this
 * by first constructing an {@code int}
 * value in exactly the manner
 * of the {@code readInt}
 * method, then converting this {@code int}
 * value to a {@code float} in
 * exactly the manner of the method {@code Float.intBitsToFloat}.
 * This method is suitable for reading
 * bytes written by the {@code writeFloat}
 * method of interface {@code DataOutput}.
 *
 * @return     the {@code float} value read.
 * @exception  EOFException  if this stream reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
```

```
 */
float readFloat() throws IOException;

/**
 * Reads eight input bytes and returns
 * a {@code double} value. It does this
 * by first constructing a {@code long}
 * value in exactly the manner
 * of the {@code readLong}
 * method, then converting this {@code long}
 * value to a {@code double} in exactly
 * the manner of the method {@code Double.longBitsToDouble}.
 * This method is suitable for reading
 * bytes written by the {@code writeDouble}
 * method of interface {@code DataOutput}.
 *
 * @return     the {@code double} value read.
 * @exception  EOFException  if this stream reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
double readDouble() throws IOException;

/**
 * Reads the next line of text from the input stream.
 * It reads successive bytes, converting
 * each byte separately into a character,
 * until it encounters a line terminator or
 * end of
 * file; the characters read are then
 * returned as a {@code String}. Note
 * that because this
 * method processes bytes,
 * it does not support input of the full Unicode
 * character set.
 * <p>
 * If end of file is encountered
 * before even one byte can be read, then {@code null}
 * is returned. Otherwise, each byte that is
 * read is converted to type {@code char}
 * by zero-extension. If the character {@code '\n'}
 * is encountered, it is discarded and reading
 * ceases. If the character {@code '\r'}
 * is encountered, it is discarded and, if
 * the following byte converts  to the
 * character {@code '\n'}, then that is
 * discarded also; reading then ceases. If
 * end of file is encountered before either
 * of the characters {@code '\n'} and
 * {@code '\r'} is encountered, reading
 * ceases. Once reading has ceased, a {@code String}
 * is returned that contains all the characters
 * read and not discarded, taken in order.
 * Note that every character in this string
 * will have a value less than {@code \u005Cu0100},
 * that is, {@code (char)256}.
 *
 * @return the next line of text from the input stream,
 *         or {@code null} if the end of file is
 *         encountered before a byte can be read.
 * @exception  IOException  if an I/O error occurs.
 */
String readLine() throws IOException;
```

```
/**
 * Reads in a string that has been encoded using a
 * <a href="#modified-utf-8">modified UTF-8</a>
 * format.
 * The general contract of {@code readUTF}
 * is that it reads a representation of a Unicode
 * character string encoded in modified
 * UTF-8 format; this string of characters
 * is then returned as a {@code String}.
 * <p>
 * First, two bytes are read and used to
 * construct an unsigned 16-bit integer in
 * exactly the manner of the {@code readUnsignedShort}
 * method . This integer value is called the
 * <i>UTF length</i> and specifies the number
 * of additional bytes to be read. These bytes
 * are then converted to characters by considering
 * them in groups. The length of each group
 * is computed from the value of the first
 * byte of the group. The byte following a
 * group, if any, is the first byte of the
 * next group.
 * <p>
 * If the first byte of a group
 * matches the bit pattern {@code 0xxxxxxx}
 * (where {@code x} means "may be {@code 0}
 * or {@code 1}"), then the group consists
 * of just that byte. The byte is zero-extended
 * to form a character.
 * <p>
 * If the first byte
 * of a group matches the bit pattern {@code 110xxxxx},
 * then the group consists of that byte {@code a}
 * and a second byte {@code b}. If there
 * is no byte {@code b} (because byte
 * {@code a} was the last of the bytes
 * to be read), or if byte {@code b} does
 * not match the bit pattern {@code 10xxxxxx},
 * then a {@code UTFDataFormatException}
 * is thrown. Otherwise, the group is converted
 * to the character:
 * <pre>{@code (char)(((a & 0x1F) << 6) | (b & 0x3F))
 * }</pre>
 * If the first byte of a group
 * matches the bit pattern {@code 1110xxxx},
 * then the group consists of that byte {@code a}
 * and two more bytes {@code b} and {@code c}.
 * If there is no byte {@code c} (because
 * byte {@code a} was one of the last
 * two of the bytes to be read), or either
 * byte {@code b} or byte {@code c}
 * does not match the bit pattern {@code 10xxxxxx},
 * then a {@code UTFDataFormatException}
 * is thrown. Otherwise, the group is converted
 * to the character:
 * <pre>{@code
 * (char)(((a & 0x0F) << 12) | ((b & 0x3F) << 6) | (c & 0x3F))
 * }</pre>
 * If the first byte of a group matches the
 * pattern {@code 1111xxxx} or the pattern
 * {@code 10xxxxxx}, then a {@code UTFDataFormatException}
 * is thrown.
```

```
     * <p>
     * If end of file is encountered
     * at any time during this entire process,
     * then an {@code EOFException} is thrown.
     * <p>
     * After every group has been converted to
     * a character by this process, the characters
     * are gathered, in the same order in which
     * their corresponding groups were read from
     * the input stream, to form a {@code String},
     * which is returned.
     * <p>
     * The {@code writeUTF}
     * method of interface {@code DataOutput}
     * may be used to write data that is suitable
     * for reading by this method.
     * @return     a Unicode string.
     * @exception  EOFException            if this stream reaches the end
     *               before reading all the bytes.
     * @exception  IOException             if an I/O error occurs.
     * @exception  UTFDataFormatException  if the bytes do not represent a
     *               valid modified UTF-8 encoding of a string.
     */
    String readUTF() throws IOException;
}
```

DataInputStream.java

```
/*
 * Copyright (c) 1994, 2006, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * A data input stream lets an application read primitive Java data
 * types from an underlying input stream in a machine-independent
 * way. An application uses a data output stream to write data that
 * can later be read by a data input stream.
 * <p>
 * DataInputStream is not necessarily safe for multithreaded access.
 * Thread safety is optional and is the responsibility of users of
 * methods in this class.
 *
 * @author  Arthur van Hoff
 * @see     java.io.DataOutputStream
 * @since   JDK1.0
 */
public
class DataInputStream extends FilterInputStream implements DataInput {

    /**
     * Creates a DataInputStream that uses the specified
     * underlying InputStream.
     *
     * @param  in   the specified input stream
     */
    public DataInputStream(InputStream in) {
        super(in);
    }

    /**
     * working arrays initialized on demand by readUTF
     */
    private byte bytearr[] = new byte[80];
    private char chararr[] = new char[80];
```

```
/**
 * Reads some number of bytes from the contained input stream and
 * stores them into the buffer array <code>b</code>. The number of
 * bytes actually read is returned as an integer. This method blocks
 * until input data is available, end of file is detected, or an
 * exception is thrown.
 *
 * <p>If <code>b</code> is null, a <code>NullPointerException</code> is
 * thrown. If the length of <code>b</code> is zero, then no bytes are
 * read and <code>0</code> is returned; otherwise, there is an attempt
 * to read at least one byte. If no byte is available because the
 * stream is at end of file, the value <code>-1</code> is returned;
 * otherwise, at least one byte is read and stored into <code>b</code>.
 *
 * <p>The first byte read is stored into element <code>b[0]</code>, the
 * next one into <code>b[1]</code>, and so on. The number of bytes read
 * is, at most, equal to the length of <code>b</code>. Let <code>k</code>
 * be the number of bytes actually read; these bytes will be stored in
 * elements <code>b[0]</code> through <code>b[k-1]</code>, leaving
 * elements <code>b[k]</code> through <code>b[b.length-1]</code>
 * unaffected.
 *
 * <p>The <code>read(b)</code> method has the same effect as:
 * <blockquote><pre>
 * read(b, 0, b.length)
 * </pre></blockquote>
 *
 * @param      b   the buffer into which the data is read.
 * @return     the total number of bytes read into the buffer, or
 *             <code>-1</code> if there is no more data because the end
 *             of the stream has been reached.
 * @exception  IOException if the first byte cannot be read for any reason
 * other than end of file, the stream has been closed and the underlying
 * input stream does not support reading after close, or another I/O
 * error occurs.
 * @see        java.io.FilterInputStream#in
 * @see        java.io.InputStream#read(byte[], int, int)
 */
public final int read(byte b[]) throws IOException {
    return in.read(b, 0, b.length);
}

/**
 * Reads up to <code>len</code> bytes of data from the contained
 * input stream into an array of bytes.  An attempt is made to read
 * as many as <code>len</code> bytes, but a smaller number may be read,
 * possibly zero. The number of bytes actually read is returned as an
 * integer.
 *
 * <p> This method blocks until input data is available, end of file is
 * detected, or an exception is thrown.
 *
 * <p> If <code>len</code> is zero, then no bytes are read and
 * <code>0</code> is returned; otherwise, there is an attempt to read at
 * least one byte. If no byte is available because the stream is at end of
 * file, the value <code>-1</code> is returned; otherwise, at least one
 * byte is read and stored into <code>b</code>.
 *
 * <p> The first byte read is stored into element <code>b[off]</code>, the
 * next one into <code>b[off+1]</code>, and so on. The number of bytes read
 * is, at most, equal to <code>len</code>. Let <i>k</i> be the number of
 * bytes actually read; these bytes will be stored in elements
 * <code>b[off]</code> through <code>b[off+</code><i>k</i><code>-1]</code>,
```

```
 * leaving elements <code>b[off+</code><i>k</i><code>]</code> through
 * <code>b[off+len-1]</code> unaffected.
 *
 * <p> In every case, elements <code>b[0]</code> through
 * <code>b[off]</code> and elements <code>b[off+len]</code> through
 * <code>b[b.length-1]</code> are unaffected.
 *
 * @param      b     the buffer into which the data is read.
 * @param off the start offset in the destination array <code>b</code>
 * @param      len   the maximum number of bytes read.
 * @return     the total number of bytes read into the buffer, or
 *             <code>-1</code> if there is no more data because the end
 *             of the stream has been reached.
 * @exception  NullPointerException If <code>b</code> is <code>null</code>.
 * @exception  IndexOutOfBoundsException If <code>off</code> is negative,
 * <code>len</code> is negative, or <code>len</code> is greater than
 * <code>b.length - off</code>
 * @exception  IOException if the first byte cannot be read for any reason
 * other than end of file, the stream has been closed and the underlying
 * input stream does not support reading after close, or another I/O
 * error occurs.
 * @see        java.io.FilterInputStream#in
 * @see        java.io.InputStream#read(byte[], int, int)
 */
public final int read(byte b[], int off, int len) throws IOException {
    return in.read(b, off, len);
}

/**
 * See the general contract of the <code>readFully</code>
 * method of <code>DataInput</code>.
 * <p>
 * Bytes
 * for this operation are read from the contained
 * input stream.
 *
 * @param      b   the buffer into which the data is read.
 * @exception  EOFException  if this input stream reaches the end before
 *             reading all the bytes.
 * @exception  IOException   the stream has been closed and the contained
 *             input stream does not support reading after close, or
 *             another I/O error occurs.
 * @see        java.io.FilterInputStream#in
 */
public final void readFully(byte b[]) throws IOException {
    readFully(b, 0, b.length);
}

/**
 * See the general contract of the <code>readFully</code>
 * method of <code>DataInput</code>.
 * <p>
 * Bytes
 * for this operation are read from the contained
 * input stream.
 *
 * @param      b     the buffer into which the data is read.
 * @param      off   the start offset of the data.
 * @param      len   the number of bytes to read.
 * @exception  EOFException  if this input stream reaches the end before
 *               reading all the bytes.
 * @exception  IOException   the stream has been closed and the contained
 *             input stream does not support reading after close, or
```

```
     *             another I/O error occurs.
     * @see        java.io.FilterInputStream#in
     */
    public final void readFully(byte b[], int off, int len) throws IOException {
        if (len < 0)
            throw new IndexOutOfBoundsException();
        int n = 0;
        while (n < len) {
            int count = in.read(b, off + n, len - n);
            if (count < 0)
                throw new EOFException();
            n += count;
        }
    }

    /**
     * See the general contract of the <code>skipBytes</code>
     * method of <code>DataInput</code>.
     * <p>
     * Bytes for this operation are read from the contained
     * input stream.
     *
     * @param      n   the number of bytes to be skipped.
     * @return     the actual number of bytes skipped.
     * @exception  IOException  if the contained input stream does not support
     *             seek, or the stream has been closed and
     *             the contained input stream does not support
     *             reading after close, or another I/O error occurs.
     */
    public final int skipBytes(int n) throws IOException {
        int total = 0;
        int cur = 0;

        while ((total<n) && ((cur = (int) in.skip(n-total)) > 0)) {
            total += cur;
        }

        return total;
    }

    /**
     * See the general contract of the <code>readBoolean</code>
     * method of <code>DataInput</code>.
     * <p>
     * Bytes for this operation are read from the contained
     * input stream.
     *
     * @return     the <code>boolean</code> value read.
     * @exception  EOFException  if this input stream has reached the end.
     * @exception  IOException   the stream has been closed and the contained
     *             input stream does not support reading after close, or
     *             another I/O error occurs.
     * @see        java.io.FilterInputStream#in
     */
    public final boolean readBoolean() throws IOException {
        int ch = in.read();
        if (ch < 0)
            throw new EOFException();
        return (ch != 0);
    }

    /**
     * See the general contract of the <code>readByte</code>
```

```
 * method of <code>DataInput</code>.
 * <p>
 * Bytes
 * for this operation are read from the contained
 * input stream.
 *
 * @return     the next byte of this input stream as a signed 8-bit
 *             <code>byte</code>.
 * @exception  EOFException  if this input stream has reached the end.
 * @exception  IOException   the stream has been closed and the contained
 *             input stream does not support reading after close, or
 *             another I/O error occurs.
 * @see        java.io.FilterInputStream#in
 */
public final byte readByte() throws IOException {
    int ch = in.read();
    if (ch < 0)
        throw new EOFException();
    return (byte)(ch);
}

/**
 * See the general contract of the <code>readUnsignedByte</code>
 * method of <code>DataInput</code>.
 * <p>
 * Bytes
 * for this operation are read from the contained
 * input stream.
 *
 * @return     the next byte of this input stream, interpreted as an
 *             unsigned 8-bit number.
 * @exception  EOFException  if this input stream has reached the end.
 * @exception  IOException   the stream has been closed and the contained
 *             input stream does not support reading after close, or
 *             another I/O error occurs.
 * @see         java.io.FilterInputStream#in
 */
public final int readUnsignedByte() throws IOException {
    int ch = in.read();
    if (ch < 0)
        throw new EOFException();
    return ch;
}

/**
 * See the general contract of the <code>readShort</code>
 * method of <code>DataInput</code>.
 * <p>
 * Bytes
 * for this operation are read from the contained
 * input stream.
 *
 * @return     the next two bytes of this input stream, interpreted as a
 *             signed 16-bit number.
 * @exception  EOFException  if this input stream reaches the end before
 *               reading two bytes.
 * @exception  IOException   the stream has been closed and the contained
 *             input stream does not support reading after close, or
 *             another I/O error occurs.
 * @see        java.io.FilterInputStream#in
 */
public final short readShort() throws IOException {
    int ch1 = in.read();
```

```
    int ch2 = in.read();
    if ((ch1 | ch2) < 0)
        throw new EOFException();
    return (short)((ch1 << 8) + (ch2 << 0));
}

/**
 * See the general contract of the <code>readUnsignedShort</code>
 * method of <code>DataInput</code>.
 * <p>
 * Bytes
 * for this operation are read from the contained
 * input stream.
 *
 * @return     the next two bytes of this input stream, interpreted as an
 *             unsigned 16-bit integer.
 * @exception  EOFException  if this input stream reaches the end before
 *             reading two bytes.
 * @exception  IOException   the stream has been closed and the contained
 *             input stream does not support reading after close, or
 *             another I/O error occurs.
 * @see        java.io.FilterInputStream#in
 */
public final int readUnsignedShort() throws IOException {
    int ch1 = in.read();
    int ch2 = in.read();
    if ((ch1 | ch2) < 0)
        throw new EOFException();
    return (ch1 << 8) + (ch2 << 0);
}

/**
 * See the general contract of the <code>readChar</code>
 * method of <code>DataInput</code>.
 * <p>
 * Bytes
 * for this operation are read from the contained
 * input stream.
 *
 * @return     the next two bytes of this input stream, interpreted as a
 *             <code>char</code>.
 * @exception  EOFException  if this input stream reaches the end before
 *               reading two bytes.
 * @exception  IOException   the stream has been closed and the contained
 *             input stream does not support reading after close, or
 *             another I/O error occurs.
 * @see        java.io.FilterInputStream#in
 */
public final char readChar() throws IOException {
    int ch1 = in.read();
    int ch2 = in.read();
    if ((ch1 | ch2) < 0)
        throw new EOFException();
    return (char)((ch1 << 8) + (ch2 << 0));
}

/**
 * See the general contract of the <code>readInt</code>
 * method of <code>DataInput</code>.
 * <p>
 * Bytes
 * for this operation are read from the contained
 * input stream.
```

```
     *
     * @return     the next four bytes of this input stream, interpreted as an
     *             <code>int</code>.
     * @exception  EOFException  if this input stream reaches the end before
     *               reading four bytes.
     * @exception  IOException   the stream has been closed and the contained
     *             input stream does not support reading after close, or
     *             another I/O error occurs.
     * @see        java.io.FilterInputStream#in
     */
    public final int readInt() throws IOException {
        int ch1 = in.read();
        int ch2 = in.read();
        int ch3 = in.read();
        int ch4 = in.read();
        if ((ch1 | ch2 | ch3 | ch4) < 0)
            throw new EOFException();
        return ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4 << 0));
    }

    private byte readBuffer[] = new byte[8];

    /**
     * See the general contract of the <code>readLong</code>
     * method of <code>DataInput</code>.
     * <p>
     * Bytes
     * for this operation are read from the contained
     * input stream.
     *
     * @return     the next eight bytes of this input stream, interpreted as a
     *             <code>long</code>.
     * @exception  EOFException  if this input stream reaches the end before
     *               reading eight bytes.
     * @exception  IOException   the stream has been closed and the contained
     *             input stream does not support reading after close, or
     *             another I/O error occurs.
     * @see        java.io.FilterInputStream#in
     */
    public final long readLong() throws IOException {
        readFully(readBuffer, 0, 8);
        return (((long)readBuffer[0] << 56) +
                ((long)(readBuffer[1] & 255) << 48) +
                ((long)(readBuffer[2] & 255) << 40) +
                ((long)(readBuffer[3] & 255) << 32) +
                ((long)(readBuffer[4] & 255) << 24) +
                ((readBuffer[5] & 255) << 16) +
                ((readBuffer[6] & 255) <<  8) +
                ((readBuffer[7] & 255) <<  0));
    }

    /**
     * See the general contract of the <code>readFloat</code>
     * method of <code>DataInput</code>.
     * <p>
     * Bytes
     * for this operation are read from the contained
     * input stream.
     *
     * @return     the next four bytes of this input stream, interpreted as a
     *             <code>float</code>.
     * @exception  EOFException  if this input stream reaches the end before
     *               reading four bytes.
```

```
     * @exception  IOException   the stream has been closed and the contained
     *             input stream does not support reading after close, or
     *             another I/O error occurs.
     * @see        java.io.DataInputStream#readInt()
     * @see        java.lang.Float#intBitsToFloat(int)
     */
    public final float readFloat() throws IOException {
        return Float.intBitsToFloat(readInt());
    }

    /**
     * See the general contract of the <code>readDouble</code>
     * method of <code>DataInput</code>.
     * <p>
     * Bytes
     * for this operation are read from the contained
     * input stream.
     *
     * @return     the next eight bytes of this input stream, interpreted as a
     *             <code>double</code>.
     * @exception  EOFException  if this input stream reaches the end before
     *               reading eight bytes.
     * @exception  IOException   the stream has been closed and the contained
     *             input stream does not support reading after close, or
     *             another I/O error occurs.
     * @see        java.io.DataInputStream#readLong()
     * @see        java.lang.Double#longBitsToDouble(long)
     */
    public final double readDouble() throws IOException {
        return Double.longBitsToDouble(readLong());
    }

    private char lineBuffer[];

    /**
     * See the general contract of the <code>readLine</code>
     * method of <code>DataInput</code>.
     * <p>
     * Bytes
     * for this operation are read from the contained
     * input stream.
     *
     * @deprecated This method does not properly convert bytes to characters.
     * As of JDK 1.1, the preferred way to read lines of text is via the
     * <code>BufferedReader.readLine()</code> method.  Programs that use the
     * <code>DataInputStream</code> class to read lines can be converted to use
     * the <code>BufferedReader</code> class by replacing code of the form:
     * <blockquote><pre>
     *     DataInputStream d = new DataInputStream(in);
     * </pre></blockquote>
     * with:
     * <blockquote><pre>
     *     BufferedReader d
     *          = new BufferedReader(new InputStreamReader(in));
     * </pre></blockquote>
     *
     * @return     the next line of text from this input stream.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.BufferedReader#readLine()
     * @see        java.io.FilterInputStream#in
     */
    @Deprecated
    public final String readLine() throws IOException {
```

```
        char buf[] = lineBuffer;

        if (buf == null) {
            buf = lineBuffer = new char[128];
        }

        int room = buf.length;
        int offset = 0;
        int c;

loop:   while (true) {
            switch (c = in.read()) {
              case -1:
              case '\n':
                break loop;

              case '\r':
                int c2 = in.read();
                if ((c2 != '\n') && (c2 != -1)) {
                    if (!(in instanceof PushbackInputStream)) {
                        this.in = new PushbackInputStream(in);
                    }
                    ((PushbackInputStream)in).unread(c2);
                }
                break loop;

              default:
                if (--room < 0) {
                    buf = new char[offset + 128];
                    room = buf.length - offset - 1;
                    System.arraycopy(lineBuffer, 0, buf, 0, offset);
                    lineBuffer = buf;
                }
                buf[offset++] = (char) c;
                break;
            }
        }
        if ((c == -1) && (offset == 0)) {
            return null;
        }
        return String.copyValueOf(buf, 0, offset);
    }

    /**
     * See the general contract of the <code>readUTF</code>
     * method of <code>DataInput</code>.
     * <p>
     * Bytes
     * for this operation are read from the contained
     * input stream.
     *
     * @return     a Unicode string.
     * @exception  EOFException  if this input stream reaches the end before
     *               reading all the bytes.
     * @exception  IOException   the stream has been closed and the contained
     *             input stream does not support reading after close, or
     *             another I/O error occurs.
     * @exception  UTFDataFormatException if the bytes do not represent a valid
     *             modified UTF-8 encoding of a string.
     * @see        java.io.DataInputStream#readUTF(java.io.DataInput)
     */
    public final String readUTF() throws IOException {
        return readUTF(this);
```

```
}

/**
 * Reads from the
 * stream <code>in</code> a representation
 * of a Unicode  character string encoded in
 * <a href="DataInput.html#modified-utf-8">modified UTF-8</a> format;
 * this string of characters is then returned as a <code>String</code>.
 * The details of the modified UTF-8 representation
 * are  exactly the same as for the <code>readUTF</code>
 * method of <code>DataInput</code>.
 *
 * @param      in   a data input stream.
 * @return     a Unicode string.
 * @exception  EOFException            if the input stream reaches the end
 *               before all the bytes.
 * @exception  IOException   the stream has been closed and the contained
 *             input stream does not support reading after close, or
 *             another I/O error occurs.
 * @exception  UTFDataFormatException  if the bytes do not represent a
 *               valid modified UTF-8 encoding of a Unicode string.
 * @see        java.io.DataInputStream#readUnsignedShort()
 */
public final static String readUTF(DataInput in) throws IOException {
    int utflen = in.readUnsignedShort();
    byte[] bytearr = null;
    char[] chararr = null;
    if (in instanceof DataInputStream) {
        DataInputStream dis = (DataInputStream)in;
        if (dis.bytearr.length < utflen){
            dis.bytearr = new byte[utflen*2];
            dis.chararr = new char[utflen*2];
        }
        chararr = dis.chararr;
        bytearr = dis.bytearr;
    } else {
        bytearr = new byte[utflen];
        chararr = new char[utflen];
    }

    int c, char2, char3;
    int count = 0;
    int chararr_count=0;

    in.readFully(bytearr, 0, utflen);

    while (count < utflen) {
        c = (int) bytearr[count] & 0xff;
        if (c > 127) break;
        count++;
        chararr[chararr_count++]=(char)c;
    }

    while (count < utflen) {
        c = (int) bytearr[count] & 0xff;
        switch (c >> 4) {
            case 0: case 1: case 2: case 3: case 4: case 5: case 6: case 7:
                /* 0xxxxxxx*/
                count++;
                chararr[chararr_count++]=(char)c;
                break;
            case 12: case 13:
                /* 110x xxxx   10xx xxxx*/
```

```
                    count += 2;
                    if (count > utflen)
                        throw new UTFDataFormatException(
                            "malformed input: partial character at end");
                    char2 = (int) bytearr[count-1];
                    if ((char2 & 0xC0) != 0x80)
                        throw new UTFDataFormatException(
                            "malformed input around byte " + count);
                    chararr[chararr_count++]=(char)(((c & 0x1F) << 6) |
                                                    (char2 & 0x3F));
                    break;
                case 14:
                    /* 1110 xxxx  10xx xxxx  10xx xxxx */
                    count += 3;
                    if (count > utflen)
                        throw new UTFDataFormatException(
                            "malformed input: partial character at end");
                    char2 = (int) bytearr[count-2];
                    char3 = (int) bytearr[count-1];
                    if (((char2 & 0xC0) != 0x80) || ((char3 & 0xC0) != 0x80))
                        throw new UTFDataFormatException(
                            "malformed input around byte " + (count-1));
                    chararr[chararr_count++]=(char)(((c     & 0x0F) << 12) |
                                                    ((char2 & 0x3F) << 6)  |
                                                    ((char3 & 0x3F) << 0));
                    break;
                default:
                    /* 10xx xxxx,  1111 xxxx */
                    throw new UTFDataFormatException(
                        "malformed input around byte " + count);
            }
        }
        // The number of chars produced may be less than utflen
        return new String(chararr, 0, chararr_count);
    }
}
```

DataOutput.java

```
/*
 * Copyright (c) 1995, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * The <code>DataOutput</code> interface provides
 * for converting data from any of the Java
 * primitive types to a series of bytes and
 * writing these bytes to a binary stream.
 * There is  also a facility for converting
 * a <code>String</code> into
 * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
 * format and writing the resulting series
 * of bytes.
 * <p>
 * For all the methods in this interface that
 * write bytes, it is generally true that if
 * a byte cannot be written for any reason,
 * an <code>IOException</code> is thrown.
 *
 * @author  Frank Yellin
 * @see     java.io.DataInput
 * @see     java.io.DataOutputStream
 * @since   JDK1.0
 */
public
interface DataOutput {
    /**
     * Writes to the output stream the eight
     * low-order bits of the argument <code>b</code>.
     * The 24 high-order  bits of <code>b</code>
     * are ignored.
     *
     * @param      b   the byte to be written.
     * @throws     IOException  if an I/O error occurs.
     */
    void write(int b) throws IOException;
```

```
/**
 * Writes to the output stream all the bytes in array <code>b</code>.
 * If <code>b</code> is <code>null</code>,
 * a <code>NullPointerException</code> is thrown.
 * If <code>b.length</code> is zero, then
 * no bytes are written. Otherwise, the byte
 * <code>b[0]</code> is written first, then
 * <code>b[1]</code>, and so on; the last byte
 * written is <code>b[b.length-1]</code>.
 *
 * @param      b   the data.
 * @throws     IOException  if an I/O error occurs.
 */
void write(byte b[]) throws IOException;

/**
 * Writes <code>len</code> bytes from array
 * <code>b</code>, in order,  to
 * the output stream.  If <code>b</code>
 * is <code>null</code>, a <code>NullPointerException</code>
 * is thrown.  If <code>off</code> is negative,
 * or <code>len</code> is negative, or <code>off+len</code>
 * is greater than the length of the array
 * <code>b</code>, then an <code>IndexOutOfBoundsException</code>
 * is thrown.  If <code>len</code> is zero,
 * then no bytes are written. Otherwise, the
 * byte <code>b[off]</code> is written first,
 * then <code>b[off+1]</code>, and so on; the
 * last byte written is <code>b[off+len-1]</code>.
 *
 * @param      b     the data.
 * @param      off   the start offset in the data.
 * @param      len   the number of bytes to write.
 * @throws     IOException  if an I/O error occurs.
 */
void write(byte b[], int off, int len) throws IOException;

/**
 * Writes a <code>boolean</code> value to this output stream.
 * If the argument <code>v</code>
 * is <code>true</code>, the value <code>(byte)1</code>
 * is written; if <code>v</code> is <code>false</code>,
 * the  value <code>(byte)0</code> is written.
 * The byte written by this method may
 * be read by the <code>readBoolean</code>
 * method of interface <code>DataInput</code>,
 * which will then return a <code>boolean</code>
 * equal to <code>v</code>.
 *
 * @param      v   the boolean to be written.
 * @throws     IOException  if an I/O error occurs.
 */
void writeBoolean(boolean v) throws IOException;

/**
 * Writes to the output stream the eight low-
 * order bits of the argument <code>v</code>.
 * The 24 high-order bits of <code>v</code>
 * are ignored. (This means  that <code>writeByte</code>
 * does exactly the same thing as <code>write</code>
 * for an integer argument.) The byte written
 * by this method may be read by the <code>readByte</code>
```

```
 * method of interface <code>DataInput</code>,
 * which will then return a <code>byte</code>
 * equal to <code>(byte)v</code>.
 *
 * @param      v   the byte value to be written.
 * @throws     IOException  if an I/O error occurs.
 */
void writeByte(int v) throws IOException;

/**
 * Writes two bytes to the output
 * stream to represent the value of the argument.
 * The byte values to be written, in the  order
 * shown, are:
 * <pre>{@code
 * (byte)(0xff & (v >> 8))
 * (byte)(0xff & v)
 * }</pre> <p>
 * The bytes written by this method may be
 * read by the <code>readShort</code> method
 * of interface <code>DataInput</code> , which
 * will then return a <code>short</code> equal
 * to <code>(short)v</code>.
 *
 * @param      v   the <code>short</code> value to be written.
 * @throws     IOException  if an I/O error occurs.
 */
void writeShort(int v) throws IOException;

/**
 * Writes a <code>char</code> value, which
 * is comprised of two bytes, to the
 * output stream.
 * The byte values to be written, in the  order
 * shown, are:
 * <pre>{@code
 * (byte)(0xff & (v >> 8))
 * (byte)(0xff & v)
 * }</pre><p>
 * The bytes written by this method may be
 * read by the <code>readChar</code> method
 * of interface <code>DataInput</code> , which
 * will then return a <code>char</code> equal
 * to <code>(char)v</code>.
 *
 * @param      v   the <code>char</code> value to be written.
 * @throws     IOException  if an I/O error occurs.
 */
void writeChar(int v) throws IOException;

/**
 * Writes an <code>int</code> value, which is
 * comprised of four bytes, to the output stream.
 * The byte values to be written, in the  order
 * shown, are:
 * <pre>{@code
 * (byte)(0xff & (v >> 24))
 * (byte)(0xff & (v >> 16))
 * (byte)(0xff & (v >>  8))
 * (byte)(0xff & v)
 * }</pre><p>
 * The bytes written by this method may be read
 * by the <code>readInt</code> method of interface
```

```
 * <code>DataInput</code> , which will then
 * return an <code>int</code> equal to <code>v</code>.
 *
 * @param      v   the <code>int</code> value to be written.
 * @throws     IOException  if an I/O error occurs.
 */
void writeInt(int v) throws IOException;

/**
 * Writes a <code>long</code> value, which is
 * comprised of eight bytes, to the output stream.
 * The byte values to be written, in the  order
 * shown, are:
 * <pre>{@code
 * (byte)(0xff & (v >> 56))
 * (byte)(0xff & (v >> 48))
 * (byte)(0xff & (v >> 40))
 * (byte)(0xff & (v >> 32))
 * (byte)(0xff & (v >> 24))
 * (byte)(0xff & (v >> 16))
 * (byte)(0xff & (v >>  8))
 * (byte)(0xff & v)
 * }</pre><p>
 * The bytes written by this method may be
 * read by the <code>readLong</code> method
 * of interface <code>DataInput</code> , which
 * will then return a <code>long</code> equal
 * to <code>v</code>.
 *
 * @param      v   the <code>long</code> value to be written.
 * @throws     IOException  if an I/O error occurs.
 */
void writeLong(long v) throws IOException;

/**
 * Writes a <code>float</code> value,
 * which is comprised of four bytes, to the output stream.
 * It does this as if it first converts this
 * <code>float</code> value to an <code>int</code>
 * in exactly the manner of the <code>Float.floatToIntBits</code>
 * method  and then writes the <code>int</code>
 * value in exactly the manner of the  <code>writeInt</code>
 * method.  The bytes written by this method
 * may be read by the <code>readFloat</code>
 * method of interface <code>DataInput</code>,
 * which will then return a <code>float</code>
 * equal to <code>v</code>.
 *
 * @param      v   the <code>float</code> value to be written.
 * @throws     IOException  if an I/O error occurs.
 */
void writeFloat(float v) throws IOException;

/**
 * Writes a <code>double</code> value,
 * which is comprised of eight bytes, to the output stream.
 * It does this as if it first converts this
 * <code>double</code> value to a <code>long</code>
 * in exactly the manner of the <code>Double.doubleToLongBits</code>
 * method  and then writes the <code>long</code>
 * value in exactly the manner of the  <code>writeLong</code>
 * method. The bytes written by this method
 * may be read by the <code>readDouble</code>
```

```
     * method of interface <code>DataInput</code>,
     * which will then return a <code>double</code>
     * equal to <code>v</code>.
     *
     * @param      v   the <code>double</code> value to be written.
     * @throws     IOException  if an I/O error occurs.
     */
    void writeDouble(double v) throws IOException;

    /**
     * Writes a string to the output stream.
     * For every character in the string
     * <code>s</code>,  taken in order, one byte
     * is written to the output stream.  If
     * <code>s</code> is <code>null</code>, a <code>NullPointerException</code>
     * is thrown.<p>  If <code>s.length</code>
     * is zero, then no bytes are written. Otherwise,
     * the character <code>s[0]</code> is written
     * first, then <code>s[1]</code>, and so on;
     * the last character written is <code>s[s.length-1]</code>.
     * For each character, one byte is written,
     * the low-order byte, in exactly the manner
     * of the <code>writeByte</code> method . The
     * high-order eight bits of each character
     * in the string are ignored.
     *
     * @param      s   the string of bytes to be written.
     * @throws     IOException  if an I/O error occurs.
     */
    void writeBytes(String s) throws IOException;

    /**
     * Writes every character in the string <code>s</code>,
     * to the output stream, in order,
     * two bytes per character. If <code>s</code>
     * is <code>null</code>, a <code>NullPointerException</code>
     * is thrown.  If <code>s.length</code>
     * is zero, then no characters are written.
     * Otherwise, the character <code>s[0]</code>
     * is written first, then <code>s[1]</code>,
     * and so on; the last character written is
     * <code>s[s.length-1]</code>. For each character,
     * two bytes are actually written, high-order
     * byte first, in exactly the manner of the
     * <code>writeChar</code> method.
     *
     * @param      s   the string value to be written.
     * @throws     IOException  if an I/O error occurs.
     */
    void writeChars(String s) throws IOException;

    /**
     * Writes two bytes of length information
     * to the output stream, followed
     * by the
     * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
     * representation
     * of  every character in the string <code>s</code>.
     * If <code>s</code> is <code>null</code>,
     * a <code>NullPointerException</code> is thrown.
     * Each character in the string <code>s</code>
     * is converted to a group of one, two, or
     * three bytes, depending on the value of the
```

```
     * character.<p>
     * If a character <code>c</code>
     * is in the range <code>\u0001</code> through
     * <code>\u007f</code>, it is represented
     * by one byte:
     * <pre>(byte)c </pre>  <p>
     * If a character <code>c</code> is <code>\u0000</code>
     * or is in the range <code>\u0080</code>
     * through <code>\u07ff</code>, then it is
     * represented by two bytes, to be written
     * in the order shown: <pre>{@code
     * (byte)(0xc0 | (0x1f & (c >> 6)))
     * (byte)(0x80 | (0x3f & c))
     * }</pre> <p> If a character
     * <code>c</code> is in the range <code>\u0800</code>
     * through <code>uffff</code>, then it is
     * represented by three bytes, to be written
     * in the order shown: <pre>{@code
     * (byte)(0xe0 | (0x0f & (c >> 12)))
     * (byte)(0x80 | (0x3f & (c >>  6)))
     * (byte)(0x80 | (0x3f & c))
     * }</pre>  <p> First,
     * the total number of bytes needed to represent
     * all the characters of <code>s</code> is
     * calculated. If this number is larger than
     * <code>65535</code>, then a <code>UTFDataFormatException</code>
     * is thrown. Otherwise, this length is written
     * to the output stream in exactly the manner
     * of the <code>writeShort</code> method;
     * after this, the one-, two-, or three-byte
     * representation of each character in the
     * string <code>s</code> is written.<p>  The
     * bytes written by this method may be read
     * by the <code>readUTF</code> method of interface
     * <code>DataInput</code> , which will then
     * return a <code>String</code> equal to <code>s</code>.
     *
     * @param      s   the string value to be written.
     * @throws     IOException  if an I/O error occurs.
     */
    void writeUTF(String s) throws IOException;
}
```

DataOutputStream.java

```
/*
 * Copyright (c) 1994, 2004, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * A data output stream lets an application write primitive Java data
 * types to an output stream in a portable way. An application can
 * then use a data input stream to read the data back in.
 *
 * @author  unascribed
 * @see     java.io.DataInputStream
 * @since   JDK1.0
 */
public
class DataOutputStream extends FilterOutputStream implements DataOutput {
    /**
     * The number of bytes written to the data output stream so far.
     * If this counter overflows, it will be wrapped to Integer.MAX_VALUE.
     */
    protected int written;

    /**
     * bytearr is initialized on demand by writeUTF
     */
    private byte[] bytearr = null;

    /**
     * Creates a new data output stream to write data to the specified
     * underlying output stream. The counter <code>written</code> is
     * set to zero.
     *
     * @param   out   the underlying output stream, to be saved for later
     *                use.
     * @see     java.io.FilterOutputStream#out
     */
    public DataOutputStream(OutputStream out) {
        super(out);
```

```
}

/**
 * Increases the written counter by the specified value
 * until it reaches Integer.MAX_VALUE.
 */
private void incCount(int value) {
    int temp = written + value;
    if (temp < 0) {
        temp = Integer.MAX_VALUE;
    }
    written = temp;
}

/**
 * Writes the specified byte (the low eight bits of the argument
 * <code>b</code>) to the underlying output stream. If no exception
 * is thrown, the counter <code>written</code> is incremented by
 * <code>1</code>.
 * <p>
 * Implements the <code>write</code> method of <code>OutputStream</code>.
 *
 * @param      b   the <code>byte</code> to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 */
public synchronized void write(int b) throws IOException {
    out.write(b);
    incCount(1);
}

/**
 * Writes <code>len</code> bytes from the specified byte array
 * starting at offset <code>off</code> to the underlying output stream.
 * If no exception is thrown, the counter <code>written</code> is
 * incremented by <code>len</code>.
 *
 * @param      b     the data.
 * @param      off   the start offset in the data.
 * @param      len   the number of bytes to write.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 */
public synchronized void write(byte b[], int off, int len)
    throws IOException
{
    out.write(b, off, len);
    incCount(len);
}

/**
 * Flushes this data output stream. This forces any buffered output
 * bytes to be written out to the stream.
 * <p>
 * The <code>flush</code> method of <code>DataOutputStream</code>
 * calls the <code>flush</code> method of its underlying output stream.
 *
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 * @see        java.io.OutputStream#flush()
 */
public void flush() throws IOException {
    out.flush();
```

```
}

/**
 * Writes a <code>boolean</code> to the underlying output stream as
 * a 1-byte value. The value <code>true</code> is written out as the
 * value <code>(byte)1</code>; the value <code>false</code> is
 * written out as the value <code>(byte)0</code>. If no exception is
 * thrown, the counter <code>written</code> is incremented by
 * <code>1</code>.
 *
 * @param      v   a <code>boolean</code> value to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 */
public final void writeBoolean(boolean v) throws IOException {
    out.write(v ? 1 : 0);
    incCount(1);
}

/**
 * Writes out a <code>byte</code> to the underlying output stream as
 * a 1-byte value. If no exception is thrown, the counter
 * <code>written</code> is incremented by <code>1</code>.
 *
 * @param      v   a <code>byte</code> value to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 */
public final void writeByte(int v) throws IOException {
    out.write(v);
    incCount(1);
}

/**
 * Writes a <code>short</code> to the underlying output stream as two
 * bytes, high byte first. If no exception is thrown, the counter
 * <code>written</code> is incremented by <code>2</code>.
 *
 * @param      v   a <code>short</code> to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 */
public final void writeShort(int v) throws IOException {
    out.write((v >>> 8) & 0xFF);
    out.write((v >>> 0) & 0xFF);
    incCount(2);
}

/**
 * Writes a <code>char</code> to the underlying output stream as a
 * 2-byte value, high byte first. If no exception is thrown, the
 * counter <code>written</code> is incremented by <code>2</code>.
 *
 * @param      v   a <code>char</code> value to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 */
public final void writeChar(int v) throws IOException {
    out.write((v >>> 8) & 0xFF);
    out.write((v >>> 0) & 0xFF);
    incCount(2);
}
```

```
/**
 * Writes an <code>int</code> to the underlying output stream as four
 * bytes, high byte first. If no exception is thrown, the counter
 * <code>written</code> is incremented by <code>4</code>.
 *
 * @param      v   an <code>int</code> to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 */
public final void writeInt(int v) throws IOException {
    out.write((v >>> 24) & 0xFF);
    out.write((v >>> 16) & 0xFF);
    out.write((v >>>  8) & 0xFF);
    out.write((v >>>  0) & 0xFF);
    incCount(4);
}

private byte writeBuffer[] = new byte[8];

/**
 * Writes a <code>long</code> to the underlying output stream as eight
 * bytes, high byte first. In no exception is thrown, the counter
 * <code>written</code> is incremented by <code>8</code>.
 *
 * @param      v   a <code>long</code> to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 */
public final void writeLong(long v) throws IOException {
    writeBuffer[0] = (byte)(v >>> 56);
    writeBuffer[1] = (byte)(v >>> 48);
    writeBuffer[2] = (byte)(v >>> 40);
    writeBuffer[3] = (byte)(v >>> 32);
    writeBuffer[4] = (byte)(v >>> 24);
    writeBuffer[5] = (byte)(v >>> 16);
    writeBuffer[6] = (byte)(v >>>  8);
    writeBuffer[7] = (byte)(v >>>  0);
    out.write(writeBuffer, 0, 8);
    incCount(8);
}

/**
 * Converts the float argument to an <code>int</code> using the
 * <code>floatToIntBits</code> method in class <code>Float</code>,
 * and then writes that <code>int</code> value to the underlying
 * output stream as a 4-byte quantity, high byte first. If no
 * exception is thrown, the counter <code>written</code> is
 * incremented by <code>4</code>.
 *
 * @param      v   a <code>float</code> value to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#out
 * @see        java.lang.Float#floatToIntBits(float)
 */
public final void writeFloat(float v) throws IOException {
    writeInt(Float.floatToIntBits(v));
}

/**
 * Converts the double argument to a <code>long</code> using the
 * <code>doubleToLongBits</code> method in class <code>Double</code>,
 * and then writes that <code>long</code> value to the underlying
 * output stream as an 8-byte quantity, high byte first. If no
```

```
     * exception is thrown, the counter <code>written</code> is
     * incremented by <code>8</code>.
     *
     * @param      v   a <code>double</code> value to be written.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#out
     * @see        java.lang.Double#doubleToLongBits(double)
     */
    public final void writeDouble(double v) throws IOException {
        writeLong(Double.doubleToLongBits(v));
    }

    /**
     * Writes out the string to the underlying output stream as a
     * sequence of bytes. Each character in the string is written out, in
     * sequence, by discarding its high eight bits. If no exception is
     * thrown, the counter <code>written</code> is incremented by the
     * length of <code>s</code>.
     *
     * @param      s   a string of bytes to be written.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#out
     */
    public final void writeBytes(String s) throws IOException {
        int len = s.length();
        for (int i = 0 ; i < len ; i++) {
            out.write((byte)s.charAt(i));
        }
        incCount(len);
    }

    /**
     * Writes a string to the underlying output stream as a sequence of
     * characters. Each character is written to the data output stream as
     * if by the <code>writeChar</code> method. If no exception is
     * thrown, the counter <code>written</code> is incremented by twice
     * the length of <code>s</code>.
     *
     * @param      s   a <code>String</code> value to be written.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.DataOutputStream#writeChar(int)
     * @see        java.io.FilterOutputStream#out
     */
    public final void writeChars(String s) throws IOException {
        int len = s.length();
        for (int i = 0 ; i < len ; i++) {
            int v = s.charAt(i);
            out.write((v >>> 8) & 0xFF);
            out.write((v >>> 0) & 0xFF);
        }
        incCount(len * 2);
    }

    /**
     * Writes a string to the underlying output stream using
     * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
     * encoding in a machine-independent manner.
     * <p>
     * First, two bytes are written to the output stream as if by the
     * <code>writeShort</code> method giving the number of bytes to
     * follow. This value is the number of bytes actually written out,
     * not the length of the string. Following the length, each character
     * of the string is output, in sequence, using the modified UTF-8 encoding
```

```
     * for the character. If no exception is thrown, the counter
     * <code>written</code> is incremented by the total number of
     * bytes written to the output stream. This will be at least two
     * plus the length of <code>str</code>, and at most two plus
     * thrice the length of <code>str</code>.
     *
     * @param      str   a string to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public final void writeUTF(String str) throws IOException {
        writeUTF(str, this);
    }

    /**
     * Writes a string to the specified DataOutput using
     * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
     * encoding in a machine-independent manner.
     * <p>
     * First, two bytes are written to out as if by the <code>writeShort</code>
     * method giving the number of bytes to follow. This value is the number of
     * bytes actually written out, not the length of the string. Following the
     * length, each character of the string is output, in sequence, using the
     * modified UTF-8 encoding for the character. If no exception is thrown, the
     * counter <code>written</code> is incremented by the total number of
     * bytes written to the output stream. This will be at least two
     * plus the length of <code>str</code>, and at most two plus
     * thrice the length of <code>str</code>.
     *
     * @param      str   a string to be written.
     * @param      out   destination to write to
     * @return     The number of bytes written out.
     * @exception  IOException  if an I/O error occurs.
     */
    static int writeUTF(String str, DataOutput out) throws IOException {
        int strlen = str.length();
        int utflen = 0;
        int c, count = 0;

        /* use charAt instead of copying String to char array */
        for (int i = 0; i < strlen; i++) {
            c = str.charAt(i);
            if ((c >= 0x0001) && (c <= 0x007F)) {
                utflen++;
            } else if (c > 0x07FF) {
                utflen += 3;
            } else {
                utflen += 2;
            }
        }

        if (utflen > 65535)
            throw new UTFDataFormatException(
                "encoded string too long: " + utflen + " bytes");

        byte[] bytearr = null;
        if (out instanceof DataOutputStream) {
            DataOutputStream dos = (DataOutputStream)out;
            if(dos.bytearr == null || (dos.bytearr.length < (utflen+2)))
                dos.bytearr = new byte[(utflen*2) + 2];
            bytearr = dos.bytearr;
        } else {
            bytearr = new byte[utflen+2];
        }
```

```
        bytearr[count++] = (byte) ((utflen >>> 8) & 0xFF);
        bytearr[count++] = (byte) ((utflen >>> 0) & 0xFF);

        int i=0;
        for (i=0; i<strlen; i++) {
           c = str.charAt(i);
           if (!((c >= 0x0001) && (c <= 0x007F))) break;
           bytearr[count++] = (byte) c;
        }

        for (;i < strlen; i++){
            c = str.charAt(i);
            if ((c >= 0x0001) && (c <= 0x007F)) {
                bytearr[count++] = (byte) c;

            } else if (c > 0x07FF) {
                bytearr[count++] = (byte) (0xE0 | ((c >> 12) & 0x0F));
                bytearr[count++] = (byte) (0x80 | ((c >>  6) & 0x3F));
                bytearr[count++] = (byte) (0x80 | ((c >>  0) & 0x3F));
            } else {
                bytearr[count++] = (byte) (0xC0 | ((c >>  6) & 0x1F));
                bytearr[count++] = (byte) (0x80 | ((c >>  0) & 0x3F));
            }
        }
        out.write(bytearr, 0, utflen+2);
        return utflen + 2;
    }

    /**
     * Returns the current value of the counter <code>written</code>,
     * the number of bytes written to this data output stream so far.
     * If the counter overflows, it will be wrapped to Integer.MAX_VALUE.
     *
     * @return  the value of the <code>written</code> field.
     * @see     java.io.DataOutputStream#written
     */
    public final int size() {
        return written;
    }
}
```

DefaultFileSystem.java

```
/*
 * Copyright (c) 2012, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 *
 * @since 1.8
 */
class DefaultFileSystem {

    /**
     * Return the FileSystem object for Windows platform.
     */
    public static FileSystem getFileSystem() {
        return new WinNTFileSystem();
    }
}
```

DeleteOnExitHook.java

```
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
package java.io;

import java.util.*;
import java.io.File;

/**
 * This class holds a set of filenames to be deleted on VM exit through a shutdown hook.
 * A set is used both to prevent double-insertion of the same file as well as offer
 * quick removal.
 */

class DeleteOnExitHook {
    private static LinkedHashSet<String> files = new LinkedHashSet<>();
    static {
        // DeleteOnExitHook must be the last shutdown hook to be invoked.
        // Application shutdown hooks may add the first file to the
        // delete on exit list and cause the DeleteOnExitHook to be
        // registered during shutdown in progress. So set the
        // registerShutdownInProgress parameter to true.
        sun.misc.SharedSecrets.getJavaLangAccess()
            .registerShutdownHook(2 /* Shutdown hook invocation order */,
                true /* register even if shutdown in progress */,
                new Runnable() {
                    public void run() {
                       runHooks();
                    }
                }
        );
    }

    private DeleteOnExitHook() {}

    static synchronized void add(String file) {
        if(files == null) {
            // DeleteOnExitHook is running. Too late to add a file
            throw new IllegalStateException("Shutdown in progress");
```

```
        }

        files.add(file);
    }

    static void runHooks() {
        LinkedHashSet<String> theFiles;

        synchronized (DeleteOnExitHook.class) {
            theFiles = files;
            files = null;
        }

        ArrayList<String> toBeDeleted = new ArrayList<>(theFiles);

        // reverse the list to maintain previous jdk deletion order.
        // Last in first deleted.
        Collections.reverse(toBeDeleted);
        for (String filename : toBeDeleted) {
            (new File(filename)).delete();
        }
    }
}
```

EOFException.java

```
/*
 * Copyright (c) 1995, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Signals that an end of file or end of stream has been reached
 * unexpectedly during input.
 * <p>
 * This exception is mainly used by data input streams to signal end of
 * stream. Note that many other input operations return a special value on
 * end of stream rather than throwing an exception.
 *
 * @author  Frank Yellin
 * @see     java.io.DataInputStream
 * @see     java.io.IOException
 * @since   JDK1.0
 */
public
class EOFException extends IOException {
    private static final long serialVersionUID = 6433858223774886977L;

    /**
     * Constructs an <code>EOFException</code> with <code>null</code>
     * as its error detail message.
     */
    public EOFException() {
        super();
    }

    /**
     * Constructs an <code>EOFException</code> with the specified detail
     * message. The string <code>s</code> may later be retrieved by the
     * <code>{@link java.lang.Throwable#getMessage}</code> method of class
     * <code>java.lang.Throwable</code>.
     *
     * @param   s   the detail message.
     */
```

```
    public EOFException(String s) {
        super(s);
    }
}
```

ExpiringCache.java

```
/*
 * Copyright (c) 2002, 2011, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 */

package java.io;

import java.util.Iterator;
import java.util.Map;
import java.util.LinkedHashMap;
import java.util.Set;

class ExpiringCache {
    private long millisUntilExpiration;
    private Map<String,Entry> map;
    // Clear out old entries every few queries
    private int queryCount;
    private int queryOverflow = 300;
    private int MAX_ENTRIES = 200;

    static class Entry {
        private long   timestamp;
        private String val;

        Entry(long timestamp, String val) {
            this.timestamp = timestamp;
            this.val = val;
        }

        long   timestamp()                 { return timestamp;           }
        void   setTimestamp(long timestamp) { this.timestamp = timestamp; }

        String val()                       { return val;                 }
        void   setVal(String val)          { this.val = val;             }
    }

    ExpiringCache() {
```

```
    this(30000);
}

@SuppressWarnings("serial")
ExpiringCache(long millisUntilExpiration) {
    this.millisUntilExpiration = millisUntilExpiration;
    map = new LinkedHashMap<String,Entry>() {
        protected boolean removeEldestEntry(Map.Entry<String,Entry> eldest) {
          return size() > MAX_ENTRIES;
        }
      };
}

synchronized String get(String key) {
    if (++queryCount >= queryOverflow) {
        cleanup();
    }
    Entry entry = entryFor(key);
    if (entry != null) {
        return entry.val();
    }
    return null;
}

synchronized void put(String key, String val) {
    if (++queryCount >= queryOverflow) {
        cleanup();
    }
    Entry entry = entryFor(key);
    if (entry != null) {
        entry.setTimestamp(System.currentTimeMillis());
        entry.setVal(val);
    } else {
        map.put(key, new Entry(System.currentTimeMillis(), val));
    }
}

synchronized void clear() {
    map.clear();
}

private Entry entryFor(String key) {
    Entry entry = map.get(key);
    if (entry != null) {
        long delta = System.currentTimeMillis() - entry.timestamp();
        if (delta < 0 || delta >= millisUntilExpiration) {
            map.remove(key);
            entry = null;
        }
    }
    return entry;
}

private void cleanup() {
    Set<String> keySet = map.keySet();
    // Avoid ConcurrentModificationExceptions
    String[] keys = new String[keySet.size()];
    int i = 0;
    for (String key: keySet) {
        keys[i++] = key;
    }
    for (int j = 0; j < keys.length; j++) {
        entryFor(keys[j]);
```

```
        }
        queryCount = 0;
    }
}
```

Externalizable.java

```
/*
 * Copyright (c) 1996, 2004, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.io.ObjectOutput;
import java.io.ObjectInput;

/**
 * Only the identity of the class of an Externalizable instance is
 * written in the serialization stream and it is the responsibility
 * of the class to save and restore the contents of its instances.
 *
 * The writeExternal and readExternal methods of the Externalizable
 * interface are implemented by a class to give the class complete
 * control over the format and contents of the stream for an object
 * and its supertypes. These methods must explicitly
 * coordinate with the supertype to save its state. These methods supersede
 * customized implementations of writeObject and readObject methods.<br>
 *
 * Object Serialization uses the Serializable and Externalizable
 * interfaces.  Object persistence mechanisms can use them as well.  Each
 * object to be stored is tested for the Externalizable interface. If
 * the object supports Externalizable, the writeExternal method is called. If the
 * object does not support Externalizable and does implement
 * Serializable, the object is saved using
 * ObjectOutputStream. <br> When an Externalizable object is
 * reconstructed, an instance is created using the public no-arg
 * constructor, then the readExternal method called.  Serializable
 * objects are restored by reading them from an ObjectInputStream.<br>
 *
 * An Externalizable instance can designate a substitution object via
 * the writeReplace and readResolve methods documented in the Serializable
 * interface.<br>
 *
 * @author  unascribed
 * @see java.io.ObjectOutputStream
 * @see java.io.ObjectInputStream
```

```
 * @see java.io.ObjectOutput
 * @see java.io.ObjectInput
 * @see java.io.Serializable
 * @since   JDK1.1
 */
public interface Externalizable extends java.io.Serializable {
    /**
     * The object implements the writeExternal method to save its contents
     * by calling the methods of DataOutput for its primitive values or
     * calling the writeObject method of ObjectOutput for objects, strings,
     * and arrays.
     *
     * @serialData Overriding methods should use this tag to describe
     *             the data layout of this Externalizable object.
     *             List the sequence of element types and, if possible,
     *             relate the element to a public/protected field and/or
     *             method of this Externalizable class.
     *
     * @param out the stream to write the object to
     * @exception IOException Includes any I/O exceptions that may occur
     */
    void writeExternal(ObjectOutput out) throws IOException;

    /**
     * The object implements the readExternal method to restore its
     * contents by calling the methods of DataInput for primitive
     * types and readObject for objects, strings and arrays.  The
     * readExternal method must read the values in the same sequence
     * and with the same types as were written by writeExternal.
     *
     * @param in the stream to read data from in order to restore the object
     * @exception IOException if I/O errors occur
     * @exception ClassNotFoundException If the class for an object being
     *              restored cannot be found.
     */
    void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
}
```

File.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.net.URI;
import java.net.URL;
import java.net.MalformedURLException;
import java.net.URISyntaxException;
import java.util.List;
import java.util.ArrayList;
import java.security.AccessController;
import java.security.SecureRandom;
import java.nio.file.Path;
import java.nio.file.FileSystems;
import sun.security.action.GetPropertyAction;

/**
 * An abstract representation of file and directory pathnames.
 *
 * <p> User interfaces and operating systems use system-dependent <em>pathname
 * strings</em> to name files and directories.  This class presents an
 * abstract, system-independent view of hierarchical pathnames.  An
 * <em>abstract pathname</em> has two components:
 *
 * <ol>
 * <li> An optional system-dependent <em>prefix</em> string,
 *      such as a disk-drive specifier, <code>"/"</code> for the UNIX root
 *      directory, or <code>"\\\\"</code> for a Microsoft Windows UNC pathname, and
 * <li> A sequence of zero or more string <em>names</em>.
 * </ol>
 *
 * The first name in an abstract pathname may be a directory name or, in the
 * case of Microsoft Windows UNC pathnames, a hostname.  Each subsequent name
 * in an abstract pathname denotes a directory; the last name may denote
 * either a directory or a file.  The <em>empty</em> abstract pathname has no
 * prefix and an empty name sequence.
 *
```

```
* <p> The conversion of a pathname string to or from an abstract pathname is
* inherently system-dependent.  When an abstract pathname is converted into a
* pathname string, each name is separated from the next by a single copy of
* the default <em>separator character</em>.  The default name-separator
* character is defined by the system property <code>file.separator</code>, and
* is made available in the public static fields <code>{@link
* #separator}</code> and <code>{@link #separatorChar}</code> of this class.
* When a pathname string is converted into an abstract pathname, the names
* within it may be separated by the default name-separator character or by any
* other name-separator character that is supported by the underlying system.
*
* <p> A pathname, whether abstract or in string form, may be either
* <em>absolute</em> or <em>relative</em>.  An absolute pathname is complete in
* that no other information is required in order to locate the file that it
* denotes.  A relative pathname, in contrast, must be interpreted in terms of
* information taken from some other pathname.  By default the classes in the
* <code>java.io</code> package always resolve relative pathnames against the
* current user directory.  This directory is named by the system property
* <code>user.dir</code>, and is typically the directory in which the Java
* virtual machine was invoked.
*
* <p> The <em>parent</em> of an abstract pathname may be obtained by invoking
* the {@link #getParent} method of this class and consists of the pathname's
* prefix and each name in the pathname's name sequence except for the last.
* Each directory's absolute pathname is an ancestor of any <tt>File</tt>
* object with an absolute abstract pathname which begins with the directory's
* absolute pathname.  For example, the directory denoted by the abstract
* pathname <tt>"/usr"</tt> is an ancestor of the directory denoted by the
* pathname <tt>"/usr/local/bin"</tt>.
*
* <p> The prefix concept is used to handle root directories on UNIX platforms,
* and drive specifiers, root directories and UNC pathnames on Microsoft Windows platforms,
* as follows:
*
* <ul>
*
* <li> For UNIX platforms, the prefix of an absolute pathname is always
* <code>"/"</code>.  Relative pathnames have no prefix.  The abstract pathname
* denoting the root directory has the prefix <code>"/"</code> and an empty
* name sequence.
*
* <li> For Microsoft Windows platforms, the prefix of a pathname that contains a drive
* specifier consists of the drive letter followed by <code>":"</code> and
* possibly followed by <code>"\\"</code> if the pathname is absolute.  The
* prefix of a UNC pathname is <code>"\\\\"</code>; the hostname and the share
* name are the first two names in the name sequence.  A relative pathname that
* does not specify a drive has no prefix.
*
* </ul>
*
* <p> Instances of this class may or may not denote an actual file-system
* object such as a file or a directory.  If it does denote such an object
* then that object resides in a <i>partition</i>.  A partition is an
* operating system-specific portion of storage for a file system.  A single
* storage device (e.g. a physical disk-drive, flash memory, CD-ROM) may
* contain multiple partitions.  The object, if any, will reside on the
* partition <a name="partName">named</a> by some ancestor of the absolute
* form of this pathname.
*
* <p> A file system may implement restrictions to certain operations on the
* actual file-system object, such as reading, writing, and executing.  These
* restrictions are collectively known as <i>access permissions</i>.  The file
* system may have multiple sets of access permissions on a single object.
```

```
 * For example, one set may apply to the object's <i>owner</i>, and another
 * may apply to all other users.  The access permissions on an object may
 * cause some methods in this class to fail.
 *
 * <p> Instances of the <code>File</code> class are immutable; that is, once
 * created, the abstract pathname represented by a <code>File</code> object
 * will never change.
 *
 * <h3>Interoperability with {@code java.nio.file} package</h3>
 *
 * <p> The <a href="../../java/nio/file/package-summary.html">{@code java.nio.file}</a>
 * package defines interfaces and classes for the Java virtual machine to access
 * files, file attributes, and file systems. This API may be used to overcome
 * many of the limitations of the {@code java.io.File} class.
 * The {@link #toPath toPath} method may be used to obtain a {@link
 * Path} that uses the abstract path represented by a {@code File} object to
 * locate a file. The resulting {@code Path} may be used with the {@link
 * java.nio.file.Files} class to provide more efficient and extensive access to
 * additional file operations, file attributes, and I/O exceptions to help
 * diagnose errors when an operation on a file fails.
 *
 * @author  unascribed
 * @since   JDK1.0
 */

public class File
    implements Serializable, Comparable<File>
{

    /**
     * The FileSystem object representing the platform's local file system.
     */
    private static final FileSystem fs = DefaultFileSystem.getFileSystem();

    /**
     * This abstract pathname's normalized pathname string. A normalized
     * pathname string uses the default name-separator character and does not
     * contain any duplicate or redundant separators.
     *
     * @serial
     */
    private final String path;

    /**
     * Enum type that indicates the status of a file path.
     */
    private static enum PathStatus { INVALID, CHECKED };

    /**
     * The flag indicating whether the file path is invalid.
     */
    private transient PathStatus status = null;

    /**
     * Check if the file has an invalid path. Currently, the inspection of
     * a file path is very limited, and it only covers Nul character check.
     * Returning true means the path is definitely invalid/garbage. But
     * returning false does not guarantee that the path is valid.
     *
     * @return true if the file path is invalid.
     */
    final boolean isInvalid() {
        if (status == null) {
```

```
        status = (this.path.indexOf('\u0000') < 0) ? PathStatus.CHECKED
                                                   : PathStatus.INVALID;
    }
    return status == PathStatus.INVALID;
}

/**
 * The length of this abstract pathname's prefix, or zero if it has no
 * prefix.
 */
private final transient int prefixLength;

/**
 * Returns the length of this abstract pathname's prefix.
 * For use by FileSystem classes.
 */
int getPrefixLength() {
    return prefixLength;
}

/**
 * The system-dependent default name-separator character.  This field is
 * initialized to contain the first character of the value of the system
 * property <code>file.separator</code>.  On UNIX systems the value of this
 * field is <code>'/'</code>; on Microsoft Windows systems it is <code>'\\'</code>.
 *
 * @see     java.lang.System#getProperty(java.lang.String)
 */
public static final char separatorChar = fs.getSeparator();

/**
 * The system-dependent default name-separator character, represented as a
 * string for convenience.  This string contains a single character, namely
 * <code>{@link #separatorChar}</code>.
 */
public static final String separator = "" + separatorChar;

/**
 * The system-dependent path-separator character.  This field is
 * initialized to contain the first character of the value of the system
 * property <code>path.separator</code>.  This character is used to
 * separate filenames in a sequence of files given as a <em>path list</em>.
 * On UNIX systems, this character is <code>':'</code>; on Microsoft Windows systems it
 * is <code>';'</code>.
 *
 * @see     java.lang.System#getProperty(java.lang.String)
 */
public static final char pathSeparatorChar = fs.getPathSeparator();

/**
 * The system-dependent path-separator character, represented as a string
 * for convenience.  This string contains a single character, namely
 * <code>{@link #pathSeparatorChar}</code>.
 */
public static final String pathSeparator = "" + pathSeparatorChar;


/* -- Constructors -- */

/**
 * Internal constructor for already-normalized pathname strings.
 */
private File(String pathname, int prefixLength) {
```

```
    this.path = pathname;
    this.prefixLength = prefixLength;
}

/**
 * Internal constructor for already-normalized pathname strings.
 * The parameter order is used to disambiguate this method from the
 * public(File, String) constructor.
 */
private File(String child, File parent) {
    assert parent.path != null;
    assert (!parent.path.equals(""));
    this.path = fs.resolve(parent.path, child);
    this.prefixLength = parent.prefixLength;
}

/**
 * Creates a new <code>File</code> instance by converting the given
 * pathname string into an abstract pathname.  If the given string is
 * the empty string, then the result is the empty abstract pathname.
 *
 * @param   pathname  A pathname string
 * @throws  NullPointerException
 *          If the <code>pathname</code> argument is <code>null</code>
 */
public File(String pathname) {
    if (pathname == null) {
        throw new NullPointerException();
    }
    this.path = fs.normalize(pathname);
    this.prefixLength = fs.prefixLength(this.path);
}

/* Note: The two-argument File constructors do not interpret an empty
   parent abstract pathname as the current user directory.  An empty parent
   instead causes the child to be resolved against the system-dependent
   directory defined by the FileSystem.getDefaultParent method.  On Unix
   this default is "/", while on Microsoft Windows it is "\\".  This is required for
   compatibility with the original behavior of this class. */

/**
 * Creates a new <code>File</code> instance from a parent pathname string
 * and a child pathname string.
 *
 * <p> If <code>parent</code> is <code>null</code> then the new
 * <code>File</code> instance is created as if by invoking the
 * single-argument <code>File</code> constructor on the given
 * <code>child</code> pathname string.
 *
 * <p> Otherwise the <code>parent</code> pathname string is taken to denote
 * a directory, and the <code>child</code> pathname string is taken to
 * denote either a directory or a file.  If the <code>child</code> pathname
 * string is absolute then it is converted into a relative pathname in a
 * system-dependent way.  If <code>parent</code> is the empty string then
 * the new <code>File</code> instance is created by converting
 * <code>child</code> into an abstract pathname and resolving the result
 * against a system-dependent default directory.  Otherwise each pathname
 * string is converted into an abstract pathname and the child abstract
 * pathname is resolved against the parent.
 *
 * @param   parent  The parent pathname string
 * @param   child   The child pathname string
 * @throws  NullPointerException
```

```
     *          If <code>child</code> is <code>null</code>
     */
    public File(String parent, String child) {
        if (child == null) {
            throw new NullPointerException();
        }
        if (parent != null) {
            if (parent.equals("")) {
                this.path = fs.resolve(fs.getDefaultParent(),
                                       fs.normalize(child));
            } else {
                this.path = fs.resolve(fs.normalize(parent),
                                       fs.normalize(child));
            }
        } else {
            this.path = fs.normalize(child);
        }
        this.prefixLength = fs.prefixLength(this.path);
    }

    /**
     * Creates a new <code>File</code> instance from a parent abstract
     * pathname and a child pathname string.
     *
     * <p> If <code>parent</code> is <code>null</code> then the new
     * <code>File</code> instance is created as if by invoking the
     * single-argument <code>File</code> constructor on the given
     * <code>child</code> pathname string.
     *
     * <p> Otherwise the <code>parent</code> abstract pathname is taken to
     * denote a directory, and the <code>child</code> pathname string is taken
     * to denote either a directory or a file.  If the <code>child</code>
     * pathname string is absolute then it is converted into a relative
     * pathname in a system-dependent way.  If <code>parent</code> is the empty
     * abstract pathname then the new <code>File</code> instance is created by
     * converting <code>child</code> into an abstract pathname and resolving
     * the result against a system-dependent default directory.  Otherwise each
     * pathname string is converted into an abstract pathname and the child
     * abstract pathname is resolved against the parent.
     *
     * @param   parent  The parent abstract pathname
     * @param   child   The child pathname string
     * @throws  NullPointerException
     *          If <code>child</code> is <code>null</code>
     */
    public File(File parent, String child) {
        if (child == null) {
            throw new NullPointerException();
        }
        if (parent != null) {
            if (parent.path.equals("")) {
                this.path = fs.resolve(fs.getDefaultParent(),
                                       fs.normalize(child));
            } else {
                this.path = fs.resolve(parent.path,
                                       fs.normalize(child));
            }
        } else {
            this.path = fs.normalize(child);
        }
        this.prefixLength = fs.prefixLength(this.path);
    }
```

```
    /**
     * Creates a new <tt>File</tt> instance by converting the given
     * <tt>file:</tt> URI into an abstract pathname.
     *
     * <p> The exact form of a <tt>file:</tt> URI is system-dependent, hence
     * the transformation performed by this constructor is also
     * system-dependent.
     *
     * <p> For a given abstract pathname <i>f</i> it is guaranteed that
     *
     * <blockquote><tt>
     * new File(</tt><i> f</i><tt>.{@link #toURI() toURI}()).equals(</tt><i> f</i><tt>.{@link
#getAbsoluteFile() getAbsoluteFile}())
     * </tt></blockquote>
     *
     * so long as the original abstract pathname, the URI, and the new abstract
     * pathname are all created in (possibly different invocations of) the same
     * Java virtual machine.  This relationship typically does not hold,
     * however, when a <tt>file:</tt> URI that is created in a virtual machine
     * on one operating system is converted into an abstract pathname in a
     * virtual machine on a different operating system.
     *
     * @param  uri
     *         An absolute, hierarchical URI with a scheme equal to
     *         <tt>"file"</tt>, a non-empty path component, and undefined
     *         authority, query, and fragment components
     *
     * @throws  NullPointerException
     *          If <tt>uri</tt> is <tt>null</tt>
     *
     * @throws  IllegalArgumentException
     *          If the preconditions on the parameter do not hold
     *
     * @see #toURI()
     * @see java.net.URI
     * @since 1.4
     */
    public File(URI uri) {

        // Check our many preconditions
        if (!uri.isAbsolute())
            throw new IllegalArgumentException("URI is not absolute");
        if (uri.isOpaque())
            throw new IllegalArgumentException("URI is not hierarchical");
        String scheme = uri.getScheme();
        if ((scheme == null) || !scheme.equalsIgnoreCase("file"))
            throw new IllegalArgumentException("URI scheme is not \"file\"");
        if (uri.getAuthority() != null)
            throw new IllegalArgumentException("URI has an authority component");
        if (uri.getFragment() != null)
            throw new IllegalArgumentException("URI has a fragment component");
        if (uri.getQuery() != null)
            throw new IllegalArgumentException("URI has a query component");
        String p = uri.getPath();
        if (p.equals(""))
            throw new IllegalArgumentException("URI path component is empty");

        // Okay, now initialize
        p = fs.fromURIPath(p);
        if (File.separatorChar != '/')
            p = p.replace('/', File.separatorChar);
        this.path = fs.normalize(p);
        this.prefixLength = fs.prefixLength(this.path);
```

```
}


/* -- Path-component accessors -- */

/**
 * Returns the name of the file or directory denoted by this abstract
 * pathname.  This is just the last name in the pathname's name
 * sequence.  If the pathname's name sequence is empty, then the empty
 * string is returned.
 *
 * @return  The name of the file or directory denoted by this abstract
 *          pathname, or the empty string if this pathname's name sequence
 *          is empty
 */
public String getName() {
    int index = path.lastIndexOf(separatorChar);
    if (index < prefixLength) return path.substring(prefixLength);
    return path.substring(index + 1);
}

/**
 * Returns the pathname string of this abstract pathname's parent, or
 * <code>null</code> if this pathname does not name a parent directory.
 *
 * <p> The <em>parent</em> of an abstract pathname consists of the
 * pathname's prefix, if any, and each name in the pathname's name
 * sequence except for the last.  If the name sequence is empty then
 * the pathname does not name a parent directory.
 *
 * @return  The pathname string of the parent directory named by this
 *          abstract pathname, or <code>null</code> if this pathname
 *          does not name a parent
 */
public String getParent() {
    int index = path.lastIndexOf(separatorChar);
    if (index < prefixLength) {
        if ((prefixLength > 0) && (path.length() > prefixLength))
            return path.substring(0, prefixLength);
        return null;
    }
    return path.substring(0, index);
}

/**
 * Returns the abstract pathname of this abstract pathname's parent,
 * or <code>null</code> if this pathname does not name a parent
 * directory.
 *
 * <p> The <em>parent</em> of an abstract pathname consists of the
 * pathname's prefix, if any, and each name in the pathname's name
 * sequence except for the last.  If the name sequence is empty then
 * the pathname does not name a parent directory.
 *
 * @return  The abstract pathname of the parent directory named by this
 *          abstract pathname, or <code>null</code> if this pathname
 *          does not name a parent
 *
 * @since 1.2
 */
public File getParentFile() {
    String p = this.getParent();
    if (p == null) return null;
```

```
    return new File(p, this.prefixLength);
}

/**
 * Converts this abstract pathname into a pathname string.  The resulting
 * string uses the {@link #separator default name-separator character} to
 * separate the names in the name sequence.
 *
 * @return  The string form of this abstract pathname
 */
public String getPath() {
    return path;
}


/* -- Path operations -- */

/**
 * Tests whether this abstract pathname is absolute.  The definition of
 * absolute pathname is system dependent.  On UNIX systems, a pathname is
 * absolute if its prefix is <code>"/"</code>.  On Microsoft Windows systems, a
 * pathname is absolute if its prefix is a drive specifier followed by
 * <code>"\\"</code>, or if its prefix is <code>"\\\\"</code>.
 *
 * @return  <code>true</code> if this abstract pathname is absolute,
 *          <code>false</code> otherwise
 */
public boolean isAbsolute() {
    return fs.isAbsolute(this);
}

/**
 * Returns the absolute pathname string of this abstract pathname.
 *
 * <p> If this abstract pathname is already absolute, then the pathname
 * string is simply returned as if by the <code>{@link #getPath}</code>
 * method.  If this abstract pathname is the empty abstract pathname then
 * the pathname string of the current user directory, which is named by the
 * system property <code>user.dir</code>, is returned.  Otherwise this
 * pathname is resolved in a system-dependent way.  On UNIX systems, a
 * relative pathname is made absolute by resolving it against the current
 * user directory.  On Microsoft Windows systems, a relative pathname is made absolute
 * by resolving it against the current directory of the drive named by the
 * pathname, if any; if not, it is resolved against the current user
 * directory.
 *
 * @return  The absolute pathname string denoting the same file or
 *          directory as this abstract pathname
 *
 * @throws  SecurityException
 *          If a required system property value cannot be accessed.
 *
 * @see     java.io.File#isAbsolute()
 */
public String getAbsolutePath() {
    return fs.resolve(this);
}

/**
 * Returns the absolute form of this abstract pathname.  Equivalent to
 * <code>new File(this.{@link #getAbsolutePath})</code>.
 *
 * @return  The absolute abstract pathname denoting the same file or
```

```
     *          directory as this abstract pathname
     *
     * @throws  SecurityException
     *          If a required system property value cannot be accessed.
     *
     * @since 1.2
     */
    public File getAbsoluteFile() {
        String absPath = getAbsolutePath();
        return new File(absPath, fs.prefixLength(absPath));
    }

    /**
     * Returns the canonical pathname string of this abstract pathname.
     *
     * <p> A canonical pathname is both absolute and unique.  The precise
     * definition of canonical form is system-dependent.  This method first
     * converts this pathname to absolute form if necessary, as if by invoking the
     * {@link #getAbsolutePath} method, and then maps it to its unique form in a
     * system-dependent way.  This typically involves removing redundant names
     * such as <tt>"."</tt> and <tt>".."</tt> from the pathname, resolving
     * symbolic links (on UNIX platforms), and converting drive letters to a
     * standard case (on Microsoft Windows platforms).
     *
     * <p> Every pathname that denotes an existing file or directory has a
     * unique canonical form.  Every pathname that denotes a nonexistent file
     * or directory also has a unique canonical form.  The canonical form of
     * the pathname of a nonexistent file or directory may be different from
     * the canonical form of the same pathname after the file or directory is
     * created.  Similarly, the canonical form of the pathname of an existing
     * file or directory may be different from the canonical form of the same
     * pathname after the file or directory is deleted.
     *
     * @return  The canonical pathname string denoting the same file or
     *          directory as this abstract pathname
     *
     * @throws  IOException
     *          If an I/O error occurs, which is possible because the
     *          construction of the canonical pathname may require
     *          filesystem queries
     *
     * @throws  SecurityException
     *          If a required system property value cannot be accessed, or
     *          if a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkRead}</code> method denies
     *          read access to the file
     *
     * @since   JDK1.1
     * @see     Path#toRealPath
     */
    public String getCanonicalPath() throws IOException {
        if (isInvalid()) {
            throw new IOException("Invalid file path");
        }
        return fs.canonicalize(fs.resolve(this));
    }

    /**
     * Returns the canonical form of this abstract pathname.  Equivalent to
     * <code>new File(this.{@link #getCanonicalPath})</code>.
     *
     * @return  The canonical pathname string denoting the same file or
     *          directory as this abstract pathname
```

```
     *
     * @throws  IOException
     *          If an I/O error occurs, which is possible because the
     *          construction of the canonical pathname may require
     *          filesystem queries
     *
     * @throws  SecurityException
     *          If a required system property value cannot be accessed, or
     *          if a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkRead}</code> method denies
     *          read access to the file
     *
     * @since 1.2
     * @see     Path#toRealPath
     */
    public File getCanonicalFile() throws IOException {
        String canonPath = getCanonicalPath();
        return new File(canonPath, fs.prefixLength(canonPath));
    }

    private static String slashify(String path, boolean isDirectory) {
        String p = path;
        if (File.separatorChar != '/')
            p = p.replace(File.separatorChar, '/');
        if (!p.startsWith("/"))
            p = "/" + p;
        if (!p.endsWith("/") && isDirectory)
            p = p + "/";
        return p;
    }

    /**
     * Converts this abstract pathname into a <code>file:</code> URL.  The
     * exact form of the URL is system-dependent.  If it can be determined that
     * the file denoted by this abstract pathname is a directory, then the
     * resulting URL will end with a slash.
     *
     * @return  A URL object representing the equivalent file URL
     *
     * @throws  MalformedURLException
     *          If the path cannot be parsed as a URL
     *
     * @see     #toURI()
     * @see     java.net.URI
     * @see     java.net.URI#toURL()
     * @see     java.net.URL
     * @since   1.2
     *
     * @deprecated This method does not automatically escape characters that
     * are illegal in URLs.  It is recommended that new code convert an
     * abstract pathname into a URL by first converting it into a URI, via the
     * {@link #toURI() toURI} method, and then converting the URI into a URL
     * via the {@link java.net.URI#toURL() URI.toURL} method.
     */
    @Deprecated
    public URL toURL() throws MalformedURLException {
        if (isInvalid()) {
            throw new MalformedURLException("Invalid file path");
        }
        return new URL("file", "", slashify(getAbsolutePath(), isDirectory()));
    }

    /**
```

```
     * Constructs a <tt>file:</tt> URI that represents this abstract pathname.
     *
     * <p> The exact form of the URI is system-dependent.  If it can be
     * determined that the file denoted by this abstract pathname is a
     * directory, then the resulting URI will end with a slash.
     *
     * <p> For a given abstract pathname <i>f</i>, it is guaranteed that
     *
     * <blockquote><tt>
     * new {@link #File(java.net.URI) File}(</tt><i> f</i><tt>.toURI()).equals(</tt><i> f</i><tt>.{@link
#getAbsoluteFile() getAbsoluteFile}())
     * </tt></blockquote>
     *
     * so long as the original abstract pathname, the URI, and the new abstract
     * pathname are all created in (possibly different invocations of) the same
     * Java virtual machine.  Due to the system-dependent nature of abstract
     * pathnames, however, this relationship typically does not hold when a
     * <tt>file:</tt> URI that is created in a virtual machine on one operating
     * system is converted into an abstract pathname in a virtual machine on a
     * different operating system.
     *
     * <p> Note that when this abstract pathname represents a UNC pathname then
     * all components of the UNC (including the server name component) are encoded
     * in the {@code URI} path. The authority component is undefined, meaning
     * that it is represented as {@code null}. The {@link Path} class defines the
     * {@link Path#toUri toUri} method to encode the server name in the authority
     * component of the resulting {@code URI}. The {@link #toPath toPath} method
     * may be used to obtain a {@code Path} representing this abstract pathname.
     *
     * @return  An absolute, hierarchical URI with a scheme equal to
     *          <tt>"file"</tt>, a path representing this abstract pathname,
     *          and undefined authority, query, and fragment components
     * @throws SecurityException If a required system property value cannot
     * be accessed.
     *
     * @see #File(java.net.URI)
     * @see java.net.URI
     * @see java.net.URI#toURL()
     * @since 1.4
     */
    public URI toURI() {
        try {
            File f = getAbsoluteFile();
            String sp = slashify(f.getPath(), f.isDirectory());
            if (sp.startsWith("//"))
                sp = "//" + sp;
            return new URI("file", null, sp, null);
        } catch (URISyntaxException x) {
            throw new Error(x);         // Can't happen
        }
    }


    /* -- Attribute accessors -- */

    /**
     * Tests whether the application can read the file denoted by this
     * abstract pathname. On some platforms it may be possible to start the
     * Java virtual machine with special privileges that allow it to read
     * files that are marked as unreadable. Consequently this method may return
     * {@code true} even though the file does not have read permissions.
     *
     * @return  <code>true</code> if and only if the file specified by this
```

```
     *          abstract pathname exists <em>and</em> can be read by the
     *          application; <code>false</code> otherwise
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkRead(java.lang.String)}</code>
     *          method denies read access to the file
     */
    public boolean canRead() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkRead(path);
        }
        if (isInvalid()) {
            return false;
        }
        return fs.checkAccess(this, FileSystem.ACCESS_READ);
    }

    /**
     * Tests whether the application can modify the file denoted by this
     * abstract pathname. On some platforms it may be possible to start the
     * Java virtual machine with special privileges that allow it to modify
     * files that are marked read-only. Consequently this method may return
     * {@code true} even though the file is marked read-only.
     *
     * @return  <code>true</code> if and only if the file system actually
     *          contains a file denoted by this abstract pathname <em>and</em>
     *          the application is allowed to write to the file;
     *          <code>false</code> otherwise.
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method denies write access to the file
     */
    public boolean canWrite() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(path);
        }
        if (isInvalid()) {
            return false;
        }
        return fs.checkAccess(this, FileSystem.ACCESS_WRITE);
    }

    /**
     * Tests whether the file or directory denoted by this abstract pathname
     * exists.
     *
     * @return  <code>true</code> if and only if the file or directory denoted
     *          by this abstract pathname exists; <code>false</code> otherwise
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkRead(java.lang.String)}</code>
     *          method denies read access to the file or directory
     */
    public boolean exists() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkRead(path);
```

```
    }
    if (isInvalid()) {
        return false;
    }
    return ((fs.getBooleanAttributes(this) & FileSystem.BA_EXISTS) != 0);
}

/**
 * Tests whether the file denoted by this abstract pathname is a
 * directory.
 *
 * <p> Where it is required to distinguish an I/O exception from the case
 * that the file is not a directory, or where several attributes of the
 * same file are required at the same time, then the {@link
 * java.nio.file.Files#readAttributes(Path,Class,LinkOption[])
 * Files.readAttributes} method may be used.
 *
 * @return <code>true</code> if and only if the file denoted by this
 *          abstract pathname exists <em>and</em> is a directory;
 *          <code>false</code> otherwise
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkRead(java.lang.String)}</code>
 *          method denies read access to the file
 */
public boolean isDirectory() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkRead(path);
    }
    if (isInvalid()) {
        return false;
    }
    return ((fs.getBooleanAttributes(this) & FileSystem.BA_DIRECTORY)
            != 0);
}

/**
 * Tests whether the file denoted by this abstract pathname is a normal
 * file.  A file is <em>normal</em> if it is not a directory and, in
 * addition, satisfies other system-dependent criteria.  Any non-directory
 * file created by a Java application is guaranteed to be a normal file.
 *
 * <p> Where it is required to distinguish an I/O exception from the case
 * that the file is not a normal file, or where several attributes of the
 * same file are required at the same time, then the {@link
 * java.nio.file.Files#readAttributes(Path,Class,LinkOption[])
 * Files.readAttributes} method may be used.
 *
 * @return  <code>true</code> if and only if the file denoted by this
 *          abstract pathname exists <em>and</em> is a normal file;
 *          <code>false</code> otherwise
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkRead(java.lang.String)}</code>
 *          method denies read access to the file
 */
public boolean isFile() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkRead(path);
```

```
    }
    if (isInvalid()) {
        return false;
    }
    return ((fs.getBooleanAttributes(this) & FileSystem.BA_REGULAR) != 0);
}

/**
 * Tests whether the file named by this abstract pathname is a hidden
 * file.  The exact definition of <em>hidden</em> is system-dependent.  On
 * UNIX systems, a file is considered to be hidden if its name begins with
 * a period character (<code>'.'</code>).  On Microsoft Windows systems, a file is
 * considered to be hidden if it has been marked as such in the filesystem.
 *
 * @return  <code>true</code> if and only if the file denoted by this
 *          abstract pathname is hidden according to the conventions of the
 *          underlying platform
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkRead(java.lang.String)}</code>
 *          method denies read access to the file
 *
 * @since 1.2
 */
public boolean isHidden() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkRead(path);
    }
    if (isInvalid()) {
        return false;
    }
    return ((fs.getBooleanAttributes(this) & FileSystem.BA_HIDDEN) != 0);
}

/**
 * Returns the time that the file denoted by this abstract pathname was
 * last modified.
 *
 * <p> Where it is required to distinguish an I/O exception from the case
 * where {@code 0L} is returned, or where several attributes of the
 * same file are required at the same time, or where the time of last
 * access or the creation time are required, then the {@link
 * java.nio.file.Files#readAttributes(Path,Class,LinkOption[])
 * Files.readAttributes} method may be used.
 *
 * @return  A <code>long</code> value representing the time the file was
 *          last modified, measured in milliseconds since the epoch
 *          (00:00:00 GMT, January 1, 1970), or <code>0L</code> if the
 *          file does not exist or if an I/O error occurs
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkRead(java.lang.String)}</code>
 *          method denies read access to the file
 */
public long lastModified() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkRead(path);
    }
    if (isInvalid()) {
```

```
        return 0L;
    }
    return fs.getLastModifiedTime(this);
}

/**
 * Returns the length of the file denoted by this abstract pathname.
 * The return value is unspecified if this pathname denotes a directory.
 *
 * <p> Where it is required to distinguish an I/O exception from the case
 * that {@code 0L} is returned, or where several attributes of the same file
 * are required at the same time, then the {@link
 * java.nio.file.Files#readAttributes(Path,Class,LinkOption[])
 * Files.readAttributes} method may be used.
 *
 * @return  The length, in bytes, of the file denoted by this abstract
 *          pathname, or <code>0L</code> if the file does not exist.  Some
 *          operating systems may return <code>0L</code> for pathnames
 *          denoting system-dependent entities such as devices or pipes.
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkRead(java.lang.String)}</code>
 *          method denies read access to the file
 */
public long length() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkRead(path);
    }
    if (isInvalid()) {
        return 0L;
    }
    return fs.getLength(this);
}


/* -- File operations -- */

/**
 * Atomically creates a new, empty file named by this abstract pathname if
 * and only if a file with this name does not yet exist.  The check for the
 * existence of the file and the creation of the file if it does not exist
 * are a single operation that is atomic with respect to all other
 * filesystem activities that might affect the file.
 * <P>
 * Note: this method should <i>not</i> be used for file-locking, as
 * the resulting protocol cannot be made to work reliably. The
 * {@link java.nio.channels.FileLock FileLock}
 * facility should be used instead.
 *
 * @return  <code>true</code> if the named file does not exist and was
 *          successfully created; <code>false</code> if the named file
 *          already exists
 *
 * @throws  IOException
 *          If an I/O error occurred
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
 *          method denies write access to the file
 *
```

```
 * @since 1.2
 */
public boolean createNewFile() throws IOException {
    SecurityManager security = System.getSecurityManager();
    if (security != null) security.checkWrite(path);
    if (isInvalid()) {
        throw new IOException("Invalid file path");
    }
    return fs.createFileExclusively(path);
}

/**
 * Deletes the file or directory denoted by this abstract pathname.  If
 * this pathname denotes a directory, then the directory must be empty in
 * order to be deleted.
 *
 * <p> Note that the {@link java.nio.file.Files} class defines the {@link
 * java.nio.file.Files#delete(Path) delete} method to throw an {@link IOException}
 * when a file cannot be deleted. This is useful for error reporting and to
 * diagnose why a file cannot be deleted.
 *
 * @return  <code>true</code> if and only if the file or directory is
 *          successfully deleted; <code>false</code> otherwise
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkDelete}</code> method denies
 *          delete access to the file
 */
public boolean delete() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkDelete(path);
    }
    if (isInvalid()) {
        return false;
    }
    return fs.delete(this);
}

/**
 * Requests that the file or directory denoted by this abstract
 * pathname be deleted when the virtual machine terminates.
 * Files (or directories) are deleted in the reverse order that
 * they are registered. Invoking this method to delete a file or
 * directory that is already registered for deletion has no effect.
 * Deletion will be attempted only for normal termination of the
 * virtual machine, as defined by the Java Language Specification.
 *
 * <p> Once deletion has been requested, it is not possible to cancel the
 * request.  This method should therefore be used with care.
 *
 * <P>
 * Note: this method should <i>not</i> be used for file-locking, as
 * the resulting protocol cannot be made to work reliably. The
 * {@link java.nio.channels.FileLock FileLock}
 * facility should be used instead.
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkDelete}</code> method denies
 *          delete access to the file
 *
```

```
     * @see #delete
     *
     * @since 1.2
     */
    public void deleteOnExit() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkDelete(path);
        }
        if (isInvalid()) {
            return;
        }
        DeleteOnExitHook.add(path);
    }

    /**
     * Returns an array of strings naming the files and directories in the
     * directory denoted by this abstract pathname.
     *
     * <p> If this abstract pathname does not denote a directory, then this
     * method returns {@code null}.  Otherwise an array of strings is
     * returned, one for each file or directory in the directory.  Names
     * denoting the directory itself and the directory's parent directory are
     * not included in the result.  Each string is a file name rather than a
     * complete path.
     *
     * <p> There is no guarantee that the name strings in the resulting array
     * will appear in any specific order; they are not, in particular,
     * guaranteed to appear in alphabetical order.
     *
     * <p> Note that the {@link java.nio.file.Files} class defines the {@link
     * java.nio.file.Files#newDirectoryStream(Path) newDirectoryStream} method to
     * open a directory and iterate over the names of the files in the directory.
     * This may use less resources when working with very large directories, and
     * may be more responsive when working with remote directories.
     *
     * @return  An array of strings naming the files and directories in the
     *          directory denoted by this abstract pathname.  The array will be
     *          empty if the directory is empty.  Returns {@code null} if
     *          this abstract pathname does not denote a directory, or if an
     *          I/O error occurs.
     *
     * @throws  SecurityException
     *          If a security manager exists and its {@link
     *          SecurityManager#checkRead(String)} method denies read access to
     *          the directory
     */
    public String[] list() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkRead(path);
        }
        if (isInvalid()) {
            return null;
        }
        return fs.list(this);
    }

    /**
     * Returns an array of strings naming the files and directories in the
     * directory denoted by this abstract pathname that satisfy the specified
     * filter.  The behavior of this method is the same as that of the
     * {@link #list()} method, except that the strings in the returned array
```

```
     * must satisfy the filter.  If the given {@code filter} is {@code null}
     * then all names are accepted.  Otherwise, a name satisfies the filter if
     * and only if the value {@code true} results when the {@link
     * FilenameFilter#accept FilenameFilter.accept(File, String)} method
     * of the filter is invoked on this abstract pathname and the name of a
     * file or directory in the directory that it denotes.
     *
     * @param  filter
     *         A filename filter
     *
     * @return  An array of strings naming the files and directories in the
     *          directory denoted by this abstract pathname that were accepted
     *          by the given {@code filter}.  The array will be empty if the
     *          directory is empty or if no names were accepted by the filter.
     *          Returns {@code null} if this abstract pathname does not denote
     *          a directory, or if an I/O error occurs.
     *
     * @throws  SecurityException
     *          If a security manager exists and its {@link
     *          SecurityManager#checkRead(String)} method denies read access to
     *          the directory
     *
     * @see java.nio.file.Files#newDirectoryStream(Path,String)
     */
    public String[] list(FilenameFilter filter) {
        String names[] = list();
        if ((names == null) || (filter == null)) {
            return names;
        }
        List<String> v = new ArrayList<>();
        for (int i = 0 ; i < names.length ; i++) {
            if (filter.accept(this, names[i])) {
                v.add(names[i]);
            }
        }
        return v.toArray(new String[v.size()]);
    }

    /**
     * Returns an array of abstract pathnames denoting the files in the
     * directory denoted by this abstract pathname.
     *
     * <p> If this abstract pathname does not denote a directory, then this
     * method returns {@code null}.  Otherwise an array of {@code File} objects
     * is returned, one for each file or directory in the directory.  Pathnames
     * denoting the directory itself and the directory's parent directory are
     * not included in the result.  Each resulting abstract pathname is
     * constructed from this abstract pathname using the {@link #File(File,
     * String) File(File, String)} constructor.  Therefore if this
     * pathname is absolute then each resulting pathname is absolute; if this
     * pathname is relative then each resulting pathname will be relative to
     * the same directory.
     *
     * <p> There is no guarantee that the name strings in the resulting array
     * will appear in any specific order; they are not, in particular,
     * guaranteed to appear in alphabetical order.
     *
     * <p> Note that the {@link java.nio.file.Files} class defines the {@link
     * java.nio.file.Files#newDirectoryStream(Path) newDirectoryStream} method
     * to open a directory and iterate over the names of the files in the
     * directory. This may use less resources when working with very large
     * directories.
     *
```

```
     * @return  An array of abstract pathnames denoting the files and
     *          directories in the directory denoted by this abstract pathname.
     *          The array will be empty if the directory is empty.  Returns
     *          {@code null} if this abstract pathname does not denote a
     *          directory, or if an I/O error occurs.
     *
     * @throws  SecurityException
     *          If a security manager exists and its {@link
     *          SecurityManager#checkRead(String)} method denies read access to
     *          the directory
     *
     * @since  1.2
     */
    public File[] listFiles() {
        String[] ss = list();
        if (ss == null) return null;
        int n = ss.length;
        File[] fs = new File[n];
        for (int i = 0; i < n; i++) {
            fs[i] = new File(ss[i], this);
        }
        return fs;
    }

    /**
     * Returns an array of abstract pathnames denoting the files and
     * directories in the directory denoted by this abstract pathname that
     * satisfy the specified filter.  The behavior of this method is the same
     * as that of the {@link #listFiles()} method, except that the pathnames in
     * the returned array must satisfy the filter.  If the given {@code filter}
     * is {@code null} then all pathnames are accepted.  Otherwise, a pathname
     * satisfies the filter if and only if the value {@code true} results when
     * the {@link FilenameFilter#accept
     * FilenameFilter.accept(File, String)} method of the filter is
     * invoked on this abstract pathname and the name of a file or directory in
     * the directory that it denotes.
     *
     * @param  filter
     *         A filename filter
     *
     * @return  An array of abstract pathnames denoting the files and
     *          directories in the directory denoted by this abstract pathname.
     *          The array will be empty if the directory is empty.  Returns
     *          {@code null} if this abstract pathname does not denote a
     *          directory, or if an I/O error occurs.
     *
     * @throws  SecurityException
     *          If a security manager exists and its {@link
     *          SecurityManager#checkRead(String)} method denies read access to
     *          the directory
     *
     * @since  1.2
     * @see java.nio.file.Files#newDirectoryStream(Path,String)
     */
    public File[] listFiles(FilenameFilter filter) {
        String ss[] = list();
        if (ss == null) return null;
        ArrayList<File> files = new ArrayList<>();
        for (String s : ss)
            if ((filter == null) || filter.accept(this, s))
                files.add(new File(s, this));
        return files.toArray(new File[files.size()]);
    }
```

```
/**
 * Returns an array of abstract pathnames denoting the files and
 * directories in the directory denoted by this abstract pathname that
 * satisfy the specified filter.  The behavior of this method is the same
 * as that of the {@link #listFiles()} method, except that the pathnames in
 * the returned array must satisfy the filter.  If the given {@code filter}
 * is {@code null} then all pathnames are accepted.  Otherwise, a pathname
 * satisfies the filter if and only if the value {@code true} results when
 * the {@link FileFilter#accept FileFilter.accept(File)} method of the
 * filter is invoked on the pathname.
 *
 * @param  filter
 *         A file filter
 *
 * @return  An array of abstract pathnames denoting the files and
 *          directories in the directory denoted by this abstract pathname.
 *          The array will be empty if the directory is empty.  Returns
 *          {@code null} if this abstract pathname does not denote a
 *          directory, or if an I/O error occurs.
 *
 * @throws  SecurityException
 *          If a security manager exists and its {@link
 *          SecurityManager#checkRead(String)} method denies read access to
 *          the directory
 *
 * @since  1.2
 * @see java.nio.file.Files#newDirectoryStream(Path,java.nio.file.DirectoryStream.Filter)
 */
public File[] listFiles(FileFilter filter) {
    String ss[] = list();
    if (ss == null) return null;
    ArrayList<File> files = new ArrayList<>();
    for (String s : ss) {
        File f = new File(s, this);
        if ((filter == null) || filter.accept(f))
            files.add(f);
    }
    return files.toArray(new File[files.size()]);
}

/**
 * Creates the directory named by this abstract pathname.
 *
 * @return  <code>true</code> if and only if the directory was
 *          created; <code>false</code> otherwise
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
 *          method does not permit the named directory to be created
 */
public boolean mkdir() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkWrite(path);
    }
    if (isInvalid()) {
        return false;
    }
    return fs.createDirectory(this);
}
```

```
/**
 * Creates the directory named by this abstract pathname, including any
 * necessary but nonexistent parent directories.  Note that if this
 * operation fails it may have succeeded in creating some of the necessary
 * parent directories.
 *
 * @return  <code>true</code> if and only if the directory was created,
 *          along with all necessary parent directories; <code>false</code>
 *          otherwise
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkRead(java.lang.String)}</code>
 *          method does not permit verification of the existence of the
 *          named directory and all necessary parent directories; or if
 *          the <code>{@link
 *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
 *          method does not permit the named directory and all necessary
 *          parent directories to be created
 */
public boolean mkdirs() {
    if (exists()) {
        return false;
    }
    if (mkdir()) {
        return true;
    }
    File canonFile = null;
    try {
        canonFile = getCanonicalFile();
    } catch (IOException e) {
        return false;
    }

    File parent = canonFile.getParentFile();
    return (parent != null && (parent.mkdirs() || parent.exists()) &&
            canonFile.mkdir());
}

/**
 * Renames the file denoted by this abstract pathname.
 *
 * <p> Many aspects of the behavior of this method are inherently
 * platform-dependent: The rename operation might not be able to move a
 * file from one filesystem to another, it might not be atomic, and it
 * might not succeed if a file with the destination abstract pathname
 * already exists.  The return value should always be checked to make sure
 * that the rename operation was successful.
 *
 * <p> Note that the {@link java.nio.file.Files} class defines the {@link
 * java.nio.file.Files#move move} method to move or rename a file in a
 * platform independent manner.
 *
 * @param  dest  The new abstract pathname for the named file
 *
 * @return  <code>true</code> if and only if the renaming succeeded;
 *          <code>false</code> otherwise
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
 *          method denies write access to either the old or new pathnames
 *
```

```
     * @throws  NullPointerException
     *          If parameter <code>dest</code> is <code>null</code>
     */
    public boolean renameTo(File dest) {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(path);
            security.checkWrite(dest.path);
        }
        if (dest == null) {
            throw new NullPointerException();
        }
        if (this.isInvalid() || dest.isInvalid()) {
            return false;
        }
        return fs.rename(this, dest);
    }

    /**
     * Sets the last-modified time of the file or directory named by this
     * abstract pathname.
     *
     * <p> All platforms support file-modification times to the nearest second,
     * but some provide more precision.  The argument will be truncated to fit
     * the supported precision.  If the operation succeeds and no intervening
     * operations on the file take place, then the next invocation of the
     * <code>{@link #lastModified}</code> method will return the (possibly
     * truncated) <code>time</code> argument that was passed to this method.
     *
     * @param  time  The new last-modified time, measured in milliseconds since
     *               the epoch (00:00:00 GMT, January 1, 1970)
     *
     * @return <code>true</code> if and only if the operation succeeded;
     *         <code>false</code> otherwise
     *
     * @throws  IllegalArgumentException  If the argument is negative
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method denies write access to the named file
     *
     * @since 1.2
     */
    public boolean setLastModified(long time) {
        if (time < 0) throw new IllegalArgumentException("Negative time");
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(path);
        }
        if (isInvalid()) {
            return false;
        }
        return fs.setLastModifiedTime(this, time);
    }

    /**
     * Marks the file or directory named by this abstract pathname so that
     * only read operations are allowed. After invoking this method the file
     * or directory will not change until it is either deleted or marked
     * to allow write access. On some platforms it may be possible to start the
     * Java virtual machine with special privileges that allow it to modify
     * files that are marked read-only. Whether or not a read-only file or
```

```
     * directory may be deleted depends upon the underlying system.
     *
     * @return <code>true</code> if and only if the operation succeeded;
     *         <code>false</code> otherwise
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method denies write access to the named file
     *
     * @since 1.2
     */
    public boolean setReadOnly() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(path);
        }
        if (isInvalid()) {
            return false;
        }
        return fs.setReadOnly(this);
    }

    /**
     * Sets the owner's or everybody's write permission for this abstract
     * pathname. On some platforms it may be possible to start the Java virtual
     * machine with special privileges that allow it to modify files that
     * disallow write operations.
     *
     * <p> The {@link java.nio.file.Files} class defines methods that operate on
     * file attributes including file permissions. This may be used when finer
     * manipulation of file permissions is required.
     *
     * @param   writable
     *          If <code>true</code>, sets the access permission to allow write
     *          operations; if <code>false</code> to disallow write operations
     *
     * @param   ownerOnly
     *          If <code>true</code>, the write permission applies only to the
     *          owner's write permission; otherwise, it applies to everybody.  If
     *          the underlying file system can not distinguish the owner's write
     *          permission from that of others, then the permission will apply to
     *          everybody, regardless of this value.
     *
     * @return  <code>true</code> if and only if the operation succeeded. The
     *          operation will fail if the user does not have permission to change
     *          the access permissions of this abstract pathname.
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method denies write access to the named file
     *
     * @since 1.6
     */
    public boolean setWritable(boolean writable, boolean ownerOnly) {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(path);
        }
        if (isInvalid()) {
            return false;
        }
```

```
    return fs.setPermission(this, FileSystem.ACCESS_WRITE, writable, ownerOnly);
}

/**
 * A convenience method to set the owner's write permission for this abstract
 * pathname. On some platforms it may be possible to start the Java virtual
 * machine with special privileges that allow it to modify files that
 * disallow write operations.
 *
 * <p> An invocation of this method of the form <tt>file.setWritable(arg)</tt>
 * behaves in exactly the same way as the invocation
 *
 * <pre>
 *     file.setWritable(arg, true) </pre>
 *
 * @param   writable
 *          If <code>true</code>, sets the access permission to allow write
 *          operations; if <code>false</code> to disallow write operations
 *
 * @return  <code>true</code> if and only if the operation succeeded.  The
 *          operation will fail if the user does not have permission to
 *          change the access permissions of this abstract pathname.
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
 *          method denies write access to the file
 *
 * @since 1.6
 */
public boolean setWritable(boolean writable) {
    return setWritable(writable, true);
}

/**
 * Sets the owner's or everybody's read permission for this abstract
 * pathname. On some platforms it may be possible to start the Java virtual
 * machine with special privileges that allow it to read files that are
 * marked as unreadable.
 *
 * <p> The {@link java.nio.file.Files} class defines methods that operate on
 * file attributes including file permissions. This may be used when finer
 * manipulation of file permissions is required.
 *
 * @param   readable
 *          If <code>true</code>, sets the access permission to allow read
 *          operations; if <code>false</code> to disallow read operations
 *
 * @param   ownerOnly
 *          If <code>true</code>, the read permission applies only to the
 *          owner's read permission; otherwise, it applies to everybody.  If
 *          the underlying file system can not distinguish the owner's read
 *          permission from that of others, then the permission will apply to
 *          everybody, regardless of this value.
 *
 * @return  <code>true</code> if and only if the operation succeeded.  The
 *          operation will fail if the user does not have permission to
 *          change the access permissions of this abstract pathname.  If
 *          <code>readable</code> is <code>false</code> and the underlying
 *          file system does not implement a read permission, then the
 *          operation will fail.
 *
 * @throws  SecurityException
```

```
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method denies write access to the file
     *
     * @since 1.6
     */
    public boolean setReadable(boolean readable, boolean ownerOnly) {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(path);
        }
        if (isInvalid()) {
            return false;
        }
        return fs.setPermission(this, FileSystem.ACCESS_READ, readable, ownerOnly);
    }

    /**
     * A convenience method to set the owner's read permission for this abstract
     * pathname. On some platforms it may be possible to start the Java virtual
     * machine with special privileges that allow it to read files that that are
     * marked as unreadable.
     *
     * <p>An invocation of this method of the form <tt>file.setReadable(arg)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     file.setReadable(arg, true) </pre>
     *
     * @param  readable
     *          If <code>true</code>, sets the access permission to allow read
     *          operations; if <code>false</code> to disallow read operations
     *
     * @return  <code>true</code> if and only if the operation succeeded.  The
     *          operation will fail if the user does not have permission to
     *          change the access permissions of this abstract pathname.  If
     *          <code>readable</code> is <code>false</code> and the underlying
     *          file system does not implement a read permission, then the
     *          operation will fail.
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method denies write access to the file
     *
     * @since 1.6
     */
    public boolean setReadable(boolean readable) {
        return setReadable(readable, true);
    }

    /**
     * Sets the owner's or everybody's execute permission for this abstract
     * pathname. On some platforms it may be possible to start the Java virtual
     * machine with special privileges that allow it to execute files that are
     * not marked executable.
     *
     * <p> The {@link java.nio.file.Files} class defines methods that operate on
     * file attributes including file permissions. This may be used when finer
     * manipulation of file permissions is required.
     *
     * @param   executable
     *          If <code>true</code>, sets the access permission to allow execute
```

```
     *          operations; if <code>false</code> to disallow execute operations
     *
     * @param   ownerOnly
     *          If <code>true</code>, the execute permission applies only to the
     *          owner's execute permission; otherwise, it applies to everybody.
     *          If the underlying file system can not distinguish the owner's
     *          execute permission from that of others, then the permission will
     *          apply to everybody, regardless of this value.
     *
     * @return  <code>true</code> if and only if the operation succeeded.  The
     *          operation will fail if the user does not have permission to
     *          change the access permissions of this abstract pathname.  If
     *          <code>executable</code> is <code>false</code> and the underlying
     *          file system does not implement an execute permission, then the
     *          operation will fail.
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method denies write access to the file
     *
     * @since 1.6
     */
    public boolean setExecutable(boolean executable, boolean ownerOnly) {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(path);
        }
        if (isInvalid()) {
            return false;
        }
        return fs.setPermission(this, FileSystem.ACCESS_EXECUTE, executable, ownerOnly);
    }

    /**
     * A convenience method to set the owner's execute permission for this
     * abstract pathname. On some platforms it may be possible to start the Java
     * virtual machine with special privileges that allow it to execute files
     * that are not marked executable.
     *
     * <p>An invocation of this method of the form <tt>file.setExcutable(arg)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     file.setExecutable(arg, true) </pre>
     *
     * @param   executable
     *          If <code>true</code>, sets the access permission to allow execute
     *          operations; if <code>false</code> to disallow execute operations
     *
     * @return   <code>true</code> if and only if the operation succeeded.  The
     *           operation will fail if the user does not have permission to
     *           change the access permissions of this abstract pathname.  If
     *           <code>executable</code> is <code>false</code> and the underlying
     *           file system does not implement an execute permission, then the
     *           operation will fail.
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method denies write access to the file
     *
     * @since 1.6
```

```
 */
public boolean setExecutable(boolean executable) {
    return setExecutable(executable, true);
}

/**
 * Tests whether the application can execute the file denoted by this
 * abstract pathname. On some platforms it may be possible to start the
 * Java virtual machine with special privileges that allow it to execute
 * files that are not marked executable. Consequently this method may return
 * {@code true} even though the file does not have execute permissions.
 *
 * @return  <code>true</code> if and only if the abstract pathname exists
 *          <em>and</em> the application is allowed to execute the file
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkExec(java.lang.String)}</code>
 *          method denies execute access to the file
 *
 * @since 1.6
 */
public boolean canExecute() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkExec(path);
    }
    if (isInvalid()) {
        return false;
    }
    return fs.checkAccess(this, FileSystem.ACCESS_EXECUTE);
}


/* -- Filesystem interface -- */

/**
 * List the available filesystem roots.
 *
 * <p> A particular Java platform may support zero or more
 * hierarchically-organized file systems.  Each file system has a
 * {@code root} directory from which all other files in that file system
 * can be reached.  Windows platforms, for example, have a root directory
 * for each active drive; UNIX platforms have a single root directory,
 * namely {@code "/"}.  The set of available filesystem roots is affected
 * by various system-level operations such as the insertion or ejection of
 * removable media and the disconnecting or unmounting of physical or
 * virtual disk drives.
 *
 * <p> This method returns an array of {@code File} objects that denote the
 * root directories of the available filesystem roots.  It is guaranteed
 * that the canonical pathname of any file physically present on the local
 * machine will begin with one of the roots returned by this method.
 *
 * <p> The canonical pathname of a file that resides on some other machine
 * and is accessed via a remote-filesystem protocol such as SMB or NFS may
 * or may not begin with one of the roots returned by this method.  If the
 * pathname of a remote file is syntactically indistinguishable from the
 * pathname of a local file then it will begin with one of the roots
 * returned by this method.  Thus, for example, {@code File} objects
 * denoting the root directories of the mapped network drives of a Windows
 * platform will be returned by this method, while {@code File} objects
 * containing UNC pathnames will not be returned by this method.
```

```
     *
     * <p> Unlike most methods in this class, this method does not throw
     * security exceptions.  If a security manager exists and its {@link
     * SecurityManager#checkRead(String)} method denies read access to a
     * particular root directory, then that directory will not appear in the
     * result.
     *
     * @return  An array of {@code File} objects denoting the available
     *          filesystem roots, or {@code null} if the set of roots could not
     *          be determined.  The array will be empty if there are no
     *          filesystem roots.
     *
     * @since  1.2
     * @see java.nio.file.FileStore
     */
    public static File[] listRoots() {
        return fs.listRoots();
    }


    /* -- Disk usage -- */

    /**
     * Returns the size of the partition <a href="#partName">named</a> by this
     * abstract pathname.
     *
     * @return  The size, in bytes, of the partition or <tt>0L</tt> if this
     *          abstract pathname does not name a partition
     *
     * @throws  SecurityException
     *          If a security manager has been installed and it denies
     *          {@link RuntimePermission}<tt>("getFileSystemAttributes")</tt>
     *          or its {@link SecurityManager#checkRead(String)} method denies
     *          read access to the file named by this abstract pathname
     *
     * @since  1.6
     */
    public long getTotalSpace() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(new RuntimePermission("getFileSystemAttributes"));
            sm.checkRead(path);
        }
        if (isInvalid()) {
            return 0L;
        }
        return fs.getSpace(this, FileSystem.SPACE_TOTAL);
    }

    /**
     * Returns the number of unallocated bytes in the partition <a
     * href="#partName">named</a> by this abstract path name.
     *
     * <p> The returned number of unallocated bytes is a hint, but not
     * a guarantee, that it is possible to use most or any of these
     * bytes.  The number of unallocated bytes is most likely to be
     * accurate immediately after this call.  It is likely to be made
     * inaccurate by any external I/O operations including those made
     * on the system outside of this virtual machine.  This method
     * makes no guarantee that write operations to this file system
     * will succeed.
     *
     * @return  The number of unallocated bytes on the partition or <tt>0L</tt>
```

```
     *          if the abstract pathname does not name a partition.  This
     *          value will be less than or equal to the total file system size
     *          returned by {@link #getTotalSpace}.
     *
     * @throws  SecurityException
     *          If a security manager has been installed and it denies
     *          {@link RuntimePermission}<tt>("getFileSystemAttributes")</tt>
     *          or its {@link SecurityManager#checkRead(String)} method denies
     *          read access to the file named by this abstract pathname
     *
     * @since  1.6
     */
    public long getFreeSpace() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(new RuntimePermission("getFileSystemAttributes"));
            sm.checkRead(path);
        }
        if (isInvalid()) {
            return 0L;
        }
        return fs.getSpace(this, FileSystem.SPACE_FREE);
    }

    /**
     * Returns the number of bytes available to this virtual machine on the
     * partition <a href="#partName">named</a> by this abstract pathname.  When
     * possible, this method checks for write permissions and other operating
     * system restrictions and will therefore usually provide a more accurate
     * estimate of how much new data can actually be written than {@link
     * #getFreeSpace}.
     *
     * <p> The returned number of available bytes is a hint, but not a
     * guarantee, that it is possible to use most or any of these bytes.  The
     * number of unallocated bytes is most likely to be accurate immediately
     * after this call.  It is likely to be made inaccurate by any external
     * I/O operations including those made on the system outside of this
     * virtual machine.  This method makes no guarantee that write operations
     * to this file system will succeed.
     *
     * @return  The number of available bytes on the partition or <tt>0L</tt>
     *          if the abstract pathname does not name a partition.  On
     *          systems where this information is not available, this method
     *          will be equivalent to a call to {@link #getFreeSpace}.
     *
     * @throws  SecurityException
     *          If a security manager has been installed and it denies
     *          {@link RuntimePermission}<tt>("getFileSystemAttributes")</tt>
     *          or its {@link SecurityManager#checkRead(String)} method denies
     *          read access to the file named by this abstract pathname
     *
     * @since  1.6
     */
    public long getUsableSpace() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(new RuntimePermission("getFileSystemAttributes"));
            sm.checkRead(path);
        }
        if (isInvalid()) {
            return 0L;
        }
        return fs.getSpace(this, FileSystem.SPACE_USABLE);
```

```
}

/* -- Temporary files -- */

private static class TempDirectory {
    private TempDirectory() { }

    // temporary directory location
    private static final File tmpdir = new File(AccessController
        .doPrivileged(new GetPropertyAction("java.io.tmpdir")));
    static File location() {
        return tmpdir;
    }

    // file name generation
    private static final SecureRandom random = new SecureRandom();
    static File generateFile(String prefix, String suffix, File dir)
        throws IOException
    {
        long n = random.nextLong();
        if (n == Long.MIN_VALUE) {
            n = 0;      // corner case
        } else {
            n = Math.abs(n);
        }

        // Use only the file name from the supplied prefix
        prefix = (new File(prefix)).getName();

        String name = prefix + Long.toString(n) + suffix;
        File f = new File(dir, name);
        if (!name.equals(f.getName()) || f.isInvalid()) {
            if (System.getSecurityManager() != null)
                throw new IOException("Unable to create temporary file");
            else
                throw new IOException("Unable to create temporary file, " + f);
        }
        return f;
    }
}

/**
 * <p> Creates a new empty file in the specified directory, using the
 * given prefix and suffix strings to generate its name.  If this method
 * returns successfully then it is guaranteed that:
 *
 * <ol>
 * <li> The file denoted by the returned abstract pathname did not exist
 *      before this method was invoked, and
 * <li> Neither this method nor any of its variants will return the same
 *      abstract pathname again in the current invocation of the virtual
 *      machine.
 * </ol>
 *
 * This method provides only part of a temporary-file facility.  To arrange
 * for a file created by this method to be deleted automatically, use the
 * <code>{@link #deleteOnExit}</code> method.
 *
 * <p> The <code>prefix</code> argument must be at least three characters
 * long.  It is recommended that the prefix be a short, meaningful string
 * such as <code>"hjb"</code> or <code>"mail"</code>.  The
 * <code>suffix</code> argument may be <code>null</code>, in which case the
 * suffix <code>".tmp"</code> will be used.
```

```
     *
     * <p> To create the new file, the prefix and the suffix may first be
     * adjusted to fit the limitations of the underlying platform.  If the
     * prefix is too long then it will be truncated, but its first three
     * characters will always be preserved.  If the suffix is too long then it
     * too will be truncated, but if it begins with a period character
     * (<code>'.'</code>) then the period and the first three characters
     * following it will always be preserved.  Once these adjustments have been
     * made the name of the new file will be generated by concatenating the
     * prefix, five or more internally-generated characters, and the suffix.
     *
     * <p> If the <code>directory</code> argument is <code>null</code> then the
     * system-dependent default temporary-file directory will be used.  The
     * default temporary-file directory is specified by the system property
     * <code>java.io.tmpdir</code>.  On UNIX systems the default value of this
     * property is typically <code>"/tmp"</code> or <code>"/var/tmp"</code>; on
     * Microsoft Windows systems it is typically <code>"C:\\WINNT\\TEMP"</code>.  A different
     * value may be given to this system property when the Java virtual machine
     * is invoked, but programmatic changes to this property are not guaranteed
     * to have any effect upon the temporary directory used by this method.
     *
     * @param  prefix     The prefix string to be used in generating the file's
     *                    name; must be at least three characters long
     *
     * @param  suffix     The suffix string to be used in generating the file's
     *                    name; may be <code>null</code>, in which case the
     *                    suffix <code>".tmp"</code> will be used
     *
     * @param  directory  The directory in which the file is to be created, or
     *                    <code>null</code> if the default temporary-file
     *                    directory is to be used
     *
     * @return  An abstract pathname denoting a newly-created empty file
     *
     * @throws  IllegalArgumentException
     *          If the <code>prefix</code> argument contains fewer than three
     *          characters
     *
     * @throws  IOException  If a file could not be created
     *
     * @throws  SecurityException
     *          If a security manager exists and its <code>{@link
     *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
     *          method does not allow a file to be created
     *
     * @since 1.2
     */
    public static File createTempFile(String prefix, String suffix,
                                      File directory)
        throws IOException
    {
        if (prefix.length() < 3)
            throw new IllegalArgumentException("Prefix string too short");
        if (suffix == null)
            suffix = ".tmp";

        File tmpdir = (directory != null) ? directory
                                          : TempDirectory.location();
        SecurityManager sm = System.getSecurityManager();
        File f;
        do {
            f = TempDirectory.generateFile(prefix, suffix, tmpdir);
```

```
        if (sm != null) {
            try {
                sm.checkWrite(f.getPath());
            } catch (SecurityException se) {
                // don't reveal temporary directory location
                if (directory == null)
                    throw new SecurityException("Unable to create temporary file");
                throw se;
            }
        }
    } while ((fs.getBooleanAttributes(f) & FileSystem.BA_EXISTS) != 0);

    if (!fs.createFileExclusively(f.getPath()))
        throw new IOException("Unable to create temporary file");

    return f;
}

/**
 * Creates an empty file in the default temporary-file directory, using
 * the given prefix and suffix to generate its name. Invoking this method
 * is equivalent to invoking <code>{@link #createTempFile(java.lang.String,
 * java.lang.String, java.io.File)
 * createTempFile(prefix, suffix, null)}</code>.
 *
 * <p> The {@link
 * java.nio.file.Files#createTempFile(String,String,java.nio.file.attribute.FileAttribute[])
 * Files.createTempFile} method provides an alternative method to create an
 * empty file in the temporary-file directory. Files created by that method
 * may have more restrictive access permissions to files created by this
 * method and so may be more suited to security-sensitive applications.
 *
 * @param  prefix     The prefix string to be used in generating the file's
 *                    name; must be at least three characters long
 *
 * @param  suffix     The suffix string to be used in generating the file's
 *                    name; may be <code>null</code>, in which case the
 *                    suffix <code>".tmp"</code> will be used
 *
 * @return  An abstract pathname denoting a newly-created empty file
 *
 * @throws  IllegalArgumentException
 *          If the <code>prefix</code> argument contains fewer than three
 *          characters
 *
 * @throws  IOException  If a file could not be created
 *
 * @throws  SecurityException
 *          If a security manager exists and its <code>{@link
 *          java.lang.SecurityManager#checkWrite(java.lang.String)}</code>
 *          method does not allow a file to be created
 *
 * @since 1.2
 * @see java.nio.file.Files#createTempDirectory(String,FileAttribute[])
 */
public static File createTempFile(String prefix, String suffix)
    throws IOException
{
    return createTempFile(prefix, suffix, null);
}

/* -- Basic infrastructure -- */
```

```
/**
 * Compares two abstract pathnames lexicographically.  The ordering
 * defined by this method depends upon the underlying system.  On UNIX
 * systems, alphabetic case is significant in comparing pathnames; on Microsoft Windows
 * systems it is not.
 *
 * @param   pathname  The abstract pathname to be compared to this abstract
 *                    pathname
 *
 * @return  Zero if the argument is equal to this abstract pathname, a
 *          value less than zero if this abstract pathname is
 *          lexicographically less than the argument, or a value greater
 *          than zero if this abstract pathname is lexicographically
 *          greater than the argument
 *
 * @since   1.2
 */
public int compareTo(File pathname) {
    return fs.compare(this, pathname);
}

/**
 * Tests this abstract pathname for equality with the given object.
 * Returns <code>true</code> if and only if the argument is not
 * <code>null</code> and is an abstract pathname that denotes the same file
 * or directory as this abstract pathname.  Whether or not two abstract
 * pathnames are equal depends upon the underlying system.  On UNIX
 * systems, alphabetic case is significant in comparing pathnames; on Microsoft Windows
 * systems it is not.
 *
 * @param   obj   The object to be compared with this abstract pathname
 *
 * @return  <code>true</code> if and only if the objects are the same;
 *          <code>false</code> otherwise
 */
public boolean equals(Object obj) {
    if ((obj != null) && (obj instanceof File)) {
        return compareTo((File)obj) == 0;
    }
    return false;
}

/**
 * Computes a hash code for this abstract pathname.  Because equality of
 * abstract pathnames is inherently system-dependent, so is the computation
 * of their hash codes.  On UNIX systems, the hash code of an abstract
 * pathname is equal to the exclusive <em>or</em> of the hash code
 * of its pathname string and the decimal value
 * <code>1234321</code>.  On Microsoft Windows systems, the hash
 * code is equal to the exclusive <em>or</em> of the hash code of
 * its pathname string converted to lower case and the decimal
 * value <code>1234321</code>.  Locale is not taken into account on
 * lowercasing the pathname string.
 *
 * @return  A hash code for this abstract pathname
 */
public int hashCode() {
    return fs.hashCode(this);
}

/**
 * Returns the pathname string of this abstract pathname.  This is just the
 * string returned by the <code>{@link #getPath}</code> method.
```

```
     *
     * @return  The string form of this abstract pathname
     */
    public String toString() {
        return getPath();
    }

    /**
     * WriteObject is called to save this filename.
     * The separator character is saved also so it can be replaced
     * in case the path is reconstituted on a different host type.
     * <p>
     * @serialData  Default fields followed by separator character.
     */
    private synchronized void writeObject(java.io.ObjectOutputStream s)
        throws IOException
    {
        s.defaultWriteObject();
        s.writeChar(separatorChar); // Add the separator character
    }

    /**
     * readObject is called to restore this filename.
     * The original separator character is read.  If it is different
     * than the separator character on this system, then the old separator
     * is replaced by the local separator.
     */
    private synchronized void readObject(java.io.ObjectInputStream s)
         throws IOException, ClassNotFoundException
    {
        ObjectInputStream.GetField fields = s.readFields();
        String pathField = (String)fields.get("path", null);
        char sep = s.readChar(); // read the previous separator char
        if (sep != separatorChar)
            pathField = pathField.replace(sep, separatorChar);
        String path = fs.normalize(pathField);
        UNSAFE.putObject(this, PATH_OFFSET, path);
        UNSAFE.putIntVolatile(this, PREFIX_LENGTH_OFFSET, fs.prefixLength(path));
    }

    private static final long PATH_OFFSET;
    private static final long PREFIX_LENGTH_OFFSET;
    private static final sun.misc.Unsafe UNSAFE;
    static {
        try {
            sun.misc.Unsafe unsafe = sun.misc.Unsafe.getUnsafe();
            PATH_OFFSET = unsafe.objectFieldOffset(
                    File.class.getDeclaredField("path"));
            PREFIX_LENGTH_OFFSET = unsafe.objectFieldOffset(
                    File.class.getDeclaredField("prefixLength"));
            UNSAFE = unsafe;
        } catch (ReflectiveOperationException e) {
            throw new Error(e);
        }
    }


    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = 301077366599181567L;

    // -- Integration with java.nio.file --

    private volatile transient Path filePath;
```

```
    /**
     * Returns a {@link Path java.nio.file.Path} object constructed from the
     * this abstract path. The resulting {@code Path} is associated with the
     * {@link java.nio.file.FileSystems#getDefault default-filesystem}.
     *
     * <p> The first invocation of this method works as if invoking it were
     * equivalent to evaluating the expression:
     * <blockquote><pre>
     * {@link java.nio.file.FileSystems#getDefault FileSystems.getDefault}().{@link
     * java.nio.file.FileSystem#getPath getPath}(this.{@link #getPath getPath}());
     * </pre></blockquote>
     * Subsequent invocations of this method return the same {@code Path}.
     *
     * <p> If this abstract pathname is the empty abstract pathname then this
     * method returns a {@code Path} that may be used to access the current
     * user directory.
     *
     * @return  a {@code Path} constructed from this abstract path
     *
     * @throws  java.nio.file.InvalidPathException
     *          if a {@code Path} object cannot be constructed from the abstract
     *          path (see {@link java.nio.file.FileSystem#getPath FileSystem.getPath})
     *
     * @since   1.7
     * @see Path#toFile
     */
    public Path toPath() {
        Path result = filePath;
        if (result == null) {
            synchronized (this) {
                result = filePath;
                if (result == null) {
                    result = FileSystems.getDefault().getPath(path);
                    filePath = result;
                }
            }
        }
        return result;
    }
}
```

FileDescriptor.java

```
/*
 * Copyright (c) 2003, 2011, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.util.ArrayList;
import java.util.List;

/**
 * Instances of the file descriptor class serve as an opaque handle
 * to the underlying machine-specific structure representing an
 * open file, an open socket, or another source or sink of bytes.
 * The main practical use for a file descriptor is to create a
 * {@link FileInputStream} or {@link FileOutputStream} to contain it.
 *
 * <p>Applications should not create their own file descriptors.
 *
 * @author  Pavani Diwanji
 * @since   JDK1.0
 */
public final class FileDescriptor {

    private int fd;

    private long handle;

    private Closeable parent;
    private List<Closeable> otherParents;
    private boolean closed;

    /**
     * Constructs an (invalid) FileDescriptor
     * object.
     */
    public /**/ FileDescriptor() {
        fd = -1;
        handle = -1;
    }
```

```
static {
    initIDs();
}

// Set up JavaIOFileDescriptorAccess in SharedSecrets
static {
    sun.misc.SharedSecrets.setJavaIOFileDescriptorAccess(
        new sun.misc.JavaIOFileDescriptorAccess() {
            public void set(FileDescriptor obj, int fd) {
                obj.fd = fd;
            }

            public int get(FileDescriptor obj) {
                return obj.fd;
            }

            public void setHandle(FileDescriptor obj, long handle) {
                obj.handle = handle;
            }

            public long getHandle(FileDescriptor obj) {
                return obj.handle;
            }
        }
    );
}

/**
 * A handle to the standard input stream. Usually, this file
 * descriptor is not used directly, but rather via the input stream
 * known as {@code System.in}.
 *
 * @see     java.lang.System#in
 */
public static final FileDescriptor in = standardStream(0);

/**
 * A handle to the standard output stream. Usually, this file
 * descriptor is not used directly, but rather via the output stream
 * known as {@code System.out}.
 * @see     java.lang.System#out
 */
public static final FileDescriptor out = standardStream(1);

/**
 * A handle to the standard error stream. Usually, this file
 * descriptor is not used directly, but rather via the output stream
 * known as {@code System.err}.
 *
 * @see     java.lang.System#err
 */
public static final FileDescriptor err = standardStream(2);

/**
 * Tests if this file descriptor object is valid.
 *
 * @return  {@code true} if the file descriptor object represents a
 *          valid, open file, socket, or other active I/O connection;
 *          {@code false} otherwise.
 */
public boolean valid() {
    return ((handle != -1) || (fd != -1));
```

```
}

/**
 * Force all system buffers to synchronize with the underlying
 * device.  This method returns after all modified data and
 * attributes of this FileDescriptor have been written to the
 * relevant device(s).  In particular, if this FileDescriptor
 * refers to a physical storage medium, such as a file in a file
 * system, sync will not return until all in-memory modified copies
 * of buffers associated with this FileDesecriptor have been
 * written to the physical medium.
 *
 * sync is meant to be used by code that requires physical
 * storage (such as a file) to be in a known state  For
 * example, a class that provided a simple transaction facility
 * might use sync to ensure that all changes to a file caused
 * by a given transaction were recorded on a storage medium.
 *
 * sync only affects buffers downstream of this FileDescriptor.  If
 * any in-memory buffering is being done by the application (for
 * example, by a BufferedOutputStream object), those buffers must
 * be flushed into the FileDescriptor (for example, by invoking
 * OutputStream.flush) before that data will be affected by sync.
 *
 * @exception SyncFailedException
 *        Thrown when the buffers cannot be flushed,
 *        or because the system cannot guarantee that all the
 *        buffers have been synchronized with physical media.
 * @since     JDK1.1
 */
public native void sync() throws SyncFailedException;

/* This routine initializes JNI field offsets for the class */
private static native void initIDs();

private static native long set(int d);

private static FileDescriptor standardStream(int fd) {
    FileDescriptor desc = new FileDescriptor();
    desc.handle = set(fd);
    return desc;
}

/*
 * Package private methods to track referents.
 * If multiple streams point to the same FileDescriptor, we cycle
 * through the list of all referents and call close()
 */

/**
 * Attach a Closeable to this FD for tracking.
 * parent reference is added to otherParents when
 * needed to make closeAll simpler.
 */
synchronized void attach(Closeable c) {
    if (parent == null) {
        // first caller gets to do this
        parent = c;
    } else if (otherParents == null) {
        otherParents = new ArrayList<>();
        otherParents.add(parent);
        otherParents.add(c);
    } else {
```

```
            otherParents.add(c);
        }
    }

    /**
     * Cycle through all Closeables sharing this FD and call
     * close() on each one.
     *
     * The caller closeable gets to call close0().
     */
    @SuppressWarnings("try")
    synchronized void closeAll(Closeable releaser) throws IOException {
        if (!closed) {
            closed = true;
            IOException ioe = null;
            try (Closeable c = releaser) {
                if (otherParents != null) {
                    for (Closeable referent : otherParents) {
                        try {
                            referent.close();
                        } catch(IOException x) {
                            if (ioe == null) {
                                ioe = x;
                            } else {
                                ioe.addSuppressed(x);
                            }
                        }
                    }
                }
            } catch(IOException ex) {
                /*
                 * If releaser close() throws IOException
                 * add other exceptions as suppressed.
                 */
                if (ioe != null)
                    ex.addSuppressed(ioe);
                ioe = ex;
            } finally {
                if (ioe != null)
                    throw ioe;
            }
        }
    }
}
```

FileFilter.java

```
/*
 * Copyright (c) 1998, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * A filter for abstract pathnames.
 *
 * <p> Instances of this interface may be passed to the <code>{@link
 * File#listFiles(java.io.FileFilter) listFiles(FileFilter)}</code> method
 * of the <code>{@link java.io.File}</code> class.
 *
 * @since 1.2
 */
@FunctionalInterface
public interface FileFilter {

    /**
     * Tests whether or not the specified abstract pathname should be
     * included in a pathname list.
     *
     * @param  pathname  The abstract pathname to be tested
     * @return  <code>true</code> if and only if <code>pathname</code>
     *          should be included
     */
    boolean accept(File pathname);
}
```

FileInputStream.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.nio.channels.FileChannel;
import sun.nio.ch.FileChannelImpl;


/**
 * A <code>FileInputStream</code> obtains input bytes
 * from a file in a file system. What files
 * are  available depends on the host environment.
 *
 * <p><code>FileInputStream</code> is meant for reading streams of raw bytes
 * such as image data. For reading streams of characters, consider using
 * <code>FileReader</code>.
 *
 * @author  Arthur van Hoff
 * @see     java.io.File
 * @see     java.io.FileDescriptor
 * @see     java.io.FileOutputStream
 * @see     java.nio.file.Files#newInputStream
 * @since   JDK1.0
 */
public
class FileInputStream extends InputStream
{
    /* File Descriptor - handle to the open file */
    private final FileDescriptor fd;

    /**
     * The path of the referenced file
     * (null if the stream is created with a file descriptor)
     */
    private final String path;

    private FileChannel channel = null;
```

```
private final Object closeLock = new Object();
private volatile boolean closed = false;

/**
 * Creates a <code>FileInputStream</code> by
 * opening a connection to an actual file,
 * the file named by the path name <code>name</code>
 * in the file system.  A new <code>FileDescriptor</code>
 * object is created to represent this file
 * connection.
 * <p>
 * First, if there is a security
 * manager, its <code>checkRead</code> method
 * is called with the <code>name</code> argument
 * as its argument.
 * <p>
 * If the named file does not exist, is a directory rather than a regular
 * file, or for some other reason cannot be opened for reading then a
 * <code>FileNotFoundException</code> is thrown.
 *
 * @param      name   the system-dependent file name.
 * @exception  FileNotFoundException  if the file does not exist,
 *                   is a directory rather than a regular file,
 *                   or for some other reason cannot be opened for
 *                   reading.
 * @exception  SecurityException      if a security manager exists and its
 *               <code>checkRead</code> method denies read access
 *               to the file.
 * @see        java.lang.SecurityManager#checkRead(java.lang.String)
 */
public FileInputStream(String name) throws FileNotFoundException {
    this(name != null ? new File(name) : null);
}

/**
 * Creates a <code>FileInputStream</code> by
 * opening a connection to an actual file,
 * the file named by the <code>File</code>
 * object <code>file</code> in the file system.
 * A new <code>FileDescriptor</code> object
 * is created to represent this file connection.
 * <p>
 * First, if there is a security manager,
 * its <code>checkRead</code> method  is called
 * with the path represented by the <code>file</code>
 * argument as its argument.
 * <p>
 * If the named file does not exist, is a directory rather than a regular
 * file, or for some other reason cannot be opened for reading then a
 * <code>FileNotFoundException</code> is thrown.
 *
 * @param      file   the file to be opened for reading.
 * @exception  FileNotFoundException  if the file does not exist,
 *                   is a directory rather than a regular file,
 *                   or for some other reason cannot be opened for
 *                   reading.
 * @exception  SecurityException      if a security manager exists and its
 *               <code>checkRead</code> method denies read access to the file.
 * @see        java.io.File#getPath()
 * @see        java.lang.SecurityManager#checkRead(java.lang.String)
 */
public FileInputStream(File file) throws FileNotFoundException {
```

```
    String name = (file != null ? file.getPath() : null);
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkRead(name);
    }
    if (name == null) {
        throw new NullPointerException();
    }
    if (file.isInvalid()) {
        throw new FileNotFoundException("Invalid file path");
    }
    fd = new FileDescriptor();
    fd.attach(this);
    path = name;
    open(name);
}

/**
 * Creates a <code>FileInputStream</code> by using the file descriptor
 * <code>fdObj</code>, which represents an existing connection to an
 * actual file in the file system.
 * <p>
 * If there is a security manager, its <code>checkRead</code> method is
 * called with the file descriptor <code>fdObj</code> as its argument to
 * see if it's ok to read the file descriptor. If read access is denied
 * to the file descriptor a <code>SecurityException</code> is thrown.
 * <p>
 * If <code>fdObj</code> is null then a <code>NullPointerException</code>
 * is thrown.
 * <p>
 * This constructor does not throw an exception if <code>fdObj</code>
 * is {@link java.io.FileDescriptor#valid() invalid}.
 * However, if the methods are invoked on the resulting stream to attempt
 * I/O on the stream, an <code>IOException</code> is thrown.
 *
 * @param      fdObj   the file descriptor to be opened for reading.
 * @throws     SecurityException      if a security manager exists and its
 *                <code>checkRead</code> method denies read access to the
 *                file descriptor.
 * @see        SecurityManager#checkRead(java.io.FileDescriptor)
 */
public FileInputStream(FileDescriptor fdObj) {
    SecurityManager security = System.getSecurityManager();
    if (fdObj == null) {
        throw new NullPointerException();
    }
    if (security != null) {
        security.checkRead(fdObj);
    }
    fd = fdObj;
    path = null;

    /*
     * FileDescriptor is being shared by streams.
     * Register this stream with FileDescriptor tracker.
     */
    fd.attach(this);
}

/**
 * Opens the specified file for reading.
 * @param name the name of the file
 */
```

```
private native void open(String name) throws FileNotFoundException;

/**
 * Reads a byte of data from this input stream. This method blocks
 * if no input is yet available.
 *
 * @return     the next byte of data, or <code>-1</code> if the end of the
 *             file is reached.
 * @exception  IOException  if an I/O error occurs.
 */
public int read() throws IOException {
    return read0();
}

private native int read0() throws IOException;

/**
 * Reads a subarray as a sequence of bytes.
 * @param b the data to be written
 * @param off the start offset in the data
 * @param len the number of bytes that are written
 * @exception IOException If an I/O error has occurred.
 */
private native int readBytes(byte b[], int off, int len) throws IOException;

/**
 * Reads up to <code>b.length</code> bytes of data from this input
 * stream into an array of bytes. This method blocks until some input
 * is available.
 *
 * @param      b   the buffer into which the data is read.
 * @return     the total number of bytes read into the buffer, or
 *             <code>-1</code> if there is no more data because the end of
 *             the file has been reached.
 * @exception  IOException  if an I/O error occurs.
 */
public int read(byte b[]) throws IOException {
    return readBytes(b, 0, b.length);
}

/**
 * Reads up to <code>len</code> bytes of data from this input stream
 * into an array of bytes. If <code>len</code> is not zero, the method
 * blocks until some input is available; otherwise, no
 * bytes are read and <code>0</code> is returned.
 *
 * @param      b     the buffer into which the data is read.
 * @param      off   the start offset in the destination array <code>b</code>
 * @param      len   the maximum number of bytes read.
 * @return     the total number of bytes read into the buffer, or
 *             <code>-1</code> if there is no more data because the end of
 *             the file has been reached.
 * @exception  NullPointerException If <code>b</code> is <code>null</code>.
 * @exception  IndexOutOfBoundsException If <code>off</code> is negative,
 * <code>len</code> is negative, or <code>len</code> is greater than
 * <code>b.length - off</code>
 * @exception  IOException  if an I/O error occurs.
 */
public int read(byte b[], int off, int len) throws IOException {
    return readBytes(b, off, len);
}

/**
```

```
     * Skips over and discards <code>n</code> bytes of data from the
     * input stream.
     *
     * <p>The <code>skip</code> method may, for a variety of
     * reasons, end up skipping over some smaller number of bytes,
     * possibly <code>0</code>. If <code>n</code> is negative, the method
     * will try to skip backwards. In case the backing file does not support
     * backward skip at its current position, an <code>IOException</code> is
     * thrown. The actual number of bytes skipped is returned. If it skips
     * forwards, it returns a positive value. If it skips backwards, it
     * returns a negative value.
     *
     * <p>This method may skip more bytes than what are remaining in the
     * backing file. This produces no exception and the number of bytes skipped
     * may include some number of bytes that were beyond the EOF of the
     * backing file. Attempting to read from the stream after skipping past
     * the end will result in -1 indicating the end of the file.
     *
     * @param      n   the number of bytes to be skipped.
     * @return     the actual number of bytes skipped.
     * @exception  IOException  if n is negative, if the stream does not
     *             support seek, or if an I/O error occurs.
     */
    public native long skip(long n) throws IOException;

    /**
     * Returns an estimate of the number of remaining bytes that can be read (or
     * skipped over) from this input stream without blocking by the next
     * invocation of a method for this input stream. Returns 0 when the file
     * position is beyond EOF. The next invocation might be the same thread
     * or another thread. A single read or skip of this many bytes will not
     * block, but may read or skip fewer bytes.
     *
     * <p> In some cases, a non-blocking read (or skip) may appear to be
     * blocked when it is merely slow, for example when reading large
     * files over slow networks.
     *
     * @return     an estimate of the number of remaining bytes that can be read
     *             (or skipped over) from this input stream without blocking.
     * @exception  IOException  if this file input stream has been closed by calling
     *             {@code close} or an I/O error occurs.
     */
    public native int available() throws IOException;

    /**
     * Closes this file input stream and releases any system resources
     * associated with the stream.
     *
     * <p> If this stream has an associated channel then the channel is closed
     * as well.
     *
     * @exception  IOException  if an I/O error occurs.
     *
     * @revised 1.4
     * @spec JSR-51
     */
    public void close() throws IOException {
        synchronized (closeLock) {
            if (closed) {
                return;
            }
            closed = true;
        }
```

```
    if (channel != null) {
       channel.close();
    }

    fd.closeAll(new Closeable() {
        public void close() throws IOException {
           close0();
       }
    });
}

/**
 * Returns the <code>FileDescriptor</code>
 * object  that represents the connection to
 * the actual file in the file system being
 * used by this <code>FileInputStream</code>.
 *
 * @return     the file descriptor object associated with this stream.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FileDescriptor
 */
public final FileDescriptor getFD() throws IOException {
    if (fd != null) {
        return fd;
    }
    throw new IOException();
}

/**
 * Returns the unique {@link java.nio.channels.FileChannel FileChannel}
 * object associated with this file input stream.
 *
 * <p> The initial {@link java.nio.channels.FileChannel#position()
 * position} of the returned channel will be equal to the
 * number of bytes read from the file so far.  Reading bytes from this
 * stream will increment the channel's position.  Changing the channel's
 * position, either explicitly or by reading, will change this stream's
 * file position.
 *
 * @return  the file channel associated with this file input stream
 *
 * @since 1.4
 * @spec JSR-51
 */
public FileChannel getChannel() {
    synchronized (this) {
        if (channel == null) {
            channel = FileChannelImpl.open(fd, path, true, false, this);
        }
        return channel;
    }
}

private static native void initIDs();

private native void close0() throws IOException;

static {
    initIDs();
}

/**
 * Ensures that the <code>close</code> method of this file input stream is
```

```
     * called when there are no more references to it.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FileInputStream#close()
     */
    protected void finalize() throws IOException {
        if ((fd != null) &&  (fd != FileDescriptor.in)) {
            /* if fd is shared, the references in FileDescriptor
             * will ensure that finalizer is only called when
             * safe to do so. All references using the fd have
             * become unreachable. We can call close()
             */
            close();
        }
    }
}
```

FilenameFilter.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Instances of classes that implement this interface are used to
 * filter filenames. These instances are used to filter directory
 * listings in the <code>list</code> method of class
 * <code>File</code>, and by the Abstract Window Toolkit's file
 * dialog component.
 *
 * @author  Arthur van Hoff
 * @author  Jonathan Payne
 * @see     java.awt.FileDialog#setFilenameFilter(java.io.FilenameFilter)
 * @see     java.io.File
 * @see     java.io.File#list(java.io.FilenameFilter)
 * @since   JDK1.0
 */
@FunctionalInterface
public interface FilenameFilter {
    /**
     * Tests if a specified file should be included in a file list.
     *
     * @param   dir    the directory in which the file was found.
     * @param   name   the name of the file.
     * @return  <code>true</code> if and only if the name should be
     * included in the file list; <code>false</code> otherwise.
     */
    boolean accept(File dir, String name);
}
```

FileNotFoundException.java

```
/*
 * Copyright (c) 1994, 2008, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Signals that an attempt to open the file denoted by a specified pathname
 * has failed.
 *
 * <p> This exception will be thrown by the {@link FileInputStream}, {@link
 * FileOutputStream}, and {@link RandomAccessFile} constructors when a file
 * with the specified pathname does not exist.  It will also be thrown by these
 * constructors if the file does exist but for some reason is inaccessible, for
 * example when an attempt is made to open a read-only file for writing.
 *
 * @author  unascribed
 * @since   JDK1.0
 */

public class FileNotFoundException extends IOException {
    private static final long serialVersionUID = -897856973823710492L;

    /**
     * Constructs a <code>FileNotFoundException</code> with
     * <code>null</code> as its error detail message.
     */
    public FileNotFoundException() {
        super();
    }

    /**
     * Constructs a <code>FileNotFoundException</code> with the
     * specified detail message. The string <code>s</code> can be
     * retrieved later by the
     * <code>{@link java.lang.Throwable#getMessage}</code>
     * method of class <code>java.lang.Throwable</code>.
     *
```

```
     * @param   s   the detail message.
     */
    public FileNotFoundException(String s) {
        super(s);
    }

    /**
     * Constructs a <code>FileNotFoundException</code> with a detail message
     * consisting of the given pathname string followed by the given reason
     * string.  If the <code>reason</code> argument is <code>null</code> then
     * it will be omitted.  This private constructor is invoked only by native
     * I/O methods.
     *
     * @since 1.2
     */
    private FileNotFoundException(String path, String reason) {
        super(path + ((reason == null)
                      ? ""
                      : " (" + reason + ")"));
    }

}
```

FileOutputStream.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.nio.channels.FileChannel;
import sun.nio.ch.FileChannelImpl;


/**
 * A file output stream is an output stream for writing data to a
 * <code>File</code> or to a <code>FileDescriptor</code>. Whether or not
 * a file is available or may be created depends upon the underlying
 * platform.  Some platforms, in particular, allow a file to be opened
 * for writing by only one <tt>FileOutputStream</tt> (or other
 * file-writing object) at a time.  In such situations the constructors in
 * this class will fail if the file involved is already open.
 *
 * <p><code>FileOutputStream</code> is meant for writing streams of raw bytes
 * such as image data. For writing streams of characters, consider using
 * <code>FileWriter</code>.
 *
 * @author  Arthur van Hoff
 * @see     java.io.File
 * @see     java.io.FileDescriptor
 * @see     java.io.FileInputStream
 * @see     java.nio.file.Files#newOutputStream
 * @since   JDK1.0
 */
public
class FileOutputStream extends OutputStream
{
    /**
     * The system dependent file descriptor.
     */
    private final FileDescriptor fd;

    /**
```

```
     * True if the file is opened for append.
     */
    private final boolean append;

    /**
     * The associated channel, initialized lazily.
     */
    private FileChannel channel;

    /**
     * The path of the referenced file
     * (null if the stream is created with a file descriptor)
     */
    private final String path;

    private final Object closeLock = new Object();
    private volatile boolean closed = false;

    /**
     * Creates a file output stream to write to the file with the
     * specified name. A new <code>FileDescriptor</code> object is
     * created to represent this file connection.
     * <p>
     * First, if there is a security manager, its <code>checkWrite</code>
     * method is called with <code>name</code> as its argument.
     * <p>
     * If the file exists but is a directory rather than a regular file, does
     * not exist but cannot be created, or cannot be opened for any other
     * reason then a <code>FileNotFoundException</code> is thrown.
     *
     * @param      name   the system-dependent filename
     * @exception  FileNotFoundException  if the file exists but is a directory
     *                   rather than a regular file, does not exist but cannot
     *                   be created, or cannot be opened for any other reason
     * @exception  SecurityException  if a security manager exists and its
     *               <code>checkWrite</code> method denies write access
     *               to the file.
     * @see        java.lang.SecurityManager#checkWrite(java.lang.String)
     */
    public FileOutputStream(String name) throws FileNotFoundException {
        this(name != null ? new File(name) : null, false);
    }

    /**
     * Creates a file output stream to write to the file with the specified
     * name.  If the second argument is <code>true</code>, then
     * bytes will be written to the end of the file rather than the beginning.
     * A new <code>FileDescriptor</code> object is created to represent this
     * file connection.
     * <p>
     * First, if there is a security manager, its <code>checkWrite</code>
     * method is called with <code>name</code> as its argument.
     * <p>
     * If the file exists but is a directory rather than a regular file, does
     * not exist but cannot be created, or cannot be opened for any other
     * reason then a <code>FileNotFoundException</code> is thrown.
     *
     * @param     name        the system-dependent file name
     * @param     append      if <code>true</code>, then bytes will be written
     *                   to the end of the file rather than the beginning
     * @exception  FileNotFoundException  if the file exists but is a directory
     *                   rather than a regular file, does not exist but cannot
     *                   be created, or cannot be opened for any other reason.
```

```
     * @exception  SecurityException  if a security manager exists and its
     *               <code>checkWrite</code> method denies write access
     *               to the file.
     * @see        java.lang.SecurityManager#checkWrite(java.lang.String)
     * @since     JDK1.1
     */
    public FileOutputStream(String name, boolean append)
        throws FileNotFoundException
    {
        this(name != null ? new File(name) : null, append);
    }

    /**
     * Creates a file output stream to write to the file represented by
     * the specified <code>File</code> object. A new
     * <code>FileDescriptor</code> object is created to represent this
     * file connection.
     * <p>
     * First, if there is a security manager, its <code>checkWrite</code>
     * method is called with the path represented by the <code>file</code>
     * argument as its argument.
     * <p>
     * If the file exists but is a directory rather than a regular file, does
     * not exist but cannot be created, or cannot be opened for any other
     * reason then a <code>FileNotFoundException</code> is thrown.
     *
     * @param      file               the file to be opened for writing.
     * @exception  FileNotFoundException  if the file exists but is a directory
     *                   rather than a regular file, does not exist but cannot
     *                   be created, or cannot be opened for any other reason
     * @exception  SecurityException  if a security manager exists and its
     *               <code>checkWrite</code> method denies write access
     *               to the file.
     * @see        java.io.File#getPath()
     * @see        java.lang.SecurityException
     * @see        java.lang.SecurityManager#checkWrite(java.lang.String)
     */
    public FileOutputStream(File file) throws FileNotFoundException {
        this(file, false);
    }

    /**
     * Creates a file output stream to write to the file represented by
     * the specified <code>File</code> object. If the second argument is
     * <code>true</code>, then bytes will be written to the end of the file
     * rather than the beginning. A new <code>FileDescriptor</code> object is
     * created to represent this file connection.
     * <p>
     * First, if there is a security manager, its <code>checkWrite</code>
     * method is called with the path represented by the <code>file</code>
     * argument as its argument.
     * <p>
     * If the file exists but is a directory rather than a regular file, does
     * not exist but cannot be created, or cannot be opened for any other
     * reason then a <code>FileNotFoundException</code> is thrown.
     *
     * @param      file               the file to be opened for writing.
     * @param     append      if <code>true</code>, then bytes will be written
     *                   to the end of the file rather than the beginning
     * @exception  FileNotFoundException  if the file exists but is a directory
     *                   rather than a regular file, does not exist but cannot
     *                   be created, or cannot be opened for any other reason
     * @exception  SecurityException  if a security manager exists and its
```

```
     *               <code>checkWrite</code> method denies write access
     *               to the file.
     * @see         java.io.File#getPath()
     * @see         java.lang.SecurityException
     * @see         java.lang.SecurityManager#checkWrite(java.lang.String)
     * @since 1.4
     */
    public FileOutputStream(File file, boolean append)
        throws FileNotFoundException
    {
        String name = (file != null ? file.getPath() : null);
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(name);
        }
        if (name == null) {
            throw new NullPointerException();
        }
        if (file.isInvalid()) {
            throw new FileNotFoundException("Invalid file path");
        }
        this.fd = new FileDescriptor();
        fd.attach(this);
        this.append = append;
        this.path = name;

        open(name, append);
    }

    /**
     * Creates a file output stream to write to the specified file
     * descriptor, which represents an existing connection to an actual
     * file in the file system.
     * <p>
     * First, if there is a security manager, its <code>checkWrite</code>
     * method is called with the file descriptor <code>fdObj</code>
     * argument as its argument.
     * <p>
     * If <code>fdObj</code> is null then a <code>NullPointerException</code>
     * is thrown.
     * <p>
     * This constructor does not throw an exception if <code>fdObj</code>
     * is {@link java.io.FileDescriptor#valid() invalid}.
     * However, if the methods are invoked on the resulting stream to attempt
     * I/O on the stream, an <code>IOException</code> is thrown.
     *
     * @param      fdObj   the file descriptor to be opened for writing
     * @exception  SecurityException  if a security manager exists and its
     *               <code>checkWrite</code> method denies
     *               write access to the file descriptor
     * @see         java.lang.SecurityManager#checkWrite(java.io.FileDescriptor)
     */
    public FileOutputStream(FileDescriptor fdObj) {
        SecurityManager security = System.getSecurityManager();
        if (fdObj == null) {
            throw new NullPointerException();
        }
        if (security != null) {
            security.checkWrite(fdObj);
        }
        this.fd = fdObj;
        this.append = false;
        this.path = null;
```

```
    fd.attach(this);
}

/**
 * Opens a file, with the specified name, for overwriting or appending.
 * @param name name of file to be opened
 * @param append whether the file is to be opened in append mode
 */
private native void open(String name, boolean append)
    throws FileNotFoundException;

/**
 * Writes the specified byte to this file output stream.
 *
 * @param   b   the byte to be written.
 * @param   append   {@code true} if the write operation first
 *     advances the position to the end of file
 */
private native void write(int b, boolean append) throws IOException;

/**
 * Writes the specified byte to this file output stream. Implements
 * the <code>write</code> method of <code>OutputStream</code>.
 *
 * @param      b   the byte to be written.
 * @exception  IOException  if an I/O error occurs.
 */
public void write(int b) throws IOException {
    write(b, append);
}

/**
 * Writes a sub array as a sequence of bytes.
 * @param b the data to be written
 * @param off the start offset in the data
 * @param len the number of bytes that are written
 * @param append {@code true} to first advance the position to the
 *     end of file
 * @exception IOException If an I/O error has occurred.
 */
private native void writeBytes(byte b[], int off, int len, boolean append)
    throws IOException;

/**
 * Writes <code>b.length</code> bytes from the specified byte array
 * to this file output stream.
 *
 * @param      b   the data.
 * @exception  IOException  if an I/O error occurs.
 */
public void write(byte b[]) throws IOException {
    writeBytes(b, 0, b.length, append);
}

/**
 * Writes <code>len</code> bytes from the specified byte array
 * starting at offset <code>off</code> to this file output stream.
 *
 * @param      b     the data.
 * @param      off   the start offset in the data.
 * @param      len   the number of bytes to write.
 * @exception  IOException  if an I/O error occurs.
```

```
 */
public void write(byte b[], int off, int len) throws IOException {
    writeBytes(b, off, len, append);
}

/**
 * Closes this file output stream and releases any system resources
 * associated with this stream. This file output stream may no longer
 * be used for writing bytes.
 *
 * <p> If this stream has an associated channel then the channel is closed
 * as well.
 *
 * @exception  IOException  if an I/O error occurs.
 *
 * @revised 1.4
 * @spec JSR-51
 */
public void close() throws IOException {
    synchronized (closeLock) {
        if (closed) {
            return;
        }
        closed = true;
    }

    if (channel != null) {
        channel.close();
    }

    fd.closeAll(new Closeable() {
        public void close() throws IOException {
           close0();
       }
    });
}

/**
 * Returns the file descriptor associated with this stream.
 *
 * @return  the <code>FileDescriptor</code> object that represents
 *          the connection to the file in the file system being used
 *          by this <code>FileOutputStream</code> object.
 *
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FileDescriptor
 */
 public final FileDescriptor getFD()  throws IOException {
    if (fd != null) {
        return fd;
    }
    throw new IOException();
 }

/**
 * Returns the unique {@link java.nio.channels.FileChannel FileChannel}
 * object associated with this file output stream.
 *
 * <p> The initial {@link java.nio.channels.FileChannel#position()
 * position} of the returned channel will be equal to the
 * number of bytes written to the file so far unless this stream is in
 * append mode, in which case it will be equal to the size of the file.
 * Writing bytes to this stream will increment the channel's position
```

```
     * accordingly.  Changing the channel's position, either explicitly or by
     * writing, will change this stream's file position.
     *
     * @return  the file channel associated with this file output stream
     *
     * @since 1.4
     * @spec JSR-51
     */
    public FileChannel getChannel() {
        synchronized (this) {
            if (channel == null) {
                channel = FileChannelImpl.open(fd, path, false, true, append, this);
            }
            return channel;
        }
    }

    /**
     * Cleans up the connection to the file, and ensures that the
     * <code>close</code> method of this file output stream is
     * called when there are no more references to this stream.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FileInputStream#close()
     */
    protected void finalize() throws IOException {
        if (fd != null) {
            if (fd == FileDescriptor.out || fd == FileDescriptor.err) {
                flush();
            } else {
                /* if fd is shared, the references in FileDescriptor
                 * will ensure that finalizer is only called when
                 * safe to do so. All references using the fd have
                 * become unreachable. We can call close()
                 */
                close();
            }
        }
    }

    private native void close0() throws IOException;

    private static native void initIDs();

    static {
        initIDs();
    }

}
```

FilePermission.java

```
/*
 * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.security.*;
import java.util.Enumeration;
import java.util.List;
import java.util.ArrayList;
import java.util.Vector;
import java.util.Collections;
import sun.security.util.SecurityConstants;

/**
 * This class represents access to a file or directory.  A FilePermission consists
 * of a pathname and a set of actions valid for that pathname.
 * <P>
 * Pathname is the pathname of the file or directory granted the specified
 * actions. A pathname that ends in "/*" (where "/" is
 * the file separator character, <code>File.separatorChar</code>) indicates
 * all the files and directories contained in that directory. A pathname
 * that ends with "/-" indicates (recursively) all files
 * and subdirectories contained in that directory. A pathname consisting of
 * the special token "<<ALL FILES>>" matches <b>any</b> file.
 * <P>
 * Note: A pathname consisting of a single "*" indicates all the files
 * in the current directory, while a pathname consisting of a single "-"
 * indicates all the files in the current directory and
 * (recursively) all files and subdirectories contained in the current
 * directory.
 * <P>
 * The actions to be granted are passed to the constructor in a string containing
 * a list of one or more comma-separated keywords. The possible keywords are
 * "read", "write", "execute", "delete", and "readlink". Their meaning is
 * defined as follows:
 *
 * <DL>
 *    <DT> read <DD> read permission
```

```
 *    <DT> write <DD> write permission
 *    <DT> execute
 *    <DD> execute permission. Allows <code>Runtime.exec</code> to
 *         be called. Corresponds to <code>SecurityManager.checkExec</code>.
 *    <DT> delete
 *    <DD> delete permission. Allows <code>File.delete</code> to
 *         be called. Corresponds to <code>SecurityManager.checkDelete</code>.
 *    <DT> readlink
 *    <DD> read link permission. Allows the target of a
 *         <a href="../nio/file/package-summary.html#links">symbolic link</a>
 *         to be read by invoking the {@link java.nio.file.Files#readSymbolicLink
 *         readSymbolicLink } method.
 * </DL>
 * <P>
 * The actions string is converted to lowercase before processing.
 * <P>
 * Be careful when granting FilePermissions. Think about the implications
 * of granting read and especially write access to various files and
 * directories. The "<<ALL FILES>>" permission with write action is
 * especially dangerous. This grants permission to write to the entire
 * file system. One thing this effectively allows is replacement of the
 * system binary, including the JVM runtime environment.
 *
 * <p>Please note: Code can always read a file from the same
 * directory it's in (or a subdirectory of that directory); it does not
 * need explicit permission to do so.
 *
 * @see java.security.Permission
 * @see java.security.Permissions
 * @see java.security.PermissionCollection
 *
 *
 * @author Marianne Mueller
 * @author Roland Schemers
 * @since 1.2
 *
 * @serial exclude
 */

public final class FilePermission extends Permission implements Serializable {

    /**
     * Execute action.
     */
    private final static int EXECUTE = 0x1;
    /**
     * Write action.
     */
    private final static int WRITE   = 0x2;
    /**
     * Read action.
     */
    private final static int READ    = 0x4;
    /**
     * Delete action.
     */
    private final static int DELETE  = 0x8;
    /**
     * Read link action.
     */
    private final static int READLINK    = 0x10;

    /**
```

```
     * All actions (read,write,execute,delete,readlink)
     */
    private final static int ALL     = READ|WRITE|EXECUTE|DELETE|READLINK;
    /**
     * No actions.
     */
    private final static int NONE    = 0x0;

    // the actions mask
    private transient int mask;

    // does path indicate a directory? (wildcard or recursive)
    private transient boolean directory;

    // is it a recursive directory specification?
    private transient boolean recursive;

    /**
     * the actions string.
     *
     * @serial
     */
    private String actions; // Left null as long as possible, then
                            // created and re-used in the getAction function.

    // canonicalized dir path. In the case of
    // directories, it is the name "/blah/*" or "/blah/-" without
    // the last character (the "*" or "-").

    private transient String cpath;

    // static Strings used by init(int mask)
    private static final char RECURSIVE_CHAR = '-';
    private static final char WILD_CHAR = '*';

/*
    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        sb.append("***\n");
        sb.append("cpath = "+cpath+"\n");
        sb.append("mask = "+mask+"\n");
        sb.append("actions = "+getActions()+"\n");
        sb.append("directory = "+directory+"\n");
        sb.append("recursive = "+recursive+"\n");
        sb.append("***\n");
        return sb.toString();
    }
*/

    private static final long serialVersionUID = 7930732926638008763L;

    /**
     * initialize a FilePermission object. Common to all constructors.
     * Also called during de-serialization.
     *
     * @param mask the actions mask to use.
     *
     */
    private void init(int mask) {
        if ((mask & ALL) != mask)
                throw new IllegalArgumentException("invalid actions mask");
```

```
    if (mask == NONE)
            throw new IllegalArgumentException("invalid actions mask");

    if ((cpath = getName()) == null)
            throw new NullPointerException("name can't be null");

    this.mask = mask;

    if (cpath.equals("<<ALL FILES>>")) {
        directory = true;
        recursive = true;
        cpath = "";
        return;
    }

    // store only the canonical cpath if possible
    cpath = AccessController.doPrivileged(new PrivilegedAction<String>() {
        public String run() {
            try {
                String path = cpath;
                if (cpath.endsWith("*")) {
                    // call getCanonicalPath with a path with wildcard character
                    // replaced to avoid calling it with paths that are
                    // intended to match all entries in a directory
                    path = path.substring(0, path.length()-1) + "-";
                    path = new File(path).getCanonicalPath();
                    return path.substring(0, path.length()-1) + "*";
                } else {
                    return new File(path).getCanonicalPath();
                }
            } catch (IOException ioe) {
                return cpath;
            }
        }
    });

    int len = cpath.length();
    char last = ((len > 0) ? cpath.charAt(len - 1) : 0);

    if (last == RECURSIVE_CHAR &&
        cpath.charAt(len - 2) == File.separatorChar) {
        directory = true;
        recursive = true;
        cpath = cpath.substring(0, --len);
    } else if (last == WILD_CHAR &&
        cpath.charAt(len - 2) == File.separatorChar) {
        directory = true;
        //recursive = false;
        cpath = cpath.substring(0, --len);
    } else {
        // overkill since they are initialized to false, but
        // commented out here to remind us...
        //directory = false;
        //recursive = false;
    }

    // XXX: at this point the path should be absolute. die if it isn't?
}

/**
 * Creates a new FilePermission object with the specified actions.
 * <i>path</i> is the pathname of a file or directory, and <i>actions</i>
 * contains a comma-separated list of the desired actions granted on the
```

```
 * file or directory. Possible actions are
 * "read", "write", "execute", "delete", and "readlink".
 *
 * <p>A pathname that ends in "/*" (where "/" is
 * the file separator character, <code>File.separatorChar</code>)
 * indicates all the files and directories contained in that directory.
 * A pathname that ends with "/-" indicates (recursively) all files and
 * subdirectories contained in that directory. The special pathname
 * "<<ALL FILES>>" matches any file.
 *
 * <p>A pathname consisting of a single "*" indicates all the files
 * in the current directory, while a pathname consisting of a single "-"
 * indicates all the files in the current directory and
 * (recursively) all files and subdirectories contained in the current
 * directory.
 *
 * <p>A pathname containing an empty string represents an empty path.
 *
 * @param path the pathname of the file/directory.
 * @param actions the action string.
 *
 * @throws IllegalArgumentException
 *          If actions is <code>null</code>, empty or contains an action
 *          other than the specified possible actions.
 */
public FilePermission(String path, String actions) {
    super(path);
    init(getMask(actions));
}

/**
 * Creates a new FilePermission object using an action mask.
 * More efficient than the FilePermission(String, String) constructor.
 * Can be used from within
 * code that needs to create a FilePermission object to pass into the
 * <code>implies</code> method.
 *
 * @param path the pathname of the file/directory.
 * @param mask the action mask to use.
 */

// package private for use by the FilePermissionCollection add method
FilePermission(String path, int mask) {
    super(path);
    init(mask);
}

/**
 * Checks if this FilePermission object "implies" the specified permission.
 * <P>
 * More specifically, this method returns true if:
 * <ul>
 * <li> <i>p</i> is an instanceof FilePermission,
 * <li> <i>p</i>'s actions are a proper subset of this
 * object's actions, and
 * <li> <i>p</i>'s pathname is implied by this object's
 *      pathname. For example, "/tmp/*" implies "/tmp/foo", since
 *      "/tmp/*" encompasses all files in the "/tmp" directory,
 *      including the one named "foo".
 * </ul>
 *
 * @param p the permission to check against.
 *
```

```
     * @return <code>true</code> if the specified permission is not
     *                 <code>null</code> and is implied by this object,
     *                 <code>false</code> otherwise.
     */
    public boolean implies(Permission p) {
        if (!(p instanceof FilePermission))
            return false;

        FilePermission that = (FilePermission) p;

        // we get the effective mask. i.e., the "and" of this and that.
        // They must be equal to that.mask for implies to return true.

        return ((this.mask & that.mask) == that.mask) && impliesIgnoreMask(that);
    }

    /**
     * Checks if the Permission's actions are a proper subset of the
     * this object's actions. Returns the effective mask iff the
     * this FilePermission's path also implies that FilePermission's path.
     *
     * @param that the FilePermission to check against.
     * @return the effective mask
     */
    boolean impliesIgnoreMask(FilePermission that) {
        if (this.directory) {
            if (this.recursive) {
                // make sure that.path is longer then path so
                // something like /foo/- does not imply /foo
                if (that.directory) {
                    return (that.cpath.length() >= this.cpath.length()) &&
                            that.cpath.startsWith(this.cpath);
                }  else {
                    return ((that.cpath.length() > this.cpath.length()) &&
                        that.cpath.startsWith(this.cpath));
                }
            } else {
                if (that.directory) {
                    // if the permission passed in is a directory
                    // specification, make sure that a non-recursive
                    // permission (i.e., this object) can't imply a recursive
                    // permission.
                    if (that.recursive)
                        return false;
                    else
                        return (this.cpath.equals(that.cpath));
                } else {
                    int last = that.cpath.lastIndexOf(File.separatorChar);
                    if (last == -1)
                        return false;
                    else {
                        // this.cpath.equals(that.cpath.substring(0, last+1));
                        // Use regionMatches to avoid creating new string
                        return (this.cpath.length() == (last + 1)) &&
                            this.cpath.regionMatches(0, that.cpath, 0, last+1);
                    }
                }
            }
        } else if (that.directory) {
            // if this is NOT recursive/wildcarded,
            // do not let it imply a recursive/wildcarded permission
            return false;
        } else {
```

```
        return (this.cpath.equals(that.cpath));
    }
}

/**
 * Checks two FilePermission objects for equality. Checks that <i>obj</i> is
 * a FilePermission, and has the same pathname and actions as this object.
 *
 * @param obj the object we are testing for equality with this object.
 * @return <code>true</code> if obj is a FilePermission, and has the same
 *          pathname and actions as this FilePermission object,
 *          <code>false</code> otherwise.
 */
public boolean equals(Object obj) {
    if (obj == this)
        return true;

    if (! (obj instanceof FilePermission))
        return false;

    FilePermission that = (FilePermission) obj;

    return (this.mask == that.mask) &&
        this.cpath.equals(that.cpath) &&
        (this.directory == that.directory) &&
        (this.recursive == that.recursive);
}

/**
 * Returns the hash code value for this object.
 *
 * @return a hash code value for this object.
 */
public int hashCode() {
    return 0;
}

/**
 * Converts an actions String to an actions mask.
 *
 * @param actions the action string.
 * @return the actions mask.
 */
private static int getMask(String actions) {
    int mask = NONE;

    // Null action valid?
    if (actions == null) {
        return mask;
    }

    // Use object identity comparison against known-interned strings for
    // performance benefit (these values are used heavily within the JDK).
    if (actions == SecurityConstants.FILE_READ_ACTION) {
        return READ;
    } else if (actions == SecurityConstants.FILE_WRITE_ACTION) {
        return WRITE;
    } else if (actions == SecurityConstants.FILE_EXECUTE_ACTION) {
        return EXECUTE;
    } else if (actions == SecurityConstants.FILE_DELETE_ACTION) {
        return DELETE;
    } else if (actions == SecurityConstants.FILE_READLINK_ACTION) {
        return READLINK;
```

```
}

char[] a = actions.toCharArray();

int i = a.length - 1;
if (i < 0)
    return mask;

while (i != -1) {
    char c;

    // skip whitespace
    while ((i!=-1) && ((c = a[i]) == ' ' ||
                       c == '\r' ||
                       c == '\n' ||
                       c == '\f' ||
                       c == '\t'))
        i--;

    // check for the known strings
    int matchlen;

    if (i >= 3 && (a[i-3] == 'r' || a[i-3] == 'R') &&
                  (a[i-2] == 'e' || a[i-2] == 'E') &&
                  (a[i-1] == 'a' || a[i-1] == 'A') &&
                  (a[i] == 'd' || a[i] == 'D'))
    {
        matchlen = 4;
        mask |= READ;

    } else if (i >= 4 && (a[i-4] == 'w' || a[i-4] == 'W') &&
                         (a[i-3] == 'r' || a[i-3] == 'R') &&
                         (a[i-2] == 'i' || a[i-2] == 'I') &&
                         (a[i-1] == 't' || a[i-1] == 'T') &&
                         (a[i] == 'e' || a[i] == 'E'))
    {
        matchlen = 5;
        mask |= WRITE;

    } else if (i >= 6 && (a[i-6] == 'e' || a[i-6] == 'E') &&
                         (a[i-5] == 'x' || a[i-5] == 'X') &&
                         (a[i-4] == 'e' || a[i-4] == 'E') &&
                         (a[i-3] == 'c' || a[i-3] == 'C') &&
                         (a[i-2] == 'u' || a[i-2] == 'U') &&
                         (a[i-1] == 't' || a[i-1] == 'T') &&
                         (a[i] == 'e' || a[i] == 'E'))
    {
        matchlen = 7;
        mask |= EXECUTE;

    } else if (i >= 5 && (a[i-5] == 'd' || a[i-5] == 'D') &&
                         (a[i-4] == 'e' || a[i-4] == 'E') &&
                         (a[i-3] == 'l' || a[i-3] == 'L') &&
                         (a[i-2] == 'e' || a[i-2] == 'E') &&
                         (a[i-1] == 't' || a[i-1] == 'T') &&
                         (a[i] == 'e' || a[i] == 'E'))
    {
        matchlen = 6;
        mask |= DELETE;

    } else if (i >= 7 && (a[i-7] == 'r' || a[i-7] == 'R') &&
                         (a[i-6] == 'e' || a[i-6] == 'E') &&
                         (a[i-5] == 'a' || a[i-5] == 'A') &&
```

```
                            (a[i-4] == 'd' || a[i-4] == 'D') &&
                            (a[i-3] == 'l' || a[i-3] == 'L') &&
                            (a[i-2] == 'i' || a[i-2] == 'I') &&
                            (a[i-1] == 'n' || a[i-1] == 'N') &&
                            (a[i] == 'k' || a[i] == 'K'))
        {
            matchlen = 8;
            mask |= READLINK;

        } else {
            // parse error
            throw new IllegalArgumentException(
                    "invalid permission: " + actions);
        }

        // make sure we didn't just match the tail of a word
        // like "ackbarfaccept".  Also, skip to the comma.
        boolean seencomma = false;
        while (i >= matchlen && !seencomma) {
            switch(a[i-matchlen]) {
            case ',':
                seencomma = true;
                break;
            case ' ': case '\r': case '\n':
            case '\f': case '\t':
                break;
            default:
                throw new IllegalArgumentException(
                        "invalid permission: " + actions);
            }
            i--;
        }

        // point i at the location of the comma minus one (or -1).
        i -= matchlen;
    }

    return mask;
}

/**
 * Return the current action mask. Used by the FilePermissionCollection.
 *
 * @return the actions mask.
 */
int getMask() {
    return mask;
}

/**
 * Return the canonical string representation of the actions.
 * Always returns present actions in the following order:
 * read, write, execute, delete, readlink.
 *
 * @return the canonical string representation of the actions.
 */
private static String getActions(int mask) {
    StringBuilder sb = new StringBuilder();
    boolean comma = false;

    if ((mask & READ) == READ) {
        comma = true;
        sb.append("read");
```

```
    }

    if ((mask & WRITE) == WRITE) {
        if (comma) sb.append(',');
        else comma = true;
        sb.append("write");
    }

    if ((mask & EXECUTE) == EXECUTE) {
        if (comma) sb.append(',');
        else comma = true;
        sb.append("execute");
    }

    if ((mask & DELETE) == DELETE) {
        if (comma) sb.append(',');
        else comma = true;
        sb.append("delete");
    }

    if ((mask & READLINK) == READLINK) {
        if (comma) sb.append(',');
        else comma = true;
        sb.append("readlink");
    }

    return sb.toString();
}

/**
 * Returns the "canonical string representation" of the actions.
 * That is, this method always returns present actions in the following order:
 * read, write, execute, delete, readlink. For example, if this FilePermission
 * object allows both write and read actions, a call to <code>getActions</code>
 * will return the string "read,write".
 *
 * @return the canonical string representation of the actions.
 */
public String getActions() {
    if (actions == null)
        actions = getActions(this.mask);

    return actions;
}

/**
 * Returns a new PermissionCollection object for storing FilePermission
 * objects.
 * <p>
 * FilePermission objects must be stored in a manner that allows them
 * to be inserted into the collection in any order, but that also enables the
 * PermissionCollection <code>implies</code>
 * method to be implemented in an efficient (and consistent) manner.
 *
 * <p>For example, if you have two FilePermissions:
 * <OL>
 * <LI>  <code>"/tmp/-", "read"</code>
 * <LI>  <code>"/tmp/scratch/foo", "write"</code>
 * </OL>
 *
 * <p>and you are calling the <code>implies</code> method with the FilePermission:
 *
 * <pre>
```

```
     *   "/tmp/scratch/foo", "read,write",
     * </pre>
     *
     * then the <code>implies</code> function must
     * take into account both the "/tmp/-" and "/tmp/scratch/foo"
     * permissions, so the effective permission is "read,write",
     * and <code>implies</code> returns true. The "implies" semantics for
     * FilePermissions are handled properly by the PermissionCollection object
     * returned by this <code>newPermissionCollection</code> method.
     *
     * @return a new PermissionCollection object suitable for storing
     * FilePermissions.
     */
    public PermissionCollection newPermissionCollection() {
        return new FilePermissionCollection();
    }

    /**
     * WriteObject is called to save the state of the FilePermission
     * to a stream. The actions are serialized, and the superclass
     * takes care of the name.
     */
    private void writeObject(ObjectOutputStream s)
        throws IOException
    {
        // Write out the actions. The superclass takes care of the name
        // call getActions to make sure actions field is initialized
        if (actions == null)
            getActions();
        s.defaultWriteObject();
    }

    /**
     * readObject is called to restore the state of the FilePermission from
     * a stream.
     */
    private void readObject(ObjectInputStream s)
         throws IOException, ClassNotFoundException
    {
        // Read in the actions, then restore everything else by calling init.
        s.defaultReadObject();
        init(getMask(actions));
    }
}

/**
 * A FilePermissionCollection stores a set of FilePermission permissions.
 * FilePermission objects
 * must be stored in a manner that allows them to be inserted in any
 * order, but enable the implies function to evaluate the implies
 * method.
 * For example, if you have two FilePermissions:
 * <OL>
 * <LI> "/tmp/-", "read"
 * <LI> "/tmp/scratch/foo", "write"
 * </OL>
 * And you are calling the implies function with the FilePermission:
 * "/tmp/scratch/foo", "read,write", then the implies function must
 * take into account both the /tmp/- and /tmp/scratch/foo
 * permissions, so the effective permission is "read,write".
 *
 * @see java.security.Permission
 * @see java.security.Permissions
```

```
 * @see java.security.PermissionCollection
 *
 *
 * @author Marianne Mueller
 * @author Roland Schemers
 *
 * @serial include
 *
 */

final class FilePermissionCollection extends PermissionCollection
    implements Serializable
{
    // Not serialized; see serialization section at end of class
    private transient List<Permission> perms;

    /**
     * Create an empty FilePermissionCollection object.
     */
    public FilePermissionCollection() {
        perms = new ArrayList<>();
    }

    /**
     * Adds a permission to the FilePermissionCollection. The key for the hash is
     * permission.path.
     *
     * @param permission the Permission object to add.
     *
     * @exception IllegalArgumentException - if the permission is not a
     *                                       FilePermission
     *
     * @exception SecurityException - if this FilePermissionCollection object
     *                                has been marked readonly
     */
    public void add(Permission permission) {
        if (! (permission instanceof FilePermission))
            throw new IllegalArgumentException("invalid permission: "+
                                               permission);
        if (isReadOnly())
            throw new SecurityException(
                "attempt to add a Permission to a readonly PermissionCollection");

        synchronized (this) {
            perms.add(permission);
        }
    }

    /**
     * Check and see if this set of permissions implies the permissions
     * expressed in "permission".
     *
     * @param permission the Permission object to compare
     *
     * @return true if "permission" is a proper subset of a permission in
     * the set, false if not.
     */
    public boolean implies(Permission permission) {
        if (! (permission instanceof FilePermission))
            return false;

        FilePermission fp = (FilePermission) permission;
```

```
    int desired = fp.getMask();
    int effective = 0;
    int needed = desired;

    synchronized (this) {
        int len = perms.size();
        for (int i = 0; i < len; i++) {
            FilePermission x = (FilePermission) perms.get(i);
            if (((needed & x.getMask()) != 0) && x.impliesIgnoreMask(fp)) {
                effective |=  x.getMask();
                if ((effective & desired) == desired)
                    return true;
                needed = (desired ^ effective);
            }
        }
    }
    return false;
}

/**
 * Returns an enumeration of all the FilePermission objects in the
 * container.
 *
 * @return an enumeration of all the FilePermission objects.
 */
public Enumeration<Permission> elements() {
    // Convert Iterator into Enumeration
    synchronized (this) {
        return Collections.enumeration(perms);
    }
}

private static final long serialVersionUID = 2202956749081564585L;

// Need to maintain serialization interoperability with earlier releases,
// which had the serializable field:
//    private Vector permissions;

/**
 * @serialField permissions java.util.Vector
 *     A list of FilePermission objects.
 */
private static final ObjectStreamField[] serialPersistentFields = {
    new ObjectStreamField("permissions", Vector.class),
};

/**
 * @serialData "permissions" field (a Vector containing the FilePermissions).
 */
/*
 * Writes the contents of the perms field out as a Vector for
 * serialization compatibility with earlier releases.
 */
private void writeObject(ObjectOutputStream out) throws IOException {
    // Don't call out.defaultWriteObject()

    // Write out Vector
    Vector<Permission> permissions = new Vector<>(perms.size());
    synchronized (this) {
        permissions.addAll(perms);
    }

    ObjectOutputStream.PutField pfields = out.putFields();
```

```
        pfields.put("permissions", permissions);
        out.writeFields();
    }

    /*
     * Reads in a Vector of FilePermissions and saves them in the perms field.
     */
    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException
    {
        // Don't call defaultReadObject()

        // Read in serialized fields
        ObjectInputStream.GetField gfields = in.readFields();

        // Get the one we want
        @SuppressWarnings("unchecked")
        Vector<Permission> permissions = (Vector<Permission>)gfields.get("permissions", null);
        perms = new ArrayList<>(permissions.size());
        perms.addAll(permissions);
    }
}
```

FileReader.java

```
/*
 * Copyright (c) 1996, 2001, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Convenience class for reading character files.  The constructors of this
 * class assume that the default character encoding and the default byte-buffer
 * size are appropriate.  To specify these values yourself, construct an
 * InputStreamReader on a FileInputStream.
 *
 * <p><code>FileReader</code> is meant for reading streams of characters.
 * For reading streams of raw bytes, consider using a
 * <code>FileInputStream</code>.
 *
 * @see InputStreamReader
 * @see FileInputStream
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */
public class FileReader extends InputStreamReader {

    /**
     * Creates a new <tt>FileReader</tt>, given the name of the
     * file to read from.
     *
     * @param fileName the name of the file to read from
     * @exception  FileNotFoundException  if the named file does not exist,
     *                   is a directory rather than a regular file,
     *                   or for some other reason cannot be opened for
     *                   reading.
     */
    public FileReader(String fileName) throws FileNotFoundException {
        super(new FileInputStream(fileName));
    }
```

```
    /**
     * Creates a new <tt>FileReader</tt>, given the <tt>File</tt>
     * to read from.
     *
     * @param file the <tt>File</tt> to read from
     * @exception  FileNotFoundException  if the file does not exist,
     *                   is a directory rather than a regular file,
     *                   or for some other reason cannot be opened for
     *                   reading.
     */
     public FileReader(File file) throws FileNotFoundException {
         super(new FileInputStream(file));
     }

    /**
     * Creates a new <tt>FileReader</tt>, given the
     * <tt>FileDescriptor</tt> to read from.
     *
     * @param fd the FileDescriptor to read from
     */
     public FileReader(FileDescriptor fd) {
         super(new FileInputStream(fd));
     }

}
```

FileSystem.java

```
/*
 * Copyright (c) 1998, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.lang.annotation.Native;

/**
 * Package-private abstract class for the local filesystem abstraction.
 */

abstract class FileSystem {

    /* -- Normalization and construction -- */

    /**
     * Return the local filesystem's name-separator character.
     */
    public abstract char getSeparator();

    /**
     * Return the local filesystem's path-separator character.
     */
    public abstract char getPathSeparator();

    /**
     * Convert the given pathname string to normal form.  If the string is
     * already in normal form then it is simply returned.
     */
    public abstract String normalize(String path);

    /**
     * Compute the length of this pathname string's prefix.  The pathname
     * string must be in normal form.
     */
    public abstract int prefixLength(String path);

    /**
```

```
     * Resolve the child pathname string against the parent.
     * Both strings must be in normal form, and the result
     * will be in normal form.
     */
    public abstract String resolve(String parent, String child);

    /**
     * Return the parent pathname string to be used when the parent-directory
     * argument in one of the two-argument File constructors is the empty
     * pathname.
     */
    public abstract String getDefaultParent();

    /**
     * Post-process the given URI path string if necessary.  This is used on
     * win32, e.g., to transform "/c:/foo" into "c:/foo".  The path string
     * still has slash separators; code in the File class will translate them
     * after this method returns.
     */
    public abstract String fromURIPath(String path);


    /* -- Path operations -- */

    /**
     * Tell whether or not the given abstract pathname is absolute.
     */
    public abstract boolean isAbsolute(File f);

    /**
     * Resolve the given abstract pathname into absolute form.  Invoked by the
     * getAbsolutePath and getCanonicalPath methods in the File class.
     */
    public abstract String resolve(File f);

    public abstract String canonicalize(String path) throws IOException;


    /* -- Attribute accessors -- */

    /* Constants for simple boolean attributes */
    @Native public static final int BA_EXISTS    = 0x01;
    @Native public static final int BA_REGULAR   = 0x02;
    @Native public static final int BA_DIRECTORY = 0x04;
    @Native public static final int BA_HIDDEN    = 0x08;

    /**
     * Return the simple boolean attributes for the file or directory denoted
     * by the given abstract pathname, or zero if it does not exist or some
     * other I/O error occurs.
     */
    public abstract int getBooleanAttributes(File f);

    @Native public static final int ACCESS_READ    = 0x04;
    @Native public static final int ACCESS_WRITE   = 0x02;
    @Native public static final int ACCESS_EXECUTE = 0x01;

    /**
     * Check whether the file or directory denoted by the given abstract
     * pathname may be accessed by this process.  The second argument specifies
     * which access, ACCESS_READ, ACCESS_WRITE or ACCESS_EXECUTE, to check.
     * Return false if access is denied or an I/O error occurs
     */
```

```
public abstract boolean checkAccess(File f, int access);
/**
 * Set on or off the access permission (to owner only or to all) to the file
 * or directory denoted by the given abstract pathname, based on the parameters
 * enable, access and oweronly.
 */
public abstract boolean setPermission(File f, int access, boolean enable, boolean owneronly);

/**
 * Return the time at which the file or directory denoted by the given
 * abstract pathname was last modified, or zero if it does not exist or
 * some other I/O error occurs.
 */
public abstract long getLastModifiedTime(File f);

/**
 * Return the length in bytes of the file denoted by the given abstract
 * pathname, or zero if it does not exist, is a directory, or some other
 * I/O error occurs.
 */
public abstract long getLength(File f);


/* -- File operations -- */

/**
 * Create a new empty file with the given pathname.  Return
 * <code>true</code> if the file was created and <code>false</code> if a
 * file or directory with the given pathname already exists.  Throw an
 * IOException if an I/O error occurs.
 */
public abstract boolean createFileExclusively(String pathname)
    throws IOException;

/**
 * Delete the file or directory denoted by the given abstract pathname,
 * returning <code>true</code> if and only if the operation succeeds.
 */
public abstract boolean delete(File f);

/**
 * List the elements of the directory denoted by the given abstract
 * pathname.  Return an array of strings naming the elements of the
 * directory if successful; otherwise, return <code>null</code>.
 */
public abstract String[] list(File f);

/**
 * Create a new directory denoted by the given abstract pathname,
 * returning <code>true</code> if and only if the operation succeeds.
 */
public abstract boolean createDirectory(File f);

/**
 * Rename the file or directory denoted by the first abstract pathname to
 * the second abstract pathname, returning <code>true</code> if and only if
 * the operation succeeds.
 */
public abstract boolean rename(File f1, File f2);

/**
 * Set the last-modified time of the file or directory denoted by the
 * given abstract pathname, returning <code>true</code> if and only if the
```

```
     * operation succeeds.
     */
    public abstract boolean setLastModifiedTime(File f, long time);

    /**
     * Mark the file or directory denoted by the given abstract pathname as
     * read-only, returning <code>true</code> if and only if the operation
     * succeeds.
     */
    public abstract boolean setReadOnly(File f);


    /* -- Filesystem interface -- */

    /**
     * List the available filesystem roots.
     */
    public abstract File[] listRoots();

    /* -- Disk usage -- */
    @Native public static final int SPACE_TOTAL  = 0;
    @Native public static final int SPACE_FREE   = 1;
    @Native public static final int SPACE_USABLE = 2;

    public abstract long getSpace(File f, int t);

    /* -- Basic infrastructure -- */

    /**
     * Compare two abstract pathnames lexicographically.
     */
    public abstract int compare(File f1, File f2);

    /**
     * Compute the hash code of an abstract pathname.
     */
    public abstract int hashCode(File f);

    // Flags for enabling/disabling performance optimizations for file
    // name canonicalization
    static boolean useCanonCaches      = true;
    static boolean useCanonPrefixCache = true;

    private static boolean getBooleanProperty(String prop, boolean defaultVal) {
        String val = System.getProperty(prop);
        if (val == null) return defaultVal;
        if (val.equalsIgnoreCase("true")) {
            return true;
        } else {
            return false;
        }
    }

    static {
        useCanonCaches      = getBooleanProperty("sun.io.useCanonCaches",
                                                 useCanonCaches);
        useCanonPrefixCache = getBooleanProperty("sun.io.useCanonPrefixCache",
                                                 useCanonPrefixCache);
    }
}
```

FileWriter.java

```
/*
 * Copyright (c) 1996, 2001, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Convenience class for writing character files.  The constructors of this
 * class assume that the default character encoding and the default byte-buffer
 * size are acceptable.  To specify these values yourself, construct an
 * OutputStreamWriter on a FileOutputStream.
 *
 * <p>Whether or not a file is available or may be created depends upon the
 * underlying platform.  Some platforms, in particular, allow a file to be
 * opened for writing by only one <tt>FileWriter</tt> (or other file-writing
 * object) at a time.  In such situations the constructors in this class
 * will fail if the file involved is already open.
 *
 * <p><code>FileWriter</code> is meant for writing streams of characters.
 * For writing streams of raw bytes, consider using a
 * <code>FileOutputStream</code>.
 *
 * @see OutputStreamWriter
 * @see FileOutputStream
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class FileWriter extends OutputStreamWriter {

    /**
     * Constructs a FileWriter object given a file name.
     *
     * @param fileName  String The system-dependent filename.
     * @throws IOException  if the named file exists but is a directory rather
     *                  than a regular file, does not exist but cannot be
     *                  created, or cannot be opened for any other reason
```

```
     */
    public FileWriter(String fileName) throws IOException {
        super(new FileOutputStream(fileName));
    }

    /**
     * Constructs a FileWriter object given a file name with a boolean
     * indicating whether or not to append the data written.
     *
     * @param fileName  String The system-dependent filename.
     * @param append    boolean if <code>true</code>, then data will be written
     *                  to the end of the file rather than the beginning.
     * @throws IOException  if the named file exists but is a directory rather
     *                  than a regular file, does not exist but cannot be
     *                  created, or cannot be opened for any other reason
     */
    public FileWriter(String fileName, boolean append) throws IOException {
        super(new FileOutputStream(fileName, append));
    }

    /**
     * Constructs a FileWriter object given a File object.
     *
     * @param file  a File object to write to.
     * @throws IOException  if the file exists but is a directory rather than
     *                  a regular file, does not exist but cannot be created,
     *                  or cannot be opened for any other reason
     */
    public FileWriter(File file) throws IOException {
        super(new FileOutputStream(file));
    }

    /**
     * Constructs a FileWriter object given a File object. If the second
     * argument is <code>true</code>, then bytes will be written to the end
     * of the file rather than the beginning.
     *
     * @param file  a File object to write to
     * @param     append    if <code>true</code>, then bytes will be written
     *                      to the end of the file rather than the beginning
     * @throws IOException  if the file exists but is a directory rather than
     *                  a regular file, does not exist but cannot be created,
     *                  or cannot be opened for any other reason
     * @since 1.4
     */
    public FileWriter(File file, boolean append) throws IOException {
        super(new FileOutputStream(file, append));
    }

    /**
     * Constructs a FileWriter object associated with a file descriptor.
     *
     * @param fd  FileDescriptor object to write to.
     */
    public FileWriter(FileDescriptor fd) {
        super(new FileOutputStream(fd));
    }

}
```

FilterInputStream.java

```
/*
 * Copyright (c) 1994, 2010, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * A <code>FilterInputStream</code> contains
 * some other input stream, which it uses as
 * its  basic source of data, possibly transforming
 * the data along the way or providing  additional
 * functionality. The class <code>FilterInputStream</code>
 * itself simply overrides all  methods of
 * <code>InputStream</code> with versions that
 * pass all requests to the contained  input
 * stream. Subclasses of <code>FilterInputStream</code>
 * may further override some of  these methods
 * and may also provide additional methods
 * and fields.
 *
 * @author  Jonathan Payne
 * @since   JDK1.0
 */
public
class FilterInputStream extends InputStream {
    /**
     * The input stream to be filtered.
     */
    protected volatile InputStream in;

    /**
     * Creates a <code>FilterInputStream</code>
     * by assigning the  argument <code>in</code>
     * to the field <code>this.in</code> so as
     * to remember it for later use.
     *
     * @param   in   the underlying input stream, or <code>null</code> if
     *          this instance is to be created without an underlying stream.
     */
```

```
protected FilterInputStream(InputStream in) {
    this.in = in;
}

/**
 * Reads the next byte of data from this input stream. The value
 * byte is returned as an <code>int</code> in the range
 * <code>0</code> to <code>255</code>. If no byte is available
 * because the end of the stream has been reached, the value
 * <code>-1</code> is returned. This method blocks until input data
 * is available, the end of the stream is detected, or an exception
 * is thrown.
 * <p>
 * This method
 * simply performs <code>in.read()</code> and returns the result.
 *
 * @return     the next byte of data, or <code>-1</code> if the end of the
 *             stream is reached.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterInputStream#in
 */
public int read() throws IOException {
    return in.read();
}

/**
 * Reads up to <code>byte.length</code> bytes of data from this
 * input stream into an array of bytes. This method blocks until some
 * input is available.
 * <p>
 * This method simply performs the call
 * <code>read(b, 0, b.length)</code> and returns
 * the  result. It is important that it does
 * <i>not</i> do <code>in.read(b)</code> instead;
 * certain subclasses of  <code>FilterInputStream</code>
 * depend on the implementation strategy actually
 * used.
 *
 * @param      b   the buffer into which the data is read.
 * @return     the total number of bytes read into the buffer, or
 *             <code>-1</code> if there is no more data because the end of
 *             the stream has been reached.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterInputStream#read(byte[], int, int)
 */
public int read(byte b[]) throws IOException {
    return read(b, 0, b.length);
}

/**
 * Reads up to <code>len</code> bytes of data from this input stream
 * into an array of bytes. If <code>len</code> is not zero, the method
 * blocks until some input is available; otherwise, no
 * bytes are read and <code>0</code> is returned.
 * <p>
 * This method simply performs <code>in.read(b, off, len)</code>
 * and returns the result.
 *
 * @param      b     the buffer into which the data is read.
 * @param      off   the start offset in the destination array <code>b</code>
 * @param      len   the maximum number of bytes read.
 * @return     the total number of bytes read into the buffer, or
 *             <code>-1</code> if there is no more data because the end of
```

```
     *             the stream has been reached.
     * @exception  NullPointerException If <code>b</code> is <code>null</code>.
     * @exception  IndexOutOfBoundsException If <code>off</code> is negative,
     * <code>len</code> is negative, or <code>len</code> is greater than
     * <code>b.length - off</code>
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterInputStream#in
     */
    public int read(byte b[], int off, int len) throws IOException {
        return in.read(b, off, len);
    }

    /**
     * Skips over and discards <code>n</code> bytes of data from the
     * input stream. The <code>skip</code> method may, for a variety of
     * reasons, end up skipping over some smaller number of bytes,
     * possibly <code>0</code>. The actual number of bytes skipped is
     * returned.
     * <p>
     * This method simply performs <code>in.skip(n)</code>.
     *
     * @param      n   the number of bytes to be skipped.
     * @return     the actual number of bytes skipped.
     * @exception  IOException  if the stream does not support seek,
     *                          or if some other I/O error occurs.
     */
    public long skip(long n) throws IOException {
        return in.skip(n);
    }

    /**
     * Returns an estimate of the number of bytes that can be read (or
     * skipped over) from this input stream without blocking by the next
     * caller of a method for this input stream. The next caller might be
     * the same thread or another thread.  A single read or skip of this
     * many bytes will not block, but may read or skip fewer bytes.
     * <p>
     * This method returns the result of {@link #in in}.available().
     *
     * @return     an estimate of the number of bytes that can be read (or skipped
     *             over) from this input stream without blocking.
     * @exception  IOException  if an I/O error occurs.
     */
    public int available() throws IOException {
        return in.available();
    }

    /**
     * Closes this input stream and releases any system resources
     * associated with the stream.
     * This
     * method simply performs <code>in.close()</code>.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterInputStream#in
     */
    public void close() throws IOException {
        in.close();
    }

    /**
     * Marks the current position in this input stream. A subsequent
     * call to the <code>reset</code> method repositions this stream at
```

```
     * the last marked position so that subsequent reads re-read the same bytes.
     * <p>
     * The <code>readlimit</code> argument tells this input stream to
     * allow that many bytes to be read before the mark position gets
     * invalidated.
     * <p>
     * This method simply performs <code>in.mark(readlimit)</code>.
     *
     * @param   readlimit   the maximum limit of bytes that can be read before
     *                      the mark position becomes invalid.
     * @see     java.io.FilterInputStream#in
     * @see     java.io.FilterInputStream#reset()
     */
    public synchronized void mark(int readlimit) {
        in.mark(readlimit);
    }

    /**
     * Repositions this stream to the position at the time the
     * <code>mark</code> method was last called on this input stream.
     * <p>
     * This method
     * simply performs <code>in.reset()</code>.
     * <p>
     * Stream marks are intended to be used in
     * situations where you need to read ahead a little to see what's in
     * the stream. Often this is most easily done by invoking some
     * general parser. If the stream is of the type handled by the
     * parse, it just chugs along happily. If the stream is not of
     * that type, the parser should toss an exception when it fails.
     * If this happens within readlimit bytes, it allows the outer
     * code to reset the stream and try another parser.
     *
     * @exception  IOException  if the stream has not been marked or if the
     *               mark has been invalidated.
     * @see        java.io.FilterInputStream#in
     * @see        java.io.FilterInputStream#mark(int)
     */
    public synchronized void reset() throws IOException {
        in.reset();
    }

    /**
     * Tests if this input stream supports the <code>mark</code>
     * and <code>reset</code> methods.
     * This method
     * simply performs <code>in.markSupported()</code>.
     *
     * @return  <code>true</code> if this stream type supports the
     *          <code>mark</code> and <code>reset</code> method;
     *          <code>false</code> otherwise.
     * @see     java.io.FilterInputStream#in
     * @see     java.io.InputStream#mark(int)
     * @see     java.io.InputStream#reset()
     */
    public boolean markSupported() {
        return in.markSupported();
    }
}
```

FilterOutputStream.java

```java
/*
 * Copyright (c) 1994, 2011, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * This class is the superclass of all classes that filter output
 * streams. These streams sit on top of an already existing output
 * stream (the <i>underlying</i> output stream) which it uses as its
 * basic sink of data, but possibly transforming the data along the
 * way or providing additional functionality.
 * <p>
 * The class <code>FilterOutputStream</code> itself simply overrides
 * all methods of <code>OutputStream</code> with versions that pass
 * all requests to the underlying output stream. Subclasses of
 * <code>FilterOutputStream</code> may further override some of these
 * methods as well as provide additional methods and fields.
 *
 * @author  Jonathan Payne
 * @since   JDK1.0
 */
public
class FilterOutputStream extends OutputStream {
    /**
     * The underlying output stream to be filtered.
     */
    protected OutputStream out;

    /**
     * Creates an output stream filter built on top of the specified
     * underlying output stream.
     *
     * @param   out   the underlying output stream to be assigned to
     *                the field <tt>this.out</tt> for later use, or
     *                <code>null</code> if this instance is to be
     *                created without an underlying stream.
     */
    public FilterOutputStream(OutputStream out) {
```

```
    this.out = out;
}

/**
 * Writes the specified <code>byte</code> to this output stream.
 * <p>
 * The <code>write</code> method of <code>FilterOutputStream</code>
 * calls the <code>write</code> method of its underlying output stream,
 * that is, it performs <tt>out.write(b)</tt>.
 * <p>
 * Implements the abstract <tt>write</tt> method of <tt>OutputStream</tt>.
 *
 * @param      b   the <code>byte</code>.
 * @exception  IOException  if an I/O error occurs.
 */
public void write(int b) throws IOException {
    out.write(b);
}

/**
 * Writes <code>b.length</code> bytes to this output stream.
 * <p>
 * The <code>write</code> method of <code>FilterOutputStream</code>
 * calls its <code>write</code> method of three arguments with the
 * arguments <code>b</code>, <code>0</code>, and
 * <code>b.length</code>.
 * <p>
 * Note that this method does not call the one-argument
 * <code>write</code> method of its underlying stream with the single
 * argument <code>b</code>.
 *
 * @param      b   the data to be written.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#write(byte[], int, int)
 */
public void write(byte b[]) throws IOException {
    write(b, 0, b.length);
}

/**
 * Writes <code>len</code> bytes from the specified
 * <code>byte</code> array starting at offset <code>off</code> to
 * this output stream.
 * <p>
 * The <code>write</code> method of <code>FilterOutputStream</code>
 * calls the <code>write</code> method of one argument on each
 * <code>byte</code> to output.
 * <p>
 * Note that this method does not call the <code>write</code> method
 * of its underlying input stream with the same arguments. Subclasses
 * of <code>FilterOutputStream</code> should provide a more efficient
 * implementation of this method.
 *
 * @param      b     the data.
 * @param      off   the start offset in the data.
 * @param      len   the number of bytes to write.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterOutputStream#write(int)
 */
public void write(byte b[], int off, int len) throws IOException {
    if ((off | len | (b.length - (len + off)) | (off + len)) < 0)
        throw new IndexOutOfBoundsException();
```

```
        for (int i = 0 ; i < len ; i++) {
            write(b[off + i]);
        }
    }

    /**
     * Flushes this output stream and forces any buffered output bytes
     * to be written out to the stream.
     * <p>
     * The <code>flush</code> method of <code>FilterOutputStream</code>
     * calls the <code>flush</code> method of its underlying output stream.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#out
     */
    public void flush() throws IOException {
        out.flush();
    }

    /**
     * Closes this output stream and releases any system resources
     * associated with the stream.
     * <p>
     * The <code>close</code> method of <code>FilterOutputStream</code>
     * calls its <code>flush</code> method, and then calls the
     * <code>close</code> method of its underlying output stream.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#flush()
     * @see        java.io.FilterOutputStream#out
     */
    @SuppressWarnings("try")
    public void close() throws IOException {
        try (OutputStream ostream = out) {
            flush();
        }
    }
}
```

FilterReader.java

```
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Abstract class for reading filtered character streams.
 * The abstract class <code>FilterReader</code> itself
 * provides default methods that pass all requests to
 * the contained stream. Subclasses of <code>FilterReader</code>
 * should override some of these methods and may also provide
 * additional methods and fields.
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public abstract class FilterReader extends Reader {

    /**
     * The underlying character-input stream.
     */
    protected Reader in;

    /**
     * Creates a new filtered reader.
     *
     * @param in  a Reader object providing the underlying stream.
     * @throws NullPointerException if <code>in</code> is <code>null</code>
     */
    protected FilterReader(Reader in) {
        super(in);
        this.in = in;
    }

    /**
     * Reads a single character.
```

```
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read() throws IOException {
        return in.read();
    }

    /**
     * Reads characters into a portion of an array.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read(char cbuf[], int off, int len) throws IOException {
        return in.read(cbuf, off, len);
    }

    /**
     * Skips characters.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public long skip(long n) throws IOException {
        return in.skip(n);
    }

    /**
     * Tells whether this stream is ready to be read.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public boolean ready() throws IOException {
        return in.ready();
    }

    /**
     * Tells whether this stream supports the mark() operation.
     */
    public boolean markSupported() {
        return in.markSupported();
    }

    /**
     * Marks the present position in the stream.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void mark(int readAheadLimit) throws IOException {
        in.mark(readAheadLimit);
    }

    /**
     * Resets the stream.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void reset() throws IOException {
        in.reset();
    }

    public void close() throws IOException {
        in.close();
    }
```

```
}
```

FilterWriter.java

```java
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Abstract class for writing filtered character streams.
 * The abstract class <code>FilterWriter</code> itself
 * provides default methods that pass all requests to the
 * contained stream. Subclasses of <code>FilterWriter</code>
 * should override some of these methods and may also
 * provide additional methods and fields.
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public abstract class FilterWriter extends Writer {

    /**
     * The underlying character-output stream.
     */
    protected Writer out;

    /**
     * Create a new filtered writer.
     *
     * @param out  a Writer object to provide the underlying stream.
     * @throws NullPointerException if <code>out</code> is <code>null</code>
     */
    protected FilterWriter(Writer out) {
        super(out);
        this.out = out;
    }

    /**
     * Writes a single character.
```

```
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void write(int c) throws IOException {
        out.write(c);
    }

    /**
     * Writes a portion of an array of characters.
     *
     * @param  cbuf  Buffer of characters to be written
     * @param  off   Offset from which to start reading characters
     * @param  len   Number of characters to be written
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void write(char cbuf[], int off, int len) throws IOException {
        out.write(cbuf, off, len);
    }

    /**
     * Writes a portion of a string.
     *
     * @param  str  String to be written
     * @param  off  Offset from which to start reading characters
     * @param  len  Number of characters to be written
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void write(String str, int off, int len) throws IOException {
        out.write(str, off, len);
    }

    /**
     * Flushes the stream.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void flush() throws IOException {
        out.flush();
    }

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

}
```

Flushable.java

```
/*
 * Copyright (c) 2004, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.io.IOException;

/**
 * A <tt>Flushable</tt> is a destination of data that can be flushed.  The
 * flush method is invoked to write any buffered output to the underlying
 * stream.
 *
 * @since 1.5
 */
public interface Flushable {

    /**
     * Flushes this stream by writing any buffered output to the underlying
     * stream.
     *
     * @throws IOException If an I/O error occurs
     */
    void flush() throws IOException;
}
```

InputStream.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * This abstract class is the superclass of all classes representing
 * an input stream of bytes.
 *
 * <p> Applications that need to define a subclass of <code>InputStream</code>
 * must always provide a method that returns the next byte of input.
 *
 * @author  Arthur van Hoff
 * @see     java.io.BufferedInputStream
 * @see     java.io.ByteArrayInputStream
 * @see     java.io.DataInputStream
 * @see     java.io.FilterInputStream
 * @see     java.io.InputStream#read()
 * @see     java.io.OutputStream
 * @see     java.io.PushbackInputStream
 * @since   JDK1.0
 */
public abstract class InputStream implements Closeable {

    // MAX_SKIP_BUFFER_SIZE is used to determine the maximum buffer size to
    // use when skipping.
    private static final int MAX_SKIP_BUFFER_SIZE = 2048;

    /**
     * Reads the next byte of data from the input stream. The value byte is
     * returned as an <code>int</code> in the range <code>0</code> to
     * <code>255</code>. If no byte is available because the end of the stream
     * has been reached, the value <code>-1</code> is returned. This method
     * blocks until input data is available, the end of the stream is detected,
     * or an exception is thrown.
     *
     * <p> A subclass must provide an implementation of this method.
     *
```

```
     * @return     the next byte of data, or <code>-1</code> if the end of the
     *             stream is reached.
     * @exception  IOException  if an I/O error occurs.
     */
    public abstract int read() throws IOException;

    /**
     * Reads some number of bytes from the input stream and stores them into
     * the buffer array <code>b</code>. The number of bytes actually read is
     * returned as an integer.  This method blocks until input data is
     * available, end of file is detected, or an exception is thrown.
     *
     * <p> If the length of <code>b</code> is zero, then no bytes are read and
     * <code>0</code> is returned; otherwise, there is an attempt to read at
     * least one byte. If no byte is available because the stream is at the
     * end of the file, the value <code>-1</code> is returned; otherwise, at
     * least one byte is read and stored into <code>b</code>.
     *
     * <p> The first byte read is stored into element <code>b[0]</code>, the
     * next one into <code>b[1]</code>, and so on. The number of bytes read is,
     * at most, equal to the length of <code>b</code>. Let <i>k</i> be the
     * number of bytes actually read; these bytes will be stored in elements
     * <code>b[0]</code> through <code>b[</code><i>k</i><code>-1]</code>,
     * leaving elements <code>b[</code><i>k</i><code>]</code> through
     * <code>b[b.length-1]</code> unaffected.
     *
     * <p> The <code>read(b)</code> method for class <code>InputStream</code>
     * has the same effect as: <pre><code> read(b, 0, b.length) </code></pre>
     *
     * @param      b   the buffer into which the data is read.
     * @return     the total number of bytes read into the buffer, or
     *             <code>-1</code> if there is no more data because the end of
     *             the stream has been reached.
     * @exception  IOException  If the first byte cannot be read for any reason
     * other than the end of the file, if the input stream has been closed, or
     * if some other I/O error occurs.
     * @exception  NullPointerException  if <code>b</code> is <code>null</code>.
     * @see        java.io.InputStream#read(byte[], int, int)
     */
    public int read(byte b[]) throws IOException {
        return read(b, 0, b.length);
    }

    /**
     * Reads up to <code>len</code> bytes of data from the input stream into
     * an array of bytes.  An attempt is made to read as many as
     * <code>len</code> bytes, but a smaller number may be read.
     * The number of bytes actually read is returned as an integer.
     *
     * <p> This method blocks until input data is available, end of file is
     * detected, or an exception is thrown.
     *
     * <p> If <code>len</code> is zero, then no bytes are read and
     * <code>0</code> is returned; otherwise, there is an attempt to read at
     * least one byte. If no byte is available because the stream is at end of
     * file, the value <code>-1</code> is returned; otherwise, at least one
     * byte is read and stored into <code>b</code>.
     *
     * <p> The first byte read is stored into element <code>b[off]</code>, the
     * next one into <code>b[off+1]</code>, and so on. The number of bytes read
     * is, at most, equal to <code>len</code>. Let <i>k</i> be the number of
     * bytes actually read; these bytes will be stored in elements
     * <code>b[off]</code> through <code>b[off+</code><i>k</i><code>-1]</code>,
```

```
     * leaving elements <code>b[off+</code><i>k</i><code>]</code> through
     * <code>b[off+len-1]</code> unaffected.
     *
     * <p> In every case, elements <code>b[0]</code> through
     * <code>b[off]</code> and elements <code>b[off+len]</code> through
     * <code>b[b.length-1]</code> are unaffected.
     *
     * <p> The <code>read(b,</code> <code>off,</code> <code>len)</code> method
     * for class <code>InputStream</code> simply calls the method
     * <code>read()</code> repeatedly. If the first such call results in an
     * <code>IOException</code>, that exception is returned from the call to
     * the <code>read(b,</code> <code>off,</code> <code>len)</code> method.  If
     * any subsequent call to <code>read()</code> results in a
     * <code>IOException</code>, the exception is caught and treated as if it
     * were end of file; the bytes read up to that point are stored into
     * <code>b</code> and the number of bytes read before the exception
     * occurred is returned. The default implementation of this method blocks
     * until the requested amount of input data <code>len</code> has been read,
     * end of file is detected, or an exception is thrown. Subclasses are encouraged
     * to provide a more efficient implementation of this method.
     *
     * @param      b     the buffer into which the data is read.
     * @param      off   the start offset in array <code>b</code>
     *                   at which the data is written.
     * @param      len   the maximum number of bytes to read.
     * @return     the total number of bytes read into the buffer, or
     *             <code>-1</code> if there is no more data because the end of
     *             the stream has been reached.
     * @exception  IOException If the first byte cannot be read for any reason
     * other than end of file, or if the input stream has been closed, or if
     * some other I/O error occurs.
     * @exception  NullPointerException If <code>b</code> is <code>null</code>.
     * @exception  IndexOutOfBoundsException If <code>off</code> is negative,
     * <code>len</code> is negative, or <code>len</code> is greater than
     * <code>b.length - off</code>
     * @see        java.io.InputStream#read()
     */
    public int read(byte b[], int off, int len) throws IOException {
        if (b == null) {
            throw new NullPointerException();
        } else if (off < 0 || len < 0 || len > b.length - off) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return 0;
        }

        int c = read();
        if (c == -1) {
            return -1;
        }
        b[off] = (byte)c;

        int i = 1;
        try {
            for (; i < len ; i++) {
                c = read();
                if (c == -1) {
                    break;
                }
                b[off + i] = (byte)c;
            }
        } catch (IOException ee) {
        }
```

```
    return i;
}

/**
 * Skips over and discards <code>n</code> bytes of data from this input
 * stream. The <code>skip</code> method may, for a variety of reasons, end
 * up skipping over some smaller number of bytes, possibly <code>0</code>.
 * This may result from any of a number of conditions; reaching end of file
 * before <code>n</code> bytes have been skipped is only one possibility.
 * The actual number of bytes skipped is returned. If {@code n} is
 * negative, the {@code skip} method for class {@code InputStream} always
 * returns 0, and no bytes are skipped. Subclasses may handle the negative
 * value differently.
 *
 * <p> The <code>skip</code> method of this class creates a
 * byte array and then repeatedly reads into it until <code>n</code> bytes
 * have been read or the end of the stream has been reached. Subclasses are
 * encouraged to provide a more efficient implementation of this method.
 * For instance, the implementation may depend on the ability to seek.
 *
 * @param      n   the number of bytes to be skipped.
 * @return     the actual number of bytes skipped.
 * @exception  IOException  if the stream does not support seek,
 *                          or if some other I/O error occurs.
 */
public long skip(long n) throws IOException {

    long remaining = n;
    int nr;

    if (n <= 0) {
        return 0;
    }

    int size = (int)Math.min(MAX_SKIP_BUFFER_SIZE, remaining);
    byte[] skipBuffer = new byte[size];
    while (remaining > 0) {
        nr = read(skipBuffer, 0, (int)Math.min(size, remaining));
        if (nr < 0) {
            break;
        }
        remaining -= nr;
    }

    return n - remaining;
}

/**
 * Returns an estimate of the number of bytes that can be read (or
 * skipped over) from this input stream without blocking by the next
 * invocation of a method for this input stream. The next invocation
 * might be the same thread or another thread.  A single read or skip of this
 * many bytes will not block, but may read or skip fewer bytes.
 *
 * <p> Note that while some implementations of {@code InputStream} will return
 * the total number of bytes in the stream, many will not.  It is
 * never correct to use the return value of this method to allocate
 * a buffer intended to hold all data in this stream.
 *
 * <p> A subclass' implementation of this method may choose to throw an
 * {@link IOException} if this input stream has been closed by
 * invoking the {@link #close()} method.
 *
```

```
     * <p> The {@code available} method for class {@code InputStream} always
     * returns {@code 0}.
     *
     * <p> This method should be overridden by subclasses.
     *
     * @return     an estimate of the number of bytes that can be read (or skipped
     *             over) from this input stream without blocking or {@code 0} when
     *             it reaches the end of the input stream.
     * @exception  IOException if an I/O error occurs.
     */
    public int available() throws IOException {
        return 0;
    }

    /**
     * Closes this input stream and releases any system resources associated
     * with the stream.
     *
     * <p> The <code>close</code> method of <code>InputStream</code> does
     * nothing.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void close() throws IOException {}

    /**
     * Marks the current position in this input stream. A subsequent call to
     * the <code>reset</code> method repositions this stream at the last marked
     * position so that subsequent reads re-read the same bytes.
     *
     * <p> The <code>readlimit</code> arguments tells this input stream to
     * allow that many bytes to be read before the mark position gets
     * invalidated.
     *
     * <p> The general contract of <code>mark</code> is that, if the method
     * <code>markSupported</code> returns <code>true</code>, the stream somehow
     * remembers all the bytes read after the call to <code>mark</code> and
     * stands ready to supply those same bytes again if and whenever the method
     * <code>reset</code> is called.  However, the stream is not required to
     * remember any data at all if more than <code>readlimit</code> bytes are
     * read from the stream before <code>reset</code> is called.
     *
     * <p> Marking a closed stream should not have any effect on the stream.
     *
     * <p> The <code>mark</code> method of <code>InputStream</code> does
     * nothing.
     *
     * @param   readlimit   the maximum limit of bytes that can be read before
     *                      the mark position becomes invalid.
     * @see     java.io.InputStream#reset()
     */
    public synchronized void mark(int readlimit) {}

    /**
     * Repositions this stream to the position at the time the
     * <code>mark</code> method was last called on this input stream.
     *
     * <p> The general contract of <code>reset</code> is:
     *
     * <ul>
     * <li> If the method <code>markSupported</code> returns
     * <code>true</code>, then:
     *
```

```
     *     <ul><li> If the method <code>mark</code> has not been called since
     *     the stream was created, or the number of bytes read from the stream
     *     since <code>mark</code> was last called is larger than the argument
     *     to <code>mark</code> at that last call, then an
     *     <code>IOException</code> might be thrown.
     *
     *     <li> If such an <code>IOException</code> is not thrown, then the
     *     stream is reset to a state such that all the bytes read since the
     *     most recent call to <code>mark</code> (or since the start of the
     *     file, if <code>mark</code> has not been called) will be resupplied
     *     to subsequent callers of the <code>read</code> method, followed by
     *     any bytes that otherwise would have been the next input data as of
     *     the time of the call to <code>reset</code>. </ul>
     *
     * <li> If the method <code>markSupported</code> returns
     * <code>false</code>, then:
     *
     *     <ul><li> The call to <code>reset</code> may throw an
     *     <code>IOException</code>.
     *
     *     <li> If an <code>IOException</code> is not thrown, then the stream
     *     is reset to a fixed state that depends on the particular type of the
     *     input stream and how it was created. The bytes that will be supplied
     *     to subsequent callers of the <code>read</code> method depend on the
     *     particular type of the input stream. </ul></ul>
     *
     * <p>The method <code>reset</code> for class <code>InputStream</code>
     * does nothing except throw an <code>IOException</code>.
     *
     * @exception  IOException  if this stream has not been marked or if the
     *               mark has been invalidated.
     * @see     java.io.InputStream#mark(int)
     * @see     java.io.IOException
     */
    public synchronized void reset() throws IOException {
        throw new IOException("mark/reset not supported");
    }

    /**
     * Tests if this input stream supports the <code>mark</code> and
     * <code>reset</code> methods. Whether or not <code>mark</code> and
     * <code>reset</code> are supported is an invariant property of a
     * particular input stream instance. The <code>markSupported</code> method
     * of <code>InputStream</code> returns <code>false</code>.
     *
     * @return  <code>true</code> if this stream instance supports the mark
     *          and reset methods; <code>false</code> otherwise.
     * @see     java.io.InputStream#mark(int)
     * @see     java.io.InputStream#reset()
     */
    public boolean markSupported() {
        return false;
    }

}
```

InputStreamReader.java

```java
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import sun.nio.cs.StreamDecoder;


/**
 * An InputStreamReader is a bridge from byte streams to character streams: It
 * reads bytes and decodes them into characters using a specified {@link
 * java.nio.charset.Charset charset}.  The charset that it uses
 * may be specified by name or may be given explicitly, or the platform's
 * default charset may be accepted.
 *
 * <p> Each invocation of one of an InputStreamReader's read() methods may
 * cause one or more bytes to be read from the underlying byte-input stream.
 * To enable the efficient conversion of bytes to characters, more bytes may
 * be read ahead from the underlying stream than are necessary to satisfy the
 * current read operation.
 *
 * <p> For top efficiency, consider wrapping an InputStreamReader within a
 * BufferedReader.  For example:
 *
 * <pre>
 * BufferedReader in
 *   = new BufferedReader(new InputStreamReader(System.in));
 * </pre>
 *
 * @see BufferedReader
 * @see InputStream
 * @see java.nio.charset.Charset
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */
```

```
public class InputStreamReader extends Reader {

    private final StreamDecoder sd;

    /**
     * Creates an InputStreamReader that uses the default charset.
     *
     * @param  in   An InputStream
     */
    public InputStreamReader(InputStream in) {
        super(in);
        try {
            sd = StreamDecoder.forInputStreamReader(in, this, (String)null); // ## check lock object
        } catch (UnsupportedEncodingException e) {
            // The default encoding should always be available
            throw new Error(e);
        }
    }

    /**
     * Creates an InputStreamReader that uses the named charset.
     *
     * @param  in
     *         An InputStream
     *
     * @param  charsetName
     *         The name of a supported
     *         {@link java.nio.charset.Charset charset}
     *
     * @exception  UnsupportedEncodingException
     *             If the named charset is not supported
     */
    public InputStreamReader(InputStream in, String charsetName)
        throws UnsupportedEncodingException
    {
        super(in);
        if (charsetName == null)
            throw new NullPointerException("charsetName");
        sd = StreamDecoder.forInputStreamReader(in, this, charsetName);
    }

    /**
     * Creates an InputStreamReader that uses the given charset.
     *
     * @param  in       An InputStream
     * @param  cs       A charset
     *
     * @since 1.4
     * @spec JSR-51
     */
    public InputStreamReader(InputStream in, Charset cs) {
        super(in);
        if (cs == null)
            throw new NullPointerException("charset");
        sd = StreamDecoder.forInputStreamReader(in, this, cs);
    }

    /**
     * Creates an InputStreamReader that uses the given charset decoder.
     *
     * @param  in       An InputStream
     * @param  dec      A charset decoder
```

```
     *
     * @since 1.4
     * @spec JSR-51
     */
    public InputStreamReader(InputStream in, CharsetDecoder dec) {
        super(in);
        if (dec == null)
            throw new NullPointerException("charset decoder");
        sd = StreamDecoder.forInputStreamReader(in, this, dec);
    }

    /**
     * Returns the name of the character encoding being used by this stream.
     *
     * <p> If the encoding has an historical name then that name is returned;
     * otherwise the encoding's canonical name is returned.
     *
     * <p> If this instance was created with the {@link
     * #InputStreamReader(InputStream, String)} constructor then the returned
     * name, being unique for the encoding, may differ from the name passed to
     * the constructor. This method will return <code>null</code> if the
     * stream has been closed.
     * </p>
     * @return The historical name of this encoding, or
     *         <code>null</code> if the stream has been closed
     *
     * @see java.nio.charset.Charset
     *
     * @revised 1.4
     * @spec JSR-51
     */
    public String getEncoding() {
        return sd.getEncoding();
    }

    /**
     * Reads a single character.
     *
     * @return The character read, or -1 if the end of the stream has been
     *         reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read() throws IOException {
        return sd.read();
    }

    /**
     * Reads characters into a portion of an array.
     *
     * @param      cbuf     Destination buffer
     * @param      offset   Offset at which to start storing characters
     * @param      length   Maximum number of characters to read
     *
     * @return     The number of characters read, or -1 if the end of the
     *             stream has been reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read(char cbuf[], int offset, int length) throws IOException {
        return sd.read(cbuf, offset, length);
    }
```

```
    /**
     * Tells whether this stream is ready to be read.  An InputStreamReader is
     * ready if its input buffer is not empty, or if bytes are available to be
     * read from the underlying byte stream.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public boolean ready() throws IOException {
        return sd.ready();
    }

    public void close() throws IOException {
        sd.close();
    }
}
```

InterruptedIOException.java

```
/*
 * Copyright (c) 1995, 2008, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Signals that an I/O operation has been interrupted. An
 * <code>InterruptedIOException</code> is thrown to indicate that an
 * input or output transfer has been terminated because the thread
 * performing it was interrupted. The field {@link #bytesTransferred}
 * indicates how many bytes were successfully transferred before
 * the interruption occurred.
 *
 * @author  unascribed
 * @see     java.io.InputStream
 * @see     java.io.OutputStream
 * @see     java.lang.Thread#interrupt()
 * @since   JDK1.0
 */
public
class InterruptedIOException extends IOException {
    private static final long serialVersionUID = 4020568460727500567L;

    /**
     * Constructs an <code>InterruptedIOException</code> with
     * <code>null</code> as its error detail message.
     */
    public InterruptedIOException() {
        super();
    }

    /**
     * Constructs an <code>InterruptedIOException</code> with the
     * specified detail message. The string <code>s</code> can be
     * retrieved later by the
     * <code>{@link java.lang.Throwable#getMessage}</code>
     * method of class <code>java.lang.Throwable</code>.
     *
```

```
     * @param   s   the detail message.
     */
    public InterruptedIOException(String s) {
        super(s);
    }

    /**
     * Reports how many bytes had been transferred as part of the I/O
     * operation before it was interrupted.
     *
     * @serial
     */
    public int bytesTransferred = 0;
}
```

InvalidClassException.java

```java
/*
 * Copyright (c) 1996, 2006, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Thrown when the Serialization runtime detects one of the following
 * problems with a Class.
 * <UL>
 * <LI> The serial version of the class does not match that of the class
 *      descriptor read from the stream
 * <LI> The class contains unknown datatypes
 * <LI> The class does not have an accessible no-arg constructor
 * </UL>
 *
 * @author  unascribed
 * @since   JDK1.1
 */
public class InvalidClassException extends ObjectStreamException {

    private static final long serialVersionUID = -4333316296251054416L;

    /**
     * Name of the invalid class.
     *
     * @serial Name of the invalid class.
     */
    public String classname;

    /**
     * Report an InvalidClassException for the reason specified.
     *
     * @param reason  String describing the reason for the exception.
     */
    public InvalidClassException(String reason) {
        super(reason);
    }
```

```
    /**
     * Constructs an InvalidClassException object.
     *
     * @param cname   a String naming the invalid class.
     * @param reason  a String describing the reason for the exception.
     */
    public InvalidClassException(String cname, String reason) {
        super(reason);
        classname = cname;
    }

    /**
     * Produce the message and include the classname, if present.
     */
    public String getMessage() {
        if (classname == null)
            return super.getMessage();
        else
            return classname + "; " + super.getMessage();
    }
}
```

InvalidObjectException.java

```java
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Indicates that one or more deserialized objects failed validation
 * tests.  The argument should provide the reason for the failure.
 *
 * @see ObjectInputValidation
 * @since JDK1.1
 *
 * @author  unascribed
 * @since   JDK1.1
 */
public class InvalidObjectException extends ObjectStreamException {

    private static final long serialVersionUID = 3233174318281839583L;

    /**
     * Constructs an <code>InvalidObjectException</code>.
     * @param reason Detailed message explaining the reason for the failure.
     *
     * @see ObjectInputValidation
     */
    public  InvalidObjectException(String reason) {
        super(reason);
    }
}
```

IOError.java

```
/*
 * Copyright (c) 2005, 2006, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Thrown when a serious I/O error has occurred.
 *
 * @author  Xueming Shen
 * @since   1.6
 */
public class IOError extends Error {
    /**
     * Constructs a new instance of IOError with the specified cause. The
     * IOError is created with the detail message of
     * (cause==null ? null : cause.toString()) (which typically
     * contains the class and detail message of cause).
     *
     * @param  cause
     *         The cause of this error, or null if the cause
     *         is not known
     */
    public IOError(Throwable cause) {
        super(cause);
    }

    private static final long serialVersionUID = 67100927991680413L;
}
```

IOException.java

```java
/*
 * Copyright (c) 1994, 2006, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Signals that an I/O exception of some sort has occurred. This
 * class is the general class of exceptions produced by failed or
 * interrupted I/O operations.
 *
 * @author  unascribed
 * @see     java.io.InputStream
 * @see     java.io.OutputStream
 * @since   JDK1.0
 */
public
class IOException extends Exception {
    static final long serialVersionUID = 7818375828146090155L;

    /**
     * Constructs an {@code IOException} with {@code null}
     * as its error detail message.
     */
    public IOException() {
        super();
    }

    /**
     * Constructs an {@code IOException} with the specified detail message.
     *
     * @param message
     *        The detail message (which is saved for later retrieval
     *        by the {@link #getMessage()} method)
     */
    public IOException(String message) {
        super(message);
    }
```

```
    /**
     * Constructs an {@code IOException} with the specified detail message
     * and cause.
     *
     * <p> Note that the detail message associated with {@code cause} is
     * <i>not</i> automatically incorporated into this exception's detail
     * message.
     *
     * @param message
     *        The detail message (which is saved for later retrieval
     *        by the {@link #getMessage()} method)
     *
     * @param cause
     *        The cause (which is saved for later retrieval by the
     *        {@link #getCause()} method).  (A null value is permitted,
     *        and indicates that the cause is nonexistent or unknown.)
     *
     * @since 1.6
     */
    public IOException(String message, Throwable cause) {
        super(message, cause);
    }

    /**
     * Constructs an {@code IOException} with the specified cause and a
     * detail message of {@code (cause==null ? null : cause.toString())}
     * (which typically contains the class and detail message of {@code cause}).
     * This constructor is useful for IO exceptions that are little more
     * than wrappers for other throwables.
     *
     * @param cause
     *        The cause (which is saved for later retrieval by the
     *        {@link #getCause()} method).  (A null value is permitted,
     *        and indicates that the cause is nonexistent or unknown.)
     *
     * @since 1.6
     */
    public IOException(Throwable cause) {
        super(cause);
    }
}
```

LineNumberInputStream.java

```
/*
 * Copyright (c) 1995, 2012, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * This class is an input stream filter that provides the added
 * functionality of keeping track of the current line number.
 * <p>
 * A line is a sequence of bytes ending with a carriage return
 * character ({@code '\u005Cr'}), a newline character
 * ({@code '\u005Cn'}), or a carriage return character followed
 * immediately by a linefeed character. In all three cases, the line
 * terminating character(s) are returned as a single newline character.
 * <p>
 * The line number begins at {@code 0}, and is incremented by
 * {@code 1} when a {@code read} returns a newline character.
 *
 * @author     Arthur van Hoff
 * @see        java.io.LineNumberReader
 * @since      JDK1.0
 * @deprecated This class incorrectly assumes that bytes adequately represent
 *             characters.  As of JDK 1.1, the preferred way to operate on
 *             character streams is via the new character-stream classes, which
 *             include a class for counting line numbers.
 */
@Deprecated
public
class LineNumberInputStream extends FilterInputStream {
    int pushBack = -1;
    int lineNumber;
    int markLineNumber;
    int markPushBack = -1;

    /**
     * Constructs a newline number input stream that reads its input
     * from the specified input stream.
     *
```

```
     * @param      in   the underlying input stream.
     */
    public LineNumberInputStream(InputStream in) {
        super(in);
    }

    /**
     * Reads the next byte of data from this input stream. The value
     * byte is returned as an {@code int} in the range
     * {@code 0} to {@code 255}. If no byte is available
     * because the end of the stream has been reached, the value
     * {@code -1} is returned. This method blocks until input data
     * is available, the end of the stream is detected, or an exception
     * is thrown.
     * <p>
     * The {@code read} method of
     * {@code LineNumberInputStream} calls the {@code read}
     * method of the underlying input stream. It checks for carriage
     * returns and newline characters in the input, and modifies the
     * current line number as appropriate. A carriage-return character or
     * a carriage return followed by a newline character are both
     * converted into a single newline character.
     *
     * @return     the next byte of data, or {@code -1} if the end of this
     *             stream is reached.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterInputStream#in
     * @see        java.io.LineNumberInputStream#getLineNumber()
     */
    @SuppressWarnings("fallthrough")
    public int read() throws IOException {
        int c = pushBack;

        if (c != -1) {
            pushBack = -1;
        } else {
            c = in.read();
        }

        switch (c) {
          case '\r':
            pushBack = in.read();
            if (pushBack == '\n') {
                pushBack = -1;
            }
          case '\n':
            lineNumber++;
            return '\n';
        }
        return c;
    }

    /**
     * Reads up to {@code len} bytes of data from this input stream
     * into an array of bytes. This method blocks until some input is available.
     * <p>
     * The {@code read} method of
     * {@code LineNumberInputStream} repeatedly calls the
     * {@code read} method of zero arguments to fill in the byte array.
     *
     * @param      b     the buffer into which the data is read.
     * @param      off   the start offset of the data.
     * @param      len   the maximum number of bytes read.
```

```
     * @return     the total number of bytes read into the buffer, or
     *             {@code -1} if there is no more data because the end of
     *             this stream has been reached.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.LineNumberInputStream#read()
     */
    public int read(byte b[], int off, int len) throws IOException {
        if (b == null) {
            throw new NullPointerException();
        } else if ((off < 0) || (off > b.length) || (len < 0) ||
                   ((off + len) > b.length) || ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return 0;
        }

        int c = read();
        if (c == -1) {
            return -1;
        }
        b[off] = (byte)c;

        int i = 1;
        try {
            for (; i < len ; i++) {
                c = read();
                if (c == -1) {
                    break;
                }
                if (b != null) {
                    b[off + i] = (byte)c;
                }
            }
        } catch (IOException ee) {
        }
        return i;
    }

    /**
     * Skips over and discards {@code n} bytes of data from this
     * input stream. The {@code skip} method may, for a variety of
     * reasons, end up skipping over some smaller number of bytes,
     * possibly {@code 0}. The actual number of bytes skipped is
     * returned.  If {@code n} is negative, no bytes are skipped.
     * <p>
     * The {@code skip} method of {@code LineNumberInputStream} creates
     * a byte array and then repeatedly reads into it until
     * {@code n} bytes have been read or the end of the stream has
     * been reached.
     *
     * @param      n   the number of bytes to be skipped.
     * @return     the actual number of bytes skipped.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterInputStream#in
     */
    public long skip(long n) throws IOException {
        int chunk = 2048;
        long remaining = n;
        byte data[];
        int nr;

        if (n <= 0) {
            return 0;
```

```
    }

    data = new byte[chunk];
    while (remaining > 0) {
        nr = read(data, 0, (int) Math.min(chunk, remaining));
        if (nr < 0) {
            break;
        }
        remaining -= nr;
    }

    return n - remaining;
}

/**
 * Sets the line number to the specified argument.
 *
 * @param      lineNumber   the new line number.
 * @see #getLineNumber
 */
public void setLineNumber(int lineNumber) {
    this.lineNumber = lineNumber;
}

/**
 * Returns the current line number.
 *
 * @return     the current line number.
 * @see #setLineNumber
 */
public int getLineNumber() {
    return lineNumber;
}


/**
 * Returns the number of bytes that can be read from this input
 * stream without blocking.
 * <p>
 * Note that if the underlying input stream is able to supply
 * <i>k</i> input characters without blocking, the
 * {@code LineNumberInputStream} can guarantee only to provide
 * <i>k</i>/2 characters without blocking, because the
 * <i>k</i> characters from the underlying input stream might
 * consist of <i>k</i>/2 pairs of {@code '\u005Cr'} and
 * {@code '\u005Cn'}, which are converted to just
 * <i>k</i>/2 {@code '\u005Cn'} characters.
 *
 * @return     the number of bytes that can be read from this input stream
 *             without blocking.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.FilterInputStream#in
 */
public int available() throws IOException {
    return (pushBack == -1) ? super.available()/2 : super.available()/2 + 1;
}

/**
 * Marks the current position in this input stream. A subsequent
 * call to the {@code reset} method repositions this stream at
 * the last marked position so that subsequent reads re-read the same bytes.
 * <p>
 * The {@code mark} method of
```

```
     * {@code LineNumberInputStream} remembers the current line
     * number in a private variable, and then calls the {@code mark}
     * method of the underlying input stream.
     *
     * @param   readlimit   the maximum limit of bytes that can be read before
     *                      the mark position becomes invalid.
     * @see     java.io.FilterInputStream#in
     * @see     java.io.LineNumberInputStream#reset()
     */
    public void mark(int readlimit) {
        markLineNumber = lineNumber;
        markPushBack   = pushBack;
        in.mark(readlimit);
    }

    /**
     * Repositions this stream to the position at the time the
     * {@code mark} method was last called on this input stream.
     * <p>
     * The {@code reset} method of
     * {@code LineNumberInputStream} resets the line number to be
     * the line number at the time the {@code mark} method was
     * called, and then calls the {@code reset} method of the
     * underlying input stream.
     * <p>
     * Stream marks are intended to be used in
     * situations where you need to read ahead a little to see what's in
     * the stream. Often this is most easily done by invoking some
     * general parser. If the stream is of the type handled by the
     * parser, it just chugs along happily. If the stream is not of
     * that type, the parser should toss an exception when it fails,
     * which, if it happens within readlimit bytes, allows the outer
     * code to reset the stream and try another parser.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterInputStream#in
     * @see        java.io.LineNumberInputStream#mark(int)
     */
    public void reset() throws IOException {
        lineNumber = markLineNumber;
        pushBack   = markPushBack;
        in.reset();
    }
}
```

LineNumberReader.java

```
/*
 * Copyright (c) 1996, 2011, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * A buffered character-input stream that keeps track of line numbers.  This
 * class defines methods {@link #setLineNumber(int)} and {@link
 * #getLineNumber()} for setting and getting the current line number
 * respectively.
 *
 * <p> By default, line numbering begins at 0. This number increments at every
 * <a href="#lt">line terminator</a> as the data is read, and can be changed
 * with a call to <tt>setLineNumber(int)</tt>.  Note however, that
 * <tt>setLineNumber(int)</tt> does not actually change the current position in
 * the stream; it only changes the value that will be returned by
 * <tt>getLineNumber()</tt>.
 *
 * <p> A line is considered to be <a name="lt">terminated</a> by any one of a
 * line feed ('\n'), a carriage return ('\r'), or a carriage return followed
 * immediately by a linefeed.
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class LineNumberReader extends BufferedReader {

    /** The current line number */
    private int lineNumber = 0;

    /** The line number of the mark, if any */
    private int markedLineNumber; // Defaults to 0

    /** If the next character is a line feed, skip it */
    private boolean skipLF;
```

```
/** The skipLF flag when the mark was set */
private boolean markedSkipLF;

/**
 * Create a new line-numbering reader, using the default input-buffer
 * size.
 *
 * @param  in
 *         A Reader object to provide the underlying stream
 */
public LineNumberReader(Reader in) {
    super(in);
}

/**
 * Create a new line-numbering reader, reading characters into a buffer of
 * the given size.
 *
 * @param  in
 *         A Reader object to provide the underlying stream
 *
 * @param  sz
 *         An int specifying the size of the buffer
 */
public LineNumberReader(Reader in, int sz) {
    super(in, sz);
}

/**
 * Set the current line number.
 *
 * @param  lineNumber
 *         An int specifying the line number
 *
 * @see #getLineNumber
 */
public void setLineNumber(int lineNumber) {
    this.lineNumber = lineNumber;
}

/**
 * Get the current line number.
 *
 * @return  The current line number
 *
 * @see #setLineNumber
 */
public int getLineNumber() {
    return lineNumber;
}

/**
 * Read a single character.  <a href="#lt">Line terminators</a> are
 * compressed into single newline ('\n') characters.  Whenever a line
 * terminator is read the current line number is incremented.
 *
 * @return  The character read, or -1 if the end of the stream has been
 *          reached
 *
 * @throws  IOException
 *          If an I/O error occurs
 */
@SuppressWarnings("fallthrough")
```

```
public int read() throws IOException {
    synchronized (lock) {
        int c = super.read();
        if (skipLF) {
            if (c == '\n')
                c = super.read();
            skipLF = false;
        }
        switch (c) {
        case '\r':
            skipLF = true;
        case '\n':          /* Fall through */
            lineNumber++;
            return '\n';
        }
        return c;
    }
}

/**
 * Read characters into a portion of an array.  Whenever a <a
 * href="#lt">line terminator</a> is read the current line number is
 * incremented.
 *
 * @param  cbuf
 *         Destination buffer
 *
 * @param  off
 *         Offset at which to start storing characters
 *
 * @param  len
 *         Maximum number of characters to read
 *
 * @return  The number of bytes read, or -1 if the end of the stream has
 *          already been reached
 *
 * @throws  IOException
 *          If an I/O error occurs
 */
@SuppressWarnings("fallthrough")
public int read(char cbuf[], int off, int len) throws IOException {
    synchronized (lock) {
        int n = super.read(cbuf, off, len);

        for (int i = off; i < off + n; i++) {
            int c = cbuf[i];
            if (skipLF) {
                skipLF = false;
                if (c == '\n')
                    continue;
            }
            switch (c) {
            case '\r':
                skipLF = true;
            case '\n':      /* Fall through */
                lineNumber++;
                break;
            }
        }

        return n;
    }
}
```

```
/**
 * Read a line of text.  Whenever a <a href="#lt">line terminator</a> is
 * read the current line number is incremented.
 *
 * @return  A String containing the contents of the line, not including
 *          any <a href="#lt">line termination characters</a>, or
 *          <tt>null</tt> if the end of the stream has been reached
 *
 * @throws  IOException
 *          If an I/O error occurs
 */
public String readLine() throws IOException {
    synchronized (lock) {
        String l = super.readLine(skipLF);
        skipLF = false;
        if (l != null)
            lineNumber++;
        return l;
    }
}

/** Maximum skip-buffer size */
private static final int maxSkipBufferSize = 8192;

/** Skip buffer, null until allocated */
private char skipBuffer[] = null;

/**
 * Skip characters.
 *
 * @param  n
 *         The number of characters to skip
 *
 * @return  The number of characters actually skipped
 *
 * @throws  IOException
 *          If an I/O error occurs
 *
 * @throws  IllegalArgumentException
 *          If <tt>n</tt> is negative
 */
public long skip(long n) throws IOException {
    if (n < 0)
        throw new IllegalArgumentException("skip() value is negative");
    int nn = (int) Math.min(n, maxSkipBufferSize);
    synchronized (lock) {
        if ((skipBuffer == null) || (skipBuffer.length < nn))
            skipBuffer = new char[nn];
        long r = n;
        while (r > 0) {
            int nc = read(skipBuffer, 0, (int) Math.min(r, nn));
            if (nc == -1)
                break;
            r -= nc;
        }
        return n - r;
    }
}

/**
 * Mark the present position in the stream.  Subsequent calls to reset()
 * will attempt to reposition the stream to this point, and will also reset
```

```
     * the line number appropriately.
     *
     * @param  readAheadLimit
     *         Limit on the number of characters that may be read while still
     *         preserving the mark.  After reading this many characters,
     *         attempting to reset the stream may fail.
     *
     * @throws  IOException
     *          If an I/O error occurs
     */
    public void mark(int readAheadLimit) throws IOException {
        synchronized (lock) {
            super.mark(readAheadLimit);
            markedLineNumber = lineNumber;
            markedSkipLF     = skipLF;
        }
    }

    /**
     * Reset the stream to the most recent mark.
     *
     * @throws  IOException
     *          If the stream has not been marked, or if the mark has been
     *          invalidated
     */
    public void reset() throws IOException {
        synchronized (lock) {
            super.reset();
            lineNumber = markedLineNumber;
            skipLF     = markedSkipLF;
        }
    }

}
```

NotActiveException.java

```java
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Thrown when serialization or deserialization is not active.
 *
 * @author  unascribed
 * @since   JDK1.1
 */
public class NotActiveException extends ObjectStreamException {

    private static final long serialVersionUID = -3893467273049808895L;

    /**
     * Constructor to create a new NotActiveException with the reason given.
     *
     * @param reason  a String describing the reason for the exception.
     */
    public NotActiveException(String reason) {
        super(reason);
    }

    /**
     * Constructor to create a new NotActiveException without a reason.
     */
    public NotActiveException() {
        super();
    }
}
```

NotSerializableException.java

```java
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Thrown when an instance is required to have a Serializable interface.
 * The serialization runtime or the class of the instance can throw
 * this exception. The argument should be the name of the class.
 *
 * @author  unascribed
 * @since   JDK1.1
 */
public class NotSerializableException extends ObjectStreamException {

    private static final long serialVersionUID = 2906642554793891381L;

    /**
     * Constructs a NotSerializableException object with message string.
     *
     * @param classname Class of the instance being serialized/deserialized.
     */
    public NotSerializableException(String classname) {
        super(classname);
    }

    /**
     *  Constructs a NotSerializableException object.
     */
    public NotSerializableException() {
        super();
    }
}
```

ObjectInput.java

```
/*
 * Copyright (c) 1996, 2010, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * ObjectInput extends the DataInput interface to include the reading of
 * objects. DataInput includes methods for the input of primitive types,
 * ObjectInput extends that interface to include objects, arrays, and Strings.
 *
 * @author  unascribed
 * @see java.io.InputStream
 * @see java.io.ObjectOutputStream
 * @see java.io.ObjectInputStream
 * @since   JDK1.1
 */
public interface ObjectInput extends DataInput, AutoCloseable {
    /**
     * Read and return an object. The class that implements this interface
     * defines where the object is "read" from.
     *
     * @return the object read from the stream
     * @exception java.lang.ClassNotFoundException If the class of a serialized
     *      object cannot be found.
     * @exception IOException If any of the usual Input/Output
     * related exceptions occur.
     */
    public Object readObject()
        throws ClassNotFoundException, IOException;

    /**
     * Reads a byte of data. This method will block if no input is
     * available.
     * @return  the byte read, or -1 if the end of the
     *          stream is reached.
     * @exception IOException If an I/O error has occurred.
     */
    public int read() throws IOException;
```

```
    /**
     * Reads into an array of bytes.  This method will
     * block until some input is available.
     * @param b the buffer into which the data is read
     * @return  the actual number of bytes read, -1 is
     *          returned when the end of the stream is reached.
     * @exception IOException If an I/O error has occurred.
     */
    public int read(byte b[]) throws IOException;

    /**
     * Reads into an array of bytes.  This method will
     * block until some input is available.
     * @param b the buffer into which the data is read
     * @param off the start offset of the data
     * @param len the maximum number of bytes read
     * @return  the actual number of bytes read, -1 is
     *          returned when the end of the stream is reached.
     * @exception IOException If an I/O error has occurred.
     */
    public int read(byte b[], int off, int len) throws IOException;

    /**
     * Skips n bytes of input.
     * @param n the number of bytes to be skipped
     * @return  the actual number of bytes skipped.
     * @exception IOException If an I/O error has occurred.
     */
    public long skip(long n) throws IOException;

    /**
     * Returns the number of bytes that can be read
     * without blocking.
     * @return the number of available bytes.
     * @exception IOException If an I/O error has occurred.
     */
    public int available() throws IOException;

    /**
     * Closes the input stream. Must be called
     * to release any resources associated with
     * the stream.
     * @exception IOException If an I/O error has occurred.
     */
    public void close() throws IOException;
}
```

ObjectInputStream.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.io.ObjectStreamClass.WeakClassKey;
import java.lang.ref.ReferenceQueue;
import java.lang.reflect.Array;
import java.lang.reflect.Modifier;
import java.lang.reflect.Proxy;
import java.security.AccessControlContext;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.util.Arrays;
import java.util.HashMap;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.atomic.AtomicBoolean;
import static java.io.ObjectStreamClass.processQueue;
import sun.reflect.misc.ReflectUtil;

/**
 * An ObjectInputStream deserializes primitive data and objects previously
 * written using an ObjectOutputStream.
 *
 * <p>ObjectOutputStream and ObjectInputStream can provide an application with
 * persistent storage for graphs of objects when used with a FileOutputStream
 * and FileInputStream respectively.  ObjectInputStream is used to recover
 * those objects previously serialized. Other uses include passing objects
 * between hosts using a socket stream or for marshaling and unmarshaling
 * arguments and parameters in a remote communication system.
 *
 * <p>ObjectInputStream ensures that the types of all objects in the graph
 * created from the stream match the classes present in the Java Virtual
 * Machine.  Classes are loaded as required using the standard mechanisms.
 *
```

```
* <p>Only objects that support the java.io.Serializable or
* java.io.Externalizable interface can be read from streams.
*
* <p>The method <code>readObject</code> is used to read an object from the
* stream.  Java's safe casting should be used to get the desired type.  In
* Java, strings and arrays are objects and are treated as objects during
* serialization. When read they need to be cast to the expected type.
*
* <p>Primitive data types can be read from the stream using the appropriate
* method on DataInput.
*
* <p>The default deserialization mechanism for objects restores the contents
* of each field to the value and type it had when it was written.  Fields
* declared as transient or static are ignored by the deserialization process.
* References to other objects cause those objects to be read from the stream
* as necessary.  Graphs of objects are restored correctly using a reference
* sharing mechanism.  New objects are always allocated when deserializing,
* which prevents existing objects from being overwritten.
*
* <p>Reading an object is analogous to running the constructors of a new
* object.  Memory is allocated for the object and initialized to zero (NULL).
* No-arg constructors are invoked for the non-serializable classes and then
* the fields of the serializable classes are restored from the stream starting
* with the serializable class closest to java.lang.object and finishing with
* the object's most specific class.
*
* <p>For example to read from a stream as written by the example in
* ObjectOutputStream:
* <br>
* <pre>
*      FileInputStream fis = new FileInputStream("t.tmp");
*      ObjectInputStream ois = new ObjectInputStream(fis);
*
*      int i = ois.readInt();
*      String today = (String) ois.readObject();
*      Date date = (Date) ois.readObject();
*
*      ois.close();
* </pre>
*
* <p>Classes control how they are serialized by implementing either the
* java.io.Serializable or java.io.Externalizable interfaces.
*
* <p>Implementing the Serializable interface allows object serialization to
* save and restore the entire state of the object and it allows classes to
* evolve between the time the stream is written and the time it is read.  It
* automatically traverses references between objects, saving and restoring
* entire graphs.
*
* <p>Serializable classes that require special handling during the
* serialization and deserialization process should implement the following
* methods:
*
* <pre>
* private void writeObject(java.io.ObjectOutputStream stream)
*     throws IOException;
* private void readObject(java.io.ObjectInputStream stream)
*     throws IOException, ClassNotFoundException;
* private void readObjectNoData()
*     throws ObjectStreamException;
* </pre>
*
* <p>The readObject method is responsible for reading and restoring the state
```

```
* of the object for its particular class using data written to the stream by
* the corresponding writeObject method.  The method does not need to concern
* itself with the state belonging to its superclasses or subclasses.  State is
* restored by reading data from the ObjectInputStream for the individual
* fields and making assignments to the appropriate fields of the object.
* Reading primitive data types is supported by DataInput.
*
* <p>Any attempt to read object data which exceeds the boundaries of the
* custom data written by the corresponding writeObject method will cause an
* OptionalDataException to be thrown with an eof field value of true.
* Non-object reads which exceed the end of the allotted data will reflect the
* end of data in the same way that they would indicate the end of the stream:
* bytewise reads will return -1 as the byte read or number of bytes read, and
* primitive reads will throw EOFExceptions.  If there is no corresponding
* writeObject method, then the end of default serialized data marks the end of
* the allotted data.
*
* <p>Primitive and object read calls issued from within a readExternal method
* behave in the same manner--if the stream is already positioned at the end of
* data written by the corresponding writeExternal method, object reads will
* throw OptionalDataExceptions with eof set to true, bytewise reads will
* return -1, and primitive reads will throw EOFExceptions.  Note that this
* behavior does not hold for streams written with the old
* <code>ObjectStreamConstants.PROTOCOL_VERSION_1</code> protocol, in which the
* end of data written by writeExternal methods is not demarcated, and hence
* cannot be detected.
*
* <p>The readObjectNoData method is responsible for initializing the state of
* the object for its particular class in the event that the serialization
* stream does not list the given class as a superclass of the object being
* deserialized.  This may occur in cases where the receiving party uses a
* different version of the deserialized instance's class than the sending
* party, and the receiver's version extends classes that are not extended by
* the sender's version.  This may also occur if the serialization stream has
* been tampered; hence, readObjectNoData is useful for initializing
* deserialized objects properly despite a "hostile" or incomplete source
* stream.
*
* <p>Serialization does not read or assign values to the fields of any object
* that does not implement the java.io.Serializable interface.  Subclasses of
* Objects that are not serializable can be serializable. In this case the
* non-serializable class must have a no-arg constructor to allow its fields to
* be initialized.  In this case it is the responsibility of the subclass to
* save and restore the state of the non-serializable class. It is frequently
* the case that the fields of that class are accessible (public, package, or
* protected) or that there are get and set methods that can be used to restore
* the state.
*
* <p>Any exception that occurs while deserializing an object will be caught by
* the ObjectInputStream and abort the reading process.
*
* <p>Implementing the Externalizable interface allows the object to assume
* complete control over the contents and format of the object's serialized
* form.  The methods of the Externalizable interface, writeExternal and
* readExternal, are called to save and restore the objects state.  When
* implemented by a class they can write and read their own state using all of
* the methods of ObjectOutput and ObjectInput.  It is the responsibility of
* the objects to handle any versioning that occurs.
*
* <p>Enum constants are deserialized differently than ordinary serializable or
* externalizable objects.  The serialized form of an enum constant consists
* solely of its name; field values of the constant are not transmitted.  To
* deserialize an enum constant, ObjectInputStream reads the constant name from
```

```
 * the stream; the deserialized constant is then obtained by calling the static
 * method <code>Enum.valueOf(Class, String)</code> with the enum constant's
 * base type and the received constant name as arguments.  Like other
 * serializable or externalizable objects, enum constants can function as the
 * targets of back references appearing subsequently in the serialization
 * stream.  The process by which enum constants are deserialized cannot be
 * customized: any class-specific readObject, readObjectNoData, and readResolve
 * methods defined by enum types are ignored during deserialization.
 * Similarly, any serialPersistentFields or serialVersionUID field declarations
 * are also ignored--all enum types have a fixed serialVersionUID of 0L.
 *
 * @author      Mike Warres
 * @author      Roger Riggs
 * @see java.io.DataInput
 * @see java.io.ObjectOutputStream
 * @see java.io.Serializable
 * @see <a href="../../../platform/serialization/spec/input.html"> Object Serialization Specification, Section
3, Object Input Classes</a>
 * @since   JDK1.1
 */
public class ObjectInputStream
    extends InputStream implements ObjectInput, ObjectStreamConstants
{
    /** handle value representing null */
    private static final int NULL_HANDLE = -1;

    /** marker for unshared objects in internal handle table */
    private static final Object unsharedMarker = new Object();

    /** table mapping primitive type names to corresponding class objects */
    private static final HashMap<String, Class<?>> primClasses
        = new HashMap<>(8, 1.0F);
    static {
        primClasses.put("boolean", boolean.class);
        primClasses.put("byte", byte.class);
        primClasses.put("char", char.class);
        primClasses.put("short", short.class);
        primClasses.put("int", int.class);
        primClasses.put("long", long.class);
        primClasses.put("float", float.class);
        primClasses.put("double", double.class);
        primClasses.put("void", void.class);
    }

    private static class Caches {
        /** cache of subclass security audit results */
        static final ConcurrentMap<WeakClassKey,Boolean> subclassAudits =
            new ConcurrentHashMap<>();

        /** queue for WeakReferences to audited subclasses */
        static final ReferenceQueue<Class<?>> subclassAuditsQueue =
            new ReferenceQueue<>();
    }

    /** filter stream for handling block data conversion */
    private final BlockDataInputStream bin;
    /** validation callback list */
    private final ValidationList vlist;
    /** recursion depth */
    private int depth;
    /** whether stream is closed */
    private boolean closed;
```

```
/** wire handle -> obj/exception map */
private final HandleTable handles;
/** scratch field for passing handle values up/down call stack */
private int passHandle = NULL_HANDLE;
/** flag set when at end of field value block with no TC_ENDBLOCKDATA */
private boolean defaultDataEnd = false;

/** buffer for reading primitive field values */
private byte[] primVals;

/** if true, invoke readObjectOverride() instead of readObject() */
private final boolean enableOverride;
/** if true, invoke resolveObject() */
private boolean enableResolve;

/**
 * Context during upcalls to class-defined readObject methods; holds
 * object currently being deserialized and descriptor for current class.
 * Null when not during readObject upcall.
 */
private SerialCallbackContext curContext;

/**
 * Creates an ObjectInputStream that reads from the specified InputStream.
 * A serialization stream header is read from the stream and verified.
 * This constructor will block until the corresponding ObjectOutputStream
 * has written and flushed the header.
 *
 * <p>If a security manager is installed, this constructor will check for
 * the "enableSubclassImplementation" SerializablePermission when invoked
 * directly or indirectly by the constructor of a subclass which overrides
 * the ObjectInputStream.readFields or ObjectInputStream.readUnshared
 * methods.
 *
 * @param   in input stream to read from
 * @throws  StreamCorruptedException if the stream header is incorrect
 * @throws  IOException if an I/O error occurs while reading stream header
 * @throws  SecurityException if untrusted subclass illegally overrides
 *          security-sensitive methods
 * @throws  NullPointerException if <code>in</code> is <code>null</code>
 * @see     ObjectInputStream#ObjectInputStream()
 * @see     ObjectInputStream#readFields()
 * @see     ObjectOutputStream#ObjectOutputStream(OutputStream)
 */
public ObjectInputStream(InputStream in) throws IOException {
    verifySubclass();
    bin = new BlockDataInputStream(in);
    handles = new HandleTable(10);
    vlist = new ValidationList();
    enableOverride = false;
    readStreamHeader();
    bin.setBlockDataMode(true);
}

/**
 * Provide a way for subclasses that are completely reimplementing
 * ObjectInputStream to not have to allocate private data just used by this
 * implementation of ObjectInputStream.
 *
 * <p>If there is a security manager installed, this method first calls the
 * security manager's <code>checkPermission</code> method with the
 * <code>SerializablePermission("enableSubclassImplementation")</code>
 * permission to ensure it's ok to enable subclassing.
```

```
     *
     * @throws  SecurityException if a security manager exists and its
     *          <code>checkPermission</code> method denies enabling
     *          subclassing.
     * @throws  IOException if an I/O error occurs while creating this stream
     * @see SecurityManager#checkPermission
     * @see java.io.SerializablePermission
     */
    protected ObjectInputStream() throws IOException, SecurityException {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSION);
        }
        bin = null;
        handles = null;
        vlist = null;
        enableOverride = true;
    }

    /**
     * Read an object from the ObjectInputStream.  The class of the object, the
     * signature of the class, and the values of the non-transient and
     * non-static fields of the class and all of its supertypes are read.
     * Default deserializing for a class can be overriden using the writeObject
     * and readObject methods.  Objects referenced by this object are read
     * transitively so that a complete equivalent graph of objects is
     * reconstructed by readObject.
     *
     * <p>The root object is completely restored when all of its fields and the
     * objects it references are completely restored.  At this point the object
     * validation callbacks are executed in order based on their registered
     * priorities. The callbacks are registered by objects (in the readObject
     * special methods) as they are individually restored.
     *
     * <p>Exceptions are thrown for problems with the InputStream and for
     * classes that should not be deserialized.  All exceptions are fatal to
     * the InputStream and leave it in an indeterminate state; it is up to the
     * caller to ignore or recover the stream state.
     *
     * @throws  ClassNotFoundException Class of a serialized object cannot be
     *          found.
     * @throws  InvalidClassException Something is wrong with a class used by
     *          serialization.
     * @throws  StreamCorruptedException Control information in the
     *          stream is inconsistent.
     * @throws  OptionalDataException Primitive data was found in the
     *          stream instead of objects.
     * @throws  IOException Any of the usual Input/Output related exceptions.
     */
    public final Object readObject()
        throws IOException, ClassNotFoundException
    {
        if (enableOverride) {
            return readObjectOverride();
        }

        // if nested read, passHandle contains handle of enclosing object
        int outerHandle = passHandle;
        try {
            Object obj = readObject0(false);
            handles.markDependency(outerHandle, passHandle);
            ClassNotFoundException ex = handles.lookupException(passHandle);
            if (ex != null) {
```

```
            throw ex;
        }
        if (depth == 0) {
            vlist.doCallbacks();
        }
        return obj;
    } finally {
        passHandle = outerHandle;
        if (closed && depth == 0) {
            clear();
        }
    }
}

/**
 * This method is called by trusted subclasses of ObjectOutputStream that
 * constructed ObjectOutputStream using the protected no-arg constructor.
 * The subclass is expected to provide an override method with the modifier
 * "final".
 *
 * @return  the Object read from the stream.
 * @throws  ClassNotFoundException Class definition of a serialized object
 *          cannot be found.
 * @throws  OptionalDataException Primitive data was found in the stream
 *          instead of objects.
 * @throws  IOException if I/O errors occurred while reading from the
 *          underlying stream
 * @see #ObjectInputStream()
 * @see #readObject()
 * @since 1.2
 */
protected Object readObjectOverride()
    throws IOException, ClassNotFoundException
{
    return null;
}

/**
 * Reads an "unshared" object from the ObjectInputStream.  This method is
 * identical to readObject, except that it prevents subsequent calls to
 * readObject and readUnshared from returning additional references to the
 * deserialized instance obtained via this call.  Specifically:
 * <ul>
 *   <li>If readUnshared is called to deserialize a back-reference (the
 *       stream representation of an object which has been written
 *       previously to the stream), an ObjectStreamException will be
 *       thrown.
 *
 *   <li>If readUnshared returns successfully, then any subsequent attempts
 *       to deserialize back-references to the stream handle deserialized
 *       by readUnshared will cause an ObjectStreamException to be thrown.
 * </ul>
 * Deserializing an object via readUnshared invalidates the stream handle
 * associated with the returned object.  Note that this in itself does not
 * always guarantee that the reference returned by readUnshared is unique;
 * the deserialized object may define a readResolve method which returns an
 * object visible to other parties, or readUnshared may return a Class
 * object or enum constant obtainable elsewhere in the stream or through
 * external means. If the deserialized object defines a readResolve method
 * and the invocation of that method returns an array, then readUnshared
 * returns a shallow clone of that array; this guarantees that the returned
 * array object is unique and cannot be obtained a second time from an
 * invocation of readObject or readUnshared on the ObjectInputStream,
```

```
     * even if the underlying data stream has been manipulated.
     *
     * <p>ObjectInputStream subclasses which override this method can only be
     * constructed in security contexts possessing the
     * "enableSubclassImplementation" SerializablePermission; any attempt to
     * instantiate such a subclass without this permission will cause a
     * SecurityException to be thrown.
     *
     * @return  reference to deserialized object
     * @throws  ClassNotFoundException if class of an object to deserialize
     *          cannot be found
     * @throws  StreamCorruptedException if control information in the stream
     *          is inconsistent
     * @throws  ObjectStreamException if object to deserialize has already
     *          appeared in stream
     * @throws  OptionalDataException if primitive data is next in stream
     * @throws  IOException if an I/O error occurs during deserialization
     * @since   1.4
     */
    public Object readUnshared() throws IOException, ClassNotFoundException {
        // if nested read, passHandle contains handle of enclosing object
        int outerHandle = passHandle;
        try {
            Object obj = readObject0(true);
            handles.markDependency(outerHandle, passHandle);
            ClassNotFoundException ex = handles.lookupException(passHandle);
            if (ex != null) {
                throw ex;
            }
            if (depth == 0) {
                vlist.doCallbacks();
            }
            return obj;
        } finally {
            passHandle = outerHandle;
            if (closed && depth == 0) {
                clear();
            }
        }
    }

    /**
     * Read the non-static and non-transient fields of the current class from
     * this stream.  This may only be called from the readObject method of the
     * class being deserialized. It will throw the NotActiveException if it is
     * called otherwise.
     *
     * @throws  ClassNotFoundException if the class of a serialized object
     *          could not be found.
     * @throws  IOException if an I/O error occurs.
     * @throws  NotActiveException if the stream is not currently reading
     *          objects.
     */
    public void defaultReadObject()
        throws IOException, ClassNotFoundException
    {
        SerialCallbackContext ctx = curContext;
        if (ctx == null) {
            throw new NotActiveException("not in call to readObject");
        }
        Object curObj = ctx.getObj();
        ObjectStreamClass curDesc = ctx.getDesc();
        bin.setBlockDataMode(false);
```

```
    defaultReadFields(curObj, curDesc);
    bin.setBlockDataMode(true);
    if (!curDesc.hasWriteObjectData()) {
        /*
         * Fix for 4360508: since stream does not contain terminating
         * TC_ENDBLOCKDATA tag, set flag so that reading code elsewhere
         * knows to simulate end-of-custom-data behavior.
         */
        defaultDataEnd = true;
    }
    ClassNotFoundException ex = handles.lookupException(passHandle);
    if (ex != null) {
        throw ex;
    }
}

/**
 * Reads the persistent fields from the stream and makes them available by
 * name.
 *
 * @return  the <code>GetField</code> object representing the persistent
 *          fields of the object being deserialized
 * @throws  ClassNotFoundException if the class of a serialized object
 *          could not be found.
 * @throws  IOException if an I/O error occurs.
 * @throws  NotActiveException if the stream is not currently reading
 *          objects.
 * @since 1.2
 */
public ObjectInputStream.GetField readFields()
    throws IOException, ClassNotFoundException
{
    SerialCallbackContext ctx = curContext;
    if (ctx == null) {
        throw new NotActiveException("not in call to readObject");
    }
    Object curObj = ctx.getObj();
    ObjectStreamClass curDesc = ctx.getDesc();
    bin.setBlockDataMode(false);
    GetFieldImpl getField = new GetFieldImpl(curDesc);
    getField.readFields();
    bin.setBlockDataMode(true);
    if (!curDesc.hasWriteObjectData()) {
        /*
         * Fix for 4360508: since stream does not contain terminating
         * TC_ENDBLOCKDATA tag, set flag so that reading code elsewhere
         * knows to simulate end-of-custom-data behavior.
         */
        defaultDataEnd = true;
    }

    return getField;
}

/**
 * Register an object to be validated before the graph is returned.  While
 * similar to resolveObject these validations are called after the entire
 * graph has been reconstituted.  Typically, a readObject method will
 * register the object with the stream so that when all of the objects are
 * restored a final set of validations can be performed.
 *
 * @param   obj the object to receive the validation callback.
 * @param   prio controls the order of callbacks;zero is a good default.
```

```
     *          Use higher numbers to be called back earlier, lower numbers for
     *          later callbacks. Within a priority, callbacks are processed in
     *          no particular order.
     * @throws  NotActiveException The stream is not currently reading objects
     *          so it is invalid to register a callback.
     * @throws  InvalidObjectException The validation object is null.
     */
    public void registerValidation(ObjectInputValidation obj, int prio)
        throws NotActiveException, InvalidObjectException
    {
        if (depth == 0) {
            throw new NotActiveException("stream inactive");
        }
        vlist.register(obj, prio);
    }

    /**
     * Load the local class equivalent of the specified stream class
     * description.  Subclasses may implement this method to allow classes to
     * be fetched from an alternate source.
     *
     * <p>The corresponding method in <code>ObjectOutputStream</code> is
     * <code>annotateClass</code>.  This method will be invoked only once for
     * each unique class in the stream.  This method can be implemented by
     * subclasses to use an alternate loading mechanism but must return a
     * <code>Class</code> object. Once returned, if the class is not an array
     * class, its serialVersionUID is compared to the serialVersionUID of the
     * serialized class, and if there is a mismatch, the deserialization fails
     * and an {@link InvalidClassException} is thrown.
     *
     * <p>The default implementation of this method in
     * <code>ObjectInputStream</code> returns the result of calling
     * <pre>
     *     Class.forName(desc.getName(), false, loader)
     * </pre>
     * where <code>loader</code> is determined as follows: if there is a
     * method on the current thread's stack whose declaring class was
     * defined by a user-defined class loader (and was not a generated to
     * implement reflective invocations), then <code>loader</code> is class
     * loader corresponding to the closest such method to the currently
     * executing frame; otherwise, <code>loader</code> is
     * <code>null</code>. If this call results in a
     * <code>ClassNotFoundException</code> and the name of the passed
     * <code>ObjectStreamClass</code> instance is the Java language keyword
     * for a primitive type or void, then the <code>Class</code> object
     * representing that primitive type or void will be returned
     * (e.g., an <code>ObjectStreamClass</code> with the name
     * <code>"int"</code> will be resolved to <code>Integer.TYPE</code>).
     * Otherwise, the <code>ClassNotFoundException</code> will be thrown to
     * the caller of this method.
     *
     * @param   desc an instance of class <code>ObjectStreamClass</code>
     * @return  a <code>Class</code> object corresponding to <code>desc</code>
     * @throws  IOException any of the usual Input/Output exceptions.
     * @throws  ClassNotFoundException if class of a serialized object cannot
     *          be found.
     */
    protected Class<?> resolveClass(ObjectStreamClass desc)
        throws IOException, ClassNotFoundException
    {
        String name = desc.getName();
        try {
            return Class.forName(name, false, latestUserDefinedLoader());
```

```
    } catch (ClassNotFoundException ex) {
        Class<?> cl = primClasses.get(name);
        if (cl != null) {
            return cl;
        } else {
            throw ex;
        }
    }
}

/**
 * Returns a proxy class that implements the interfaces named in a proxy
 * class descriptor; subclasses may implement this method to read custom
 * data from the stream along with the descriptors for dynamic proxy
 * classes, allowing them to use an alternate loading mechanism for the
 * interfaces and the proxy class.
 *
 * <p>This method is called exactly once for each unique proxy class
 * descriptor in the stream.
 *
 * <p>The corresponding method in <code>ObjectOutputStream</code> is
 * <code>annotateProxyClass</code>.  For a given subclass of
 * <code>ObjectInputStream</code> that overrides this method, the
 * <code>annotateProxyClass</code> method in the corresponding subclass of
 * <code>ObjectOutputStream</code> must write any data or objects read by
 * this method.
 *
 * <p>The default implementation of this method in
 * <code>ObjectInputStream</code> returns the result of calling
 * <code>Proxy.getProxyClass</code> with the list of <code>Class</code>
 * objects for the interfaces that are named in the <code>interfaces</code>
 * parameter.  The <code>Class</code> object for each interface name
 * <code>i</code> is the value returned by calling
 * <pre>
 *     Class.forName(i, false, loader)
 * </pre>
 * where <code>loader</code> is that of the first non-<code>null</code>
 * class loader up the execution stack, or <code>null</code> if no
 * non-<code>null</code> class loaders are on the stack (the same class
 * loader choice used by the <code>resolveClass</code> method).  Unless any
 * of the resolved interfaces are non-public, this same value of
 * <code>loader</code> is also the class loader passed to
 * <code>Proxy.getProxyClass</code>; if non-public interfaces are present,
 * their class loader is passed instead (if more than one non-public
 * interface class loader is encountered, an
 * <code>IllegalAccessError</code> is thrown).
 * If <code>Proxy.getProxyClass</code> throws an
 * <code>IllegalArgumentException</code>, <code>resolveProxyClass</code>
 * will throw a <code>ClassNotFoundException</code> containing the
 * <code>IllegalArgumentException</code>.
 *
 * @param interfaces the list of interface names that were
 *                deserialized in the proxy class descriptor
 * @return  a proxy class for the specified interfaces
 * @throws        IOException any exception thrown by the underlying
 *                <code>InputStream</code>
 * @throws        ClassNotFoundException if the proxy class or any of the
 *                named interfaces could not be found
 * @see ObjectOutputStream#annotateProxyClass(Class)
 * @since 1.3
 */
protected Class<?> resolveProxyClass(String[] interfaces)
    throws IOException, ClassNotFoundException
```

```
{
    ClassLoader latestLoader = latestUserDefinedLoader();
    ClassLoader nonPublicLoader = null;
    boolean hasNonPublicInterface = false;

    // define proxy in class loader of non-public interface(s), if any
    Class<?>[] classObjs = new Class<?>[interfaces.length];
    for (int i = 0; i < interfaces.length; i++) {
        Class<?> cl = Class.forName(interfaces[i], false, latestLoader);
        if ((cl.getModifiers() & Modifier.PUBLIC) == 0) {
            if (hasNonPublicInterface) {
                if (nonPublicLoader != cl.getClassLoader()) {
                    throw new IllegalAccessError(
                        "conflicting non-public interface class loaders");
                }
            } else {
                nonPublicLoader = cl.getClassLoader();
                hasNonPublicInterface = true;
            }
        }
        classObjs[i] = cl;
    }
    try {
        return Proxy.getProxyClass(
            hasNonPublicInterface ? nonPublicLoader : latestLoader,
            classObjs);
    } catch (IllegalArgumentException e) {
        throw new ClassNotFoundException(null, e);
    }
}

/**
 * This method will allow trusted subclasses of ObjectInputStream to
 * substitute one object for another during deserialization. Replacing
 * objects is disabled until enableResolveObject is called. The
 * enableResolveObject method checks that the stream requesting to resolve
 * object can be trusted. Every reference to serializable objects is passed
 * to resolveObject.  To insure that the private state of objects is not
 * unintentionally exposed only trusted streams may use resolveObject.
 *
 * <p>This method is called after an object has been read but before it is
 * returned from readObject.  The default resolveObject method just returns
 * the same object.
 *
 * <p>When a subclass is replacing objects it must insure that the
 * substituted object is compatible with every field where the reference
 * will be stored.  Objects whose type is not a subclass of the type of the
 * field or array element abort the serialization by raising an exception
 * and the object is not be stored.
 *
 * <p>This method is called only once when each object is first
 * encountered.  All subsequent references to the object will be redirected
 * to the new object.
 *
 * @param   obj object to be substituted
 * @return  the substituted object
 * @throws  IOException Any of the usual Input/Output exceptions.
 */
protected Object resolveObject(Object obj) throws IOException {
    return obj;
}

/**
```

```
     * Enable the stream to allow objects read from the stream to be replaced.
     * When enabled, the resolveObject method is called for every object being
     * deserialized.
     *
     * <p>If <i>enable</i> is true, and there is a security manager installed,
     * this method first calls the security manager's
     * <code>checkPermission</code> method with the
     * <code>SerializablePermission("enableSubstitution")</code> permission to
     * ensure it's ok to enable the stream to allow objects read from the
     * stream to be replaced.
     *
     * @param   enable true for enabling use of <code>resolveObject</code> for
     *          every object being deserialized
     * @return  the previous setting before this method was invoked
     * @throws  SecurityException if a security manager exists and its
     *          <code>checkPermission</code> method denies enabling the stream
     *          to allow objects read from the stream to be replaced.
     * @see SecurityManager#checkPermission
     * @see java.io.SerializablePermission
     */
    protected boolean enableResolveObject(boolean enable)
        throws SecurityException
    {
        if (enable == enableResolve) {
            return enable;
        }
        if (enable) {
            SecurityManager sm = System.getSecurityManager();
            if (sm != null) {
                sm.checkPermission(SUBSTITUTION_PERMISSION);
            }
        }
        enableResolve = enable;
        return !enableResolve;
    }

    /**
     * The readStreamHeader method is provided to allow subclasses to read and
     * verify their own stream headers. It reads and verifies the magic number
     * and version number.
     *
     * @throws  IOException if there are I/O errors while reading from the
     *          underlying <code>InputStream</code>
     * @throws  StreamCorruptedException if control information in the stream
     *          is inconsistent
     */
    protected void readStreamHeader()
        throws IOException, StreamCorruptedException
    {
        short s0 = bin.readShort();
        short s1 = bin.readShort();
        if (s0 != STREAM_MAGIC || s1 != STREAM_VERSION) {
            throw new StreamCorruptedException(
                String.format("invalid stream header: %04X%04X", s0, s1));
        }
    }

    /**
     * Read a class descriptor from the serialization stream.  This method is
     * called when the ObjectInputStream expects a class descriptor as the next
     * item in the serialization stream.  Subclasses of ObjectInputStream may
     * override this method to read in class descriptors that have been written
     * in non-standard formats (by subclasses of ObjectOutputStream which have
```

```
     * overridden the <code>writeClassDescriptor</code> method).  By default,
     * this method reads class descriptors according to the format defined in
     * the Object Serialization specification.
     *
     * @return  the class descriptor read
     * @throws  IOException If an I/O error has occurred.
     * @throws  ClassNotFoundException If the Class of a serialized object used
     *          in the class descriptor representation cannot be found
     * @see java.io.ObjectOutputStream#writeClassDescriptor(java.io.ObjectStreamClass)
     * @since 1.3
     */
    protected ObjectStreamClass readClassDescriptor()
        throws IOException, ClassNotFoundException
    {
        ObjectStreamClass desc = new ObjectStreamClass();
        desc.readNonProxy(this);
        return desc;
    }

    /**
     * Reads a byte of data. This method will block if no input is available.
     *
     * @return  the byte read, or -1 if the end of the stream is reached.
     * @throws  IOException If an I/O error has occurred.
     */
    public int read() throws IOException {
        return bin.read();
    }

    /**
     * Reads into an array of bytes.  This method will block until some input
     * is available. Consider using java.io.DataInputStream.readFully to read
     * exactly 'length' bytes.
     *
     * @param   buf the buffer into which the data is read
     * @param   off the start offset of the data
     * @param   len the maximum number of bytes read
     * @return  the actual number of bytes read, -1 is returned when the end of
     *          the stream is reached.
     * @throws  IOException If an I/O error has occurred.
     * @see java.io.DataInputStream#readFully(byte[],int,int)
     */
    public int read(byte[] buf, int off, int len) throws IOException {
        if (buf == null) {
            throw new NullPointerException();
        }
        int endoff = off + len;
        if (off < 0 || len < 0 || endoff > buf.length || endoff < 0) {
            throw new IndexOutOfBoundsException();
        }
        return bin.read(buf, off, len, false);
    }

    /**
     * Returns the number of bytes that can be read without blocking.
     *
     * @return  the number of available bytes.
     * @throws  IOException if there are I/O errors while reading from the
     *          underlying <code>InputStream</code>
     */
    public int available() throws IOException {
        return bin.available();
    }
```

```
/**
 * Closes the input stream. Must be called to release any resources
 * associated with the stream.
 *
 * @throws  IOException If an I/O error has occurred.
 */
public void close() throws IOException {
    /*
     * Even if stream already closed, propagate redundant close to
     * underlying stream to stay consistent with previous implementations.
     */
    closed = true;
    if (depth == 0) {
        clear();
    }
    bin.close();
}

/**
 * Reads in a boolean.
 *
 * @return  the boolean read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public boolean readBoolean() throws IOException {
    return bin.readBoolean();
}

/**
 * Reads an 8 bit byte.
 *
 * @return  the 8 bit byte read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public byte readByte() throws IOException  {
    return bin.readByte();
}

/**
 * Reads an unsigned 8 bit byte.
 *
 * @return  the 8 bit byte read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public int readUnsignedByte()  throws IOException {
    return bin.readUnsignedByte();
}

/**
 * Reads a 16 bit char.
 *
 * @return  the 16 bit char read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public char readChar()  throws IOException {
    return bin.readChar();
}
```

```
/**
 * Reads a 16 bit short.
 *
 * @return  the 16 bit short read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public short readShort()  throws IOException {
    return bin.readShort();
}

/**
 * Reads an unsigned 16 bit short.
 *
 * @return  the 16 bit short read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public int readUnsignedShort() throws IOException {
    return bin.readUnsignedShort();
}

/**
 * Reads a 32 bit int.
 *
 * @return  the 32 bit integer read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public int readInt()  throws IOException {
    return bin.readInt();
}

/**
 * Reads a 64 bit long.
 *
 * @return  the read 64 bit long.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public long readLong()  throws IOException {
    return bin.readLong();
}

/**
 * Reads a 32 bit float.
 *
 * @return  the 32 bit float read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public float readFloat() throws IOException {
    return bin.readFloat();
}

/**
 * Reads a 64 bit double.
 *
 * @return  the 64 bit double read.
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public double readDouble() throws IOException {
```

```
    return bin.readDouble();
}

/**
 * Reads bytes, blocking until all bytes are read.
 *
 * @param   buf the buffer into which the data is read
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public void readFully(byte[] buf) throws IOException {
    bin.readFully(buf, 0, buf.length, false);
}

/**
 * Reads bytes, blocking until all bytes are read.
 *
 * @param   buf the buffer into which the data is read
 * @param   off the start offset of the data
 * @param   len the maximum number of bytes to read
 * @throws  EOFException If end of file is reached.
 * @throws  IOException If other I/O error has occurred.
 */
public void readFully(byte[] buf, int off, int len) throws IOException {
    int endoff = off + len;
    if (off < 0 || len < 0 || endoff > buf.length || endoff < 0) {
        throw new IndexOutOfBoundsException();
    }
    bin.readFully(buf, off, len, false);
}

/**
 * Skips bytes.
 *
 * @param   len the number of bytes to be skipped
 * @return  the actual number of bytes skipped.
 * @throws  IOException If an I/O error has occurred.
 */
public int skipBytes(int len) throws IOException {
    return bin.skipBytes(len);
}

/**
 * Reads in a line that has been terminated by a \n, \r, \r\n or EOF.
 *
 * @return  a String copy of the line.
 * @throws  IOException if there are I/O errors while reading from the
 *          underlying <code>InputStream</code>
 * @deprecated This method does not properly convert bytes to characters.
 *          see DataInputStream for the details and alternatives.
 */
@Deprecated
public String readLine() throws IOException {
    return bin.readLine();
}

/**
 * Reads a String in
 * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
 * format.
 *
 * @return  the String.
 * @throws  IOException if there are I/O errors while reading from the
```

```
     *          underlying <code>InputStream</code>
     * @throws  UTFDataFormatException if read bytes do not represent a valid
     *          modified UTF-8 encoding of a string
     */
    public String readUTF() throws IOException {
        return bin.readUTF();
    }

    /**
     * Provide access to the persistent fields read from the input stream.
     */
    public static abstract class GetField {

        /**
         * Get the ObjectStreamClass that describes the fields in the stream.
         *
         * @return  the descriptor class that describes the serializable fields
         */
        public abstract ObjectStreamClass getObjectStreamClass();

        /**
         * Return true if the named field is defaulted and has no value in this
         * stream.
         *
         * @param  name the name of the field
         * @return true, if and only if the named field is defaulted
         * @throws IOException if there are I/O errors while reading from
         *         the underlying <code>InputStream</code>
         * @throws IllegalArgumentException if <code>name</code> does not
         *         correspond to a serializable field
         */
        public abstract boolean defaulted(String name) throws IOException;

        /**
         * Get the value of the named boolean field from the persistent field.
         *
         * @param  name the name of the field
         * @param  val the default value to use if <code>name</code> does not
         *         have a value
         * @return the value of the named <code>boolean</code> field
         * @throws IOException if there are I/O errors while reading from the
         *         underlying <code>InputStream</code>
         * @throws IllegalArgumentException if type of <code>name</code> is
         *         not serializable or if the field type is incorrect
         */
        public abstract boolean get(String name, boolean val)
            throws IOException;

        /**
         * Get the value of the named byte field from the persistent field.
         *
         * @param  name the name of the field
         * @param  val the default value to use if <code>name</code> does not
         *         have a value
         * @return the value of the named <code>byte</code> field
         * @throws IOException if there are I/O errors while reading from the
         *         underlying <code>InputStream</code>
         * @throws IllegalArgumentException if type of <code>name</code> is
         *         not serializable or if the field type is incorrect
         */
        public abstract byte get(String name, byte val) throws IOException;

        /**
```

```
     * Get the value of the named char field from the persistent field.
     *
     * @param  name the name of the field
     * @param  val the default value to use if <code>name</code> does not
     *         have a value
     * @return the value of the named <code>char</code> field
     * @throws IOException if there are I/O errors while reading from the
     *         underlying <code>InputStream</code>
     * @throws IllegalArgumentException if type of <code>name</code> is
     *         not serializable or if the field type is incorrect
     */
    public abstract char get(String name, char val) throws IOException;

    /**
     * Get the value of the named short field from the persistent field.
     *
     * @param  name the name of the field
     * @param  val the default value to use if <code>name</code> does not
     *         have a value
     * @return the value of the named <code>short</code> field
     * @throws IOException if there are I/O errors while reading from the
     *         underlying <code>InputStream</code>
     * @throws IllegalArgumentException if type of <code>name</code> is
     *         not serializable or if the field type is incorrect
     */
    public abstract short get(String name, short val) throws IOException;

    /**
     * Get the value of the named int field from the persistent field.
     *
     * @param  name the name of the field
     * @param  val the default value to use if <code>name</code> does not
     *         have a value
     * @return the value of the named <code>int</code> field
     * @throws IOException if there are I/O errors while reading from the
     *         underlying <code>InputStream</code>
     * @throws IllegalArgumentException if type of <code>name</code> is
     *         not serializable or if the field type is incorrect
     */
    public abstract int get(String name, int val) throws IOException;

    /**
     * Get the value of the named long field from the persistent field.
     *
     * @param  name the name of the field
     * @param  val the default value to use if <code>name</code> does not
     *         have a value
     * @return the value of the named <code>long</code> field
     * @throws IOException if there are I/O errors while reading from the
     *         underlying <code>InputStream</code>
     * @throws IllegalArgumentException if type of <code>name</code> is
     *         not serializable or if the field type is incorrect
     */
    public abstract long get(String name, long val) throws IOException;

    /**
     * Get the value of the named float field from the persistent field.
     *
     * @param  name the name of the field
     * @param  val the default value to use if <code>name</code> does not
     *         have a value
     * @return the value of the named <code>float</code> field
     * @throws IOException if there are I/O errors while reading from the
```

```
         *         underlying <code>InputStream</code>
         * @throws IllegalArgumentException if type of <code>name</code> is
         *         not serializable or if the field type is incorrect
         */
        public abstract float get(String name, float val) throws IOException;

        /**
         * Get the value of the named double field from the persistent field.
         *
         * @param  name the name of the field
         * @param  val the default value to use if <code>name</code> does not
         *         have a value
         * @return the value of the named <code>double</code> field
         * @throws IOException if there are I/O errors while reading from the
         *         underlying <code>InputStream</code>
         * @throws IllegalArgumentException if type of <code>name</code> is
         *         not serializable or if the field type is incorrect
         */
        public abstract double get(String name, double val) throws IOException;

        /**
         * Get the value of the named Object field from the persistent field.
         *
         * @param  name the name of the field
         * @param  val the default value to use if <code>name</code> does not
         *         have a value
         * @return the value of the named <code>Object</code> field
         * @throws IOException if there are I/O errors while reading from the
         *         underlying <code>InputStream</code>
         * @throws IllegalArgumentException if type of <code>name</code> is
         *         not serializable or if the field type is incorrect
         */
        public abstract Object get(String name, Object val) throws IOException;
    }

    /**
     * Verifies that this (possibly subclass) instance can be constructed
     * without violating security constraints: the subclass must not override
     * security-sensitive non-final methods, or else the
     * "enableSubclassImplementation" SerializablePermission is checked.
     */
    private void verifySubclass() {
        Class<?> cl = getClass();
        if (cl == ObjectInputStream.class) {
            return;
        }
        SecurityManager sm = System.getSecurityManager();
        if (sm == null) {
            return;
        }
        processQueue(Caches.subclassAuditsQueue, Caches.subclassAudits);
        WeakClassKey key = new WeakClassKey(cl, Caches.subclassAuditsQueue);
        Boolean result = Caches.subclassAudits.get(key);
        if (result == null) {
            result = Boolean.valueOf(auditSubclass(cl));
            Caches.subclassAudits.putIfAbsent(key, result);
        }
        if (result.booleanValue()) {
            return;
        }
        sm.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSION);
    }
```

```
/**
 * Performs reflective checks on given subclass to verify that it doesn't
 * override security-sensitive non-final methods.  Returns true if subclass
 * is "safe", false otherwise.
 */
private static boolean auditSubclass(final Class<?> subcl) {
    Boolean result = AccessController.doPrivileged(
        new PrivilegedAction<Boolean>() {
            public Boolean run() {
                for (Class<?> cl = subcl;
                     cl != ObjectInputStream.class;
                     cl = cl.getSuperclass())
                {
                    try {
                        cl.getDeclaredMethod(
                            "readUnshared", (Class[]) null);
                        return Boolean.FALSE;
                    } catch (NoSuchMethodException ex) {
                    }
                    try {
                        cl.getDeclaredMethod("readFields", (Class[]) null);
                        return Boolean.FALSE;
                    } catch (NoSuchMethodException ex) {
                    }
                }
                return Boolean.TRUE;
            }
        }
    );
    return result.booleanValue();
}

/**
 * Clears internal data structures.
 */
private void clear() {
    handles.clear();
    vlist.clear();
}

/**
 * Underlying readObject implementation.
 */
private Object readObject0(boolean unshared) throws IOException {
    boolean oldMode = bin.getBlockDataMode();
    if (oldMode) {
        int remain = bin.currentBlockRemaining();
        if (remain > 0) {
            throw new OptionalDataException(remain);
        } else if (defaultDataEnd) {
            /*
             * Fix for 4360508: stream is currently at the end of a field
             * value block written via default serialization; since there
             * is no terminating TC_ENDBLOCKDATA tag, simulate
             * end-of-custom-data behavior explicitly.
             */
            throw new OptionalDataException(true);
        }
        bin.setBlockDataMode(false);
    }

    byte tc;
    while ((tc = bin.peekByte()) == TC_RESET) {
```

```
    bin.readByte();
    handleReset();
}

depth++;
try {
    switch (tc) {
        case TC_NULL:
            return readNull();

        case TC_REFERENCE:
            return readHandle(unshared);

        case TC_CLASS:
            return readClass(unshared);

        case TC_CLASSDESC:
        case TC_PROXYCLASSDESC:
            return readClassDesc(unshared);

        case TC_STRING:
        case TC_LONGSTRING:
            return checkResolve(readString(unshared));

        case TC_ARRAY:
            return checkResolve(readArray(unshared));

        case TC_ENUM:
            return checkResolve(readEnum(unshared));

        case TC_OBJECT:
            return checkResolve(readOrdinaryObject(unshared));

        case TC_EXCEPTION:
            IOException ex = readFatalException();
            throw new WriteAbortedException("writing aborted", ex);

        case TC_BLOCKDATA:
        case TC_BLOCKDATALONG:
            if (oldMode) {
                bin.setBlockDataMode(true);
                bin.peek();             // force header read
                throw new OptionalDataException(
                    bin.currentBlockRemaining());
            } else {
                throw new StreamCorruptedException(
                    "unexpected block data");
            }

        case TC_ENDBLOCKDATA:
            if (oldMode) {
                throw new OptionalDataException(true);
            } else {
                throw new StreamCorruptedException(
                    "unexpected end of block data");
            }

        default:
            throw new StreamCorruptedException(
                String.format("invalid type code: %02X", tc));
    }
} finally {
    depth--;
```

```
            bin.setBlockDataMode(oldMode);
        }
    }

    /**
     * If resolveObject has been enabled and given object does not have an
     * exception associated with it, calls resolveObject to determine
     * replacement for object, and updates handle table accordingly.  Returns
     * replacement object, or echoes provided object if no replacement
     * occurred.  Expects that passHandle is set to given object's handle prior
     * to calling this method.
     */
    private Object checkResolve(Object obj) throws IOException {
        if (!enableResolve || handles.lookupException(passHandle) != null) {
            return obj;
        }
        Object rep = resolveObject(obj);
        if (rep != obj) {
            handles.setObject(passHandle, rep);
        }
        return rep;
    }

    /**
     * Reads string without allowing it to be replaced in stream.  Called from
     * within ObjectStreamClass.read().
     */
    String readTypeString() throws IOException {
        int oldHandle = passHandle;
        try {
            byte tc = bin.peekByte();
            switch (tc) {
                case TC_NULL:
                    return (String) readNull();

                case TC_REFERENCE:
                    return (String) readHandle(false);

                case TC_STRING:
                case TC_LONGSTRING:
                    return readString(false);

                default:
                    throw new StreamCorruptedException(
                        String.format("invalid type code: %02X", tc));
            }
        } finally {
            passHandle = oldHandle;
        }
    }

    /**
     * Reads in null code, sets passHandle to NULL_HANDLE and returns null.
     */
    private Object readNull() throws IOException {
        if (bin.readByte() != TC_NULL) {
            throw new InternalError();
        }
        passHandle = NULL_HANDLE;
        return null;
    }

    /**
```

```
     * Reads in object handle, sets passHandle to the read handle, and returns
     * object associated with the handle.
     */
    private Object readHandle(boolean unshared) throws IOException {
        if (bin.readByte() != TC_REFERENCE) {
            throw new InternalError();
        }
        passHandle = bin.readInt() - baseWireHandle;
        if (passHandle < 0 || passHandle >= handles.size()) {
            throw new StreamCorruptedException(
                String.format("invalid handle value: %08X", passHandle +
                baseWireHandle));
        }
        if (unshared) {
            // REMIND: what type of exception to throw here?
            throw new InvalidObjectException(
                "cannot read back reference as unshared");
        }

        Object obj = handles.lookupObject(passHandle);
        if (obj == unsharedMarker) {
            // REMIND: what type of exception to throw here?
            throw new InvalidObjectException(
                "cannot read back reference to unshared object");
        }
        return obj;
    }

    /**
     * Reads in and returns class object.  Sets passHandle to class object's
     * assigned handle.  Returns null if class is unresolvable (in which case a
     * ClassNotFoundException will be associated with the class' handle in the
     * handle table).
     */
    private Class<?> readClass(boolean unshared) throws IOException {
        if (bin.readByte() != TC_CLASS) {
            throw new InternalError();
        }
        ObjectStreamClass desc = readClassDesc(false);
        Class<?> cl = desc.forClass();
        passHandle = handles.assign(unshared ? unsharedMarker : cl);

        ClassNotFoundException resolveEx = desc.getResolveException();
        if (resolveEx != null) {
            handles.markException(passHandle, resolveEx);
        }

        handles.finish(passHandle);
        return cl;
    }

    /**
     * Reads in and returns (possibly null) class descriptor.  Sets passHandle
     * to class descriptor's assigned handle.  If class descriptor cannot be
     * resolved to a class in the local VM, a ClassNotFoundException is
     * associated with the class descriptor's handle.
     */
    private ObjectStreamClass readClassDesc(boolean unshared)
        throws IOException
    {
        byte tc = bin.peekByte();
        switch (tc) {
            case TC_NULL:
```

```
            return (ObjectStreamClass) readNull();

        case TC_REFERENCE:
            return (ObjectStreamClass) readHandle(unshared);

        case TC_PROXYCLASSDESC:
            return readProxyDesc(unshared);

        case TC_CLASSDESC:
            return readNonProxyDesc(unshared);

        default:
            throw new StreamCorruptedException(
                String.format("invalid type code: %02X", tc));
    }
}

private boolean isCustomSubclass() {
    // Return true if this class is a custom subclass of ObjectInputStream
    return getClass().getClassLoader()
                != ObjectInputStream.class.getClassLoader();
}

/**
 * Reads in and returns class descriptor for a dynamic proxy class.  Sets
 * passHandle to proxy class descriptor's assigned handle.  If proxy class
 * descriptor cannot be resolved to a class in the local VM, a
 * ClassNotFoundException is associated with the descriptor's handle.
 */
private ObjectStreamClass readProxyDesc(boolean unshared)
    throws IOException
{
    if (bin.readByte() != TC_PROXYCLASSDESC) {
        throw new InternalError();
    }

    ObjectStreamClass desc = new ObjectStreamClass();
    int descHandle = handles.assign(unshared ? unsharedMarker : desc);
    passHandle = NULL_HANDLE;

    int numIfaces = bin.readInt();
    String[] ifaces = new String[numIfaces];
    for (int i = 0; i < numIfaces; i++) {
        ifaces[i] = bin.readUTF();
    }

    Class<?> cl = null;
    ClassNotFoundException resolveEx = null;
    bin.setBlockDataMode(true);
    try {
        if ((cl = resolveProxyClass(ifaces)) == null) {
            resolveEx = new ClassNotFoundException("null class");
        } else if (!Proxy.isProxyClass(cl)) {
            throw new InvalidClassException("Not a proxy");
        } else {
            // ReflectUtil.checkProxyPackageAccess makes a test
            // equivalent to isCustomSubclass so there's no need
            // to condition this call to isCustomSubclass == true here.
            ReflectUtil.checkProxyPackageAccess(
                    getClass().getClassLoader(),
                    cl.getInterfaces());
        }
    } catch (ClassNotFoundException ex) {
```

```
            resolveEx = ex;
        }
        skipCustomData();

        desc.initProxy(cl, resolveEx, readClassDesc(false));

        handles.finish(descHandle);
        passHandle = descHandle;
        return desc;
    }

    /**
     * Reads in and returns class descriptor for a class that is not a dynamic
     * proxy class.  Sets passHandle to class descriptor's assigned handle.  If
     * class descriptor cannot be resolved to a class in the local VM, a
     * ClassNotFoundException is associated with the descriptor's handle.
     */
    private ObjectStreamClass readNonProxyDesc(boolean unshared)
        throws IOException
    {
        if (bin.readByte() != TC_CLASSDESC) {
            throw new InternalError();
        }

        ObjectStreamClass desc = new ObjectStreamClass();
        int descHandle = handles.assign(unshared ? unsharedMarker : desc);
        passHandle = NULL_HANDLE;

        ObjectStreamClass readDesc = null;
        try {
            readDesc = readClassDescriptor();
        } catch (ClassNotFoundException ex) {
            throw (IOException) new InvalidClassException(
                "failed to read class descriptor").initCause(ex);
        }

        Class<?> cl = null;
        ClassNotFoundException resolveEx = null;
        bin.setBlockDataMode(true);
        final boolean checksRequired = isCustomSubclass();
        try {
            if ((cl = resolveClass(readDesc)) == null) {
                resolveEx = new ClassNotFoundException("null class");
            } else if (checksRequired) {
                ReflectUtil.checkPackageAccess(cl);
            }
        } catch (ClassNotFoundException ex) {
            resolveEx = ex;
        }
        skipCustomData();

        desc.initNonProxy(readDesc, cl, resolveEx, readClassDesc(false));

        handles.finish(descHandle);
        passHandle = descHandle;
        return desc;
    }

    /**
     * Reads in and returns new string.  Sets passHandle to new string's
     * assigned handle.
     */
    private String readString(boolean unshared) throws IOException {
```

```
    String str;
    byte tc = bin.readByte();
    switch (tc) {
        case TC_STRING:
            str = bin.readUTF();
            break;

        case TC_LONGSTRING:
            str = bin.readLongUTF();
            break;

        default:
            throw new StreamCorruptedException(
                String.format("invalid type code: %02X", tc));
    }
    passHandle = handles.assign(unshared ? unsharedMarker : str);
    handles.finish(passHandle);
    return str;
}

/**
 * Reads in and returns array object, or null if array class is
 * unresolvable.  Sets passHandle to array's assigned handle.
 */
private Object readArray(boolean unshared) throws IOException {
    if (bin.readByte() != TC_ARRAY) {
        throw new InternalError();
    }

    ObjectStreamClass desc = readClassDesc(false);
    int len = bin.readInt();

    Object array = null;
    Class<?> cl, ccl = null;
    if ((cl = desc.forClass()) != null) {
        ccl = cl.getComponentType();
        array = Array.newInstance(ccl, len);
    }

    int arrayHandle = handles.assign(unshared ? unsharedMarker : array);
    ClassNotFoundException resolveEx = desc.getResolveException();
    if (resolveEx != null) {
        handles.markException(arrayHandle, resolveEx);
    }

    if (ccl == null) {
        for (int i = 0; i < len; i++) {
            readObject0(false);
        }
    } else if (ccl.isPrimitive()) {
        if (ccl == Integer.TYPE) {
            bin.readInts((int[]) array, 0, len);
        } else if (ccl == Byte.TYPE) {
            bin.readFully((byte[]) array, 0, len, true);
        } else if (ccl == Long.TYPE) {
            bin.readLongs((long[]) array, 0, len);
        } else if (ccl == Float.TYPE) {
            bin.readFloats((float[]) array, 0, len);
        } else if (ccl == Double.TYPE) {
            bin.readDoubles((double[]) array, 0, len);
        } else if (ccl == Short.TYPE) {
            bin.readShorts((short[]) array, 0, len);
        } else if (ccl == Character.TYPE) {
```

```
            bin.readChars((char[]) array, 0, len);
        } else if (ccl == Boolean.TYPE) {
            bin.readBooleans((boolean[]) array, 0, len);
        } else {
            throw new InternalError();
        }
    } else {
        Object[] oa = (Object[]) array;
        for (int i = 0; i < len; i++) {
            oa[i] = readObject0(false);
            handles.markDependency(arrayHandle, passHandle);
        }
    }

    handles.finish(arrayHandle);
    passHandle = arrayHandle;
    return array;
}

/**
 * Reads in and returns enum constant, or null if enum type is
 * unresolvable.  Sets passHandle to enum constant's assigned handle.
 */
private Enum<?> readEnum(boolean unshared) throws IOException {
    if (bin.readByte() != TC_ENUM) {
        throw new InternalError();
    }

    ObjectStreamClass desc = readClassDesc(false);
    if (!desc.isEnum()) {
        throw new InvalidClassException("non-enum class: " + desc);
    }

    int enumHandle = handles.assign(unshared ? unsharedMarker : null);
    ClassNotFoundException resolveEx = desc.getResolveException();
    if (resolveEx != null) {
        handles.markException(enumHandle, resolveEx);
    }

    String name = readString(false);
    Enum<?> result = null;
    Class<?> cl = desc.forClass();
    if (cl != null) {
        try {
            @SuppressWarnings("unchecked")
            Enum<?> en = Enum.valueOf((Class)cl, name);
            result = en;
        } catch (IllegalArgumentException ex) {
            throw (IOException) new InvalidObjectException(
                "enum constant " + name + " does not exist in " +
                cl).initCause(ex);
        }
        if (!unshared) {
            handles.setObject(enumHandle, result);
        }
    }

    handles.finish(enumHandle);
    passHandle = enumHandle;
    return result;
}

/**
```

```
     * Reads and returns "ordinary" (i.e., not a String, Class,
     * ObjectStreamClass, array, or enum constant) object, or null if object's
     * class is unresolvable (in which case a ClassNotFoundException will be
     * associated with object's handle).  Sets passHandle to object's assigned
     * handle.
     */
    private Object readOrdinaryObject(boolean unshared)
        throws IOException
    {
        if (bin.readByte() != TC_OBJECT) {
            throw new InternalError();
        }

        ObjectStreamClass desc = readClassDesc(false);
        desc.checkDeserialize();

        Class<?> cl = desc.forClass();
        if (cl == String.class || cl == Class.class
                || cl == ObjectStreamClass.class) {
            throw new InvalidClassException("invalid class descriptor");
        }

        Object obj;
        try {
            obj = desc.isInstantiable() ? desc.newInstance() : null;
        } catch (Exception ex) {
            throw (IOException) new InvalidClassException(
                desc.forClass().getName(),
                "unable to create instance").initCause(ex);
        }

        passHandle = handles.assign(unshared ? unsharedMarker : obj);
        ClassNotFoundException resolveEx = desc.getResolveException();
        if (resolveEx != null) {
            handles.markException(passHandle, resolveEx);
        }

        if (desc.isExternalizable()) {
            readExternalData((Externalizable) obj, desc);
        } else {
            readSerialData(obj, desc);
        }

        handles.finish(passHandle);

        if (obj != null &&
            handles.lookupException(passHandle) == null &&
            desc.hasReadResolveMethod())
        {
            Object rep = desc.invokeReadResolve(obj);
            if (unshared && rep.getClass().isArray()) {
                rep = cloneArray(rep);
            }
            if (rep != obj) {
                handles.setObject(passHandle, obj = rep);
            }
        }

        return obj;
    }

    /**
     * If obj is non-null, reads externalizable data by invoking readExternal()
```

```
     * method of obj; otherwise, attempts to skip over externalizable data.
     * Expects that passHandle is set to obj's handle before this method is
     * called.
     */
    private void readExternalData(Externalizable obj, ObjectStreamClass desc)
        throws IOException
    {
        SerialCallbackContext oldContext = curContext;
        curContext = null;
        try {
            boolean blocked = desc.hasBlockExternalData();
            if (blocked) {
                bin.setBlockDataMode(true);
            }
            if (obj != null) {
                try {
                    obj.readExternal(this);
                } catch (ClassNotFoundException ex) {
                    /*
                     * In most cases, the handle table has already propagated
                     * a CNFException to passHandle at this point; this mark
                     * call is included to address cases where the readExternal
                     * method has cons'ed and thrown a new CNFException of its
                     * own.
                     */
                     handles.markException(passHandle, ex);
                }
            }
            if (blocked) {
                skipCustomData();
            }
        } finally {
            curContext = oldContext;
        }
        /*
         * At this point, if the externalizable data was not written in
         * block-data form and either the externalizable class doesn't exist
         * locally (i.e., obj == null) or readExternal() just threw a
         * CNFException, then the stream is probably in an inconsistent state,
         * since some (or all) of the externalizable data may not have been
         * consumed.  Since there's no "correct" action to take in this case,
         * we mimic the behavior of past serialization implementations and
         * blindly hope that the stream is in sync; if it isn't and additional
         * externalizable data remains in the stream, a subsequent read will
         * most likely throw a StreamCorruptedException.
         */
    }

    /**
     * Reads (or attempts to skip, if obj is null or is tagged with a
     * ClassNotFoundException) instance data for each serializable class of
     * object in stream, from superclass to subclass.  Expects that passHandle
     * is set to obj's handle before this method is called.
     */
    private void readSerialData(Object obj, ObjectStreamClass desc)
        throws IOException
    {
        ObjectStreamClass.ClassDataSlot[] slots = desc.getClassDataLayout();
        for (int i = 0; i < slots.length; i++) {
            ObjectStreamClass slotDesc = slots[i].desc;

            if (slots[i].hasData) {
                if (obj != null &&
```

```
                slotDesc.hasReadObjectMethod() &&
                handles.lookupException(passHandle) == null)
            {
                SerialCallbackContext oldContext = curContext;

                try {
                    curContext = new SerialCallbackContext(obj, slotDesc);

                    bin.setBlockDataMode(true);
                    slotDesc.invokeReadObject(obj, this);
                } catch (ClassNotFoundException ex) {
                    /*
                     * In most cases, the handle table has already
                     * propagated a CNFException to passHandle at this
                     * point; this mark call is included to address cases
                     * where the custom readObject method has cons'ed and
                     * thrown a new CNFException of its own.
                     */
                    handles.markException(passHandle, ex);
                } finally {
                    curContext.setUsed();
                    curContext = oldContext;
                }

                /*
                 * defaultDataEnd may have been set indirectly by custom
                 * readObject() method when calling defaultReadObject() or
                 * readFields(); clear it to restore normal read behavior.
                 */
                defaultDataEnd = false;
            } else {
                defaultReadFields(obj, slotDesc);
            }
            if (slotDesc.hasWriteObjectData()) {
                skipCustomData();
            } else {
                bin.setBlockDataMode(false);
            }
        } else {
            if (obj != null &&
                slotDesc.hasReadObjectNoDataMethod() &&
                handles.lookupException(passHandle) == null)
            {
                slotDesc.invokeReadObjectNoData(obj);
            }
        }
    }
}

/**
 * Skips over all block data and objects until TC_ENDBLOCKDATA is
 * encountered.
 */
private void skipCustomData() throws IOException {
    int oldHandle = passHandle;
    for (;;) {
        if (bin.getBlockDataMode()) {
            bin.skipBlockData();
            bin.setBlockDataMode(false);
        }
        switch (bin.peekByte()) {
            case TC_BLOCKDATA:
            case TC_BLOCKDATALONG:
```

```
                bin.setBlockDataMode(true);
                break;

            case TC_ENDBLOCKDATA:
                bin.readByte();
                passHandle = oldHandle;
                return;

            default:
                readObject0(false);
                break;
        }
    }
}

/**
 * Reads in values of serializable fields declared by given class
 * descriptor.  If obj is non-null, sets field values in obj.  Expects that
 * passHandle is set to obj's handle before this method is called.
 */
private void defaultReadFields(Object obj, ObjectStreamClass desc)
    throws IOException
{
    Class<?> cl = desc.forClass();
    if (cl != null && obj != null && !cl.isInstance(obj)) {
        throw new ClassCastException();
    }

    int primDataSize = desc.getPrimDataSize();
    if (primVals == null || primVals.length < primDataSize) {
        primVals = new byte[primDataSize];
    }
    bin.readFully(primVals, 0, primDataSize, false);
    if (obj != null) {
        desc.setPrimFieldValues(obj, primVals);
    }

    int objHandle = passHandle;
    ObjectStreamField[] fields = desc.getFields(false);
    Object[] objVals = new Object[desc.getNumObjFields()];
    int numPrimFields = fields.length - objVals.length;
    for (int i = 0; i < objVals.length; i++) {
        ObjectStreamField f = fields[numPrimFields + i];
        objVals[i] = readObject0(f.isUnshared());
        if (f.getField() != null) {
            handles.markDependency(objHandle, passHandle);
        }
    }
    if (obj != null) {
        desc.setObjFieldValues(obj, objVals);
    }
    passHandle = objHandle;
}

/**
 * Reads in and returns IOException that caused serialization to abort.
 * All stream state is discarded prior to reading in fatal exception.  Sets
 * passHandle to fatal exception's handle.
 */
private IOException readFatalException() throws IOException {
    if (bin.readByte() != TC_EXCEPTION) {
        throw new InternalError();
    }
```

```
    clear();
    return (IOException) readObject0(false);
}

/**
 * If recursion depth is 0, clears internal data structures; otherwise,
 * throws a StreamCorruptedException.  This method is called when a
 * TC_RESET typecode is encountered.
 */
private void handleReset() throws StreamCorruptedException {
    if (depth > 0) {
        throw new StreamCorruptedException(
            "unexpected reset; recursion depth: " + depth);
    }
    clear();
}

/**
 * Converts specified span of bytes into float values.
 */
// REMIND: remove once hotspot inlines Float.intBitsToFloat
private static native void bytesToFloats(byte[] src, int srcpos,
                                         float[] dst, int dstpos,
                                         int nfloats);

/**
 * Converts specified span of bytes into double values.
 */
// REMIND: remove once hotspot inlines Double.longBitsToDouble
private static native void bytesToDoubles(byte[] src, int srcpos,
                                          double[] dst, int dstpos,
                                          int ndoubles);

/**
 * Returns the first non-null class loader (not counting class loaders of
 * generated reflection implementation classes) up the execution stack, or
 * null if only code from the null class loader is on the stack.  This
 * method is also called via reflection by the following RMI-IIOP class:
 *
 *     com.sun.corba.se.internal.util.JDKClassLoader
 *
 * This method should not be removed or its signature changed without
 * corresponding modifications to the above class.
 */
private static ClassLoader latestUserDefinedLoader() {
    return sun.misc.VM.latestUserDefinedLoader();
}

/**
 * Default GetField implementation.
 */
private class GetFieldImpl extends GetField {

    /** class descriptor describing serializable fields */
    private final ObjectStreamClass desc;
    /** primitive field values */
    private final byte[] primVals;
    /** object field values */
    private final Object[] objVals;
    /** object field value handles */
    private final int[] objHandles;

    /**
```

```
     * Creates GetFieldImpl object for reading fields defined in given
     * class descriptor.
     */
    GetFieldImpl(ObjectStreamClass desc) {
        this.desc = desc;
        primVals = new byte[desc.getPrimDataSize()];
        objVals = new Object[desc.getNumObjFields()];
        objHandles = new int[objVals.length];
    }

    public ObjectStreamClass getObjectStreamClass() {
        return desc;
    }

    public boolean defaulted(String name) throws IOException {
        return (getFieldOffset(name, null) < 0);
    }

    public boolean get(String name, boolean val) throws IOException {
        int off = getFieldOffset(name, Boolean.TYPE);
        return (off >= 0) ? Bits.getBoolean(primVals, off) : val;
    }

    public byte get(String name, byte val) throws IOException {
        int off = getFieldOffset(name, Byte.TYPE);
        return (off >= 0) ? primVals[off] : val;
    }

    public char get(String name, char val) throws IOException {
        int off = getFieldOffset(name, Character.TYPE);
        return (off >= 0) ? Bits.getChar(primVals, off) : val;
    }

    public short get(String name, short val) throws IOException {
        int off = getFieldOffset(name, Short.TYPE);
        return (off >= 0) ? Bits.getShort(primVals, off) : val;
    }

    public int get(String name, int val) throws IOException {
        int off = getFieldOffset(name, Integer.TYPE);
        return (off >= 0) ? Bits.getInt(primVals, off) : val;
    }

    public float get(String name, float val) throws IOException {
        int off = getFieldOffset(name, Float.TYPE);
        return (off >= 0) ? Bits.getFloat(primVals, off) : val;
    }

    public long get(String name, long val) throws IOException {
        int off = getFieldOffset(name, Long.TYPE);
        return (off >= 0) ? Bits.getLong(primVals, off) : val;
    }

    public double get(String name, double val) throws IOException {
        int off = getFieldOffset(name, Double.TYPE);
        return (off >= 0) ? Bits.getDouble(primVals, off) : val;
    }

    public Object get(String name, Object val) throws IOException {
        int off = getFieldOffset(name, Object.class);
        if (off >= 0) {
            int objHandle = objHandles[off];
            handles.markDependency(passHandle, objHandle);
```

```
            return (handles.lookupException(objHandle) == null) ?
                objVals[off] : null;
        } else {
            return val;
        }
    }

    /**
     * Reads primitive and object field values from stream.
     */
    void readFields() throws IOException {
        bin.readFully(primVals, 0, primVals.length, false);

        int oldHandle = passHandle;
        ObjectStreamField[] fields = desc.getFields(false);
        int numPrimFields = fields.length - objVals.length;
        for (int i = 0; i < objVals.length; i++) {
            objVals[i] =
                readObject0(fields[numPrimFields + i].isUnshared());
            objHandles[i] = passHandle;
        }
        passHandle = oldHandle;
    }

    /**
     * Returns offset of field with given name and type.  A specified type
     * of null matches all types, Object.class matches all non-primitive
     * types, and any other non-null type matches assignable types only.
     * If no matching field is found in the (incoming) class
     * descriptor but a matching field is present in the associated local
     * class descriptor, returns -1.  Throws IllegalArgumentException if
     * neither incoming nor local class descriptor contains a match.
     */
    private int getFieldOffset(String name, Class<?> type) {
        ObjectStreamField field = desc.getField(name, type);
        if (field != null) {
            return field.getOffset();
        } else if (desc.getLocalDesc().getField(name, type) != null) {
            return -1;
        } else {
            throw new IllegalArgumentException("no such field " + name +
                                               " with type " + type);
        }
    }
}

/**
 * Prioritized list of callbacks to be performed once object graph has been
 * completely deserialized.
 */
private static class ValidationList {

    private static class Callback {
        final ObjectInputValidation obj;
        final int priority;
        Callback next;
        final AccessControlContext acc;

        Callback(ObjectInputValidation obj, int priority, Callback next,
            AccessControlContext acc)
        {
            this.obj = obj;
            this.priority = priority;
```

```
        this.next = next;
        this.acc = acc;
    }
}

/** linked list of callbacks */
private Callback list;

/**
 * Creates new (empty) ValidationList.
 */
ValidationList() {
}

/**
 * Registers callback.  Throws InvalidObjectException if callback
 * object is null.
 */
void register(ObjectInputValidation obj, int priority)
    throws InvalidObjectException
{
    if (obj == null) {
        throw new InvalidObjectException("null callback");
    }

    Callback prev = null, cur = list;
    while (cur != null && priority < cur.priority) {
        prev = cur;
        cur = cur.next;
    }
    AccessControlContext acc = AccessController.getContext();
    if (prev != null) {
        prev.next = new Callback(obj, priority, cur, acc);
    } else {
        list = new Callback(obj, priority, list, acc);
    }
}

/**
 * Invokes all registered callbacks and clears the callback list.
 * Callbacks with higher priorities are called first; those with equal
 * priorities may be called in any order.  If any of the callbacks
 * throws an InvalidObjectException, the callback process is terminated
 * and the exception propagated upwards.
 */
void doCallbacks() throws InvalidObjectException {
    try {
        while (list != null) {
            AccessController.doPrivileged(
                new PrivilegedExceptionAction<Void>()
            {
                public Void run() throws InvalidObjectException {
                    list.obj.validateObject();
                    return null;
                }
            }, list.acc);
            list = list.next;
        }
    } catch (PrivilegedActionException ex) {
        list = null;
        throw (InvalidObjectException) ex.getException();
    }
}
```

```
    /**
     * Resets the callback list to its initial (empty) state.
     */
    public void clear() {
        list = null;
    }
}

/**
 * Input stream supporting single-byte peek operations.
 */
private static class PeekInputStream extends InputStream {

    /** underlying stream */
    private final InputStream in;
    /** peeked byte */
    private int peekb = -1;

    /**
     * Creates new PeekInputStream on top of given underlying stream.
     */
    PeekInputStream(InputStream in) {
        this.in = in;
    }

    /**
     * Peeks at next byte value in stream.  Similar to read(), except
     * that it does not consume the read value.
     */
    int peek() throws IOException {
        return (peekb >= 0) ? peekb : (peekb = in.read());
    }

    public int read() throws IOException {
        if (peekb >= 0) {
            int v = peekb;
            peekb = -1;
            return v;
        } else {
            return in.read();
        }
    }

    public int read(byte[] b, int off, int len) throws IOException {
        if (len == 0) {
            return 0;
        } else if (peekb < 0) {
            return in.read(b, off, len);
        } else {
            b[off++] = (byte) peekb;
            len--;
            peekb = -1;
            int n = in.read(b, off, len);
            return (n >= 0) ? (n + 1) : 1;
        }
    }

    void readFully(byte[] b, int off, int len) throws IOException {
        int n = 0;
        while (n < len) {
            int count = read(b, off + n, len - n);
            if (count < 0) {
```

```
                throw new EOFException();
            }
            n += count;
        }
    }

    public long skip(long n) throws IOException {
        if (n <= 0) {
            return 0;
        }
        int skipped = 0;
        if (peekb >= 0) {
            peekb = -1;
            skipped++;
            n--;
        }
        return skipped + skip(n);
    }

    public int available() throws IOException {
        return in.available() + ((peekb >= 0) ? 1 : 0);
    }

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

/**
 * Input stream with two modes: in default mode, inputs data written in the
 * same format as DataOutputStream; in "block data" mode, inputs data
 * bracketed by block data markers (see object serialization specification
 * for details).  Buffering depends on block data mode: when in default
 * mode, no data is buffered in advance; when in block data mode, all data
 * for the current data block is read in at once (and buffered).
 */
private class BlockDataInputStream
    extends InputStream implements DataInput
{
    /** maximum data block length */
    private static final int MAX_BLOCK_SIZE = 1024;
    /** maximum data block header length */
    private static final int MAX_HEADER_SIZE = 5;
    /** (tunable) length of char buffer (for reading strings) */
    private static final int CHAR_BUF_SIZE = 256;
    /** readBlockHeader() return value indicating header read may block */
    private static final int HEADER_BLOCKED = -2;

    /** buffer for reading general/block data */
    private final byte[] buf = new byte[MAX_BLOCK_SIZE];
    /** buffer for reading block data headers */
    private final byte[] hbuf = new byte[MAX_HEADER_SIZE];
    /** char buffer for fast string reads */
    private final char[] cbuf = new char[CHAR_BUF_SIZE];

    /** block data mode */
    private boolean blkmode = false;

    // block data state fields; values meaningful only when blkmode true
    /** current offset into buf */
    private int pos = 0;
    /** end offset of valid data in buf, or -1 if no more block data */
    private int end = -1;
```

```
/** number of bytes in current block yet to be read from stream */
private int unread = 0;

/** underlying stream (wrapped in peekable filter stream) */
private final PeekInputStream in;
/** loopback stream (for data reads that span data blocks) */
private final DataInputStream din;

/**
 * Creates new BlockDataInputStream on top of given underlying stream.
 * Block data mode is turned off by default.
 */
BlockDataInputStream(InputStream in) {
    this.in = new PeekInputStream(in);
    din = new DataInputStream(this);
}

/**
 * Sets block data mode to the given mode (true == on, false == off)
 * and returns the previous mode value.  If the new mode is the same as
 * the old mode, no action is taken.  Throws IllegalStateException if
 * block data mode is being switched from on to off while unconsumed
 * block data is still present in the stream.
 */
boolean setBlockDataMode(boolean newmode) throws IOException {
    if (blkmode == newmode) {
        return blkmode;
    }
    if (newmode) {
        pos = 0;
        end = 0;
        unread = 0;
    } else if (pos < end) {
        throw new IllegalStateException("unread block data");
    }
    blkmode = newmode;
    return !blkmode;
}

/**
 * Returns true if the stream is currently in block data mode, false
 * otherwise.
 */
boolean getBlockDataMode() {
    return blkmode;
}

/**
 * If in block data mode, skips to the end of the current group of data
 * blocks (but does not unset block data mode).  If not in block data
 * mode, throws an IllegalStateException.
 */
void skipBlockData() throws IOException {
    if (!blkmode) {
        throw new IllegalStateException("not in block data mode");
    }
    while (end >= 0) {
        refill();
    }
}

/**
 * Attempts to read in the next block data header (if any).  If
```

```
     * canBlock is false and a full header cannot be read without possibly
     * blocking, returns HEADER_BLOCKED, else if the next element in the
     * stream is a block data header, returns the block data length
     * specified by the header, else returns -1.
     */
    private int readBlockHeader(boolean canBlock) throws IOException {
        if (defaultDataEnd) {
            /*
             * Fix for 4360508: stream is currently at the end of a field
             * value block written via default serialization; since there
             * is no terminating TC_ENDBLOCKDATA tag, simulate
             * end-of-custom-data behavior explicitly.
             */
            return -1;
        }
        try {
            for (;;) {
                int avail = canBlock ? Integer.MAX_VALUE : in.available();
                if (avail == 0) {
                    return HEADER_BLOCKED;
                }

                int tc = in.peek();
                switch (tc) {
                    case TC_BLOCKDATA:
                        if (avail < 2) {
                            return HEADER_BLOCKED;
                        }
                        in.readFully(hbuf, 0, 2);
                        return hbuf[1] & 0xFF;

                    case TC_BLOCKDATALONG:
                        if (avail < 5) {
                            return HEADER_BLOCKED;
                        }
                        in.readFully(hbuf, 0, 5);
                        int len = Bits.getInt(hbuf, 1);
                        if (len < 0) {
                            throw new StreamCorruptedException(
                                "illegal block data header length: " +
                                len);
                        }
                        return len;

                    /*
                     * TC_RESETs may occur in between data blocks.
                     * Unfortunately, this case must be parsed at a lower
                     * level than other typecodes, since primitive data
                     * reads may span data blocks separated by a TC_RESET.
                     */
                    case TC_RESET:
                        in.read();
                        handleReset();
                        break;

                    default:
                        if (tc >= 0 && (tc < TC_BASE || tc > TC_MAX)) {
                            throw new StreamCorruptedException(
                                String.format("invalid type code: %02X",
                                tc));
                        }
                        return -1;
                }
```

```
        }
    } catch (EOFException ex) {
        throw new StreamCorruptedException(
            "unexpected EOF while reading block data header");
    }
}

/**
 * Refills internal buffer buf with block data.  Any data in buf at the
 * time of the call is considered consumed.  Sets the pos, end, and
 * unread fields to reflect the new amount of available block data; if
 * the next element in the stream is not a data block, sets pos and
 * unread to 0 and end to -1.
 */
private void refill() throws IOException {
    try {
        do {
            pos = 0;
            if (unread > 0) {
                int n =
                    in.read(buf, 0, Math.min(unread, MAX_BLOCK_SIZE));
                if (n >= 0) {
                    end = n;
                    unread -= n;
                } else {
                    throw new StreamCorruptedException(
                        "unexpected EOF in middle of data block");
                }
            } else {
                int n = readBlockHeader(true);
                if (n >= 0) {
                    end = 0;
                    unread = n;
                } else {
                    end = -1;
                    unread = 0;
                }
            }
        } while (pos == end);
    } catch (IOException ex) {
        pos = 0;
        end = -1;
        unread = 0;
        throw ex;
    }
}

/**
 * If in block data mode, returns the number of unconsumed bytes
 * remaining in the current data block.  If not in block data mode,
 * throws an IllegalStateException.
 */
int currentBlockRemaining() {
    if (blkmode) {
        return (end >= 0) ? (end - pos) + unread : 0;
    } else {
        throw new IllegalStateException();
    }
}

/**
 * Peeks at (but does not consume) and returns the next byte value in
 * the stream, or -1 if the end of the stream/block data (if in block
```

```
     * data mode) has been reached.
     */
    int peek() throws IOException {
        if (blkmode) {
            if (pos == end) {
                refill();
            }
            return (end >= 0) ? (buf[pos] & 0xFF) : -1;
        } else {
            return in.peek();
        }
    }

    /**
     * Peeks at (but does not consume) and returns the next byte value in
     * the stream, or throws EOFException if end of stream/block data has
     * been reached.
     */
    byte peekByte() throws IOException {
        int val = peek();
        if (val < 0) {
            throw new EOFException();
        }
        return (byte) val;
    }


    /* ----------------- generic input stream methods ------------------ */
    /*
     * The following methods are equivalent to their counterparts in
     * InputStream, except that they interpret data block boundaries and
     * read the requested data from within data blocks when in block data
     * mode.
     */

    public int read() throws IOException {
        if (blkmode) {
            if (pos == end) {
                refill();
            }
            return (end >= 0) ? (buf[pos++] & 0xFF) : -1;
        } else {
            return in.read();
        }
    }

    public int read(byte[] b, int off, int len) throws IOException {
        return read(b, off, len, false);
    }

    public long skip(long len) throws IOException {
        long remain = len;
        while (remain > 0) {
            if (blkmode) {
                if (pos == end) {
                    refill();
                }
                if (end < 0) {
                    break;
                }
                int nread = (int) Math.min(remain, end - pos);
                remain -= nread;
                pos += nread;
```

```
        } else {
            int nread = (int) Math.min(remain, MAX_BLOCK_SIZE);
            if ((nread = in.read(buf, 0, nread)) < 0) {
                break;
            }
            remain -= nread;
        }
    }
    return len - remain;
}

public int available() throws IOException {
    if (blkmode) {
        if ((pos == end) && (unread == 0)) {
            int n;
            while ((n = readBlockHeader(false)) == 0) ;
            switch (n) {
                case HEADER_BLOCKED:
                    break;

                case -1:
                    pos = 0;
                    end = -1;
                    break;

                default:
                    pos = 0;
                    end = 0;
                    unread = n;
                    break;
            }
        }
        // avoid unnecessary call to in.available() if possible
        int unreadAvail = (unread > 0) ?
            Math.min(in.available(), unread) : 0;
        return (end >= 0) ? (end - pos) + unreadAvail : 0;
    } else {
        return in.available();
    }
}

public void close() throws IOException {
    if (blkmode) {
        pos = 0;
        end = -1;
        unread = 0;
    }
    in.close();
}

/**
 * Attempts to read len bytes into byte array b at offset off.  Returns
 * the number of bytes read, or -1 if the end of stream/block data has
 * been reached.  If copy is true, reads values into an intermediate
 * buffer before copying them to b (to avoid exposing a reference to
 * b).
 */
int read(byte[] b, int off, int len, boolean copy) throws IOException {
    if (len == 0) {
        return 0;
    } else if (blkmode) {
        if (pos == end) {
            refill();
```

```
        }
        if (end < 0) {
            return -1;
        }
        int nread = Math.min(len, end - pos);
        System.arraycopy(buf, pos, b, off, nread);
        pos += nread;
        return nread;
    } else if (copy) {
        int nread = in.read(buf, 0, Math.min(len, MAX_BLOCK_SIZE));
        if (nread > 0) {
            System.arraycopy(buf, 0, b, off, nread);
        }
        return nread;
    } else {
        return in.read(b, off, len);
    }
}

/* ----------------- primitive data input methods ------------------ */
/*
 * The following methods are equivalent to their counterparts in
 * DataInputStream, except that they interpret data block boundaries
 * and read the requested data from within data blocks when in block
 * data mode.
 */

public void readFully(byte[] b) throws IOException {
    readFully(b, 0, b.length, false);
}

public void readFully(byte[] b, int off, int len) throws IOException {
    readFully(b, off, len, false);
}

public void readFully(byte[] b, int off, int len, boolean copy)
    throws IOException
{
    while (len > 0) {
        int n = read(b, off, len, copy);
        if (n < 0) {
            throw new EOFException();
        }
        off += n;
        len -= n;
    }
}

public int skipBytes(int n) throws IOException {
    return din.skipBytes(n);
}

public boolean readBoolean() throws IOException {
    int v = read();
    if (v < 0) {
        throw new EOFException();
    }
    return (v != 0);
}

public byte readByte() throws IOException {
    int v = read();
    if (v < 0) {
```

```
            throw new EOFException();
        }
        return (byte) v;
    }

    public int readUnsignedByte() throws IOException {
        int v = read();
        if (v < 0) {
            throw new EOFException();
        }
        return v;
    }

    public char readChar() throws IOException {
        if (!blkmode) {
            pos = 0;
            in.readFully(buf, 0, 2);
        } else if (end - pos < 2) {
            return din.readChar();
        }
        char v = Bits.getChar(buf, pos);
        pos += 2;
        return v;
    }

    public short readShort() throws IOException {
        if (!blkmode) {
            pos = 0;
            in.readFully(buf, 0, 2);
        } else if (end - pos < 2) {
            return din.readShort();
        }
        short v = Bits.getShort(buf, pos);
        pos += 2;
        return v;
    }

    public int readUnsignedShort() throws IOException {
        if (!blkmode) {
            pos = 0;
            in.readFully(buf, 0, 2);
        } else if (end - pos < 2) {
            return din.readUnsignedShort();
        }
        int v = Bits.getShort(buf, pos) & 0xFFFF;
        pos += 2;
        return v;
    }

    public int readInt() throws IOException {
        if (!blkmode) {
            pos = 0;
            in.readFully(buf, 0, 4);
        } else if (end - pos < 4) {
            return din.readInt();
        }
        int v = Bits.getInt(buf, pos);
        pos += 4;
        return v;
    }

    public float readFloat() throws IOException {
        if (!blkmode) {
```

```
            pos = 0;
            in.readFully(buf, 0, 4);
        } else if (end - pos < 4) {
            return din.readFloat();
        }
        float v = Bits.getFloat(buf, pos);
        pos += 4;
        return v;
    }

    public long readLong() throws IOException {
        if (!blkmode) {
            pos = 0;
            in.readFully(buf, 0, 8);
        } else if (end - pos < 8) {
            return din.readLong();
        }
        long v = Bits.getLong(buf, pos);
        pos += 8;
        return v;
    }

    public double readDouble() throws IOException {
        if (!blkmode) {
            pos = 0;
            in.readFully(buf, 0, 8);
        } else if (end - pos < 8) {
            return din.readDouble();
        }
        double v = Bits.getDouble(buf, pos);
        pos += 8;
        return v;
    }

    public String readUTF() throws IOException {
        return readUTFBody(readUnsignedShort());
    }

    @SuppressWarnings("deprecation")
    public String readLine() throws IOException {
        return din.readLine();      // deprecated, not worth optimizing
    }

    /* -------------- primitive data array input methods --------------- */
    /*
     * The following methods read in spans of primitive data values.
     * Though equivalent to calling the corresponding primitive read
     * methods repeatedly, these methods are optimized for reading groups
     * of primitive data values more efficiently.
     */

    void readBooleans(boolean[] v, int off, int len) throws IOException {
        int stop, endoff = off + len;
        while (off < endoff) {
            if (!blkmode) {
                int span = Math.min(endoff - off, MAX_BLOCK_SIZE);
                in.readFully(buf, 0, span);
                stop = off + span;
                pos = 0;
            } else if (end - pos < 1) {
                v[off++] = din.readBoolean();
                continue;
            } else {
```

```
            stop = Math.min(endoff, off + end - pos);
        }

        while (off < stop) {
            v[off++] = Bits.getBoolean(buf, pos++);
        }
    }
}

void readChars(char[] v, int off, int len) throws IOException {
    int stop, endoff = off + len;
    while (off < endoff) {
        if (!blkmode) {
            int span = Math.min(endoff - off, MAX_BLOCK_SIZE >> 1);
            in.readFully(buf, 0, span << 1);
            stop = off + span;
            pos = 0;
        } else if (end - pos < 2) {
            v[off++] = din.readChar();
            continue;
        } else {
            stop = Math.min(endoff, off + ((end - pos) >> 1));
        }

        while (off < stop) {
            v[off++] = Bits.getChar(buf, pos);
            pos += 2;
        }
    }
}

void readShorts(short[] v, int off, int len) throws IOException {
    int stop, endoff = off + len;
    while (off < endoff) {
        if (!blkmode) {
            int span = Math.min(endoff - off, MAX_BLOCK_SIZE >> 1);
            in.readFully(buf, 0, span << 1);
            stop = off + span;
            pos = 0;
        } else if (end - pos < 2) {
            v[off++] = din.readShort();
            continue;
        } else {
            stop = Math.min(endoff, off + ((end - pos) >> 1));
        }

        while (off < stop) {
            v[off++] = Bits.getShort(buf, pos);
            pos += 2;
        }
    }
}

void readInts(int[] v, int off, int len) throws IOException {
    int stop, endoff = off + len;
    while (off < endoff) {
        if (!blkmode) {
            int span = Math.min(endoff - off, MAX_BLOCK_SIZE >> 2);
            in.readFully(buf, 0, span << 2);
            stop = off + span;
            pos = 0;
        } else if (end - pos < 4) {
            v[off++] = din.readInt();
```

```
            continue;
        } else {
            stop = Math.min(endoff, off + ((end - pos) >> 2));
        }

        while (off < stop) {
            v[off++] = Bits.getInt(buf, pos);
            pos += 4;
        }
    }
}

void readFloats(float[] v, int off, int len) throws IOException {
    int span, endoff = off + len;
    while (off < endoff) {
        if (!blkmode) {
            span = Math.min(endoff - off, MAX_BLOCK_SIZE >> 2);
            in.readFully(buf, 0, span << 2);
            pos = 0;
        } else if (end - pos < 4) {
            v[off++] = din.readFloat();
            continue;
        } else {
            span = Math.min(endoff - off, ((end - pos) >> 2));
        }

        bytesToFloats(buf, pos, v, off, span);
        off += span;
        pos += span << 2;
    }
}

void readLongs(long[] v, int off, int len) throws IOException {
    int stop, endoff = off + len;
    while (off < endoff) {
        if (!blkmode) {
            int span = Math.min(endoff - off, MAX_BLOCK_SIZE >> 3);
            in.readFully(buf, 0, span << 3);
            stop = off + span;
            pos = 0;
        } else if (end - pos < 8) {
            v[off++] = din.readLong();
            continue;
        } else {
            stop = Math.min(endoff, off + ((end - pos) >> 3));
        }

        while (off < stop) {
            v[off++] = Bits.getLong(buf, pos);
            pos += 8;
        }
    }
}

void readDoubles(double[] v, int off, int len) throws IOException {
    int span, endoff = off + len;
    while (off < endoff) {
        if (!blkmode) {
            span = Math.min(endoff - off, MAX_BLOCK_SIZE >> 3);
            in.readFully(buf, 0, span << 3);
            pos = 0;
        } else if (end - pos < 8) {
            v[off++] = din.readDouble();
```

```
            continue;
        } else {
            span = Math.min(endoff - off, ((end - pos) >> 3));
        }

        bytesToDoubles(buf, pos, v, off, span);
        off += span;
        pos += span << 3;
    }
}

/**
 * Reads in string written in "long" UTF format.  "Long" UTF format is
 * identical to standard UTF, except that it uses an 8 byte header
 * (instead of the standard 2 bytes) to convey the UTF encoding length.
 */
String readLongUTF() throws IOException {
    return readUTFBody(readLong());
}

/**
 * Reads in the "body" (i.e., the UTF representation minus the 2-byte
 * or 8-byte length header) of a UTF encoding, which occupies the next
 * utflen bytes.
 */
private String readUTFBody(long utflen) throws IOException {
    StringBuilder sbuf = new StringBuilder();
    if (!blkmode) {
        end = pos = 0;
    }

    while (utflen > 0) {
        int avail = end - pos;
        if (avail >= 3 || (long) avail == utflen) {
            utflen -= readUTFSpan(sbuf, utflen);
        } else {
            if (blkmode) {
                // near block boundary, read one byte at a time
                utflen -= readUTFChar(sbuf, utflen);
            } else {
                // shift and refill buffer manually
                if (avail > 0) {
                    System.arraycopy(buf, pos, buf, 0, avail);
                }
                pos = 0;
                end = (int) Math.min(MAX_BLOCK_SIZE, utflen);
                in.readFully(buf, avail, end - avail);
            }
        }
    }

    return sbuf.toString();
}

/**
 * Reads span of UTF-encoded characters out of internal buffer
 * (starting at offset pos and ending at or before offset end),
 * consuming no more than utflen bytes.  Appends read characters to
 * sbuf.  Returns the number of bytes consumed.
 */
private long readUTFSpan(StringBuilder sbuf, long utflen)
    throws IOException
{
```

```
int cpos = 0;
int start = pos;
int avail = Math.min(end - pos, CHAR_BUF_SIZE);
// stop short of last char unless all of utf bytes in buffer
int stop = pos + ((utflen > avail) ? avail - 2 : (int) utflen);
boolean outOfBounds = false;

try {
    while (pos < stop) {
        int b1, b2, b3;
        b1 = buf[pos++] & 0xFF;
        switch (b1 >> 4) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
            case 5:
            case 6:
            case 7:   // 1 byte format: 0xxxxxxx
                cbuf[cpos++] = (char) b1;
                break;

            case 12:
            case 13:  // 2 byte format: 110xxxxx 10xxxxxx
                b2 = buf[pos++];
                if ((b2 & 0xC0) != 0x80) {
                    throw new UTFDataFormatException();
                }
                cbuf[cpos++] = (char) (((b1 & 0x1F) << 6) |
                                       ((b2 & 0x3F) << 0));
                break;

            case 14:  // 3 byte format: 1110xxxx 10xxxxxx 10xxxxxx
                b3 = buf[pos + 1];
                b2 = buf[pos + 0];
                pos += 2;
                if ((b2 & 0xC0) != 0x80 || (b3 & 0xC0) != 0x80) {
                    throw new UTFDataFormatException();
                }
                cbuf[cpos++] = (char) (((b1 & 0x0F) << 12) |
                                       ((b2 & 0x3F) << 6) |
                                       ((b3 & 0x3F) << 0));
                break;

            default:  // 10xx xxxx, 1111 xxxx
                throw new UTFDataFormatException();
        }
    }
} catch (ArrayIndexOutOfBoundsException ex) {
    outOfBounds = true;
} finally {
    if (outOfBounds || (pos - start) > utflen) {
        /*
         * Fix for 4450867: if a malformed utf char causes the
         * conversion loop to scan past the expected end of the utf
         * string, only consume the expected number of utf bytes.
         */
        pos = start + (int) utflen;
        throw new UTFDataFormatException();
    }
}
```

```
            sbuf.append(cbuf, 0, cpos);
            return pos - start;
        }

        /**
         * Reads in single UTF-encoded character one byte at a time, appends
         * the character to sbuf, and returns the number of bytes consumed.
         * This method is used when reading in UTF strings written in block
         * data mode to handle UTF-encoded characters which (potentially)
         * straddle block-data boundaries.
         */
        private int readUTFChar(StringBuilder sbuf, long utflen)
            throws IOException
        {
            int b1, b2, b3;
            b1 = readByte() & 0xFF;
            switch (b1 >> 4) {
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                case 5:
                case 6:
                case 7:     // 1 byte format: 0xxxxxxx
                    sbuf.append((char) b1);
                    return 1;

                case 12:
                case 13:    // 2 byte format: 110xxxxx 10xxxxxx
                    if (utflen < 2) {
                        throw new UTFDataFormatException();
                    }
                    b2 = readByte();
                    if ((b2 & 0xC0) != 0x80) {
                        throw new UTFDataFormatException();
                    }
                    sbuf.append((char) (((b1 & 0x1F) << 6) |
                                        ((b2 & 0x3F) << 0)));
                    return 2;

                case 14:    // 3 byte format: 1110xxxx 10xxxxxx 10xxxxxx
                    if (utflen < 3) {
                        if (utflen == 2) {
                            readByte();         // consume remaining byte
                        }
                        throw new UTFDataFormatException();
                    }
                    b2 = readByte();
                    b3 = readByte();
                    if ((b2 & 0xC0) != 0x80 || (b3 & 0xC0) != 0x80) {
                        throw new UTFDataFormatException();
                    }
                    sbuf.append((char) (((b1 & 0x0F) << 12) |
                                        ((b2 & 0x3F) << 6) |
                                        ((b3 & 0x3F) << 0)));
                    return 3;

                default:   // 10xx xxxx, 1111 xxxx
                    throw new UTFDataFormatException();
            }
        }
    }
```

```
/**
 * Unsynchronized table which tracks wire handle to object mappings, as
 * well as ClassNotFoundExceptions associated with deserialized objects.
 * This class implements an exception-propagation algorithm for
 * determining which objects should have ClassNotFoundExceptions associated
 * with them, taking into account cycles and discontinuities (e.g., skipped
 * fields) in the object graph.
 *
 * <p>General use of the table is as follows: during deserialization, a
 * given object is first assigned a handle by calling the assign method.
 * This method leaves the assigned handle in an "open" state, wherein
 * dependencies on the exception status of other handles can be registered
 * by calling the markDependency method, or an exception can be directly
 * associated with the handle by calling markException.  When a handle is
 * tagged with an exception, the HandleTable assumes responsibility for
 * propagating the exception to any other objects which depend
 * (transitively) on the exception-tagged object.
 *
 * <p>Once all exception information/dependencies for the handle have been
 * registered, the handle should be "closed" by calling the finish method
 * on it.  The act of finishing a handle allows the exception propagation
 * algorithm to aggressively prune dependency links, lessening the
 * performance/memory impact of exception tracking.
 *
 * <p>Note that the exception propagation algorithm used depends on handles
 * being assigned/finished in LIFO order; however, for simplicity as well
 * as memory conservation, it does not enforce this constraint.
 */
// REMIND: add full description of exception propagation algorithm?
private static class HandleTable {

    /* status codes indicating whether object has associated exception */
    private static final byte STATUS_OK = 1;
    private static final byte STATUS_UNKNOWN = 2;
    private static final byte STATUS_EXCEPTION = 3;

    /** array mapping handle -> object status */
    byte[] status;
    /** array mapping handle -> object/exception (depending on status) */
    Object[] entries;
    /** array mapping handle -> list of dependent handles (if any) */
    HandleList[] deps;
    /** lowest unresolved dependency */
    int lowDep = -1;
    /** number of handles in table */
    int size = 0;

    /**
     * Creates handle table with the given initial capacity.
     */
    HandleTable(int initialCapacity) {
        status = new byte[initialCapacity];
        entries = new Object[initialCapacity];
        deps = new HandleList[initialCapacity];
    }

    /**
     * Assigns next available handle to given object, and returns assigned
     * handle.  Once object has been completely deserialized (and all
     * dependencies on other objects identified), the handle should be
     * "closed" by passing it to finish().
     */
```

```
int assign(Object obj) {
    if (size >= entries.length) {
        grow();
    }
    status[size] = STATUS_UNKNOWN;
    entries[size] = obj;
    return size++;
}

/**
 * Registers a dependency (in exception status) of one handle on
 * another.  The dependent handle must be "open" (i.e., assigned, but
 * not finished yet).  No action is taken if either dependent or target
 * handle is NULL_HANDLE.
 */
void markDependency(int dependent, int target) {
    if (dependent == NULL_HANDLE || target == NULL_HANDLE) {
        return;
    }
    switch (status[dependent]) {

        case STATUS_UNKNOWN:
            switch (status[target]) {
                case STATUS_OK:
                    // ignore dependencies on objs with no exception
                    break;

                case STATUS_EXCEPTION:
                    // eagerly propagate exception
                    markException(dependent,
                        (ClassNotFoundException) entries[target]);
                    break;

                case STATUS_UNKNOWN:
                    // add to dependency list of target
                    if (deps[target] == null) {
                        deps[target] = new HandleList();
                    }
                    deps[target].add(dependent);

                    // remember lowest unresolved target seen
                    if (lowDep < 0 || lowDep > target) {
                        lowDep = target;
                    }
                    break;

                default:
                    throw new InternalError();
            }
            break;

        case STATUS_EXCEPTION:
            break;

        default:
            throw new InternalError();
    }
}

/**
 * Associates a ClassNotFoundException (if one not already associated)
 * with the currently active handle and propagates it to other
 * referencing objects as appropriate.  The specified handle must be
```

```
     * "open" (i.e., assigned, but not finished yet).
     */
    void markException(int handle, ClassNotFoundException ex) {
        switch (status[handle]) {
            case STATUS_UNKNOWN:
                status[handle] = STATUS_EXCEPTION;
                entries[handle] = ex;

                // propagate exception to dependents
                HandleList dlist = deps[handle];
                if (dlist != null) {
                    int ndeps = dlist.size();
                    for (int i = 0; i < ndeps; i++) {
                        markException(dlist.get(i), ex);
                    }
                    deps[handle] = null;
                }
                break;

            case STATUS_EXCEPTION:
                break;

            default:
                throw new InternalError();
        }
    }

    /**
     * Marks given handle as finished, meaning that no new dependencies
     * will be marked for handle.  Calls to the assign and finish methods
     * must occur in LIFO order.
     */
    void finish(int handle) {
        int end;
        if (lowDep < 0) {
            // no pending unknowns, only resolve current handle
            end = handle + 1;
        } else if (lowDep >= handle) {
            // pending unknowns now clearable, resolve all upward handles
            end = size;
            lowDep = -1;
        } else {
            // unresolved backrefs present, can't resolve anything yet
            return;
        }

        // change STATUS_UNKNOWN -> STATUS_OK in selected span of handles
        for (int i = handle; i < end; i++) {
            switch (status[i]) {
                case STATUS_UNKNOWN:
                    status[i] = STATUS_OK;
                    deps[i] = null;
                    break;

                case STATUS_OK:
                case STATUS_EXCEPTION:
                    break;

                default:
                    throw new InternalError();
            }
        }
    }
```

```
/**
 * Assigns a new object to the given handle.  The object previously
 * associated with the handle is forgotten.  This method has no effect
 * if the given handle already has an exception associated with it.
 * This method may be called at any time after the handle is assigned.
 */
void setObject(int handle, Object obj) {
    switch (status[handle]) {
        case STATUS_UNKNOWN:
        case STATUS_OK:
            entries[handle] = obj;
            break;

        case STATUS_EXCEPTION:
            break;

        default:
            throw new InternalError();
    }
}

/**
 * Looks up and returns object associated with the given handle.
 * Returns null if the given handle is NULL_HANDLE, or if it has an
 * associated ClassNotFoundException.
 */
Object lookupObject(int handle) {
    return (handle != NULL_HANDLE &&
            status[handle] != STATUS_EXCEPTION) ?
        entries[handle] : null;
}

/**
 * Looks up and returns ClassNotFoundException associated with the
 * given handle.  Returns null if the given handle is NULL_HANDLE, or
 * if there is no ClassNotFoundException associated with the handle.
 */
ClassNotFoundException lookupException(int handle) {
    return (handle != NULL_HANDLE &&
            status[handle] == STATUS_EXCEPTION) ?
        (ClassNotFoundException) entries[handle] : null;
}

/**
 * Resets table to its initial state.
 */
void clear() {
    Arrays.fill(status, 0, size, (byte) 0);
    Arrays.fill(entries, 0, size, null);
    Arrays.fill(deps, 0, size, null);
    lowDep = -1;
    size = 0;
}

/**
 * Returns number of handles registered in table.
 */
int size() {
    return size;
}

/**
```

```
     * Expands capacity of internal arrays.
     */
    private void grow() {
        int newCapacity = (entries.length << 1) + 1;

        byte[] newStatus = new byte[newCapacity];
        Object[] newEntries = new Object[newCapacity];
        HandleList[] newDeps = new HandleList[newCapacity];

        System.arraycopy(status, 0, newStatus, 0, size);
        System.arraycopy(entries, 0, newEntries, 0, size);
        System.arraycopy(deps, 0, newDeps, 0, size);

        status = newStatus;
        entries = newEntries;
        deps = newDeps;
    }

    /**
     * Simple growable list of (integer) handles.
     */
    private static class HandleList {
        private int[] list = new int[4];
        private int size = 0;

        public HandleList() {
        }

        public void add(int handle) {
            if (size >= list.length) {
                int[] newList = new int[list.length << 1];
                System.arraycopy(list, 0, newList, 0, list.length);
                list = newList;
            }
            list[size++] = handle;
        }

        public int get(int index) {
            if (index >= size) {
                throw new ArrayIndexOutOfBoundsException();
            }
            return list[index];
        }

        public int size() {
            return size;
        }
    }
}

/**
 * Method for cloning arrays in case of using unsharing reading
 */
private static Object cloneArray(Object array) {
    if (array instanceof Object[]) {
        return ((Object[]) array).clone();
    } else if (array instanceof boolean[]) {
        return ((boolean[]) array).clone();
    } else if (array instanceof byte[]) {
        return ((byte[]) array).clone();
    } else if (array instanceof char[]) {
        return ((char[]) array).clone();
    } else if (array instanceof double[]) {
```

```
            return ((double[]) array).clone();
        } else if (array instanceof float[]) {
            return ((float[]) array).clone();
        } else if (array instanceof int[]) {
            return ((int[]) array).clone();
        } else if (array instanceof long[]) {
            return ((long[]) array).clone();
        } else if (array instanceof short[]) {
            return ((short[]) array).clone();
        } else {
            throw new AssertionError();
        }
    }

}
```

ObjectInputValidation.java

```java
/*
 * Copyright (c) 1996, 1999, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Callback interface to allow validation of objects within a graph.
 * Allows an object to be called when a complete graph of objects has
 * been deserialized.
 *
 * @author  unascribed
 * @see     ObjectInputStream
 * @see     ObjectInputStream#registerValidation(java.io.ObjectInputValidation, int)
 * @since   JDK1.1
 */
public interface ObjectInputValidation {
    /**
     * Validates the object.
     *
     * @exception InvalidObjectException If the object cannot validate itself.
     */
    public void validateObject() throws InvalidObjectException;
}
```

ObjectOutput.java

```
/*
 * Copyright (c) 1996, 2010, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * ObjectOutput extends the DataOutput interface to include writing of objects.
 * DataOutput includes methods for output of primitive types, ObjectOutput
 * extends that interface to include objects, arrays, and Strings.
 *
 * @author  unascribed
 * @see java.io.InputStream
 * @see java.io.ObjectOutputStream
 * @see java.io.ObjectInputStream
 * @since   JDK1.1
 */
public interface ObjectOutput extends DataOutput, AutoCloseable {
    /**
     * Write an object to the underlying storage or stream.  The
     * class that implements this interface defines how the object is
     * written.
     *
     * @param obj the object to be written
     * @exception IOException Any of the usual Input/Output related exceptions.
     */
    public void writeObject(Object obj)
      throws IOException;

    /**
     * Writes a byte. This method will block until the byte is actually
     * written.
     * @param b the byte
     * @exception IOException If an I/O error has occurred.
     */
    public void write(int b) throws IOException;

    /**
     * Writes an array of bytes. This method will block until the bytes
```

```
     * are actually written.
     * @param b the data to be written
     * @exception IOException If an I/O error has occurred.
     */
    public void write(byte b[]) throws IOException;

    /**
     * Writes a sub array of bytes.
     * @param b the data to be written
     * @param off       the start offset in the data
     * @param len       the number of bytes that are written
     * @exception IOException If an I/O error has occurred.
     */
    public void write(byte b[], int off, int len) throws IOException;

    /**
     * Flushes the stream. This will write any buffered
     * output bytes.
     * @exception IOException If an I/O error has occurred.
     */
    public void flush() throws IOException;

    /**
     * Closes the stream. This method must be called
     * to release any resources associated with the
     * stream.
     * @exception IOException If an I/O error has occurred.
     */
    public void close() throws IOException;
}
```

ObjectOutputStream.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.io.ObjectStreamClass.WeakClassKey;
import java.lang.ref.ReferenceQueue;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import static java.io.ObjectStreamClass.processQueue;
import java.io.SerialCallbackContext;
import sun.reflect.misc.ReflectUtil;

/**
 * An ObjectOutputStream writes primitive data types and graphs of Java objects
 * to an OutputStream.  The objects can be read (reconstituted) using an
 * ObjectInputStream.  Persistent storage of objects can be accomplished by
 * using a file for the stream.  If the stream is a network socket stream, the
 * objects can be reconstituted on another host or in another process.
 *
 * <p>Only objects that support the java.io.Serializable interface can be
 * written to streams.  The class of each serializable object is encoded
 * including the class name and signature of the class, the values of the
 * object's fields and arrays, and the closure of any other objects referenced
 * from the initial objects.
 *
 * <p>The method writeObject is used to write an object to the stream.  Any
 * object, including Strings and arrays, is written with writeObject. Multiple
 * objects or primitives can be written to the stream.  The objects must be
 * read back from the corresponding ObjectInputstream with the same types and
 * in the same order as they were written.
 *
 * <p>Primitive data types can also be written to the stream using the
```

```
* appropriate methods from DataOutput. Strings can also be written using the
* writeUTF method.
*
* <p>The default serialization mechanism for an object writes the class of the
* object, the class signature, and the values of all non-transient and
* non-static fields.  References to other objects (except in transient or
* static fields) cause those objects to be written also. Multiple references
* to a single object are encoded using a reference sharing mechanism so that
* graphs of objects can be restored to the same shape as when the original was
* written.
*
* <p>For example to write an object that can be read by the example in
* ObjectInputStream:
* <br>
* <pre>
*      FileOutputStream fos = new FileOutputStream("t.tmp");
*      ObjectOutputStream oos = new ObjectOutputStream(fos);
*
*      oos.writeInt(12345);
*      oos.writeObject("Today");
*      oos.writeObject(new Date());
*
*      oos.close();
* </pre>
*
* <p>Classes that require special handling during the serialization and
* deserialization process must implement special methods with these exact
* signatures:
* <br>
* <pre>
* private void readObject(java.io.ObjectInputStream stream)
*     throws IOException, ClassNotFoundException;
* private void writeObject(java.io.ObjectOutputStream stream)
*     throws IOException
* private void readObjectNoData()
*     throws ObjectStreamException;
* </pre>
*
* <p>The writeObject method is responsible for writing the state of the object
* for its particular class so that the corresponding readObject method can
* restore it.  The method does not need to concern itself with the state
* belonging to the object's superclasses or subclasses.  State is saved by
* writing the individual fields to the ObjectOutputStream using the
* writeObject method or by using the methods for primitive data types
* supported by DataOutput.
*
* <p>Serialization does not write out the fields of any object that does not
* implement the java.io.Serializable interface.  Subclasses of Objects that
* are not serializable can be serializable. In this case the non-serializable
* class must have a no-arg constructor to allow its fields to be initialized.
* In this case it is the responsibility of the subclass to save and restore
* the state of the non-serializable class. It is frequently the case that the
* fields of that class are accessible (public, package, or protected) or that
* there are get and set methods that can be used to restore the state.
*
* <p>Serialization of an object can be prevented by implementing writeObject
* and readObject methods that throw the NotSerializableException.  The
* exception will be caught by the ObjectOutputStream and abort the
* serialization process.
*
* <p>Implementing the Externalizable interface allows the object to assume
* complete control over the contents and format of the object's serialized
* form.  The methods of the Externalizable interface, writeExternal and
```

```
 * readExternal, are called to save and restore the objects state.  When
 * implemented by a class they can write and read their own state using all of
 * the methods of ObjectOutput and ObjectInput.  It is the responsibility of
 * the objects to handle any versioning that occurs.
 *
 * <p>Enum constants are serialized differently than ordinary serializable or
 * externalizable objects.  The serialized form of an enum constant consists
 * solely of its name; field values of the constant are not transmitted.  To
 * serialize an enum constant, ObjectOutputStream writes the string returned by
 * the constant's name method.  Like other serializable or externalizable
 * objects, enum constants can function as the targets of back references
 * appearing subsequently in the serialization stream.  The process by which
 * enum constants are serialized cannot be customized; any class-specific
 * writeObject and writeReplace methods defined by enum types are ignored
 * during serialization.  Similarly, any serialPersistentFields or
 * serialVersionUID field declarations are also ignored--all enum types have a
 * fixed serialVersionUID of 0L.
 *
 * <p>Primitive data, excluding serializable fields and externalizable data, is
 * written to the ObjectOutputStream in block-data records. A block data record
 * is composed of a header and data. The block data header consists of a marker
 * and the number of bytes to follow the header.  Consecutive primitive data
 * writes are merged into one block-data record.  The blocking factor used for
 * a block-data record will be 1024 bytes.  Each block-data record will be
 * filled up to 1024 bytes, or be written whenever there is a termination of
 * block-data mode.  Calls to the ObjectOutputStream methods writeObject,
 * defaultWriteObject and writeFields initially terminate any existing
 * block-data record.
 *
 * @author      Mike Warres
 * @author      Roger Riggs
 * @see java.io.DataOutput
 * @see java.io.ObjectInputStream
 * @see java.io.Serializable
 * @see java.io.Externalizable
 * @see <a href="../../../platform/serialization/spec/output.html">Object Serialization Specification, Section
2, Object Output Classes</a>
 * @since       JDK1.1
 */
public class ObjectOutputStream
    extends OutputStream implements ObjectOutput, ObjectStreamConstants
{

    private static class Caches {
        /** cache of subclass security audit results */
        static final ConcurrentMap<WeakClassKey,Boolean> subclassAudits =
            new ConcurrentHashMap<>();

        /** queue for WeakReferences to audited subclasses */
        static final ReferenceQueue<Class<?>> subclassAuditsQueue =
            new ReferenceQueue<>();
    }

    /** filter stream for handling block data conversion */
    private final BlockDataOutputStream bout;
    /** obj -> wire handle map */
    private final HandleTable handles;
    /** obj -> replacement obj map */
    private final ReplaceTable subs;
    /** stream protocol version */
    private int protocol = PROTOCOL_VERSION_2;
    /** recursion depth */
    private int depth;
```

```
/** buffer for writing primitive field values */
private byte[] primVals;

/** if true, invoke writeObjectOverride() instead of writeObject() */
private final boolean enableOverride;
/** if true, invoke replaceObject() */
private boolean enableReplace;

// values below valid only during upcalls to writeObject()/writeExternal()
/**
 * Context during upcalls to class-defined writeObject methods; holds
 * object currently being serialized and descriptor for current class.
 * Null when not during writeObject upcall.
 */
private SerialCallbackContext curContext;
/** current PutField object */
private PutFieldImpl curPut;

/** custom storage for debug trace info */
private final DebugTraceInfoStack debugInfoStack;

/**
 * value of "sun.io.serialization.extendedDebugInfo" property,
 * as true or false for extended information about exception's place
 */
private static final boolean extendedDebugInfo =
    java.security.AccessController.doPrivileged(
        new sun.security.action.GetBooleanAction(
            "sun.io.serialization.extendedDebugInfo")).booleanValue();

/**
 * Creates an ObjectOutputStream that writes to the specified OutputStream.
 * This constructor writes the serialization stream header to the
 * underlying stream; callers may wish to flush the stream immediately to
 * ensure that constructors for receiving ObjectInputStreams will not block
 * when reading the header.
 *
 * <p>If a security manager is installed, this constructor will check for
 * the "enableSubclassImplementation" SerializablePermission when invoked
 * directly or indirectly by the constructor of a subclass which overrides
 * the ObjectOutputStream.putFields or ObjectOutputStream.writeUnshared
 * methods.
 *
 * @param   out output stream to write to
 * @throws  IOException if an I/O error occurs while writing stream header
 * @throws  SecurityException if untrusted subclass illegally overrides
 *          security-sensitive methods
 * @throws  NullPointerException if <code>out</code> is <code>null</code>
 * @since   1.4
 * @see     ObjectOutputStream#ObjectOutputStream()
 * @see     ObjectOutputStream#putFields()
 * @see     ObjectInputStream#ObjectInputStream(InputStream)
 */
public ObjectOutputStream(OutputStream out) throws IOException {
    verifySubclass();
    bout = new BlockDataOutputStream(out);
    handles = new HandleTable(10, (float) 3.00);
    subs = new ReplaceTable(10, (float) 3.00);
    enableOverride = false;
    writeStreamHeader();
    bout.setBlockDataMode(true);
    if (extendedDebugInfo) {
```

```
        debugInfoStack = new DebugTraceInfoStack();
    } else {
        debugInfoStack = null;
    }
}

/**
 * Provide a way for subclasses that are completely reimplementing
 * ObjectOutputStream to not have to allocate private data just used by
 * this implementation of ObjectOutputStream.
 *
 * <p>If there is a security manager installed, this method first calls the
 * security manager's <code>checkPermission</code> method with a
 * <code>SerializablePermission("enableSubclassImplementation")</code>
 * permission to ensure it's ok to enable subclassing.
 *
 * @throws  SecurityException if a security manager exists and its
 *          <code>checkPermission</code> method denies enabling
 *          subclassing.
 * @throws  IOException if an I/O error occurs while creating this stream
 * @see SecurityManager#checkPermission
 * @see java.io.SerializablePermission
 */
protected ObjectOutputStream() throws IOException, SecurityException {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        sm.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSION);
    }
    bout = null;
    handles = null;
    subs = null;
    enableOverride = true;
    debugInfoStack = null;
}

/**
 * Specify stream protocol version to use when writing the stream.
 *
 * <p>This routine provides a hook to enable the current version of
 * Serialization to write in a format that is backwards compatible to a
 * previous version of the stream format.
 *
 * <p>Every effort will be made to avoid introducing additional
 * backwards incompatibilities; however, sometimes there is no
 * other alternative.
 *
 * @param   version use ProtocolVersion from java.io.ObjectStreamConstants.
 * @throws  IllegalStateException if called after any objects
 *          have been serialized.
 * @throws  IllegalArgumentException if invalid version is passed in.
 * @throws  IOException if I/O errors occur
 * @see java.io.ObjectStreamConstants#PROTOCOL_VERSION_1
 * @see java.io.ObjectStreamConstants#PROTOCOL_VERSION_2
 * @since   1.2
 */
public void useProtocolVersion(int version) throws IOException {
    if (handles.size() != 0) {
        // REMIND: implement better check for pristine stream?
        throw new IllegalStateException("stream non-empty");
    }
    switch (version) {
        case PROTOCOL_VERSION_1:
        case PROTOCOL_VERSION_2:
```

```
            protocol = version;
            break;

        default:
            throw new IllegalArgumentException(
                "unknown version: " + version);
    }
}

/**
 * Write the specified object to the ObjectOutputStream.  The class of the
 * object, the signature of the class, and the values of the non-transient
 * and non-static fields of the class and all of its supertypes are
 * written.  Default serialization for a class can be overridden using the
 * writeObject and the readObject methods.  Objects referenced by this
 * object are written transitively so that a complete equivalent graph of
 * objects can be reconstructed by an ObjectInputStream.
 *
 * <p>Exceptions are thrown for problems with the OutputStream and for
 * classes that should not be serialized.  All exceptions are fatal to the
 * OutputStream, which is left in an indeterminate state, and it is up to
 * the caller to ignore or recover the stream state.
 *
 * @throws  InvalidClassException Something is wrong with a class used by
 *          serialization.
 * @throws  NotSerializableException Some object to be serialized does not
 *          implement the java.io.Serializable interface.
 * @throws  IOException Any exception thrown by the underlying
 *          OutputStream.
 */
public final void writeObject(Object obj) throws IOException {
    if (enableOverride) {
        writeObjectOverride(obj);
        return;
    }
    try {
        writeObject0(obj, false);
    } catch (IOException ex) {
        if (depth == 0) {
            writeFatalException(ex);
        }
        throw ex;
    }
}

/**
 * Method used by subclasses to override the default writeObject method.
 * This method is called by trusted subclasses of ObjectInputStream that
 * constructed ObjectInputStream using the protected no-arg constructor.
 * The subclass is expected to provide an override method with the modifier
 * "final".
 *
 * @param   obj object to be written to the underlying stream
 * @throws  IOException if there are I/O errors while writing to the
 *          underlying stream
 * @see #ObjectOutputStream()
 * @see #writeObject(Object)
 * @since 1.2
 */
protected void writeObjectOverride(Object obj) throws IOException {
}

/**
```

```
     * Writes an "unshared" object to the ObjectOutputStream.  This method is
     * identical to writeObject, except that it always writes the given object
     * as a new, unique object in the stream (as opposed to a back-reference
     * pointing to a previously serialized instance).  Specifically:
     * <ul>
     *   <li>An object written via writeUnshared is always serialized in the
     *       same manner as a newly appearing object (an object that has not
     *       been written to the stream yet), regardless of whether or not the
     *       object has been written previously.
     *
     *   <li>If writeObject is used to write an object that has been previously
     *       written with writeUnshared, the previous writeUnshared operation
     *       is treated as if it were a write of a separate object.  In other
     *       words, ObjectOutputStream will never generate back-references to
     *       object data written by calls to writeUnshared.
     * </ul>
     * While writing an object via writeUnshared does not in itself guarantee a
     * unique reference to the object when it is deserialized, it allows a
     * single object to be defined multiple times in a stream, so that multiple
     * calls to readUnshared by the receiver will not conflict.  Note that the
     * rules described above only apply to the base-level object written with
     * writeUnshared, and not to any transitively referenced sub-objects in the
     * object graph to be serialized.
     *
     * <p>ObjectOutputStream subclasses which override this method can only be
     * constructed in security contexts possessing the
     * "enableSubclassImplementation" SerializablePermission; any attempt to
     * instantiate such a subclass without this permission will cause a
     * SecurityException to be thrown.
     *
     * @param   obj object to write to stream
     * @throws  NotSerializableException if an object in the graph to be
     *          serialized does not implement the Serializable interface
     * @throws  InvalidClassException if a problem exists with the class of an
     *          object to be serialized
     * @throws  IOException if an I/O error occurs during serialization
     * @since 1.4
     */
    public void writeUnshared(Object obj) throws IOException {
        try {
            writeObject0(obj, true);
        } catch (IOException ex) {
            if (depth == 0) {
                writeFatalException(ex);
            }
            throw ex;
        }
    }

    /**
     * Write the non-static and non-transient fields of the current class to
     * this stream.  This may only be called from the writeObject method of the
     * class being serialized. It will throw the NotActiveException if it is
     * called otherwise.
     *
     * @throws  IOException if I/O errors occur while writing to the underlying
     *          <code>OutputStream</code>
     */
    public void defaultWriteObject() throws IOException {
        SerialCallbackContext ctx = curContext;
        if (ctx == null) {
            throw new NotActiveException("not in call to writeObject");
        }
```

```
    Object curObj = ctx.getObj();
    ObjectStreamClass curDesc = ctx.getDesc();
    bout.setBlockDataMode(false);
    defaultWriteFields(curObj, curDesc);
    bout.setBlockDataMode(true);
}

/**
 * Retrieve the object used to buffer persistent fields to be written to
 * the stream.  The fields will be written to the stream when writeFields
 * method is called.
 *
 * @return  an instance of the class Putfield that holds the serializable
 *          fields
 * @throws  IOException if I/O errors occur
 * @since 1.2
 */
public ObjectOutputStream.PutField putFields() throws IOException {
    if (curPut == null) {
        SerialCallbackContext ctx = curContext;
        if (ctx == null) {
            throw new NotActiveException("not in call to writeObject");
        }
        Object curObj = ctx.getObj();
        ObjectStreamClass curDesc = ctx.getDesc();
        curPut = new PutFieldImpl(curDesc);
    }
    return curPut;
}

/**
 * Write the buffered fields to the stream.
 *
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 * @throws  NotActiveException Called when a classes writeObject method was
 *          not called to write the state of the object.
 * @since 1.2
 */
public void writeFields() throws IOException {
    if (curPut == null) {
        throw new NotActiveException("no current PutField object");
    }
    bout.setBlockDataMode(false);
    curPut.writeFields();
    bout.setBlockDataMode(true);
}

/**
 * Reset will disregard the state of any objects already written to the
 * stream.  The state is reset to be the same as a new ObjectOutputStream.
 * The current point in the stream is marked as reset so the corresponding
 * ObjectInputStream will be reset at the same point.  Objects previously
 * written to the stream will not be referred to as already being in the
 * stream.  They will be written to the stream again.
 *
 * @throws  IOException if reset() is invoked while serializing an object.
 */
public void reset() throws IOException {
    if (depth != 0) {
        throw new IOException("stream active");
    }
    bout.setBlockDataMode(false);
```

```
    bout.writeByte(TC_RESET);
    clear();
    bout.setBlockDataMode(true);
}

/**
 * Subclasses may implement this method to allow class data to be stored in
 * the stream. By default this method does nothing.  The corresponding
 * method in ObjectInputStream is resolveClass.  This method is called
 * exactly once for each unique class in the stream.  The class name and
 * signature will have already been written to the stream.  This method may
 * make free use of the ObjectOutputStream to save any representation of
 * the class it deems suitable (for example, the bytes of the class file).
 * The resolveClass method in the corresponding subclass of
 * ObjectInputStream must read and use any data or objects written by
 * annotateClass.
 *
 * @param   cl the class to annotate custom data for
 * @throws  IOException Any exception thrown by the underlying
 *          OutputStream.
 */
protected void annotateClass(Class<?> cl) throws IOException {
}

/**
 * Subclasses may implement this method to store custom data in the stream
 * along with descriptors for dynamic proxy classes.
 *
 * <p>This method is called exactly once for each unique proxy class
 * descriptor in the stream.  The default implementation of this method in
 * <code>ObjectOutputStream</code> does nothing.
 *
 * <p>The corresponding method in <code>ObjectInputStream</code> is
 * <code>resolveProxyClass</code>.  For a given subclass of
 * <code>ObjectOutputStream</code> that overrides this method, the
 * <code>resolveProxyClass</code> method in the corresponding subclass of
 * <code>ObjectInputStream</code> must read any data or objects written by
 * <code>annotateProxyClass</code>.
 *
 * @param   cl the proxy class to annotate custom data for
 * @throws  IOException any exception thrown by the underlying
 *          <code>OutputStream</code>
 * @see ObjectInputStream#resolveProxyClass(String[])
 * @since   1.3
 */
protected void annotateProxyClass(Class<?> cl) throws IOException {
}

/**
 * This method will allow trusted subclasses of ObjectOutputStream to
 * substitute one object for another during serialization. Replacing
 * objects is disabled until enableReplaceObject is called. The
 * enableReplaceObject method checks that the stream requesting to do
 * replacement can be trusted.  The first occurrence of each object written
 * into the serialization stream is passed to replaceObject.  Subsequent
 * references to the object are replaced by the object returned by the
 * original call to replaceObject.  To ensure that the private state of
 * objects is not unintentionally exposed, only trusted streams may use
 * replaceObject.
 *
 * <p>The ObjectOutputStream.writeObject method takes a parameter of type
 * Object (as opposed to type Serializable) to allow for cases where
 * non-serializable objects are replaced by serializable ones.
```

```
 *
 * <p>When a subclass is replacing objects it must insure that either a
 * complementary substitution must be made during deserialization or that
 * the substituted object is compatible with every field where the
 * reference will be stored.  Objects whose type is not a subclass of the
 * type of the field or array element abort the serialization by raising an
 * exception and the object is not be stored.
 *
 * <p>This method is called only once when each object is first
 * encountered.  All subsequent references to the object will be redirected
 * to the new object. This method should return the object to be
 * substituted or the original object.
 *
 * <p>Null can be returned as the object to be substituted, but may cause
 * NullReferenceException in classes that contain references to the
 * original object since they may be expecting an object instead of
 * null.
 *
 * @param   obj the object to be replaced
 * @return  the alternate object that replaced the specified one
 * @throws  IOException Any exception thrown by the underlying
 *          OutputStream.
 */
protected Object replaceObject(Object obj) throws IOException {
    return obj;
}

/**
 * Enable the stream to do replacement of objects in the stream.  When
 * enabled, the replaceObject method is called for every object being
 * serialized.
 *
 * <p>If <code>enable</code> is true, and there is a security manager
 * installed, this method first calls the security manager's
 * <code>checkPermission</code> method with a
 * <code>SerializablePermission("enableSubstitution")</code> permission to
 * ensure it's ok to enable the stream to do replacement of objects in the
 * stream.
 *
 * @param   enable boolean parameter to enable replacement of objects
 * @return  the previous setting before this method was invoked
 * @throws  SecurityException if a security manager exists and its
 *          <code>checkPermission</code> method denies enabling the stream
 *          to do replacement of objects in the stream.
 * @see SecurityManager#checkPermission
 * @see java.io.SerializablePermission
 */
protected boolean enableReplaceObject(boolean enable)
    throws SecurityException
{
    if (enable == enableReplace) {
        return enable;
    }
    if (enable) {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(SUBSTITUTION_PERMISSION);
        }
    }
    enableReplace = enable;
    return !enableReplace;
}
```

```
/**
 * The writeStreamHeader method is provided so subclasses can append or
 * prepend their own header to the stream.  It writes the magic number and
 * version to the stream.
 *
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 */
protected void writeStreamHeader() throws IOException {
    bout.writeShort(STREAM_MAGIC);
    bout.writeShort(STREAM_VERSION);
}

/**
 * Write the specified class descriptor to the ObjectOutputStream.  Class
 * descriptors are used to identify the classes of objects written to the
 * stream.  Subclasses of ObjectOutputStream may override this method to
 * customize the way in which class descriptors are written to the
 * serialization stream.  The corresponding method in ObjectInputStream,
 * <code>readClassDescriptor</code>, should then be overridden to
 * reconstitute the class descriptor from its custom stream representation.
 * By default, this method writes class descriptors according to the format
 * defined in the Object Serialization specification.
 *
 * <p>Note that this method will only be called if the ObjectOutputStream
 * is not using the old serialization stream format (set by calling
 * ObjectOutputStream's <code>useProtocolVersion</code> method).  If this
 * serialization stream is using the old format
 * (<code>PROTOCOL_VERSION_1</code>), the class descriptor will be written
 * internally in a manner that cannot be overridden or customized.
 *
 * @param   desc class descriptor to write to the stream
 * @throws  IOException If an I/O error has occurred.
 * @see java.io.ObjectInputStream#readClassDescriptor()
 * @see #useProtocolVersion(int)
 * @see java.io.ObjectStreamConstants#PROTOCOL_VERSION_1
 * @since 1.3
 */
protected void writeClassDescriptor(ObjectStreamClass desc)
    throws IOException
{
    desc.writeNonProxy(this);
}

/**
 * Writes a byte. This method will block until the byte is actually
 * written.
 *
 * @param   val the byte to be written to the stream
 * @throws  IOException If an I/O error has occurred.
 */
public void write(int val) throws IOException {
    bout.write(val);
}

/**
 * Writes an array of bytes. This method will block until the bytes are
 * actually written.
 *
 * @param   buf the data to be written
 * @throws  IOException If an I/O error has occurred.
 */
public void write(byte[] buf) throws IOException {
```

```
    bout.write(buf, 0, buf.length, false);
}

/**
 * Writes a sub array of bytes.
 *
 * @param   buf the data to be written
 * @param   off the start offset in the data
 * @param   len the number of bytes that are written
 * @throws  IOException If an I/O error has occurred.
 */
public void write(byte[] buf, int off, int len) throws IOException {
    if (buf == null) {
        throw new NullPointerException();
    }
    int endoff = off + len;
    if (off < 0 || len < 0 || endoff > buf.length || endoff < 0) {
        throw new IndexOutOfBoundsException();
    }
    bout.write(buf, off, len, false);
}

/**
 * Flushes the stream. This will write any buffered output bytes and flush
 * through to the underlying stream.
 *
 * @throws  IOException If an I/O error has occurred.
 */
public void flush() throws IOException {
    bout.flush();
}

/**
 * Drain any buffered data in ObjectOutputStream.  Similar to flush but
 * does not propagate the flush to the underlying stream.
 *
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 */
protected void drain() throws IOException {
    bout.drain();
}

/**
 * Closes the stream. This method must be called to release any resources
 * associated with the stream.
 *
 * @throws  IOException If an I/O error has occurred.
 */
public void close() throws IOException {
    flush();
    clear();
    bout.close();
}

/**
 * Writes a boolean.
 *
 * @param   val the boolean to be written
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 */
public void writeBoolean(boolean val) throws IOException {
```

```
    bout.writeBoolean(val);
}

/**
 * Writes an 8 bit byte.
 *
 * @param   val the byte value to be written
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 */
public void writeByte(int val) throws IOException  {
    bout.writeByte(val);
}

/**
 * Writes a 16 bit short.
 *
 * @param   val the short value to be written
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 */
public void writeShort(int val)  throws IOException {
    bout.writeShort(val);
}

/**
 * Writes a 16 bit char.
 *
 * @param   val the char value to be written
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 */
public void writeChar(int val)  throws IOException {
    bout.writeChar(val);
}

/**
 * Writes a 32 bit int.
 *
 * @param   val the integer value to be written
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 */
public void writeInt(int val)  throws IOException {
    bout.writeInt(val);
}

/**
 * Writes a 64 bit long.
 *
 * @param   val the long value to be written
 * @throws  IOException if I/O errors occur while writing to the underlying
 *          stream
 */
public void writeLong(long val)  throws IOException {
    bout.writeLong(val);
}

/**
 * Writes a 32 bit float.
 *
 * @param   val the float value to be written
 * @throws  IOException if I/O errors occur while writing to the underlying
```

```
     *          stream
     */
    public void writeFloat(float val) throws IOException {
        bout.writeFloat(val);
    }

    /**
     * Writes a 64 bit double.
     *
     * @param   val the double value to be written
     * @throws  IOException if I/O errors occur while writing to the underlying
     *          stream
     */
    public void writeDouble(double val) throws IOException {
        bout.writeDouble(val);
    }

    /**
     * Writes a String as a sequence of bytes.
     *
     * @param   str the String of bytes to be written
     * @throws  IOException if I/O errors occur while writing to the underlying
     *          stream
     */
    public void writeBytes(String str) throws IOException {
        bout.writeBytes(str);
    }

    /**
     * Writes a String as a sequence of chars.
     *
     * @param   str the String of chars to be written
     * @throws  IOException if I/O errors occur while writing to the underlying
     *          stream
     */
    public void writeChars(String str) throws IOException {
        bout.writeChars(str);
    }

    /**
     * Primitive data write of this String in
     * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
     * format.  Note that there is a
     * significant difference between writing a String into the stream as
     * primitive data or as an Object. A String instance written by writeObject
     * is written into the stream as a String initially. Future writeObject()
     * calls write references to the string into the stream.
     *
     * @param   str the String to be written
     * @throws  IOException if I/O errors occur while writing to the underlying
     *          stream
     */
    public void writeUTF(String str) throws IOException {
        bout.writeUTF(str);
    }

    /**
     * Provide programmatic access to the persistent fields to be written
     * to ObjectOutput.
     *
     * @since 1.2
     */
    public static abstract class PutField {
```

```
/**
 * Put the value of the named boolean field into the persistent field.
 *
 * @param  name the name of the serializable field
 * @param  val the value to assign to the field
 * @throws IllegalArgumentException if <code>name</code> does not
 * match the name of a serializable field for the class whose fields
 * are being written, or if the type of the named field is not
 * <code>boolean</code>
 */
public abstract void put(String name, boolean val);

/**
 * Put the value of the named byte field into the persistent field.
 *
 * @param  name the name of the serializable field
 * @param  val the value to assign to the field
 * @throws IllegalArgumentException if <code>name</code> does not
 * match the name of a serializable field for the class whose fields
 * are being written, or if the type of the named field is not
 * <code>byte</code>
 */
public abstract void put(String name, byte val);

/**
 * Put the value of the named char field into the persistent field.
 *
 * @param  name the name of the serializable field
 * @param  val the value to assign to the field
 * @throws IllegalArgumentException if <code>name</code> does not
 * match the name of a serializable field for the class whose fields
 * are being written, or if the type of the named field is not
 * <code>char</code>
 */
public abstract void put(String name, char val);

/**
 * Put the value of the named short field into the persistent field.
 *
 * @param  name the name of the serializable field
 * @param  val the value to assign to the field
 * @throws IllegalArgumentException if <code>name</code> does not
 * match the name of a serializable field for the class whose fields
 * are being written, or if the type of the named field is not
 * <code>short</code>
 */
public abstract void put(String name, short val);

/**
 * Put the value of the named int field into the persistent field.
 *
 * @param  name the name of the serializable field
 * @param  val the value to assign to the field
 * @throws IllegalArgumentException if <code>name</code> does not
 * match the name of a serializable field for the class whose fields
 * are being written, or if the type of the named field is not
 * <code>int</code>
 */
public abstract void put(String name, int val);

/**
 * Put the value of the named long field into the persistent field.
```

```
     *
     * @param  name the name of the serializable field
     * @param  val the value to assign to the field
     * @throws IllegalArgumentException if <code>name</code> does not
     * match the name of a serializable field for the class whose fields
     * are being written, or if the type of the named field is not
     * <code>long</code>
     */
    public abstract void put(String name, long val);

    /**
     * Put the value of the named float field into the persistent field.
     *
     * @param  name the name of the serializable field
     * @param  val the value to assign to the field
     * @throws IllegalArgumentException if <code>name</code> does not
     * match the name of a serializable field for the class whose fields
     * are being written, or if the type of the named field is not
     * <code>float</code>
     */
    public abstract void put(String name, float val);

    /**
     * Put the value of the named double field into the persistent field.
     *
     * @param  name the name of the serializable field
     * @param  val the value to assign to the field
     * @throws IllegalArgumentException if <code>name</code> does not
     * match the name of a serializable field for the class whose fields
     * are being written, or if the type of the named field is not
     * <code>double</code>
     */
    public abstract void put(String name, double val);

    /**
     * Put the value of the named Object field into the persistent field.
     *
     * @param  name the name of the serializable field
     * @param  val the value to assign to the field
     *         (which may be <code>null</code>)
     * @throws IllegalArgumentException if <code>name</code> does not
     * match the name of a serializable field for the class whose fields
     * are being written, or if the type of the named field is not a
     * reference type
     */
    public abstract void put(String name, Object val);

    /**
     * Write the data and fields to the specified ObjectOutput stream,
     * which must be the same stream that produced this
     * <code>PutField</code> object.
     *
     * @param  out the stream to write the data and fields to
     * @throws IOException if I/O errors occur while writing to the
     *         underlying stream
     * @throws IllegalArgumentException if the specified stream is not
     *         the same stream that produced this <code>PutField</code>
     *         object
     * @deprecated This method does not write the values contained by this
     *         <code>PutField</code> object in a proper format, and may
     *         result in corruption of the serialization stream.  The
     *         correct way to write <code>PutField</code> data is by
     *         calling the {@link java.io.ObjectOutputStream#writeFields()}
```

```
     *         method.
     */
    @Deprecated
    public abstract void write(ObjectOutput out) throws IOException;
}


/**
 * Returns protocol version in use.
 */
int getProtocolVersion() {
    return protocol;
}

/**
 * Writes string without allowing it to be replaced in stream.  Used by
 * ObjectStreamClass to write class descriptor type strings.
 */
void writeTypeString(String str) throws IOException {
    int handle;
    if (str == null) {
        writeNull();
    } else if ((handle = handles.lookup(str)) != -1) {
        writeHandle(handle);
    } else {
        writeString(str, false);
    }
}

/**
 * Verifies that this (possibly subclass) instance can be constructed
 * without violating security constraints: the subclass must not override
 * security-sensitive non-final methods, or else the
 * "enableSubclassImplementation" SerializablePermission is checked.
 */
private void verifySubclass() {
    Class<?> cl = getClass();
    if (cl == ObjectOutputStream.class) {
        return;
    }
    SecurityManager sm = System.getSecurityManager();
    if (sm == null) {
        return;
    }
    processQueue(Caches.subclassAuditsQueue, Caches.subclassAudits);
    WeakClassKey key = new WeakClassKey(cl, Caches.subclassAuditsQueue);
    Boolean result = Caches.subclassAudits.get(key);
    if (result == null) {
        result = Boolean.valueOf(auditSubclass(cl));
        Caches.subclassAudits.putIfAbsent(key, result);
    }
    if (result.booleanValue()) {
        return;
    }
    sm.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSION);
}

/**
 * Performs reflective checks on given subclass to verify that it doesn't
 * override security-sensitive non-final methods.  Returns true if subclass
 * is "safe", false otherwise.
 */
private static boolean auditSubclass(final Class<?> subcl) {
```

```
    Boolean result = AccessController.doPrivileged(
        new PrivilegedAction<Boolean>() {
            public Boolean run() {
                for (Class<?> cl = subcl;
                     cl != ObjectOutputStream.class;
                     cl = cl.getSuperclass())
                {
                    try {
                        cl.getDeclaredMethod(
                            "writeUnshared", new Class<?>[] { Object.class });
                        return Boolean.FALSE;
                    } catch (NoSuchMethodException ex) {
                    }
                    try {
                        cl.getDeclaredMethod("putFields", (Class<?>[]) null);
                        return Boolean.FALSE;
                    } catch (NoSuchMethodException ex) {
                    }
                }
                return Boolean.TRUE;
            }
        }
    );
    return result.booleanValue();
}

/**
 * Clears internal data structures.
 */
private void clear() {
    subs.clear();
    handles.clear();
}

/**
 * Underlying writeObject/writeUnshared implementation.
 */
private void writeObject0(Object obj, boolean unshared)
    throws IOException
{
    boolean oldMode = bout.setBlockDataMode(false);
    depth++;
    try {
        // handle previously written and non-replaceable objects
        int h;
        if ((obj = subs.lookup(obj)) == null) {
            writeNull();
            return;
        } else if (!unshared && (h = handles.lookup(obj)) != -1) {
            writeHandle(h);
            return;
        } else if (obj instanceof Class) {
            writeClass((Class) obj, unshared);
            return;
        } else if (obj instanceof ObjectStreamClass) {
            writeClassDesc((ObjectStreamClass) obj, unshared);
            return;
        }

        // check for replacement object
        Object orig = obj;
        Class<?> cl = obj.getClass();
        ObjectStreamClass desc;
```

```
        for (;;) {
            // REMIND: skip this check for strings/arrays?
            Class<?> repCl;
            desc = ObjectStreamClass.lookup(cl, true);
            if (!desc.hasWriteReplaceMethod() ||
                (obj = desc.invokeWriteReplace(obj)) == null ||
                (repCl = obj.getClass()) == cl)
            {
                break;
            }
            cl = repCl;
        }
        if (enableReplace) {
            Object rep = replaceObject(obj);
            if (rep != obj && rep != null) {
                cl = rep.getClass();
                desc = ObjectStreamClass.lookup(cl, true);
            }
            obj = rep;
        }

        // if object replaced, run through original checks a second time
        if (obj != orig) {
            subs.assign(orig, obj);
            if (obj == null) {
                writeNull();
                return;
            } else if (!unshared && (h = handles.lookup(obj)) != -1) {
                writeHandle(h);
                return;
            } else if (obj instanceof Class) {
                writeClass((Class) obj, unshared);
                return;
            } else if (obj instanceof ObjectStreamClass) {
                writeClassDesc((ObjectStreamClass) obj, unshared);
                return;
            }
        }

        // remaining cases
        if (obj instanceof String) {
            writeString((String) obj, unshared);
        } else if (cl.isArray()) {
            writeArray(obj, desc, unshared);
        } else if (obj instanceof Enum) {
            writeEnum((Enum<?>) obj, desc, unshared);
        } else if (obj instanceof Serializable) {
            writeOrdinaryObject(obj, desc, unshared);
        } else {
            if (extendedDebugInfo) {
                throw new NotSerializableException(
                    cl.getName() + "\n" + debugInfoStack.toString());
            } else {
                throw new NotSerializableException(cl.getName());
            }
        }
    } finally {
        depth--;
        bout.setBlockDataMode(oldMode);
    }
}

/**
```

```
 * Writes null code to stream.
 */
private void writeNull() throws IOException {
    bout.writeByte(TC_NULL);
}

/**
 * Writes given object handle to stream.
 */
private void writeHandle(int handle) throws IOException {
    bout.writeByte(TC_REFERENCE);
    bout.writeInt(baseWireHandle + handle);
}

/**
 * Writes representation of given class to stream.
 */
private void writeClass(Class<?> cl, boolean unshared) throws IOException {
    bout.writeByte(TC_CLASS);
    writeClassDesc(ObjectStreamClass.lookup(cl, true), false);
    handles.assign(unshared ? null : cl);
}

/**
 * Writes representation of given class descriptor to stream.
 */
private void writeClassDesc(ObjectStreamClass desc, boolean unshared)
    throws IOException
{
    int handle;
    if (desc == null) {
        writeNull();
    } else if (!unshared && (handle = handles.lookup(desc)) != -1) {
        writeHandle(handle);
    } else if (desc.isProxy()) {
        writeProxyDesc(desc, unshared);
    } else {
        writeNonProxyDesc(desc, unshared);
    }
}

private boolean isCustomSubclass() {
    // Return true if this class is a custom subclass of ObjectOutputStream
    return getClass().getClassLoader()
               != ObjectOutputStream.class.getClassLoader();
}

/**
 * Writes class descriptor representing a dynamic proxy class to stream.
 */
private void writeProxyDesc(ObjectStreamClass desc, boolean unshared)
    throws IOException
{
    bout.writeByte(TC_PROXYCLASSDESC);
    handles.assign(unshared ? null : desc);

    Class<?> cl = desc.forClass();
    Class<?>[] ifaces = cl.getInterfaces();
    bout.writeInt(ifaces.length);
    for (int i = 0; i < ifaces.length; i++) {
        bout.writeUTF(ifaces[i].getName());
    }
```

```
    bout.setBlockDataMode(true);
    if (cl != null && isCustomSubclass()) {
        ReflectUtil.checkPackageAccess(cl);
    }
    annotateProxyClass(cl);
    bout.setBlockDataMode(false);
    bout.writeByte(TC_ENDBLOCKDATA);

    writeClassDesc(desc.getSuperDesc(), false);
}

/**
 * Writes class descriptor representing a standard (i.e., not a dynamic
 * proxy) class to stream.
 */
private void writeNonProxyDesc(ObjectStreamClass desc, boolean unshared)
    throws IOException
{
    bout.writeByte(TC_CLASSDESC);
    handles.assign(unshared ? null : desc);

    if (protocol == PROTOCOL_VERSION_1) {
        // do not invoke class descriptor write hook with old protocol
        desc.writeNonProxy(this);
    } else {
        writeClassDescriptor(desc);
    }

    Class<?> cl = desc.forClass();
    bout.setBlockDataMode(true);
    if (cl != null && isCustomSubclass()) {
        ReflectUtil.checkPackageAccess(cl);
    }
    annotateClass(cl);
    bout.setBlockDataMode(false);
    bout.writeByte(TC_ENDBLOCKDATA);

    writeClassDesc(desc.getSuperDesc(), false);
}

/**
 * Writes given string to stream, using standard or long UTF format
 * depending on string length.
 */
private void writeString(String str, boolean unshared) throws IOException {
    handles.assign(unshared ? null : str);
    long utflen = bout.getUTFLength(str);
    if (utflen <= 0xFFFF) {
        bout.writeByte(TC_STRING);
        bout.writeUTF(str, utflen);
    } else {
        bout.writeByte(TC_LONGSTRING);
        bout.writeLongUTF(str, utflen);
    }
}

/**
 * Writes given array object to stream.
 */
private void writeArray(Object array,
                        ObjectStreamClass desc,
                        boolean unshared)
    throws IOException
```

```
{
    bout.writeByte(TC_ARRAY);
    writeClassDesc(desc, false);
    handles.assign(unshared ? null : array);

    Class<?> ccl = desc.forClass().getComponentType();
    if (ccl.isPrimitive()) {
        if (ccl == Integer.TYPE) {
            int[] ia = (int[]) array;
            bout.writeInt(ia.length);
            bout.writeInts(ia, 0, ia.length);
        } else if (ccl == Byte.TYPE) {
            byte[] ba = (byte[]) array;
            bout.writeInt(ba.length);
            bout.write(ba, 0, ba.length, true);
        } else if (ccl == Long.TYPE) {
            long[] ja = (long[]) array;
            bout.writeInt(ja.length);
            bout.writeLongs(ja, 0, ja.length);
        } else if (ccl == Float.TYPE) {
            float[] fa = (float[]) array;
            bout.writeInt(fa.length);
            bout.writeFloats(fa, 0, fa.length);
        } else if (ccl == Double.TYPE) {
            double[] da = (double[]) array;
            bout.writeInt(da.length);
            bout.writeDoubles(da, 0, da.length);
        } else if (ccl == Short.TYPE) {
            short[] sa = (short[]) array;
            bout.writeInt(sa.length);
            bout.writeShorts(sa, 0, sa.length);
        } else if (ccl == Character.TYPE) {
            char[] ca = (char[]) array;
            bout.writeInt(ca.length);
            bout.writeChars(ca, 0, ca.length);
        } else if (ccl == Boolean.TYPE) {
            boolean[] za = (boolean[]) array;
            bout.writeInt(za.length);
            bout.writeBooleans(za, 0, za.length);
        } else {
            throw new InternalError();
        }
    } else {
        Object[] objs = (Object[]) array;
        int len = objs.length;
        bout.writeInt(len);
        if (extendedDebugInfo) {
            debugInfoStack.push(
                "array (class \"" + array.getClass().getName() +
                "\", size: " + len  + ")");
        }
        try {
            for (int i = 0; i < len; i++) {
                if (extendedDebugInfo) {
                    debugInfoStack.push(
                        "element of array (index: " + i + ")");
                }
                try {
                    writeObject0(objs[i], false);
                } finally {
                    if (extendedDebugInfo) {
                        debugInfoStack.pop();
                    }
```

```
                }
            }
        } finally {
            if (extendedDebugInfo) {
                debugInfoStack.pop();
            }
        }
    }
}

/**
 * Writes given enum constant to stream.
 */
private void writeEnum(Enum<?> en,
                       ObjectStreamClass desc,
                       boolean unshared)
    throws IOException
{
    bout.writeByte(TC_ENUM);
    ObjectStreamClass sdesc = desc.getSuperDesc();
    writeClassDesc((sdesc.forClass() == Enum.class) ? desc : sdesc, false);
    handles.assign(unshared ? null : en);
    writeString(en.name(), false);
}

/**
 * Writes representation of a "ordinary" (i.e., not a String, Class,
 * ObjectStreamClass, array, or enum constant) serializable object to the
 * stream.
 */
private void writeOrdinaryObject(Object obj,
                                 ObjectStreamClass desc,
                                 boolean unshared)
    throws IOException
{
    if (extendedDebugInfo) {
        debugInfoStack.push(
            (depth == 1 ? "root " : "") + "object (class \"" +
            obj.getClass().getName() + "\", " + obj.toString() + ")");
    }
    try {
        desc.checkSerialize();

        bout.writeByte(TC_OBJECT);
        writeClassDesc(desc, false);
        handles.assign(unshared ? null : obj);
        if (desc.isExternalizable() && !desc.isProxy()) {
            writeExternalData((Externalizable) obj);
        } else {
            writeSerialData(obj, desc);
        }
    } finally {
        if (extendedDebugInfo) {
            debugInfoStack.pop();
        }
    }
}

/**
 * Writes externalizable data of given object by invoking its
 * writeExternal() method.
 */
private void writeExternalData(Externalizable obj) throws IOException {
```

```
    PutFieldImpl oldPut = curPut;
    curPut = null;

    if (extendedDebugInfo) {
        debugInfoStack.push("writeExternal data");
    }
    SerialCallbackContext oldContext = curContext;
    try {
        curContext = null;
        if (protocol == PROTOCOL_VERSION_1) {
            obj.writeExternal(this);
        } else {
            bout.setBlockDataMode(true);
            obj.writeExternal(this);
            bout.setBlockDataMode(false);
            bout.writeByte(TC_ENDBLOCKDATA);
        }
    } finally {
        curContext = oldContext;
        if (extendedDebugInfo) {
            debugInfoStack.pop();
        }
    }

    curPut = oldPut;
}

/**
 * Writes instance data for each serializable class of given object, from
 * superclass to subclass.
 */
private void writeSerialData(Object obj, ObjectStreamClass desc)
    throws IOException
{
    ObjectStreamClass.ClassDataSlot[] slots = desc.getClassDataLayout();
    for (int i = 0; i < slots.length; i++) {
        ObjectStreamClass slotDesc = slots[i].desc;
        if (slotDesc.hasWriteObjectMethod()) {
            PutFieldImpl oldPut = curPut;
            curPut = null;
            SerialCallbackContext oldContext = curContext;

            if (extendedDebugInfo) {
                debugInfoStack.push(
                    "custom writeObject data (class \"" +
                    slotDesc.getName() + "\")");
            }
            try {
                curContext = new SerialCallbackContext(obj, slotDesc);
                bout.setBlockDataMode(true);
                slotDesc.invokeWriteObject(obj, this);
                bout.setBlockDataMode(false);
                bout.writeByte(TC_ENDBLOCKDATA);
            } finally {
                curContext.setUsed();
                curContext = oldContext;
                if (extendedDebugInfo) {
                    debugInfoStack.pop();
                }
            }

            curPut = oldPut;
        } else {
```

```
            defaultWriteFields(obj, slotDesc);
        }
    }
}

/**
 * Fetches and writes values of serializable fields of given object to
 * stream.  The given class descriptor specifies which field values to
 * write, and in which order they should be written.
 */
private void defaultWriteFields(Object obj, ObjectStreamClass desc)
    throws IOException
{
    Class<?> cl = desc.forClass();
    if (cl != null && obj != null && !cl.isInstance(obj)) {
        throw new ClassCastException();
    }

    desc.checkDefaultSerialize();

    int primDataSize = desc.getPrimDataSize();
    if (primVals == null || primVals.length < primDataSize) {
        primVals = new byte[primDataSize];
    }
    desc.getPrimFieldValues(obj, primVals);
    bout.write(primVals, 0, primDataSize, false);

    ObjectStreamField[] fields = desc.getFields(false);
    Object[] objVals = new Object[desc.getNumObjFields()];
    int numPrimFields = fields.length - objVals.length;
    desc.getObjFieldValues(obj, objVals);
    for (int i = 0; i < objVals.length; i++) {
        if (extendedDebugInfo) {
            debugInfoStack.push(
                "field (class \"" + desc.getName() + "\", name: \"" +
                fields[numPrimFields + i].getName() + "\", type: \"" +
                fields[numPrimFields + i].getType() + "\")");
        }
        try {
            writeObject0(objVals[i],
                         fields[numPrimFields + i].isUnshared());
        } finally {
            if (extendedDebugInfo) {
                debugInfoStack.pop();
            }
        }
    }
}

/**
 * Attempts to write to stream fatal IOException that has caused
 * serialization to abort.
 */
private void writeFatalException(IOException ex) throws IOException {
    /*
     * Note: the serialization specification states that if a second
     * IOException occurs while attempting to serialize the original fatal
     * exception to the stream, then a StreamCorruptedException should be
     * thrown (section 2.1).  However, due to a bug in previous
     * implementations of serialization, StreamCorruptedExceptions were
     * rarely (if ever) actually thrown--the "root" exceptions from
     * underlying streams were thrown instead.  This historical behavior is
     * followed here for consistency.
```

```
         */
        clear();
        boolean oldMode = bout.setBlockDataMode(false);
        try {
            bout.writeByte(TC_EXCEPTION);
            writeObject0(ex, false);
            clear();
        } finally {
            bout.setBlockDataMode(oldMode);
        }
    }

    /**
     * Converts specified span of float values into byte values.
     */
    // REMIND: remove once hotspot inlines Float.floatToIntBits
    private static native void floatsToBytes(float[] src, int srcpos,
                                             byte[] dst, int dstpos,
                                             int nfloats);

    /**
     * Converts specified span of double values into byte values.
     */
    // REMIND: remove once hotspot inlines Double.doubleToLongBits
    private static native void doublesToBytes(double[] src, int srcpos,
                                              byte[] dst, int dstpos,
                                              int ndoubles);

    /**
     * Default PutField implementation.
     */
    private class PutFieldImpl extends PutField {

        /** class descriptor describing serializable fields */
        private final ObjectStreamClass desc;
        /** primitive field values */
        private final byte[] primVals;
        /** object field values */
        private final Object[] objVals;

        /**
         * Creates PutFieldImpl object for writing fields defined in given
         * class descriptor.
         */
        PutFieldImpl(ObjectStreamClass desc) {
            this.desc = desc;
            primVals = new byte[desc.getPrimDataSize()];
            objVals = new Object[desc.getNumObjFields()];
        }

        public void put(String name, boolean val) {
            Bits.putBoolean(primVals, getFieldOffset(name, Boolean.TYPE), val);
        }

        public void put(String name, byte val) {
            primVals[getFieldOffset(name, Byte.TYPE)] = val;
        }

        public void put(String name, char val) {
            Bits.putChar(primVals, getFieldOffset(name, Character.TYPE), val);
        }

        public void put(String name, short val) {
```

```
        Bits.putShort(primVals, getFieldOffset(name, Short.TYPE), val);
    }

    public void put(String name, int val) {
        Bits.putInt(primVals, getFieldOffset(name, Integer.TYPE), val);
    }

    public void put(String name, float val) {
        Bits.putFloat(primVals, getFieldOffset(name, Float.TYPE), val);
    }

    public void put(String name, long val) {
        Bits.putLong(primVals, getFieldOffset(name, Long.TYPE), val);
    }

    public void put(String name, double val) {
        Bits.putDouble(primVals, getFieldOffset(name, Double.TYPE), val);
    }

    public void put(String name, Object val) {
        objVals[getFieldOffset(name, Object.class)] = val;
    }

    // deprecated in ObjectOutputStream.PutField
    public void write(ObjectOutput out) throws IOException {
        /*
         * Applications should *not* use this method to write PutField
         * data, as it will lead to stream corruption if the PutField
         * object writes any primitive data (since block data mode is not
         * unset/set properly, as is done in OOS.writeFields()).  This
         * broken implementation is being retained solely for behavioral
         * compatibility, in order to support applications which use
         * OOS.PutField.write() for writing only non-primitive data.
         *
         * Serialization of unshared objects is not implemented here since
         * it is not necessary for backwards compatibility; also, unshared
         * semantics may not be supported by the given ObjectOutput
         * instance.  Applications which write unshared objects using the
         * PutField API must use OOS.writeFields().
         */
        if (ObjectOutputStream.this != out) {
            throw new IllegalArgumentException("wrong stream");
        }
        out.write(primVals, 0, primVals.length);

        ObjectStreamField[] fields = desc.getFields(false);
        int numPrimFields = fields.length - objVals.length;
        // REMIND: warn if numPrimFields > 0?
        for (int i = 0; i < objVals.length; i++) {
            if (fields[numPrimFields + i].isUnshared()) {
                throw new IOException("cannot write unshared object");
            }
            out.writeObject(objVals[i]);
        }
    }

    /**
     * Writes buffered primitive data and object fields to stream.
     */
    void writeFields() throws IOException {
        bout.write(primVals, 0, primVals.length, false);

        ObjectStreamField[] fields = desc.getFields(false);
```

```
            int numPrimFields = fields.length - objVals.length;
            for (int i = 0; i < objVals.length; i++) {
                if (extendedDebugInfo) {
                    debugInfoStack.push(
                        "field (class \"" + desc.getName() + "\", name: \"" +
                        fields[numPrimFields + i].getName() + "\", type: \"" +
                        fields[numPrimFields + i].getType() + "\")");
                }
                try {
                    writeObject0(objVals[i],
                                 fields[numPrimFields + i].isUnshared());
                } finally {
                    if (extendedDebugInfo) {
                        debugInfoStack.pop();
                    }
                }
            }
        }

        /**
         * Returns offset of field with given name and type.  A specified type
         * of null matches all types, Object.class matches all non-primitive
         * types, and any other non-null type matches assignable types only.
         * Throws IllegalArgumentException if no matching field found.
         */
        private int getFieldOffset(String name, Class<?> type) {
            ObjectStreamField field = desc.getField(name, type);
            if (field == null) {
                throw new IllegalArgumentException("no such field " + name +
                                                   " with type " + type);
            }
            return field.getOffset();
        }
    }

    /**
     * Buffered output stream with two modes: in default mode, outputs data in
     * same format as DataOutputStream; in "block data" mode, outputs data
     * bracketed by block data markers (see object serialization specification
     * for details).
     */
    private static class BlockDataOutputStream
        extends OutputStream implements DataOutput
    {
        /** maximum data block length */
        private static final int MAX_BLOCK_SIZE = 1024;
        /** maximum data block header length */
        private static final int MAX_HEADER_SIZE = 5;
        /** (tunable) length of char buffer (for writing strings) */
        private static final int CHAR_BUF_SIZE = 256;

        /** buffer for writing general/block data */
        private final byte[] buf = new byte[MAX_BLOCK_SIZE];
        /** buffer for writing block data headers */
        private final byte[] hbuf = new byte[MAX_HEADER_SIZE];
        /** char buffer for fast string writes */
        private final char[] cbuf = new char[CHAR_BUF_SIZE];

        /** block data mode */
        private boolean blkmode = false;
        /** current offset into buf */
        private int pos = 0;
```

```
/** underlying output stream */
private final OutputStream out;
/** loopback stream (for data writes that span data blocks) */
private final DataOutputStream dout;

/**
 * Creates new BlockDataOutputStream on top of given underlying stream.
 * Block data mode is turned off by default.
 */
BlockDataOutputStream(OutputStream out) {
    this.out = out;
    dout = new DataOutputStream(this);
}

/**
 * Sets block data mode to the given mode (true == on, false == off)
 * and returns the previous mode value.  If the new mode is the same as
 * the old mode, no action is taken.  If the new mode differs from the
 * old mode, any buffered data is flushed before switching to the new
 * mode.
 */
boolean setBlockDataMode(boolean mode) throws IOException {
    if (blkmode == mode) {
        return blkmode;
    }
    drain();
    blkmode = mode;
    return !blkmode;
}

/**
 * Returns true if the stream is currently in block data mode, false
 * otherwise.
 */
boolean getBlockDataMode() {
    return blkmode;
}

/* ---------------- generic output stream methods ---------------- */
/*
 * The following methods are equivalent to their counterparts in
 * OutputStream, except that they partition written data into data
 * blocks when in block data mode.
 */

public void write(int b) throws IOException {
    if (pos >= MAX_BLOCK_SIZE) {
        drain();
    }
    buf[pos++] = (byte) b;
}

public void write(byte[] b) throws IOException {
    write(b, 0, b.length, false);
}

public void write(byte[] b, int off, int len) throws IOException {
    write(b, off, len, false);
}

public void flush() throws IOException {
    drain();
    out.flush();
```

```
}

public void close() throws IOException {
    flush();
    out.close();
}

/**
 * Writes specified span of byte values from given array.  If copy is
 * true, copies the values to an intermediate buffer before writing
 * them to underlying stream (to avoid exposing a reference to the
 * original byte array).
 */
void write(byte[] b, int off, int len, boolean copy)
    throws IOException
{
    if (!(copy || blkmode)) {           // write directly
        drain();
        out.write(b, off, len);
        return;
    }

    while (len > 0) {
        if (pos >= MAX_BLOCK_SIZE) {
            drain();
        }
        if (len >= MAX_BLOCK_SIZE && !copy && pos == 0) {
            // avoid unnecessary copy
            writeBlockHeader(MAX_BLOCK_SIZE);
            out.write(b, off, MAX_BLOCK_SIZE);
            off += MAX_BLOCK_SIZE;
            len -= MAX_BLOCK_SIZE;
        } else {
            int wlen = Math.min(len, MAX_BLOCK_SIZE - pos);
            System.arraycopy(b, off, buf, pos, wlen);
            pos += wlen;
            off += wlen;
            len -= wlen;
        }
    }
}

/**
 * Writes all buffered data from this stream to the underlying stream,
 * but does not flush underlying stream.
 */
void drain() throws IOException {
    if (pos == 0) {
        return;
    }
    if (blkmode) {
        writeBlockHeader(pos);
    }
    out.write(buf, 0, pos);
    pos = 0;
}

/**
 * Writes block data header.  Data blocks shorter than 256 bytes are
 * prefixed with a 2-byte header; all others start with a 5-byte
 * header.
 */
private void writeBlockHeader(int len) throws IOException {
```

```
        if (len <= 0xFF) {
            hbuf[0] = TC_BLOCKDATA;
            hbuf[1] = (byte) len;
            out.write(hbuf, 0, 2);
        } else {
            hbuf[0] = TC_BLOCKDATALONG;
            Bits.putInt(hbuf, 1, len);
            out.write(hbuf, 0, 5);
        }
    }


    /* ----------------- primitive data output methods ----------------- */
    /*
     * The following methods are equivalent to their counterparts in
     * DataOutputStream, except that they partition written data into data
     * blocks when in block data mode.
     */

    public void writeBoolean(boolean v) throws IOException {
        if (pos >= MAX_BLOCK_SIZE) {
            drain();
        }
        Bits.putBoolean(buf, pos++, v);
    }

    public void writeByte(int v) throws IOException {
        if (pos >= MAX_BLOCK_SIZE) {
            drain();
        }
        buf[pos++] = (byte) v;
    }

    public void writeChar(int v) throws IOException {
        if (pos + 2 <= MAX_BLOCK_SIZE) {
            Bits.putChar(buf, pos, (char) v);
            pos += 2;
        } else {
            dout.writeChar(v);
        }
    }

    public void writeShort(int v) throws IOException {
        if (pos + 2 <= MAX_BLOCK_SIZE) {
            Bits.putShort(buf, pos, (short) v);
            pos += 2;
        } else {
            dout.writeShort(v);
        }
    }

    public void writeInt(int v) throws IOException {
        if (pos + 4 <= MAX_BLOCK_SIZE) {
            Bits.putInt(buf, pos, v);
            pos += 4;
        } else {
            dout.writeInt(v);
        }
    }

    public void writeFloat(float v) throws IOException {
        if (pos + 4 <= MAX_BLOCK_SIZE) {
            Bits.putFloat(buf, pos, v);
```

```
            pos += 4;
        } else {
            dout.writeFloat(v);
        }
    }

    public void writeLong(long v) throws IOException {
        if (pos + 8 <= MAX_BLOCK_SIZE) {
            Bits.putLong(buf, pos, v);
            pos += 8;
        } else {
            dout.writeLong(v);
        }
    }

    public void writeDouble(double v) throws IOException {
        if (pos + 8 <= MAX_BLOCK_SIZE) {
            Bits.putDouble(buf, pos, v);
            pos += 8;
        } else {
            dout.writeDouble(v);
        }
    }

    public void writeBytes(String s) throws IOException {
        int endoff = s.length();
        int cpos = 0;
        int csize = 0;
        for (int off = 0; off < endoff; ) {
            if (cpos >= csize) {
                cpos = 0;
                csize = Math.min(endoff - off, CHAR_BUF_SIZE);
                s.getChars(off, off + csize, cbuf, 0);
            }
            if (pos >= MAX_BLOCK_SIZE) {
                drain();
            }
            int n = Math.min(csize - cpos, MAX_BLOCK_SIZE - pos);
            int stop = pos + n;
            while (pos < stop) {
                buf[pos++] = (byte) cbuf[cpos++];
            }
            off += n;
        }
    }

    public void writeChars(String s) throws IOException {
        int endoff = s.length();
        for (int off = 0; off < endoff; ) {
            int csize = Math.min(endoff - off, CHAR_BUF_SIZE);
            s.getChars(off, off + csize, cbuf, 0);
            writeChars(cbuf, 0, csize);
            off += csize;
        }
    }

    public void writeUTF(String s) throws IOException {
        writeUTF(s, getUTFLength(s));
    }


    /* -------------- primitive data array output methods -------------- */
    /*
```

```
     * The following methods write out spans of primitive data values.
     * Though equivalent to calling the corresponding primitive write
     * methods repeatedly, these methods are optimized for writing groups
     * of primitive data values more efficiently.
     */

    void writeBooleans(boolean[] v, int off, int len) throws IOException {
        int endoff = off + len;
        while (off < endoff) {
            if (pos >= MAX_BLOCK_SIZE) {
                drain();
            }
            int stop = Math.min(endoff, off + (MAX_BLOCK_SIZE - pos));
            while (off < stop) {
                Bits.putBoolean(buf, pos++, v[off++]);
            }
        }
    }

    void writeChars(char[] v, int off, int len) throws IOException {
        int limit = MAX_BLOCK_SIZE - 2;
        int endoff = off + len;
        while (off < endoff) {
            if (pos <= limit) {
                int avail = (MAX_BLOCK_SIZE - pos) >> 1;
                int stop = Math.min(endoff, off + avail);
                while (off < stop) {
                    Bits.putChar(buf, pos, v[off++]);
                    pos += 2;
                }
            } else {
                dout.writeChar(v[off++]);
            }
        }
    }

    void writeShorts(short[] v, int off, int len) throws IOException {
        int limit = MAX_BLOCK_SIZE - 2;
        int endoff = off + len;
        while (off < endoff) {
            if (pos <= limit) {
                int avail = (MAX_BLOCK_SIZE - pos) >> 1;
                int stop = Math.min(endoff, off + avail);
                while (off < stop) {
                    Bits.putShort(buf, pos, v[off++]);
                    pos += 2;
                }
            } else {
                dout.writeShort(v[off++]);
            }
        }
    }

    void writeInts(int[] v, int off, int len) throws IOException {
        int limit = MAX_BLOCK_SIZE - 4;
        int endoff = off + len;
        while (off < endoff) {
            if (pos <= limit) {
                int avail = (MAX_BLOCK_SIZE - pos) >> 2;
                int stop = Math.min(endoff, off + avail);
                while (off < stop) {
                    Bits.putInt(buf, pos, v[off++]);
                    pos += 4;
```

```
            }
        } else {
            dout.writeInt(v[off++]);
        }
    }
}

void writeFloats(float[] v, int off, int len) throws IOException {
    int limit = MAX_BLOCK_SIZE - 4;
    int endoff = off + len;
    while (off < endoff) {
        if (pos <= limit) {
            int avail = (MAX_BLOCK_SIZE - pos) >> 2;
            int chunklen = Math.min(endoff - off, avail);
            floatsToBytes(v, off, buf, pos, chunklen);
            off += chunklen;
            pos += chunklen << 2;
        } else {
            dout.writeFloat(v[off++]);
        }
    }
}

void writeLongs(long[] v, int off, int len) throws IOException {
    int limit = MAX_BLOCK_SIZE - 8;
    int endoff = off + len;
    while (off < endoff) {
        if (pos <= limit) {
            int avail = (MAX_BLOCK_SIZE - pos) >> 3;
            int stop = Math.min(endoff, off + avail);
            while (off < stop) {
                Bits.putLong(buf, pos, v[off++]);
                pos += 8;
            }
        } else {
            dout.writeLong(v[off++]);
        }
    }
}

void writeDoubles(double[] v, int off, int len) throws IOException {
    int limit = MAX_BLOCK_SIZE - 8;
    int endoff = off + len;
    while (off < endoff) {
        if (pos <= limit) {
            int avail = (MAX_BLOCK_SIZE - pos) >> 3;
            int chunklen = Math.min(endoff - off, avail);
            doublesToBytes(v, off, buf, pos, chunklen);
            off += chunklen;
            pos += chunklen << 3;
        } else {
            dout.writeDouble(v[off++]);
        }
    }
}

/**
 * Returns the length in bytes of the UTF encoding of the given string.
 */
long getUTFLength(String s) {
    int len = s.length();
    long utflen = 0;
    for (int off = 0; off < len; ) {
```

```
        int csize = Math.min(len - off, CHAR_BUF_SIZE);
        s.getChars(off, off + csize, cbuf, 0);
        for (int cpos = 0; cpos < csize; cpos++) {
            char c = cbuf[cpos];
            if (c >= 0x0001 && c <= 0x007F) {
                utflen++;
            } else if (c > 0x07FF) {
                utflen += 3;
            } else {
                utflen += 2;
            }
        }
        off += csize;
    }
    return utflen;
}

/**
 * Writes the given string in UTF format.  This method is used in
 * situations where the UTF encoding length of the string is already
 * known; specifying it explicitly avoids a prescan of the string to
 * determine its UTF length.
 */
void writeUTF(String s, long utflen) throws IOException {
    if (utflen > 0xFFFFL) {
        throw new UTFDataFormatException();
    }
    writeShort((int) utflen);
    if (utflen == (long) s.length()) {
        writeBytes(s);
    } else {
        writeUTFBody(s);
    }
}

/**
 * Writes given string in "long" UTF format.  "Long" UTF format is
 * identical to standard UTF, except that it uses an 8 byte header
 * (instead of the standard 2 bytes) to convey the UTF encoding length.
 */
void writeLongUTF(String s) throws IOException {
    writeLongUTF(s, getUTFLength(s));
}

/**
 * Writes given string in "long" UTF format, where the UTF encoding
 * length of the string is already known.
 */
void writeLongUTF(String s, long utflen) throws IOException {
    writeLong(utflen);
    if (utflen == (long) s.length()) {
        writeBytes(s);
    } else {
        writeUTFBody(s);
    }
}

/**
 * Writes the "body" (i.e., the UTF representation minus the 2-byte or
 * 8-byte length header) of the UTF encoding for the given string.
 */
private void writeUTFBody(String s) throws IOException {
    int limit = MAX_BLOCK_SIZE - 3;
```

```
        int len = s.length();
        for (int off = 0; off < len; ) {
            int csize = Math.min(len - off, CHAR_BUF_SIZE);
            s.getChars(off, off + csize, cbuf, 0);
            for (int cpos = 0; cpos < csize; cpos++) {
                char c = cbuf[cpos];
                if (pos <= limit) {
                    if (c <= 0x007F && c != 0) {
                        buf[pos++] = (byte) c;
                    } else if (c > 0x07FF) {
                        buf[pos + 2] = (byte) (0x80 | ((c >> 0) & 0x3F));
                        buf[pos + 1] = (byte) (0x80 | ((c >> 6) & 0x3F));
                        buf[pos + 0] = (byte) (0xE0 | ((c >> 12) & 0x0F));
                        pos += 3;
                    } else {
                        buf[pos + 1] = (byte) (0x80 | ((c >> 0) & 0x3F));
                        buf[pos + 0] = (byte) (0xC0 | ((c >> 6) & 0x1F));
                        pos += 2;
                    }
                } else {    // write one byte at a time to normalize block
                    if (c <= 0x007F && c != 0) {
                        write(c);
                    } else if (c > 0x07FF) {
                        write(0xE0 | ((c >> 12) & 0x0F));
                        write(0x80 | ((c >> 6) & 0x3F));
                        write(0x80 | ((c >> 0) & 0x3F));
                    } else {
                        write(0xC0 | ((c >> 6) & 0x1F));
                        write(0x80 | ((c >> 0) & 0x3F));
                    }
                }
            }
            off += csize;
        }
    }
}

/**
 * Lightweight identity hash table which maps objects to integer handles,
 * assigned in ascending order.
 */
private static class HandleTable {

    /* number of mappings in table/next available handle */
    private int size;
    /* size threshold determining when to expand hash spine */
    private int threshold;
    /* factor for computing size threshold */
    private final float loadFactor;
    /* maps hash value -> candidate handle value */
    private int[] spine;
    /* maps handle value -> next candidate handle value */
    private int[] next;
    /* maps handle value -> associated object */
    private Object[] objs;

    /**
     * Creates new HandleTable with given capacity and load factor.
     */
    HandleTable(int initialCapacity, float loadFactor) {
        this.loadFactor = loadFactor;
        spine = new int[initialCapacity];
        next = new int[initialCapacity];
```

```
    objs = new Object[initialCapacity];
    threshold = (int) (initialCapacity * loadFactor);
    clear();
}

/**
 * Assigns next available handle to given object, and returns handle
 * value.  Handles are assigned in ascending order starting at 0.
 */
int assign(Object obj) {
    if (size >= next.length) {
        growEntries();
    }
    if (size >= threshold) {
        growSpine();
    }
    insert(obj, size);
    return size++;
}

/**
 * Looks up and returns handle associated with given object, or -1 if
 * no mapping found.
 */
int lookup(Object obj) {
    if (size == 0) {
        return -1;
    }
    int index = hash(obj) % spine.length;
    for (int i = spine[index]; i >= 0; i = next[i]) {
        if (objs[i] == obj) {
            return i;
        }
    }
    return -1;
}

/**
 * Resets table to its initial (empty) state.
 */
void clear() {
    Arrays.fill(spine, -1);
    Arrays.fill(objs, 0, size, null);
    size = 0;
}

/**
 * Returns the number of mappings currently in table.
 */
int size() {
    return size;
}

/**
 * Inserts mapping object -> handle mapping into table.  Assumes table
 * is large enough to accommodate new mapping.
 */
private void insert(Object obj, int handle) {
    int index = hash(obj) % spine.length;
    objs[handle] = obj;
    next[handle] = spine[index];
    spine[index] = handle;
}
```

```
    /**
     * Expands the hash "spine" -- equivalent to increasing the number of
     * buckets in a conventional hash table.
     */
    private void growSpine() {
        spine = new int[(spine.length << 1) + 1];
        threshold = (int) (spine.length * loadFactor);
        Arrays.fill(spine, -1);
        for (int i = 0; i < size; i++) {
            insert(objs[i], i);
        }
    }

    /**
     * Increases hash table capacity by lengthening entry arrays.
     */
    private void growEntries() {
        int newLength = (next.length << 1) + 1;
        int[] newNext = new int[newLength];
        System.arraycopy(next, 0, newNext, 0, size);
        next = newNext;

        Object[] newObjs = new Object[newLength];
        System.arraycopy(objs, 0, newObjs, 0, size);
        objs = newObjs;
    }

    /**
     * Returns hash value for given object.
     */
    private int hash(Object obj) {
        return System.identityHashCode(obj) & 0x7FFFFFFF;
    }
}

/**
 * Lightweight identity hash table which maps objects to replacement
 * objects.
 */
private static class ReplaceTable {

    /* maps object -> index */
    private final HandleTable htab;
    /* maps index -> replacement object */
    private Object[] reps;

    /**
     * Creates new ReplaceTable with given capacity and load factor.
     */
    ReplaceTable(int initialCapacity, float loadFactor) {
        htab = new HandleTable(initialCapacity, loadFactor);
        reps = new Object[initialCapacity];
    }

    /**
     * Enters mapping from object to replacement object.
     */
    void assign(Object obj, Object rep) {
        int index = htab.assign(obj);
        while (index >= reps.length) {
            grow();
        }
```

```
        reps[index] = rep;
    }

    /**
     * Looks up and returns replacement for given object.  If no
     * replacement is found, returns the lookup object itself.
     */
    Object lookup(Object obj) {
        int index = htab.lookup(obj);
        return (index >= 0) ? reps[index] : obj;
    }

    /**
     * Resets table to its initial (empty) state.
     */
    void clear() {
        Arrays.fill(reps, 0, htab.size(), null);
        htab.clear();
    }

    /**
     * Returns the number of mappings currently in table.
     */
    int size() {
        return htab.size();
    }

    /**
     * Increases table capacity.
     */
    private void grow() {
        Object[] newReps = new Object[(reps.length << 1) + 1];
        System.arraycopy(reps, 0, newReps, 0, reps.length);
        reps = newReps;
    }
}

/**
 * Stack to keep debug information about the state of the
 * serialization process, for embedding in exception messages.
 */
private static class DebugTraceInfoStack {
    private final List<String> stack;

    DebugTraceInfoStack() {
        stack = new ArrayList<>();
    }

    /**
     * Removes all of the elements from enclosed list.
     */
    void clear() {
        stack.clear();
    }

    /**
     * Removes the object at the top of enclosed list.
     */
    void pop() {
        stack.remove(stack.size()-1);
    }

    /**
```

```
         * Pushes a String onto the top of enclosed list.
         */
        void push(String entry) {
            stack.add("\t- " + entry);
        }

        /**
         * Returns a string representation of this object
         */
        public String toString() {
            StringBuilder buffer = new StringBuilder();
            if (!stack.isEmpty()) {
                for(int i = stack.size(); i > 0; i-- ) {
                    buffer.append(stack.get(i-1) + ((i != 1) ? "\n" : ""));
                }
            }
            return buffer.toString();
        }
    }

}
```

ObjectStreamClass.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.lang.ref.Reference;
import java.lang.ref.ReferenceQueue;
import java.lang.ref.SoftReference;
import java.lang.ref.WeakReference;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Member;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Proxy;
import java.security.AccessController;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.PrivilegedAction;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import sun.misc.Unsafe;
import sun.reflect.CallerSensitive;
import sun.reflect.Reflection;
import sun.reflect.ReflectionFactory;
import sun.reflect.misc.ReflectUtil;

/**
 * Serialization's descriptor for classes.  It contains the name and
 * serialVersionUID of the class.  The ObjectStreamClass for a specific class
 * loaded in this Java VM can be found/created using the lookup method.
```

```
 *
 * <p>The algorithm to compute the SerialVersionUID is described in
 * <a href="../../../platform/serialization/spec/class.html#4100">Object
 * Serialization Specification, Section 4.6, Stream Unique Identifiers</a>.
 *
 * @author      Mike Warres
 * @author      Roger Riggs
 * @see ObjectStreamField
 * @see <a href="../../../platform/serialization/spec/class.html">Object Serialization Specification, Section
4, Class Descriptors</a>
 * @since   JDK1.1
 */
public class ObjectStreamClass implements Serializable {

    /** serialPersistentFields value indicating no serializable fields */
    public static final ObjectStreamField[] NO_FIELDS =
        new ObjectStreamField[0];

    private static final long serialVersionUID = -6120832682080437368L;
    private static final ObjectStreamField[] serialPersistentFields =
        NO_FIELDS;

    /** reflection factory for obtaining serialization constructors */
    private static final ReflectionFactory reflFactory =
        AccessController.doPrivileged(
            new ReflectionFactory.GetReflectionFactoryAction());

    private static class Caches {
        /** cache mapping local classes -> descriptors */
        static final ConcurrentMap<WeakClassKey,Reference<?>> localDescs =
            new ConcurrentHashMap<>();

        /** cache mapping field group/local desc pairs -> field reflectors */
        static final ConcurrentMap<FieldReflectorKey,Reference<?>> reflectors =
            new ConcurrentHashMap<>();

        /** queue for WeakReferences to local classes */
        private static final ReferenceQueue<Class<?>> localDescsQueue =
            new ReferenceQueue<>();
        /** queue for WeakReferences to field reflectors keys */
        private static final ReferenceQueue<Class<?>> reflectorsQueue =
            new ReferenceQueue<>();
    }

    /** class associated with this descriptor (if any) */
    private Class<?> cl;
    /** name of class represented by this descriptor */
    private String name;
    /** serialVersionUID of represented class (null if not computed yet) */
    private volatile Long suid;

    /** true if represents dynamic proxy class */
    private boolean isProxy;
    /** true if represents enum type */
    private boolean isEnum;
    /** true if represented class implements Serializable */
    private boolean serializable;
    /** true if represented class implements Externalizable */
    private boolean externalizable;
    /** true if desc has data written by class-defined writeObject method */
    private boolean hasWriteObjectData;
    /**
     * true if desc has externalizable data written in block data format; this
```

```
     * must be true by default to accommodate ObjectInputStream subclasses which
     * override readClassDescriptor() to return class descriptors obtained from
     * ObjectStreamClass.lookup() (see 4461737)
     */
    private boolean hasBlockExternalData = true;

    /**
     * Contains information about InvalidClassException instances to be thrown
     * when attempting operations on an invalid class. Note that instances of
     * this class are immutable and are potentially shared among
     * ObjectStreamClass instances.
     */
    private static class ExceptionInfo {
        private final String className;
        private final String message;

        ExceptionInfo(String cn, String msg) {
            className = cn;
            message = msg;
        }

        /**
         * Returns (does not throw) an InvalidClassException instance created
         * from the information in this object, suitable for being thrown by
         * the caller.
         */
        InvalidClassException newInvalidClassException() {
            return new InvalidClassException(className, message);
        }
    }

    /** exception (if any) thrown while attempting to resolve class */
    private ClassNotFoundException resolveEx;
    /** exception (if any) to throw if non-enum deserialization attempted */
    private ExceptionInfo deserializeEx;
    /** exception (if any) to throw if non-enum serialization attempted */
    private ExceptionInfo serializeEx;
    /** exception (if any) to throw if default serialization attempted */
    private ExceptionInfo defaultSerializeEx;

    /** serializable fields */
    private ObjectStreamField[] fields;
    /** aggregate marshalled size of primitive fields */
    private int primDataSize;
    /** number of non-primitive fields */
    private int numObjFields;
    /** reflector for setting/getting serializable field values */
    private FieldReflector fieldRefl;
    /** data layout of serialized objects described by this class desc */
    private volatile ClassDataSlot[] dataLayout;

    /** serialization-appropriate constructor, or null if none */
    private Constructor<?> cons;
    /** class-defined writeObject method, or null if none */
    private Method writeObjectMethod;
    /** class-defined readObject method, or null if none */
    private Method readObjectMethod;
    /** class-defined readObjectNoData method, or null if none */
    private Method readObjectNoDataMethod;
    /** class-defined writeReplace method, or null if none */
    private Method writeReplaceMethod;
    /** class-defined readResolve method, or null if none */
    private Method readResolveMethod;
```

```
/** local class descriptor for represented class (may point to self) */
private ObjectStreamClass localDesc;
/** superclass descriptor appearing in stream */
private ObjectStreamClass superDesc;

/**
 * Initializes native code.
 */
private static native void initNative();
static {
    initNative();
}

/**
 * Find the descriptor for a class that can be serialized.  Creates an
 * ObjectStreamClass instance if one does not exist yet for class. Null is
 * returned if the specified class does not implement java.io.Serializable
 * or java.io.Externalizable.
 *
 * @param   cl class for which to get the descriptor
 * @return  the class descriptor for the specified class
 */
public static ObjectStreamClass lookup(Class<?> cl) {
    return lookup(cl, false);
}

/**
 * Returns the descriptor for any class, regardless of whether it
 * implements {@link Serializable}.
 *
 * @param        cl class for which to get the descriptor
 * @return       the class descriptor for the specified class
 * @since 1.6
 */
public static ObjectStreamClass lookupAny(Class<?> cl) {
    return lookup(cl, true);
}

/**
 * Returns the name of the class described by this descriptor.
 * This method returns the name of the class in the format that
 * is used by the {@link Class#getName} method.
 *
 * @return a string representing the name of the class
 */
public String getName() {
    return name;
}

/**
 * Return the serialVersionUID for this class.  The serialVersionUID
 * defines a set of classes all with the same name that have evolved from a
 * common root class and agree to be serialized and deserialized using a
 * common format.  NonSerializable classes have a serialVersionUID of 0L.
 *
 * @return  the SUID of the class described by this descriptor
 */
public long getSerialVersionUID() {
    // REMIND: synchronize instead of relying on volatile?
    if (suid == null) {
        suid = AccessController.doPrivileged(
            new PrivilegedAction<Long>() {
```

```
                public Long run() {
                    return computeDefaultSUID(cl);
                }
            }
        );
    }
    return suid.longValue();
}

/**
 * Return the class in the local VM that this version is mapped to.  Null
 * is returned if there is no corresponding local class.
 *
 * @return  the <code>Class</code> instance that this descriptor represents
 */
@CallerSensitive
public Class<?> forClass() {
    if (cl == null) {
        return null;
    }
    if (System.getSecurityManager() != null) {
        Class<?> caller = Reflection.getCallerClass();
        if (ReflectUtil.needsPackageAccessCheck(caller.getClassLoader(), cl.getClassLoader())) {
            ReflectUtil.checkPackageAccess(cl);
        }
    }
    return cl;
}

/**
 * Return an array of the fields of this serializable class.
 *
 * @return  an array containing an element for each persistent field of
 *          this class. Returns an array of length zero if there are no
 *          fields.
 * @since 1.2
 */
public ObjectStreamField[] getFields() {
    return getFields(true);
}

/**
 * Get the field of this class by name.
 *
 * @param   name the name of the data field to look for
 * @return  The ObjectStreamField object of the named field or null if
 *          there is no such named field.
 */
public ObjectStreamField getField(String name) {
    return getField(name, null);
}

/**
 * Return a string describing this ObjectStreamClass.
 */
public String toString() {
    return name + ": static final long serialVersionUID = " +
        getSerialVersionUID() + "L;";
}

/**
 * Looks up and returns class descriptor for given class, or null if class
 * is non-serializable and "all" is set to false.
```

```
     *
     * @param   cl class to look up
     * @param   all if true, return descriptors for all classes; if false, only
     *          return descriptors for serializable classes
     */
    static ObjectStreamClass lookup(Class<?> cl, boolean all) {
        if (!(all || Serializable.class.isAssignableFrom(cl))) {
            return null;
        }
        processQueue(Caches.localDescsQueue, Caches.localDescs);
        WeakClassKey key = new WeakClassKey(cl, Caches.localDescsQueue);
        Reference<?> ref = Caches.localDescs.get(key);
        Object entry = null;
        if (ref != null) {
            entry = ref.get();
        }
        EntryFuture future = null;
        if (entry == null) {
            EntryFuture newEntry = new EntryFuture();
            Reference<?> newRef = new SoftReference<>(newEntry);
            do {
                if (ref != null) {
                    Caches.localDescs.remove(key, ref);
                }
                ref = Caches.localDescs.putIfAbsent(key, newRef);
                if (ref != null) {
                    entry = ref.get();
                }
            } while (ref != null && entry == null);
            if (entry == null) {
                future = newEntry;
            }
        }

        if (entry instanceof ObjectStreamClass) {  // check common case first
            return (ObjectStreamClass) entry;
        }
        if (entry instanceof EntryFuture) {
            future = (EntryFuture) entry;
            if (future.getOwner() == Thread.currentThread()) {
                /*
                 * Handle nested call situation described by 4803747: waiting
                 * for future value to be set by a lookup() call further up the
                 * stack will result in deadlock, so calculate and set the
                 * future value here instead.
                 */
                entry = null;
            } else {
                entry = future.get();
            }
        }
        if (entry == null) {
            try {
                entry = new ObjectStreamClass(cl);
            } catch (Throwable th) {
                entry = th;
            }
            if (future.set(entry)) {
                Caches.localDescs.put(key, new SoftReference<Object>(entry));
            } else {
                // nested lookup call already set future
                entry = future.get();
            }
```

```
    }

    if (entry instanceof ObjectStreamClass) {
        return (ObjectStreamClass) entry;
    } else if (entry instanceof RuntimeException) {
        throw (RuntimeException) entry;
    } else if (entry instanceof Error) {
        throw (Error) entry;
    } else {
        throw new InternalError("unexpected entry: " + entry);
    }
}

/**
 * Placeholder used in class descriptor and field reflector lookup tables
 * for an entry in the process of being initialized.  (Internal) callers
 * which receive an EntryFuture belonging to another thread as the result
 * of a lookup should call the get() method of the EntryFuture; this will
 * return the actual entry once it is ready for use and has been set().  To
 * conserve objects, EntryFutures synchronize on themselves.
 */
private static class EntryFuture {

    private static final Object unset = new Object();
    private final Thread owner = Thread.currentThread();
    private Object entry = unset;

    /**
     * Attempts to set the value contained by this EntryFuture.  If the
     * EntryFuture's value has not been set already, then the value is
     * saved, any callers blocked in the get() method are notified, and
     * true is returned.  If the value has already been set, then no saving
     * or notification occurs, and false is returned.
     */
    synchronized boolean set(Object entry) {
        if (this.entry != unset) {
            return false;
        }
        this.entry = entry;
        notifyAll();
        return true;
    }

    /**
     * Returns the value contained by this EntryFuture, blocking if
     * necessary until a value is set.
     */
    synchronized Object get() {
        boolean interrupted = false;
        while (entry == unset) {
            try {
                wait();
            } catch (InterruptedException ex) {
                interrupted = true;
            }
        }
        if (interrupted) {
            AccessController.doPrivileged(
                new PrivilegedAction<Void>() {
                    public Void run() {
                        Thread.currentThread().interrupt();
                        return null;
                    }
```

```
                }
            );
        }
        return entry;
    }

    /**
     * Returns the thread that created this EntryFuture.
     */
    Thread getOwner() {
        return owner;
    }
}

/**
 * Creates local class descriptor representing given class.
 */
private ObjectStreamClass(final Class<?> cl) {
    this.cl = cl;
    name = cl.getName();
    isProxy = Proxy.isProxyClass(cl);
    isEnum = Enum.class.isAssignableFrom(cl);
    serializable = Serializable.class.isAssignableFrom(cl);
    externalizable = Externalizable.class.isAssignableFrom(cl);

    Class<?> superCl = cl.getSuperclass();
    superDesc = (superCl != null) ? lookup(superCl, false) : null;
    localDesc = this;

    if (serializable) {
        AccessController.doPrivileged(new PrivilegedAction<Void>() {
            public Void run() {
                if (isEnum) {
                    suid = Long.valueOf(0);
                    fields = NO_FIELDS;
                    return null;
                }
                if (cl.isArray()) {
                    fields = NO_FIELDS;
                    return null;
                }

                suid = getDeclaredSUID(cl);
                try {
                    fields = getSerialFields(cl);
                    computeFieldOffsets();
                } catch (InvalidClassException e) {
                    serializeEx = deserializeEx =
                        new ExceptionInfo(e.classname, e.getMessage());
                    fields = NO_FIELDS;
                }

                if (externalizable) {
                    cons = getExternalizableConstructor(cl);
                } else {
                    cons = getSerializableConstructor(cl);
                    writeObjectMethod = getPrivateMethod(cl, "writeObject",
                        new Class<?>[] { ObjectOutputStream.class },
                        Void.TYPE);
                    readObjectMethod = getPrivateMethod(cl, "readObject",
                        new Class<?>[] { ObjectInputStream.class },
                        Void.TYPE);
                    readObjectNoDataMethod = getPrivateMethod(
```

```
                        cl, "readObjectNoData", null, Void.TYPE);
                    hasWriteObjectData = (writeObjectMethod != null);
                }
                writeReplaceMethod = getInheritableMethod(
                    cl, "writeReplace", null, Object.class);
                readResolveMethod = getInheritableMethod(
                    cl, "readResolve", null, Object.class);
                return null;
            }
        });
    } else {
        suid = Long.valueOf(0);
        fields = NO_FIELDS;
    }

    try {
        fieldRefl = getReflector(fields, this);
    } catch (InvalidClassException ex) {
        // field mismatches impossible when matching local fields vs. self
        throw new InternalError(ex);
    }

    if (deserializeEx == null) {
        if (isEnum) {
            deserializeEx = new ExceptionInfo(name, "enum type");
        } else if (cons == null) {
            deserializeEx = new ExceptionInfo(name, "no valid constructor");
        }
    }
    for (int i = 0; i < fields.length; i++) {
        if (fields[i].getField() == null) {
            defaultSerializeEx = new ExceptionInfo(
                name, "unmatched serializable field(s) declared");
        }
    }
}

/**
 * Creates blank class descriptor which should be initialized via a
 * subsequent call to initProxy(), initNonProxy() or readNonProxy().
 */
ObjectStreamClass() {
}

/**
 * Initializes class descriptor representing a proxy class.
 */
void initProxy(Class<?> cl,
               ClassNotFoundException resolveEx,
               ObjectStreamClass superDesc)
    throws InvalidClassException
{
    this.cl = cl;
    this.resolveEx = resolveEx;
    this.superDesc = superDesc;
    isProxy = true;
    serializable = true;
    suid = Long.valueOf(0);
    fields = NO_FIELDS;

    if (cl != null) {
        localDesc = lookup(cl, true);
        if (!localDesc.isProxy) {
```

```
            throw new InvalidClassException(
                "cannot bind proxy descriptor to a non-proxy class");
        }
        name = localDesc.name;
        externalizable = localDesc.externalizable;
        cons = localDesc.cons;
        writeReplaceMethod = localDesc.writeReplaceMethod;
        readResolveMethod = localDesc.readResolveMethod;
        deserializeEx = localDesc.deserializeEx;
    }
    fieldRefl = getReflector(fields, localDesc);
}

/**
 * Initializes class descriptor representing a non-proxy class.
 */
void initNonProxy(ObjectStreamClass model,
                  Class<?> cl,
                  ClassNotFoundException resolveEx,
                  ObjectStreamClass superDesc)
    throws InvalidClassException
{
    this.cl = cl;
    this.resolveEx = resolveEx;
    this.superDesc = superDesc;
    name = model.name;
    suid = Long.valueOf(model.getSerialVersionUID());
    isProxy = false;
    isEnum = model.isEnum;
    serializable = model.serializable;
    externalizable = model.externalizable;
    hasBlockExternalData = model.hasBlockExternalData;
    hasWriteObjectData = model.hasWriteObjectData;
    fields = model.fields;
    primDataSize = model.primDataSize;
    numObjFields = model.numObjFields;

    if (cl != null) {
        localDesc = lookup(cl, true);
        if (localDesc.isProxy) {
            throw new InvalidClassException(
                "cannot bind non-proxy descriptor to a proxy class");
        }
        if (isEnum != localDesc.isEnum) {
            throw new InvalidClassException(isEnum ?
                "cannot bind enum descriptor to a non-enum class" :
                "cannot bind non-enum descriptor to an enum class");
        }

        if (serializable == localDesc.serializable &&
            !cl.isArray() &&
            suid.longValue() != localDesc.getSerialVersionUID())
        {
            throw new InvalidClassException(localDesc.name,
                "local class incompatible: " +
                "stream classdesc serialVersionUID = " + suid +
                ", local class serialVersionUID = " +
                localDesc.getSerialVersionUID());
        }

        if (!classNamesEqual(name, localDesc.name)) {
            throw new InvalidClassException(localDesc.name,
                "local class name incompatible with stream class " +
```

```
                    "name \"" + name + "\"");
            }

            if (!isEnum) {
                if ((serializable == localDesc.serializable) &&
                    (externalizable != localDesc.externalizable))
                {
                    throw new InvalidClassException(localDesc.name,
                        "Serializable incompatible with Externalizable");
                }

                if ((serializable != localDesc.serializable) ||
                    (externalizable != localDesc.externalizable) ||
                    !(serializable || externalizable))
                {
                    deserializeEx = new ExceptionInfo(
                        localDesc.name, "class invalid for deserialization");
                }
            }

            cons = localDesc.cons;
            writeObjectMethod = localDesc.writeObjectMethod;
            readObjectMethod = localDesc.readObjectMethod;
            readObjectNoDataMethod = localDesc.readObjectNoDataMethod;
            writeReplaceMethod = localDesc.writeReplaceMethod;
            readResolveMethod = localDesc.readResolveMethod;
            if (deserializeEx == null) {
                deserializeEx = localDesc.deserializeEx;
            }
        }
        fieldRefl = getReflector(fields, localDesc);
        // reassign to matched fields so as to reflect local unshared settings
        fields = fieldRefl.getFields();
    }

    /**
     * Reads non-proxy class descriptor information from given input stream.
     * The resulting class descriptor is not fully functional; it can only be
     * used as input to the ObjectInputStream.resolveClass() and
     * ObjectStreamClass.initNonProxy() methods.
     */
    void readNonProxy(ObjectInputStream in)
        throws IOException, ClassNotFoundException
    {
        name = in.readUTF();
        suid = Long.valueOf(in.readLong());
        isProxy = false;

        byte flags = in.readByte();
        hasWriteObjectData =
            ((flags & ObjectStreamConstants.SC_WRITE_METHOD) != 0);
        hasBlockExternalData =
            ((flags & ObjectStreamConstants.SC_BLOCK_DATA) != 0);
        externalizable =
            ((flags & ObjectStreamConstants.SC_EXTERNALIZABLE) != 0);
        boolean sflag =
            ((flags & ObjectStreamConstants.SC_SERIALIZABLE) != 0);
        if (externalizable && sflag) {
            throw new InvalidClassException(
                name, "serializable and externalizable flags conflict");
        }
        serializable = externalizable || sflag;
        isEnum = ((flags & ObjectStreamConstants.SC_ENUM) != 0);
```

```
    if (isEnum && suid.longValue() != 0L) {
        throw new InvalidClassException(name,
            "enum descriptor has non-zero serialVersionUID: " + suid);
    }

    int numFields = in.readShort();
    if (isEnum && numFields != 0) {
        throw new InvalidClassException(name,
            "enum descriptor has non-zero field count: " + numFields);
    }
    fields = (numFields > 0) ?
        new ObjectStreamField[numFields] : NO_FIELDS;
    for (int i = 0; i < numFields; i++) {
        char tcode = (char) in.readByte();
        String fname = in.readUTF();
        String signature = ((tcode == 'L') || (tcode == '[')) ?
            in.readTypeString() : new String(new char[] { tcode });
        try {
            fields[i] = new ObjectStreamField(fname, signature, false);
        } catch (RuntimeException e) {
            throw (IOException) new InvalidClassException(name,
                "invalid descriptor for field " + fname).initCause(e);
        }
    }
    computeFieldOffsets();
}

/**
 * Writes non-proxy class descriptor information to given output stream.
 */
void writeNonProxy(ObjectOutputStream out) throws IOException {
    out.writeUTF(name);
    out.writeLong(getSerialVersionUID());

    byte flags = 0;
    if (externalizable) {
        flags |= ObjectStreamConstants.SC_EXTERNALIZABLE;
        int protocol = out.getProtocolVersion();
        if (protocol != ObjectStreamConstants.PROTOCOL_VERSION_1) {
            flags |= ObjectStreamConstants.SC_BLOCK_DATA;
        }
    } else if (serializable) {
        flags |= ObjectStreamConstants.SC_SERIALIZABLE;
    }
    if (hasWriteObjectData) {
        flags |= ObjectStreamConstants.SC_WRITE_METHOD;
    }
    if (isEnum) {
        flags |= ObjectStreamConstants.SC_ENUM;
    }
    out.writeByte(flags);

    out.writeShort(fields.length);
    for (int i = 0; i < fields.length; i++) {
        ObjectStreamField f = fields[i];
        out.writeByte(f.getTypeCode());
        out.writeUTF(f.getName());
        if (!f.isPrimitive()) {
            out.writeTypeString(f.getTypeString());
        }
    }
}
```

```
/**
 * Returns ClassNotFoundException (if any) thrown while attempting to
 * resolve local class corresponding to this class descriptor.
 */
ClassNotFoundException getResolveException() {
    return resolveEx;
}

/**
 * Throws an InvalidClassException if object instances referencing this
 * class descriptor should not be allowed to deserialize.  This method does
 * not apply to deserialization of enum constants.
 */
void checkDeserialize() throws InvalidClassException {
    if (deserializeEx != null) {
        throw deserializeEx.newInvalidClassException();
    }
}

/**
 * Throws an InvalidClassException if objects whose class is represented by
 * this descriptor should not be allowed to serialize.  This method does
 * not apply to serialization of enum constants.
 */
void checkSerialize() throws InvalidClassException {
    if (serializeEx != null) {
        throw serializeEx.newInvalidClassException();
    }
}

/**
 * Throws an InvalidClassException if objects whose class is represented by
 * this descriptor should not be permitted to use default serialization
 * (e.g., if the class declares serializable fields that do not correspond
 * to actual fields, and hence must use the GetField API).  This method
 * does not apply to deserialization of enum constants.
 */
void checkDefaultSerialize() throws InvalidClassException {
    if (defaultSerializeEx != null) {
        throw defaultSerializeEx.newInvalidClassException();
    }
}

/**
 * Returns superclass descriptor.  Note that on the receiving side, the
 * superclass descriptor may be bound to a class that is not a superclass
 * of the subclass descriptor's bound class.
 */
ObjectStreamClass getSuperDesc() {
    return superDesc;
}

/**
 * Returns the "local" class descriptor for the class associated with this
 * class descriptor (i.e., the result of
 * ObjectStreamClass.lookup(this.forClass())) or null if there is no class
 * associated with this descriptor.
 */
ObjectStreamClass getLocalDesc() {
    return localDesc;
}

/**
```

```
     * Returns arrays of ObjectStreamFields representing the serializable
     * fields of the represented class.  If copy is true, a clone of this class
     * descriptor's field array is returned, otherwise the array itself is
     * returned.
     */
    ObjectStreamField[] getFields(boolean copy) {
        return copy ? fields.clone() : fields;
    }

    /**
     * Looks up a serializable field of the represented class by name and type.
     * A specified type of null matches all types, Object.class matches all
     * non-primitive types, and any other non-null type matches assignable
     * types only.  Returns matching field, or null if no match found.
     */
    ObjectStreamField getField(String name, Class<?> type) {
        for (int i = 0; i < fields.length; i++) {
            ObjectStreamField f = fields[i];
            if (f.getName().equals(name)) {
                if (type == null ||
                    (type == Object.class && !f.isPrimitive()))
                {
                    return f;
                }
                Class<?> ftype = f.getType();
                if (ftype != null && type.isAssignableFrom(ftype)) {
                    return f;
                }
            }
        }
        return null;
    }

    /**
     * Returns true if class descriptor represents a dynamic proxy class, false
     * otherwise.
     */
    boolean isProxy() {
        return isProxy;
    }

    /**
     * Returns true if class descriptor represents an enum type, false
     * otherwise.
     */
    boolean isEnum() {
        return isEnum;
    }

    /**
     * Returns true if represented class implements Externalizable, false
     * otherwise.
     */
    boolean isExternalizable() {
        return externalizable;
    }

    /**
     * Returns true if represented class implements Serializable, false
     * otherwise.
     */
    boolean isSerializable() {
        return serializable;
```

```
}

/**
 * Returns true if class descriptor represents externalizable class that
 * has written its data in 1.2 (block data) format, false otherwise.
 */
boolean hasBlockExternalData() {
    return hasBlockExternalData;
}

/**
 * Returns true if class descriptor represents serializable (but not
 * externalizable) class which has written its data via a custom
 * writeObject() method, false otherwise.
 */
boolean hasWriteObjectData() {
    return hasWriteObjectData;
}

/**
 * Returns true if represented class is serializable/externalizable and can
 * be instantiated by the serialization runtime--i.e., if it is
 * externalizable and defines a public no-arg constructor, or if it is
 * non-externalizable and its first non-serializable superclass defines an
 * accessible no-arg constructor.  Otherwise, returns false.
 */
boolean isInstantiable() {
    return (cons != null);
}

/**
 * Returns true if represented class is serializable (but not
 * externalizable) and defines a conformant writeObject method.  Otherwise,
 * returns false.
 */
boolean hasWriteObjectMethod() {
    return (writeObjectMethod != null);
}

/**
 * Returns true if represented class is serializable (but not
 * externalizable) and defines a conformant readObject method.  Otherwise,
 * returns false.
 */
boolean hasReadObjectMethod() {
    return (readObjectMethod != null);
}

/**
 * Returns true if represented class is serializable (but not
 * externalizable) and defines a conformant readObjectNoData method.
 * Otherwise, returns false.
 */
boolean hasReadObjectNoDataMethod() {
    return (readObjectNoDataMethod != null);
}

/**
 * Returns true if represented class is serializable or externalizable and
 * defines a conformant writeReplace method.  Otherwise, returns false.
 */
boolean hasWriteReplaceMethod() {
    return (writeReplaceMethod != null);
```

```
}

/**
 * Returns true if represented class is serializable or externalizable and
 * defines a conformant readResolve method.  Otherwise, returns false.
 */
boolean hasReadResolveMethod() {
    return (readResolveMethod != null);
}

/**
 * Creates a new instance of the represented class.  If the class is
 * externalizable, invokes its public no-arg constructor; otherwise, if the
 * class is serializable, invokes the no-arg constructor of the first
 * non-serializable superclass.  Throws UnsupportedOperationException if
 * this class descriptor is not associated with a class, if the associated
 * class is non-serializable or if the appropriate no-arg constructor is
 * inaccessible/unavailable.
 */
Object newInstance()
    throws InstantiationException, InvocationTargetException,
           UnsupportedOperationException
{
    if (cons != null) {
        try {
            return cons.newInstance();
        } catch (IllegalAccessException ex) {
            // should not occur, as access checks have been suppressed
            throw new InternalError(ex);
        }
    } else {
        throw new UnsupportedOperationException();
    }
}

/**
 * Invokes the writeObject method of the represented serializable class.
 * Throws UnsupportedOperationException if this class descriptor is not
 * associated with a class, or if the class is externalizable,
 * non-serializable or does not define writeObject.
 */
void invokeWriteObject(Object obj, ObjectOutputStream out)
    throws IOException, UnsupportedOperationException
{
    if (writeObjectMethod != null) {
        try {
            writeObjectMethod.invoke(obj, new Object[]{ out });
        } catch (InvocationTargetException ex) {
            Throwable th = ex.getTargetException();
            if (th instanceof IOException) {
                throw (IOException) th;
            } else {
                throwMiscException(th);
            }
        } catch (IllegalAccessException ex) {
            // should not occur, as access checks have been suppressed
            throw new InternalError(ex);
        }
    } else {
        throw new UnsupportedOperationException();
    }
}
```

```
/**
 * Invokes the readObject method of the represented serializable class.
 * Throws UnsupportedOperationException if this class descriptor is not
 * associated with a class, or if the class is externalizable,
 * non-serializable or does not define readObject.
 */
void invokeReadObject(Object obj, ObjectInputStream in)
    throws ClassNotFoundException, IOException,
           UnsupportedOperationException
{
    if (readObjectMethod != null) {
        try {
            readObjectMethod.invoke(obj, new Object[]{ in });
        } catch (InvocationTargetException ex) {
            Throwable th = ex.getTargetException();
            if (th instanceof ClassNotFoundException) {
                throw (ClassNotFoundException) th;
            } else if (th instanceof IOException) {
                throw (IOException) th;
            } else {
                throwMiscException(th);
            }
        } catch (IllegalAccessException ex) {
            // should not occur, as access checks have been suppressed
            throw new InternalError(ex);
        }
    } else {
        throw new UnsupportedOperationException();
    }
}

/**
 * Invokes the readObjectNoData method of the represented serializable
 * class.  Throws UnsupportedOperationException if this class descriptor is
 * not associated with a class, or if the class is externalizable,
 * non-serializable or does not define readObjectNoData.
 */
void invokeReadObjectNoData(Object obj)
    throws IOException, UnsupportedOperationException
{
    if (readObjectNoDataMethod != null) {
        try {
            readObjectNoDataMethod.invoke(obj, (Object[]) null);
        } catch (InvocationTargetException ex) {
            Throwable th = ex.getTargetException();
            if (th instanceof ObjectStreamException) {
                throw (ObjectStreamException) th;
            } else {
                throwMiscException(th);
            }
        } catch (IllegalAccessException ex) {
            // should not occur, as access checks have been suppressed
            throw new InternalError(ex);
        }
    } else {
        throw new UnsupportedOperationException();
    }
}

/**
 * Invokes the writeReplace method of the represented serializable class and
 * returns the result.  Throws UnsupportedOperationException if this class
 * descriptor is not associated with a class, or if the class is
```

```
     * non-serializable or does not define writeReplace.
     */
    Object invokeWriteReplace(Object obj)
        throws IOException, UnsupportedOperationException
    {
        if (writeReplaceMethod != null) {
            try {
                return writeReplaceMethod.invoke(obj, (Object[]) null);
            } catch (InvocationTargetException ex) {
                Throwable th = ex.getTargetException();
                if (th instanceof ObjectStreamException) {
                    throw (ObjectStreamException) th;
                } else {
                    throwMiscException(th);
                    throw new InternalError(th);  // never reached
                }
            } catch (IllegalAccessException ex) {
                // should not occur, as access checks have been suppressed
                throw new InternalError(ex);
            }
        } else {
            throw new UnsupportedOperationException();
        }
    }

    /**
     * Invokes the readResolve method of the represented serializable class and
     * returns the result.  Throws UnsupportedOperationException if this class
     * descriptor is not associated with a class, or if the class is
     * non-serializable or does not define readResolve.
     */
    Object invokeReadResolve(Object obj)
        throws IOException, UnsupportedOperationException
    {
        if (readResolveMethod != null) {
            try {
                return readResolveMethod.invoke(obj, (Object[]) null);
            } catch (InvocationTargetException ex) {
                Throwable th = ex.getTargetException();
                if (th instanceof ObjectStreamException) {
                    throw (ObjectStreamException) th;
                } else {
                    throwMiscException(th);
                    throw new InternalError(th);  // never reached
                }
            } catch (IllegalAccessException ex) {
                // should not occur, as access checks have been suppressed
                throw new InternalError(ex);
            }
        } else {
            throw new UnsupportedOperationException();
        }
    }

    /**
     * Class representing the portion of an object's serialized form allotted
     * to data described by a given class descriptor.  If "hasData" is false,
     * the object's serialized form does not contain data associated with the
     * class descriptor.
     */
    static class ClassDataSlot {

        /** class descriptor "occupying" this slot */
```

```
    final ObjectStreamClass desc;
    /** true if serialized form includes data for this slot's descriptor */
    final boolean hasData;

    ClassDataSlot(ObjectStreamClass desc, boolean hasData) {
        this.desc = desc;
        this.hasData = hasData;
    }
}

/**
 * Returns array of ClassDataSlot instances representing the data layout
 * (including superclass data) for serialized objects described by this
 * class descriptor.  ClassDataSlots are ordered by inheritance with those
 * containing "higher" superclasses appearing first.  The final
 * ClassDataSlot contains a reference to this descriptor.
 */
ClassDataSlot[] getClassDataLayout() throws InvalidClassException {
    // REMIND: synchronize instead of relying on volatile?
    if (dataLayout == null) {
        dataLayout = getClassDataLayout0();
    }
    return dataLayout;
}

private ClassDataSlot[] getClassDataLayout0()
    throws InvalidClassException
{
    ArrayList<ClassDataSlot> slots = new ArrayList<>();
    Class<?> start = cl, end = cl;

    // locate closest non-serializable superclass
    while (end != null && Serializable.class.isAssignableFrom(end)) {
        end = end.getSuperclass();
    }

    HashSet<String> oscNames = new HashSet<>(3);

    for (ObjectStreamClass d = this; d != null; d = d.superDesc) {
        if (oscNames.contains(d.name)) {
            throw new InvalidClassException("Circular reference.");
        } else {
            oscNames.add(d.name);
        }

        // search up inheritance hierarchy for class with matching name
        String searchName = (d.cl != null) ? d.cl.getName() : d.name;
        Class<?> match = null;
        for (Class<?> c = start; c != end; c = c.getSuperclass()) {
            if (searchName.equals(c.getName())) {
                match = c;
                break;
            }
        }

        // add "no data" slot for each unmatched class below match
        if (match != null) {
            for (Class<?> c = start; c != match; c = c.getSuperclass()) {
                slots.add(new ClassDataSlot(
                    ObjectStreamClass.lookup(c, true), false));
            }
            start = match.getSuperclass();
        }
```

```
        // record descriptor/class pairing
        slots.add(new ClassDataSlot(d.getVariantFor(match), true));
    }

    // add "no data" slot for any leftover unmatched classes
    for (Class<?> c = start; c != end; c = c.getSuperclass()) {
        slots.add(new ClassDataSlot(
            ObjectStreamClass.lookup(c, true), false));
    }

    // order slots from superclass -> subclass
    Collections.reverse(slots);
    return slots.toArray(new ClassDataSlot[slots.size()]);
}

/**
 * Returns aggregate size (in bytes) of marshalled primitive field values
 * for represented class.
 */
int getPrimDataSize() {
    return primDataSize;
}

/**
 * Returns number of non-primitive serializable fields of represented
 * class.
 */
int getNumObjFields() {
    return numObjFields;
}

/**
 * Fetches the serializable primitive field values of object obj and
 * marshals them into byte array buf starting at offset 0.  It is the
 * responsibility of the caller to ensure that obj is of the proper type if
 * non-null.
 */
void getPrimFieldValues(Object obj, byte[] buf) {
    fieldRefl.getPrimFieldValues(obj, buf);
}

/**
 * Sets the serializable primitive fields of object obj using values
 * unmarshalled from byte array buf starting at offset 0.  It is the
 * responsibility of the caller to ensure that obj is of the proper type if
 * non-null.
 */
void setPrimFieldValues(Object obj, byte[] buf) {
    fieldRefl.setPrimFieldValues(obj, buf);
}

/**
 * Fetches the serializable object field values of object obj and stores
 * them in array vals starting at offset 0.  It is the responsibility of
 * the caller to ensure that obj is of the proper type if non-null.
 */
void getObjFieldValues(Object obj, Object[] vals) {
    fieldRefl.getObjFieldValues(obj, vals);
}

/**
 * Sets the serializable object fields of object obj using values from
```

```
     * array vals starting at offset 0.  It is the responsibility of the caller
     * to ensure that obj is of the proper type if non-null.
     */
    void setObjFieldValues(Object obj, Object[] vals) {
        fieldRefl.setObjFieldValues(obj, vals);
    }

    /**
     * Calculates and sets serializable field offsets, as well as primitive
     * data size and object field count totals.  Throws InvalidClassException
     * if fields are illegally ordered.
     */
    private void computeFieldOffsets() throws InvalidClassException {
        primDataSize = 0;
        numObjFields = 0;
        int firstObjIndex = -1;

        for (int i = 0; i < fields.length; i++) {
            ObjectStreamField f = fields[i];
            switch (f.getTypeCode()) {
                case 'Z':
                case 'B':
                    f.setOffset(primDataSize++);
                    break;

                case 'C':
                case 'S':
                    f.setOffset(primDataSize);
                    primDataSize += 2;
                    break;

                case 'I':
                case 'F':
                    f.setOffset(primDataSize);
                    primDataSize += 4;
                    break;

                case 'J':
                case 'D':
                    f.setOffset(primDataSize);
                    primDataSize += 8;
                    break;

                case '[':
                case 'L':
                    f.setOffset(numObjFields++);
                    if (firstObjIndex == -1) {
                        firstObjIndex = i;
                    }
                    break;

                default:
                    throw new InternalError();
            }
        }
        if (firstObjIndex != -1 &&
            firstObjIndex + numObjFields != fields.length)
        {
            throw new InvalidClassException(name, "illegal field order");
        }
    }

    /**
```

```
     * If given class is the same as the class associated with this class
     * descriptor, returns reference to this class descriptor.  Otherwise,
     * returns variant of this class descriptor bound to given class.
     */
    private ObjectStreamClass getVariantFor(Class<?> cl)
        throws InvalidClassException
    {
        if (this.cl == cl) {
            return this;
        }
        ObjectStreamClass desc = new ObjectStreamClass();
        if (isProxy) {
            desc.initProxy(cl, null, superDesc);
        } else {
            desc.initNonProxy(this, cl, null, superDesc);
        }
        return desc;
    }

    /**
     * Returns public no-arg constructor of given class, or null if none found.
     * Access checks are disabled on the returned constructor (if any), since
     * the defining class may still be non-public.
     */
    private static Constructor<?> getExternalizableConstructor(Class<?> cl) {
        try {
            Constructor<?> cons = cl.getDeclaredConstructor((Class<?>[]) null);
            cons.setAccessible(true);
            return ((cons.getModifiers() & Modifier.PUBLIC) != 0) ?
                cons : null;
        } catch (NoSuchMethodException ex) {
            return null;
        }
    }

    /**
     * Returns subclass-accessible no-arg constructor of first non-serializable
     * superclass, or null if none found.  Access checks are disabled on the
     * returned constructor (if any).
     */
    private static Constructor<?> getSerializableConstructor(Class<?> cl) {
        Class<?> initCl = cl;
        while (Serializable.class.isAssignableFrom(initCl)) {
            if ((initCl = initCl.getSuperclass()) == null) {
                return null;
            }
        }
        try {
            Constructor<?> cons = initCl.getDeclaredConstructor((Class<?>[]) null);
            int mods = cons.getModifiers();
            if ((mods & Modifier.PRIVATE) != 0 ||
                ((mods & (Modifier.PUBLIC | Modifier.PROTECTED)) == 0 &&
                 !packageEquals(cl, initCl)))
            {
                return null;
            }
            cons = reflFactory.newConstructorForSerialization(cl, cons);
            cons.setAccessible(true);
            return cons;
        } catch (NoSuchMethodException ex) {
            return null;
        }
    }
```

```
/**
 * Returns non-static, non-abstract method with given signature provided it
 * is defined by or accessible (via inheritance) by the given class, or
 * null if no match found.  Access checks are disabled on the returned
 * method (if any).
 */
private static Method getInheritableMethod(Class<?> cl, String name,
                                           Class<?>[] argTypes,
                                           Class<?> returnType)
{
    Method meth = null;
    Class<?> defCl = cl;
    while (defCl != null) {
        try {
            meth = defCl.getDeclaredMethod(name, argTypes);
            break;
        } catch (NoSuchMethodException ex) {
            defCl = defCl.getSuperclass();
        }
    }

    if ((meth == null) || (meth.getReturnType() != returnType)) {
        return null;
    }
    meth.setAccessible(true);
    int mods = meth.getModifiers();
    if ((mods & (Modifier.STATIC | Modifier.ABSTRACT)) != 0) {
        return null;
    } else if ((mods & (Modifier.PUBLIC | Modifier.PROTECTED)) != 0) {
        return meth;
    } else if ((mods & Modifier.PRIVATE) != 0) {
        return (cl == defCl) ? meth : null;
    } else {
        return packageEquals(cl, defCl) ? meth : null;
    }
}

/**
 * Returns non-static private method with given signature defined by given
 * class, or null if none found.  Access checks are disabled on the
 * returned method (if any).
 */
private static Method getPrivateMethod(Class<?> cl, String name,
                                       Class<?>[] argTypes,
                                       Class<?> returnType)
{
    try {
        Method meth = cl.getDeclaredMethod(name, argTypes);
        meth.setAccessible(true);
        int mods = meth.getModifiers();
        return ((meth.getReturnType() == returnType) &&
                ((mods & Modifier.STATIC) == 0) &&
                ((mods & Modifier.PRIVATE) != 0)) ? meth : null;
    } catch (NoSuchMethodException ex) {
        return null;
    }
}

/**
 * Returns true if classes are defined in the same runtime package, false
 * otherwise.
 */
```

```
private static boolean packageEquals(Class<?> cl1, Class<?> cl2) {
    return (cl1.getClassLoader() == cl2.getClassLoader() &&
            getPackageName(cl1).equals(getPackageName(cl2)));
}

/**
 * Returns package name of given class.
 */
private static String getPackageName(Class<?> cl) {
    String s = cl.getName();
    int i = s.lastIndexOf('[');
    if (i >= 0) {
        s = s.substring(i + 2);
    }
    i = s.lastIndexOf('.');
    return (i >= 0) ? s.substring(0, i) : "";
}

/**
 * Compares class names for equality, ignoring package names.  Returns true
 * if class names equal, false otherwise.
 */
private static boolean classNamesEqual(String name1, String name2) {
    name1 = name1.substring(name1.lastIndexOf('.') + 1);
    name2 = name2.substring(name2.lastIndexOf('.') + 1);
    return name1.equals(name2);
}

/**
 * Returns JVM type signature for given class.
 */
private static String getClassSignature(Class<?> cl) {
    StringBuilder sbuf = new StringBuilder();
    while (cl.isArray()) {
        sbuf.append('[');
        cl = cl.getComponentType();
    }
    if (cl.isPrimitive()) {
        if (cl == Integer.TYPE) {
            sbuf.append('I');
        } else if (cl == Byte.TYPE) {
            sbuf.append('B');
        } else if (cl == Long.TYPE) {
            sbuf.append('J');
        } else if (cl == Float.TYPE) {
            sbuf.append('F');
        } else if (cl == Double.TYPE) {
            sbuf.append('D');
        } else if (cl == Short.TYPE) {
            sbuf.append('S');
        } else if (cl == Character.TYPE) {
            sbuf.append('C');
        } else if (cl == Boolean.TYPE) {
            sbuf.append('Z');
        } else if (cl == Void.TYPE) {
            sbuf.append('V');
        } else {
            throw new InternalError();
        }
    } else {
        sbuf.append('L' + cl.getName().replace('.', '/') + ';');
    }
    return sbuf.toString();
```

```
}

/**
 * Returns JVM type signature for given list of parameters and return type.
 */
private static String getMethodSignature(Class<?>[] paramTypes,
                                         Class<?> retType)
{
    StringBuilder sbuf = new StringBuilder();
    sbuf.append('(');
    for (int i = 0; i < paramTypes.length; i++) {
        sbuf.append(getClassSignature(paramTypes[i]));
    }
    sbuf.append(')');
    sbuf.append(getClassSignature(retType));
    return sbuf.toString();
}

/**
 * Convenience method for throwing an exception that is either a
 * RuntimeException, Error, or of some unexpected type (in which case it is
 * wrapped inside an IOException).
 */
private static void throwMiscException(Throwable th) throws IOException {
    if (th instanceof RuntimeException) {
        throw (RuntimeException) th;
    } else if (th instanceof Error) {
        throw (Error) th;
    } else {
        IOException ex = new IOException("unexpected exception type");
        ex.initCause(th);
        throw ex;
    }
}

/**
 * Returns ObjectStreamField array describing the serializable fields of
 * the given class.  Serializable fields backed by an actual field of the
 * class are represented by ObjectStreamFields with corresponding non-null
 * Field objects.  Throws InvalidClassException if the (explicitly
 * declared) serializable fields are invalid.
 */
private static ObjectStreamField[] getSerialFields(Class<?> cl)
    throws InvalidClassException
{
    ObjectStreamField[] fields;
    if (Serializable.class.isAssignableFrom(cl) &&
        !Externalizable.class.isAssignableFrom(cl) &&
        !Proxy.isProxyClass(cl) &&
        !cl.isInterface())
    {
        if ((fields = getDeclaredSerialFields(cl)) == null) {
            fields = getDefaultSerialFields(cl);
        }
        Arrays.sort(fields);
    } else {
        fields = NO_FIELDS;
    }
    return fields;
}

/**
 * Returns serializable fields of given class as defined explicitly by a
```

```
     * "serialPersistentFields" field, or null if no appropriate
     * "serialPersistentFields" field is defined.  Serializable fields backed
     * by an actual field of the class are represented by ObjectStreamFields
     * with corresponding non-null Field objects.  For compatibility with past
     * releases, a "serialPersistentFields" field with a null value is
     * considered equivalent to not declaring "serialPersistentFields".  Throws
     * InvalidClassException if the declared serializable fields are
     * invalid--e.g., if multiple fields share the same name.
     */
    private static ObjectStreamField[] getDeclaredSerialFields(Class<?> cl)
        throws InvalidClassException
    {
        ObjectStreamField[] serialPersistentFields = null;
        try {
            Field f = cl.getDeclaredField("serialPersistentFields");
            int mask = Modifier.PRIVATE | Modifier.STATIC | Modifier.FINAL;
            if ((f.getModifiers() & mask) == mask) {
                f.setAccessible(true);
                serialPersistentFields = (ObjectStreamField[]) f.get(null);
            }
        } catch (Exception ex) {
        }
        if (serialPersistentFields == null) {
            return null;
        } else if (serialPersistentFields.length == 0) {
            return NO_FIELDS;
        }

        ObjectStreamField[] boundFields =
            new ObjectStreamField[serialPersistentFields.length];
        Set<String> fieldNames = new HashSet<>(serialPersistentFields.length);

        for (int i = 0; i < serialPersistentFields.length; i++) {
            ObjectStreamField spf = serialPersistentFields[i];

            String fname = spf.getName();
            if (fieldNames.contains(fname)) {
                throw new InvalidClassException(
                    "multiple serializable fields named " + fname);
            }
            fieldNames.add(fname);

            try {
                Field f = cl.getDeclaredField(fname);
                if ((f.getType() == spf.getType()) &&
                    ((f.getModifiers() & Modifier.STATIC) == 0))
                {
                    boundFields[i] =
                        new ObjectStreamField(f, spf.isUnshared(), true);
                }
            } catch (NoSuchFieldException ex) {
            }
            if (boundFields[i] == null) {
                boundFields[i] = new ObjectStreamField(
                    fname, spf.getType(), spf.isUnshared());
            }
        }
        return boundFields;
    }

    /**
     * Returns array of ObjectStreamFields corresponding to all non-static
     * non-transient fields declared by given class.  Each ObjectStreamField
```

```
     * contains a Field object for the field it represents.  If no default
     * serializable fields exist, NO_FIELDS is returned.
     */
    private static ObjectStreamField[] getDefaultSerialFields(Class<?> cl) {
        Field[] clFields = cl.getDeclaredFields();
        ArrayList<ObjectStreamField> list = new ArrayList<>();
        int mask = Modifier.STATIC | Modifier.TRANSIENT;

        for (int i = 0; i < clFields.length; i++) {
            if ((clFields[i].getModifiers() & mask) == 0) {
                list.add(new ObjectStreamField(clFields[i], false, true));
            }
        }
        int size = list.size();
        return (size == 0) ? NO_FIELDS :
            list.toArray(new ObjectStreamField[size]);
    }

    /**
     * Returns explicit serial version UID value declared by given class, or
     * null if none.
     */
    private static Long getDeclaredSUID(Class<?> cl) {
        try {
            Field f = cl.getDeclaredField("serialVersionUID");
            int mask = Modifier.STATIC | Modifier.FINAL;
            if ((f.getModifiers() & mask) == mask) {
                f.setAccessible(true);
                return Long.valueOf(f.getLong(null));
            }
        } catch (Exception ex) {
        }
        return null;
    }

    /**
     * Computes the default serial version UID value for the given class.
     */
    private static long computeDefaultSUID(Class<?> cl) {
        if (!Serializable.class.isAssignableFrom(cl) || Proxy.isProxyClass(cl))
        {
            return 0L;
        }

        try {
            ByteArrayOutputStream bout = new ByteArrayOutputStream();
            DataOutputStream dout = new DataOutputStream(bout);

            dout.writeUTF(cl.getName());

            int classMods = cl.getModifiers() &
                (Modifier.PUBLIC | Modifier.FINAL |
                 Modifier.INTERFACE | Modifier.ABSTRACT);

            /*
             * compensate for javac bug in which ABSTRACT bit was set for an
             * interface only if the interface declared methods
             */
            Method[] methods = cl.getDeclaredMethods();
            if ((classMods & Modifier.INTERFACE) != 0) {
                classMods = (methods.length > 0) ?
                    (classMods | Modifier.ABSTRACT) :
                    (classMods & ~Modifier.ABSTRACT);
```

```
}
dout.writeInt(classMods);

if (!cl.isArray()) {
    /*
     * compensate for change in 1.2FCS in which
     * Class.getInterfaces() was modified to return Cloneable and
     * Serializable for array classes.
     */
    Class<?>[] interfaces = cl.getInterfaces();
    String[] ifaceNames = new String[interfaces.length];
    for (int i = 0; i < interfaces.length; i++) {
        ifaceNames[i] = interfaces[i].getName();
    }
    Arrays.sort(ifaceNames);
    for (int i = 0; i < ifaceNames.length; i++) {
        dout.writeUTF(ifaceNames[i]);
    }
}

Field[] fields = cl.getDeclaredFields();
MemberSignature[] fieldSigs = new MemberSignature[fields.length];
for (int i = 0; i < fields.length; i++) {
    fieldSigs[i] = new MemberSignature(fields[i]);
}
Arrays.sort(fieldSigs, new Comparator<MemberSignature>() {
    public int compare(MemberSignature ms1, MemberSignature ms2) {
        return ms1.name.compareTo(ms2.name);
    }
});
for (int i = 0; i < fieldSigs.length; i++) {
    MemberSignature sig = fieldSigs[i];
    int mods = sig.member.getModifiers() &
        (Modifier.PUBLIC | Modifier.PRIVATE | Modifier.PROTECTED |
         Modifier.STATIC | Modifier.FINAL | Modifier.VOLATILE |
         Modifier.TRANSIENT);
    if (((mods & Modifier.PRIVATE) == 0) ||
        ((mods & (Modifier.STATIC | Modifier.TRANSIENT)) == 0))
    {
        dout.writeUTF(sig.name);
        dout.writeInt(mods);
        dout.writeUTF(sig.signature);
    }
}

if (hasStaticInitializer(cl)) {
    dout.writeUTF("<clinit>");
    dout.writeInt(Modifier.STATIC);
    dout.writeUTF("()V");
}

Constructor<?>[] cons = cl.getDeclaredConstructors();
MemberSignature[] consSigs = new MemberSignature[cons.length];
for (int i = 0; i < cons.length; i++) {
    consSigs[i] = new MemberSignature(cons[i]);
}
Arrays.sort(consSigs, new Comparator<MemberSignature>() {
    public int compare(MemberSignature ms1, MemberSignature ms2) {
        return ms1.signature.compareTo(ms2.signature);
    }
});
for (int i = 0; i < consSigs.length; i++) {
    MemberSignature sig = consSigs[i];
```

```
            int mods = sig.member.getModifiers() &
                (Modifier.PUBLIC | Modifier.PRIVATE | Modifier.PROTECTED |
                 Modifier.STATIC | Modifier.FINAL |
                 Modifier.SYNCHRONIZED | Modifier.NATIVE |
                 Modifier.ABSTRACT | Modifier.STRICT);
            if ((mods & Modifier.PRIVATE) == 0) {
                dout.writeUTF("<init>");
                dout.writeInt(mods);
                dout.writeUTF(sig.signature.replace('/', '.'));
            }
        }

        MemberSignature[] methSigs = new MemberSignature[methods.length];
        for (int i = 0; i < methods.length; i++) {
            methSigs[i] = new MemberSignature(methods[i]);
        }
        Arrays.sort(methSigs, new Comparator<MemberSignature>() {
            public int compare(MemberSignature ms1, MemberSignature ms2) {
                int comp = ms1.name.compareTo(ms2.name);
                if (comp == 0) {
                    comp = ms1.signature.compareTo(ms2.signature);
                }
                return comp;
            }
        });
        for (int i = 0; i < methSigs.length; i++) {
            MemberSignature sig = methSigs[i];
            int mods = sig.member.getModifiers() &
                (Modifier.PUBLIC | Modifier.PRIVATE | Modifier.PROTECTED |
                 Modifier.STATIC | Modifier.FINAL |
                 Modifier.SYNCHRONIZED | Modifier.NATIVE |
                 Modifier.ABSTRACT | Modifier.STRICT);
            if ((mods & Modifier.PRIVATE) == 0) {
                dout.writeUTF(sig.name);
                dout.writeInt(mods);
                dout.writeUTF(sig.signature.replace('/', '.'));
            }
        }

        dout.flush();

        MessageDigest md = MessageDigest.getInstance("SHA");
        byte[] hashBytes = md.digest(bout.toByteArray());
        long hash = 0;
        for (int i = Math.min(hashBytes.length, 8) - 1; i >= 0; i--) {
            hash = (hash << 8) | (hashBytes[i] & 0xFF);
        }
        return hash;
    } catch (IOException ex) {
        throw new InternalError(ex);
    } catch (NoSuchAlgorithmException ex) {
        throw new SecurityException(ex.getMessage());
    }
}

/**
 * Returns true if the given class defines a static initializer method,
 * false otherwise.
 */
private native static boolean hasStaticInitializer(Class<?> cl);

/**
 * Class for computing and caching field/constructor/method signatures
```

```
     * during serialVersionUID calculation.
     */
    private static class MemberSignature {

        public final Member member;
        public final String name;
        public final String signature;

        public MemberSignature(Field field) {
            member = field;
            name = field.getName();
            signature = getClassSignature(field.getType());
        }

        public MemberSignature(Constructor<?> cons) {
            member = cons;
            name = cons.getName();
            signature = getMethodSignature(
                cons.getParameterTypes(), Void.TYPE);
        }

        public MemberSignature(Method meth) {
            member = meth;
            name = meth.getName();
            signature = getMethodSignature(
                meth.getParameterTypes(), meth.getReturnType());
        }
    }

    /**
     * Class for setting and retrieving serializable field values in batch.
     */
    // REMIND: dynamically generate these?
    private static class FieldReflector {

        /** handle for performing unsafe operations */
        private static final Unsafe unsafe = Unsafe.getUnsafe();

        /** fields to operate on */
        private final ObjectStreamField[] fields;
        /** number of primitive fields */
        private final int numPrimFields;
        /** unsafe field keys for reading fields - may contain dupes */
        private final long[] readKeys;
        /** unsafe fields keys for writing fields - no dupes */
        private final long[] writeKeys;
        /** field data offsets */
        private final int[] offsets;
        /** field type codes */
        private final char[] typeCodes;
        /** field types */
        private final Class<?>[] types;

        /**
         * Constructs FieldReflector capable of setting/getting values from the
         * subset of fields whose ObjectStreamFields contain non-null
         * reflective Field objects.  ObjectStreamFields with null Fields are
         * treated as filler, for which get operations return default values
         * and set operations discard given values.
         */
        FieldReflector(ObjectStreamField[] fields) {
            this.fields = fields;
            int nfields = fields.length;
```

```
        readKeys = new long[nfields];
        writeKeys = new long[nfields];
        offsets = new int[nfields];
        typeCodes = new char[nfields];
        ArrayList<Class<?>> typeList = new ArrayList<>();
        Set<Long> usedKeys = new HashSet<>();


        for (int i = 0; i < nfields; i++) {
            ObjectStreamField f = fields[i];
            Field rf = f.getField();
            long key = (rf != null) ?
                unsafe.objectFieldOffset(rf) : Unsafe.INVALID_FIELD_OFFSET;
            readKeys[i] = key;
            writeKeys[i] = usedKeys.add(key) ?
                key : Unsafe.INVALID_FIELD_OFFSET;
            offsets[i] = f.getOffset();
            typeCodes[i] = f.getTypeCode();
            if (!f.isPrimitive()) {
                typeList.add((rf != null) ? rf.getType() : null);
            }
        }

        types = typeList.toArray(new Class<?>[typeList.size()]);
        numPrimFields = nfields - types.length;
    }

    /**
     * Returns list of ObjectStreamFields representing fields operated on
     * by this reflector.  The shared/unshared values and Field objects
     * contained by ObjectStreamFields in the list reflect their bindings
     * to locally defined serializable fields.
     */
    ObjectStreamField[] getFields() {
        return fields;
    }

    /**
     * Fetches the serializable primitive field values of object obj and
     * marshals them into byte array buf starting at offset 0.  The caller
     * is responsible for ensuring that obj is of the proper type.
     */
    void getPrimFieldValues(Object obj, byte[] buf) {
        if (obj == null) {
            throw new NullPointerException();
        }
        /* assuming checkDefaultSerialize() has been called on the class
         * descriptor this FieldReflector was obtained from, no field keys
         * in array should be equal to Unsafe.INVALID_FIELD_OFFSET.
         */
        for (int i = 0; i < numPrimFields; i++) {
            long key = readKeys[i];
            int off = offsets[i];
            switch (typeCodes[i]) {
                case 'Z':
                    Bits.putBoolean(buf, off, unsafe.getBoolean(obj, key));
                    break;

                case 'B':
                    buf[off] = unsafe.getByte(obj, key);
                    break;

                case 'C':
```

```
                Bits.putChar(buf, off, unsafe.getChar(obj, key));
                break;

            case 'S':
                Bits.putShort(buf, off, unsafe.getShort(obj, key));
                break;

            case 'I':
                Bits.putInt(buf, off, unsafe.getInt(obj, key));
                break;

            case 'F':
                Bits.putFloat(buf, off, unsafe.getFloat(obj, key));
                break;

            case 'J':
                Bits.putLong(buf, off, unsafe.getLong(obj, key));
                break;

            case 'D':
                Bits.putDouble(buf, off, unsafe.getDouble(obj, key));
                break;

            default:
                throw new InternalError();
        }
    }
}

/**
 * Sets the serializable primitive fields of object obj using values
 * unmarshalled from byte array buf starting at offset 0.  The caller
 * is responsible for ensuring that obj is of the proper type.
 */
void setPrimFieldValues(Object obj, byte[] buf) {
    if (obj == null) {
        throw new NullPointerException();
    }
    for (int i = 0; i < numPrimFields; i++) {
        long key = writeKeys[i];
        if (key == Unsafe.INVALID_FIELD_OFFSET) {
            continue;           // discard value
        }
        int off = offsets[i];
        switch (typeCodes[i]) {
            case 'Z':
                unsafe.putBoolean(obj, key, Bits.getBoolean(buf, off));
                break;

            case 'B':
                unsafe.putByte(obj, key, buf[off]);
                break;

            case 'C':
                unsafe.putChar(obj, key, Bits.getChar(buf, off));
                break;

            case 'S':
                unsafe.putShort(obj, key, Bits.getShort(buf, off));
                break;

            case 'I':
                unsafe.putInt(obj, key, Bits.getInt(buf, off));
```

```
                break;

            case 'F':
                unsafe.putFloat(obj, key, Bits.getFloat(buf, off));
                break;

            case 'J':
                unsafe.putLong(obj, key, Bits.getLong(buf, off));
                break;

            case 'D':
                unsafe.putDouble(obj, key, Bits.getDouble(buf, off));
                break;

            default:
                throw new InternalError();
        }
    }
}

/**
 * Fetches the serializable object field values of object obj and
 * stores them in array vals starting at offset 0.  The caller is
 * responsible for ensuring that obj is of the proper type.
 */
void getObjFieldValues(Object obj, Object[] vals) {
    if (obj == null) {
        throw new NullPointerException();
    }
    /* assuming checkDefaultSerialize() has been called on the class
     * descriptor this FieldReflector was obtained from, no field keys
     * in array should be equal to Unsafe.INVALID_FIELD_OFFSET.
     */
    for (int i = numPrimFields; i < fields.length; i++) {
        switch (typeCodes[i]) {
            case 'L':
            case '[':
                vals[offsets[i]] = unsafe.getObject(obj, readKeys[i]);
                break;

            default:
                throw new InternalError();
        }
    }
}

/**
 * Sets the serializable object fields of object obj using values from
 * array vals starting at offset 0.  The caller is responsible for
 * ensuring that obj is of the proper type; however, attempts to set a
 * field with a value of the wrong type will trigger an appropriate
 * ClassCastException.
 */
void setObjFieldValues(Object obj, Object[] vals) {
    if (obj == null) {
        throw new NullPointerException();
    }
    for (int i = numPrimFields; i < fields.length; i++) {
        long key = writeKeys[i];
        if (key == Unsafe.INVALID_FIELD_OFFSET) {
            continue;           // discard value
        }
        switch (typeCodes[i]) {
```

```
                case 'L':
                case '[':
                    Object val = vals[offsets[i]];
                    if (val != null &&
                        !types[i - numPrimFields].isInstance(val))
                    {
                        Field f = fields[i].getField();
                        throw new ClassCastException(
                            "cannot assign instance of " +
                            val.getClass().getName() + " to field " +
                            f.getDeclaringClass().getName() + "." +
                            f.getName() + " of type " +
                            f.getType().getName() + " in instance of " +
                            obj.getClass().getName());
                    }
                    unsafe.putObject(obj, key, val);
                    break;

                default:
                    throw new InternalError();
            }
        }
    }
}

/**
 * Matches given set of serializable fields with serializable fields
 * described by the given local class descriptor, and returns a
 * FieldReflector instance capable of setting/getting values from the
 * subset of fields that match (non-matching fields are treated as filler,
 * for which get operations return default values and set operations
 * discard given values).  Throws InvalidClassException if unresolvable
 * type conflicts exist between the two sets of fields.
 */
private static FieldReflector getReflector(ObjectStreamField[] fields,
                                           ObjectStreamClass localDesc)
    throws InvalidClassException
{
    // class irrelevant if no fields
    Class<?> cl = (localDesc != null && fields.length > 0) ?
        localDesc.cl : null;
    processQueue(Caches.reflectorsQueue, Caches.reflectors);
    FieldReflectorKey key = new FieldReflectorKey(cl, fields,
                                                  Caches.reflectorsQueue);
    Reference<?> ref = Caches.reflectors.get(key);
    Object entry = null;
    if (ref != null) {
        entry = ref.get();
    }
    EntryFuture future = null;
    if (entry == null) {
        EntryFuture newEntry = new EntryFuture();
        Reference<?> newRef = new SoftReference<>(newEntry);
        do {
            if (ref != null) {
                Caches.reflectors.remove(key, ref);
            }
            ref = Caches.reflectors.putIfAbsent(key, newRef);
            if (ref != null) {
                entry = ref.get();
            }
        } while (ref != null && entry == null);
        if (entry == null) {
```

```
                future = newEntry;
            }
        }

        if (entry instanceof FieldReflector) {  // check common case first
            return (FieldReflector) entry;
        } else if (entry instanceof EntryFuture) {
            entry = ((EntryFuture) entry).get();
        } else if (entry == null) {
            try {
                entry = new FieldReflector(matchFields(fields, localDesc));
            } catch (Throwable th) {
                entry = th;
            }
            future.set(entry);
            Caches.reflectors.put(key, new SoftReference<Object>(entry));
        }

        if (entry instanceof FieldReflector) {
            return (FieldReflector) entry;
        } else if (entry instanceof InvalidClassException) {
            throw (InvalidClassException) entry;
        } else if (entry instanceof RuntimeException) {
            throw (RuntimeException) entry;
        } else if (entry instanceof Error) {
            throw (Error) entry;
        } else {
            throw new InternalError("unexpected entry: " + entry);
        }
    }

    /**
     * FieldReflector cache lookup key.  Keys are considered equal if they
     * refer to the same class and equivalent field formats.
     */
    private static class FieldReflectorKey extends WeakReference<Class<?>> {

        private final String sigs;
        private final int hash;
        private final boolean nullClass;

        FieldReflectorKey(Class<?> cl, ObjectStreamField[] fields,
                          ReferenceQueue<Class<?>> queue)
        {
            super(cl, queue);
            nullClass = (cl == null);
            StringBuilder sbuf = new StringBuilder();
            for (int i = 0; i < fields.length; i++) {
                ObjectStreamField f = fields[i];
                sbuf.append(f.getName()).append(f.getSignature());
            }
            sigs = sbuf.toString();
            hash = System.identityHashCode(cl) + sigs.hashCode();
        }

        public int hashCode() {
            return hash;
        }

        public boolean equals(Object obj) {
            if (obj == this) {
                return true;
            }
```

```
        if (obj instanceof FieldReflectorKey) {
            FieldReflectorKey other = (FieldReflectorKey) obj;
            Class<?> referent;
            return (nullClass ? other.nullClass
                              : ((referent = get()) != null) &&
                                (referent == other.get())) &&
                sigs.equals(other.sigs);
        } else {
            return false;
        }
    }
}

/**
 * Matches given set of serializable fields with serializable fields
 * obtained from the given local class descriptor (which contain bindings
 * to reflective Field objects).  Returns list of ObjectStreamFields in
 * which each ObjectStreamField whose signature matches that of a local
 * field contains a Field object for that field; unmatched
 * ObjectStreamFields contain null Field objects.  Shared/unshared settings
 * of the returned ObjectStreamFields also reflect those of matched local
 * ObjectStreamFields.  Throws InvalidClassException if unresolvable type
 * conflicts exist between the two sets of fields.
 */
private static ObjectStreamField[] matchFields(ObjectStreamField[] fields,
                                               ObjectStreamClass localDesc)
    throws InvalidClassException
{
    ObjectStreamField[] localFields = (localDesc != null) ?
        localDesc.fields : NO_FIELDS;

    /*
     * Even if fields == localFields, we cannot simply return localFields
     * here.  In previous implementations of serialization,
     * ObjectStreamField.getType() returned Object.class if the
     * ObjectStreamField represented a non-primitive field and belonged to
     * a non-local class descriptor.  To preserve this (questionable)
     * behavior, the ObjectStreamField instances returned by matchFields
     * cannot report non-primitive types other than Object.class; hence
     * localFields cannot be returned directly.
     */

    ObjectStreamField[] matches = new ObjectStreamField[fields.length];
    for (int i = 0; i < fields.length; i++) {
        ObjectStreamField f = fields[i], m = null;
        for (int j = 0; j < localFields.length; j++) {
            ObjectStreamField lf = localFields[j];
            if (f.getName().equals(lf.getName())) {
                if ((f.isPrimitive() || lf.isPrimitive()) &&
                    f.getTypeCode() != lf.getTypeCode())
                {
                    throw new InvalidClassException(localDesc.name,
                        "incompatible types for field " + f.getName());
                }
                if (lf.getField() != null) {
                    m = new ObjectStreamField(
                        lf.getField(), lf.isUnshared(), false);
                } else {
                    m = new ObjectStreamField(
                        lf.getName(), lf.getSignature(), lf.isUnshared());
                }
            }
```

```
        }
        if (m == null) {
            m = new ObjectStreamField(
                f.getName(), f.getSignature(), false);
        }
        m.setOffset(f.getOffset());
        matches[i] = m;
    }
    return matches;
}

/**
 * Removes from the specified map any keys that have been enqueued
 * on the specified reference queue.
 */
static void processQueue(ReferenceQueue<Class<?>> queue,
                         ConcurrentMap<? extends
                         WeakReference<Class<?>>, ?> map)
{
    Reference<? extends Class<?>> ref;
    while((ref = queue.poll()) != null) {
        map.remove(ref);
    }
}

/**
 *  Weak key for Class objects.
 *
 **/
static class WeakClassKey extends WeakReference<Class<?>> {
    /**
     * saved value of the referent's identity hash code, to maintain
     * a consistent hash code after the referent has been cleared
     */
    private final int hash;

    /**
     * Create a new WeakClassKey to the given object, registered
     * with a queue.
     */
    WeakClassKey(Class<?> cl, ReferenceQueue<Class<?>> refQueue) {
        super(cl, refQueue);
        hash = System.identityHashCode(cl);
    }

    /**
     * Returns the identity hash code of the original referent.
     */
    public int hashCode() {
        return hash;
    }

    /**
     * Returns true if the given object is this identical
     * WeakClassKey instance, or, if this object's referent has not
     * been cleared, if the given object is another WeakClassKey
     * instance with the identical non-null referent as this one.
     */
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
```

```
            if (obj instanceof WeakClassKey) {
                Object referent = get();
                return (referent != null) &&
                       (referent == ((WeakClassKey) obj).get());
            } else {
                return false;
            }
        }
    }
}
```

ObjectStreamConstants.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Constants written into the Object Serialization Stream.
 *
 * @author  unascribed
 * @since JDK 1.1
 */
public interface ObjectStreamConstants {

    /**
     * Magic number that is written to the stream header.
     */
    final static short STREAM_MAGIC = (short)0xaced;

    /**
     * Version number that is written to the stream header.
     */
    final static short STREAM_VERSION = 5;

    /* Each item in the stream is preceded by a tag
     */

    /**
     * First tag value.
     */
    final static byte TC_BASE = 0x70;

    /**
     * Null object reference.
     */
    final static byte TC_NULL =         (byte)0x70;

    /**
     * Reference to an object already written into the stream.
```

```
 */
final static byte TC_REFERENCE =    (byte)0x71;

/**
 * new Class Descriptor.
 */
final static byte TC_CLASSDESC =    (byte)0x72;

/**
 * new Object.
 */
final static byte TC_OBJECT =       (byte)0x73;

/**
 * new String.
 */
final static byte TC_STRING =       (byte)0x74;

/**
 * new Array.
 */
final static byte TC_ARRAY =        (byte)0x75;

/**
 * Reference to Class.
 */
final static byte TC_CLASS =        (byte)0x76;

/**
 * Block of optional data. Byte following tag indicates number
 * of bytes in this block data.
 */
final static byte TC_BLOCKDATA =    (byte)0x77;

/**
 * End of optional block data blocks for an object.
 */
final static byte TC_ENDBLOCKDATA = (byte)0x78;

/**
 * Reset stream context. All handles written into stream are reset.
 */
final static byte TC_RESET =        (byte)0x79;

/**
 * long Block data. The long following the tag indicates the
 * number of bytes in this block data.
 */
final static byte TC_BLOCKDATALONG= (byte)0x7A;

/**
 * Exception during write.
 */
final static byte TC_EXCEPTION =    (byte)0x7B;

/**
 * Long string.
 */
final static byte TC_LONGSTRING =   (byte)0x7C;

/**
 * new Proxy Class Descriptor.
 */
```

```
final static byte TC_PROXYCLASSDESC =       (byte)0x7D;

/**
 * new Enum constant.
 * @since 1.5
 */
final static byte TC_ENUM =         (byte)0x7E;

/**
 * Last tag value.
 */
final static byte TC_MAX =          (byte)0x7E;

/**
 * First wire handle to be assigned.
 */
final static int baseWireHandle = 0x7e0000;


/******************************************************/
/* Bit masks for ObjectStreamClass flag.*/

/**
 * Bit mask for ObjectStreamClass flag. Indicates a Serializable class
 * defines its own writeObject method.
 */
final static byte SC_WRITE_METHOD = 0x01;

/**
 * Bit mask for ObjectStreamClass flag. Indicates Externalizable data
 * written in Block Data mode.
 * Added for PROTOCOL_VERSION_2.
 *
 * @see #PROTOCOL_VERSION_2
 * @since 1.2
 */
final static byte SC_BLOCK_DATA = 0x08;

/**
 * Bit mask for ObjectStreamClass flag. Indicates class is Serializable.
 */
final static byte SC_SERIALIZABLE = 0x02;

/**
 * Bit mask for ObjectStreamClass flag. Indicates class is Externalizable.
 */
final static byte SC_EXTERNALIZABLE = 0x04;

/**
 * Bit mask for ObjectStreamClass flag. Indicates class is an enum type.
 * @since 1.5
 */
final static byte SC_ENUM = 0x10;


/* *****************************************************************/
/* Security permissions */

/**
 * Enable substitution of one object for another during
 * serialization/deserialization.
 *
 * @see java.io.ObjectOutputStream#enableReplaceObject(boolean)
```

```
     * @see java.io.ObjectInputStream#enableResolveObject(boolean)
     * @since 1.2
     */
    final static SerializablePermission SUBSTITUTION_PERMISSION =
                            new SerializablePermission("enableSubstitution");

    /**
     * Enable overriding of readObject and writeObject.
     *
     * @see java.io.ObjectOutputStream#writeObjectOverride(Object)
     * @see java.io.ObjectInputStream#readObjectOverride()
     * @since 1.2
     */
    final static SerializablePermission SUBCLASS_IMPLEMENTATION_PERMISSION =
                    new SerializablePermission("enableSubclassImplementation");
    /**
     * A Stream Protocol Version. <p>
     *
     * All externalizable data is written in JDK 1.1 external data
     * format after calling this method. This version is needed to write
     * streams containing Externalizable data that can be read by
     * pre-JDK 1.1.6 JVMs.
     *
     * @see java.io.ObjectOutputStream#useProtocolVersion(int)
     * @since 1.2
     */
    public final static int PROTOCOL_VERSION_1 = 1;


    /**
     * A Stream Protocol Version. <p>
     *
     * This protocol is written by JVM 1.2.
     *
     * Externalizable data is written in block data mode and is
     * terminated with TC_ENDBLOCKDATA. Externalizable class descriptor
     * flags has SC_BLOCK_DATA enabled. JVM 1.1.6 and greater can
     * read this format change.
     *
     * Enables writing a nonSerializable class descriptor into the
     * stream. The serialVersionUID of a nonSerializable class is
     * set to 0L.
     *
     * @see java.io.ObjectOutputStream#useProtocolVersion(int)
     * @see #SC_BLOCK_DATA
     * @since 1.2
     */
    public final static int PROTOCOL_VERSION_2 = 2;
}
```

ObjectStreamException.java

```java
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Superclass of all exceptions specific to Object Stream classes.
 *
 * @author  unascribed
 * @since   JDK1.1
 */
public abstract class ObjectStreamException extends IOException {

    private static final long serialVersionUID = 7260898174833392607L;

    /**
     * Create an ObjectStreamException with the specified argument.
     *
     * @param classname the detailed message for the exception
     */
    protected ObjectStreamException(String classname) {
        super(classname);
    }

    /**
     * Create an ObjectStreamException.
     */
    protected ObjectStreamException() {
        super();
    }
}
```

ObjectStreamField.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.lang.reflect.Field;
import sun.reflect.CallerSensitive;
import sun.reflect.Reflection;
import sun.reflect.misc.ReflectUtil;

/**
 * A description of a Serializable field from a Serializable class.  An array
 * of ObjectStreamFields is used to declare the Serializable fields of a class.
 *
 * @author      Mike Warres
 * @author      Roger Riggs
 * @see ObjectStreamClass
 * @since 1.2
 */
public class ObjectStreamField
    implements Comparable<Object>
{

    /** field name */
    private final String name;
    /** canonical JVM signature of field type */
    private final String signature;
    /** field type (Object.class if unknown non-primitive type) */
    private final Class<?> type;
    /** whether or not to (de)serialize field values as unshared */
    private final boolean unshared;
    /** corresponding reflective field object, if any */
    private final Field field;
    /** offset of field value in enclosing field group */
    private int offset = 0;

    /**
     * Create a Serializable field with the specified type.  This field should
```

```
     * be documented with a <code>serialField</code> tag.
     *
     * @param   name the name of the serializable field
     * @param   type the <code>Class</code> object of the serializable field
     */
    public ObjectStreamField(String name, Class<?> type) {
        this(name, type, false);
    }

    /**
     * Creates an ObjectStreamField representing a serializable field with the
     * given name and type.  If unshared is false, values of the represented
     * field are serialized and deserialized in the default manner--if the
     * field is non-primitive, object values are serialized and deserialized as
     * if they had been written and read by calls to writeObject and
     * readObject.  If unshared is true, values of the represented field are
     * serialized and deserialized as if they had been written and read by
     * calls to writeUnshared and readUnshared.
     *
     * @param   name field name
     * @param   type field type
     * @param   unshared if false, write/read field values in the same manner
     *          as writeObject/readObject; if true, write/read in the same
     *          manner as writeUnshared/readUnshared
     * @since   1.4
     */
    public ObjectStreamField(String name, Class<?> type, boolean unshared) {
        if (name == null) {
            throw new NullPointerException();
        }
        this.name = name;
        this.type = type;
        this.unshared = unshared;
        signature = getClassSignature(type).intern();
        field = null;
    }

    /**
     * Creates an ObjectStreamField representing a field with the given name,
     * signature and unshared setting.
     */
    ObjectStreamField(String name, String signature, boolean unshared) {
        if (name == null) {
            throw new NullPointerException();
        }
        this.name = name;
        this.signature = signature.intern();
        this.unshared = unshared;
        field = null;

        switch (signature.charAt(0)) {
            case 'Z': type = Boolean.TYPE; break;
            case 'B': type = Byte.TYPE; break;
            case 'C': type = Character.TYPE; break;
            case 'S': type = Short.TYPE; break;
            case 'I': type = Integer.TYPE; break;
            case 'J': type = Long.TYPE; break;
            case 'F': type = Float.TYPE; break;
            case 'D': type = Double.TYPE; break;
            case 'L':
            case '[': type = Object.class; break;
            default: throw new IllegalArgumentException("illegal signature");
        }
```

```
}

/**
 * Creates an ObjectStreamField representing the given field with the
 * specified unshared setting.  For compatibility with the behavior of
 * earlier serialization implementations, a "showType" parameter is
 * necessary to govern whether or not a getType() call on this
 * ObjectStreamField (if non-primitive) will return Object.class (as
 * opposed to a more specific reference type).
 */
ObjectStreamField(Field field, boolean unshared, boolean showType) {
    this.field = field;
    this.unshared = unshared;
    name = field.getName();
    Class<?> ftype = field.getType();
    type = (showType || ftype.isPrimitive()) ? ftype : Object.class;
    signature = getClassSignature(ftype).intern();
}

/**
 * Get the name of this field.
 *
 * @return  a <code>String</code> representing the name of the serializable
 *          field
 */
public String getName() {
    return name;
}

/**
 * Get the type of the field.  If the type is non-primitive and this
 * <code>ObjectStreamField</code> was obtained from a deserialized {@link
 * ObjectStreamClass} instance, then <code>Object.class</code> is returned.
 * Otherwise, the <code>Class</code> object for the type of the field is
 * returned.
 *
 * @return  a <code>Class</code> object representing the type of the
 *          serializable field
 */
@CallerSensitive
public Class<?> getType() {
    if (System.getSecurityManager() != null) {
        Class<?> caller = Reflection.getCallerClass();
        if (ReflectUtil.needsPackageAccessCheck(caller.getClassLoader(), type.getClassLoader())) {
            ReflectUtil.checkPackageAccess(type);
        }
    }
    return type;
}

/**
 * Returns character encoding of field type.  The encoding is as follows:
 * <blockquote><pre>
 * B            byte
 * C            char
 * D            double
 * F            float
 * I            int
 * J            long
 * L            class or interface
 * S            short
 * Z            boolean
 * [            array
```

```
     * </pre></blockquote>
     *
     * @return  the typecode of the serializable field
     */
    // REMIND: deprecate?
    public char getTypeCode() {
        return signature.charAt(0);
    }

    /**
     * Return the JVM type signature.
     *
     * @return  null if this field has a primitive type.
     */
    // REMIND: deprecate?
    public String getTypeString() {
        return isPrimitive() ? null : signature;
    }

    /**
     * Offset of field within instance data.
     *
     * @return  the offset of this field
     * @see #setOffset
     */
    // REMIND: deprecate?
    public int getOffset() {
        return offset;
    }

    /**
     * Offset within instance data.
     *
     * @param   offset the offset of the field
     * @see #getOffset
     */
    // REMIND: deprecate?
    protected void setOffset(int offset) {
        this.offset = offset;
    }

    /**
     * Return true if this field has a primitive type.
     *
     * @return  true if and only if this field corresponds to a primitive type
     */
    // REMIND: deprecate?
    public boolean isPrimitive() {
        char tcode = signature.charAt(0);
        return ((tcode != 'L') && (tcode != '['));
    }

    /**
     * Returns boolean value indicating whether or not the serializable field
     * represented by this ObjectStreamField instance is unshared.
     *
     * @return {@code true} if this field is unshared
     *
     * @since 1.4
     */
    public boolean isUnshared() {
        return unshared;
    }
```

```
/**
 * Compare this field with another <code>ObjectStreamField</code>.  Return
 * -1 if this is smaller, 0 if equal, 1 if greater.  Types that are
 * primitives are "smaller" than object types.  If equal, the field names
 * are compared.
 */
// REMIND: deprecate?
public int compareTo(Object obj) {
    ObjectStreamField other = (ObjectStreamField) obj;
    boolean isPrim = isPrimitive();
    if (isPrim != other.isPrimitive()) {
        return isPrim ? -1 : 1;
    }
    return name.compareTo(other.name);
}

/**
 * Return a string that describes this field.
 */
public String toString() {
    return signature + ' ' + name;
}

/**
 * Returns field represented by this ObjectStreamField, or null if
 * ObjectStreamField is not associated with an actual field.
 */
Field getField() {
    return field;
}

/**
 * Returns JVM type signature of field (similar to getTypeString, except
 * that signature strings are returned for primitive fields as well).
 */
String getSignature() {
    return signature;
}

/**
 * Returns JVM type signature for given class.
 */
private static String getClassSignature(Class<?> cl) {
    StringBuilder sbuf = new StringBuilder();
    while (cl.isArray()) {
        sbuf.append('[');
        cl = cl.getComponentType();
    }
    if (cl.isPrimitive()) {
        if (cl == Integer.TYPE) {
            sbuf.append('I');
        } else if (cl == Byte.TYPE) {
            sbuf.append('B');
        } else if (cl == Long.TYPE) {
            sbuf.append('J');
        } else if (cl == Float.TYPE) {
            sbuf.append('F');
        } else if (cl == Double.TYPE) {
            sbuf.append('D');
        } else if (cl == Short.TYPE) {
            sbuf.append('S');
        } else if (cl == Character.TYPE) {
```

```
                sbuf.append('C');
            } else if (cl == Boolean.TYPE) {
                sbuf.append('Z');
            } else if (cl == Void.TYPE) {
                sbuf.append('V');
            } else {
                throw new InternalError();
            }
        } else {
            sbuf.append('L' + cl.getName().replace('.', '/') + ';');
        }
        return sbuf.toString();
    }
}
```

OptionalDataException.java

```
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
package java.io;

/**
 * Exception indicating the failure of an object read operation due to
 * unread primitive data, or the end of data belonging to a serialized
 * object in the stream.  This exception may be thrown in two cases:
 *
 * <ul>
 *   <li>An attempt was made to read an object when the next element in the
 *       stream is primitive data.  In this case, the OptionalDataException's
 *       length field is set to the number of bytes of primitive data
 *       immediately readable from the stream, and the eof field is set to
 *       false.
 *
 *   <li>An attempt was made to read past the end of data consumable by a
 *       class-defined readObject or readExternal method.  In this case, the
 *       OptionalDataException's eof field is set to true, and the length field
 *       is set to 0.
 * </ul>
 *
 * @author  unascribed
 * @since   JDK1.1
 */
public class OptionalDataException extends ObjectStreamException {

    private static final long serialVersionUID = -8011121865681257820L;

    /*
     * Create an <code>OptionalDataException</code> with a length.
     */
    OptionalDataException(int len) {
        eof = false;
        length = len;
    }

    /*
```

```
     * Create an <code>OptionalDataException</code> signifying no
     * more primitive data is available.
     */
    OptionalDataException(boolean end) {
        length = 0;
        eof = end;
    }

    /**
     * The number of bytes of primitive data available to be read
     * in the current buffer.
     *
     * @serial
     */
    public int length;

    /**
     * True if there is no more data in the buffered part of the stream.
     *
     * @serial
     */
    public boolean eof;
}
```

OutputStream.java

```java
/*
 * Copyright (c) 1994, 2004, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * This abstract class is the superclass of all classes representing
 * an output stream of bytes. An output stream accepts output bytes
 * and sends them to some sink.
 * <p>
 * Applications that need to define a subclass of
 * <code>OutputStream</code> must always provide at least a method
 * that writes one byte of output.
 *
 * @author  Arthur van Hoff
 * @see     java.io.BufferedOutputStream
 * @see     java.io.ByteArrayOutputStream
 * @see     java.io.DataOutputStream
 * @see     java.io.FilterOutputStream
 * @see     java.io.InputStream
 * @see     java.io.OutputStream#write(int)
 * @since   JDK1.0
 */
public abstract class OutputStream implements Closeable, Flushable {
    /**
     * Writes the specified byte to this output stream. The general
     * contract for <code>write</code> is that one byte is written
     * to the output stream. The byte to be written is the eight
     * low-order bits of the argument <code>b</code>. The 24
     * high-order bits of <code>b</code> are ignored.
     * <p>
     * Subclasses of <code>OutputStream</code> must provide an
     * implementation for this method.
     *
     * @param      b   the <code>byte</code>.
     * @exception  IOException  if an I/O error occurs. In particular,
     *             an <code>IOException</code> may be thrown if the
     *             output stream has been closed.
```

```
     */
    public abstract void write(int b) throws IOException;

    /**
     * Writes <code>b.length</code> bytes from the specified byte array
     * to this output stream. The general contract for <code>write(b)</code>
     * is that it should have exactly the same effect as the call
     * <code>write(b, 0, b.length)</code>.
     *
     * @param      b   the data.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.OutputStream#write(byte[], int, int)
     */
    public void write(byte b[]) throws IOException {
        write(b, 0, b.length);
    }

    /**
     * Writes <code>len</code> bytes from the specified byte array
     * starting at offset <code>off</code> to this output stream.
     * The general contract for <code>write(b, off, len)</code> is that
     * some of the bytes in the array <code>b</code> are written to the
     * output stream in order; element <code>b[off]</code> is the first
     * byte written and <code>b[off+len-1]</code> is the last byte written
     * by this operation.
     * <p>
     * The <code>write</code> method of <code>OutputStream</code> calls
     * the write method of one argument on each of the bytes to be
     * written out. Subclasses are encouraged to override this method and
     * provide a more efficient implementation.
     * <p>
     * If <code>b</code> is <code>null</code>, a
     * <code>NullPointerException</code> is thrown.
     * <p>
     * If <code>off</code> is negative, or <code>len</code> is negative, or
     * <code>off+len</code> is greater than the length of the array
     * <code>b</code>, then an <tt>IndexOutOfBoundsException</tt> is thrown.
     *
     * @param      b     the data.
     * @param      off   the start offset in the data.
     * @param      len   the number of bytes to write.
     * @exception  IOException  if an I/O error occurs. In particular,
     *             an <code>IOException</code> is thrown if the output
     *             stream is closed.
     */
    public void write(byte b[], int off, int len) throws IOException {
        if (b == null) {
            throw new NullPointerException();
        } else if ((off < 0) || (off > b.length) || (len < 0) ||
                   ((off + len) > b.length) || ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return;
        }
        for (int i = 0 ; i < len ; i++) {
            write(b[off + i]);
        }
    }

    /**
     * Flushes this output stream and forces any buffered output bytes
     * to be written out. The general contract of <code>flush</code> is
     * that calling it is an indication that, if any bytes previously
```

```
     * written have been buffered by the implementation of the output
     * stream, such bytes should immediately be written to their
     * intended destination.
     * <p>
     * If the intended destination of this stream is an abstraction provided by
     * the underlying operating system, for example a file, then flushing the
     * stream guarantees only that bytes previously written to the stream are
     * passed to the operating system for writing; it does not guarantee that
     * they are actually written to a physical device such as a disk drive.
     * <p>
     * The <code>flush</code> method of <code>OutputStream</code> does nothing.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void flush() throws IOException {
    }

    /**
     * Closes this output stream and releases any system resources
     * associated with this stream. The general contract of <code>close</code>
     * is that it closes the output stream. A closed stream cannot perform
     * output operations and cannot be reopened.
     * <p>
     * The <code>close</code> method of <code>OutputStream</code> does nothing.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void close() throws IOException {
    }

}
```

OutputStreamWriter.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.nio.charset.Charset;
import java.nio.charset.CharsetEncoder;
import sun.nio.cs.StreamEncoder;


/**
 * An OutputStreamWriter is a bridge from character streams to byte streams:
 * Characters written to it are encoded into bytes using a specified {@link
 * java.nio.charset.Charset charset}.  The charset that it uses
 * may be specified by name or may be given explicitly, or the platform's
 * default charset may be accepted.
 *
 * <p> Each invocation of a write() method causes the encoding converter to be
 * invoked on the given character(s).  The resulting bytes are accumulated in a
 * buffer before being written to the underlying output stream.  The size of
 * this buffer may be specified, but by default it is large enough for most
 * purposes.  Note that the characters passed to the write() methods are not
 * buffered.
 *
 * <p> For top efficiency, consider wrapping an OutputStreamWriter within a
 * BufferedWriter so as to avoid frequent converter invocations.  For example:
 *
 * <pre>
 * Writer out
 *   = new BufferedWriter(new OutputStreamWriter(System.out));
 * </pre>
 *
 * <p> A <i>surrogate pair</i> is a character represented by a sequence of two
 * <tt>char</tt> values: A <i>high</i> surrogate in the range '\uD800' to
 * '\uDBFF' followed by a <i>low</i> surrogate in the range '\uDC00' to
 * '\uDFFF'.
 *
 * <p> A <i>malformed surrogate element</i> is a high surrogate that is not
```

```
 * followed by a low surrogate or a low surrogate that is not preceded by a
 * high surrogate.
 *
 * <p> This class always replaces malformed surrogate elements and unmappable
 * character sequences with the charset's default <i>substitution sequence</i>.
 * The {@linkplain java.nio.charset.CharsetEncoder} class should be used when more
 * control over the encoding process is required.
 *
 * @see BufferedWriter
 * @see OutputStream
 * @see java.nio.charset.Charset
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class OutputStreamWriter extends Writer {

    private final StreamEncoder se;

    /**
     * Creates an OutputStreamWriter that uses the named charset.
     *
     * @param  out
     *         An OutputStream
     *
     * @param  charsetName
     *         The name of a supported
     *         {@link java.nio.charset.Charset charset}
     *
     * @exception  UnsupportedEncodingException
     *             If the named encoding is not supported
     */
    public OutputStreamWriter(OutputStream out, String charsetName)
        throws UnsupportedEncodingException
    {
        super(out);
        if (charsetName == null)
            throw new NullPointerException("charsetName");
        se = StreamEncoder.forOutputStreamWriter(out, this, charsetName);
    }

    /**
     * Creates an OutputStreamWriter that uses the default character encoding.
     *
     * @param  out  An OutputStream
     */
    public OutputStreamWriter(OutputStream out) {
        super(out);
        try {
            se = StreamEncoder.forOutputStreamWriter(out, this, (String)null);
        } catch (UnsupportedEncodingException e) {
            throw new Error(e);
        }
    }

    /**
     * Creates an OutputStreamWriter that uses the given charset.
     *
     * @param  out
     *         An OutputStream
     *
     * @param  cs
```

```
     *         A charset
     *
     * @since 1.4
     * @spec JSR-51
     */
    public OutputStreamWriter(OutputStream out, Charset cs) {
        super(out);
        if (cs == null)
            throw new NullPointerException("charset");
        se = StreamEncoder.forOutputStreamWriter(out, this, cs);
    }

    /**
     * Creates an OutputStreamWriter that uses the given charset encoder.
     *
     * @param  out
     *         An OutputStream
     *
     * @param  enc
     *         A charset encoder
     *
     * @since 1.4
     * @spec JSR-51
     */
    public OutputStreamWriter(OutputStream out, CharsetEncoder enc) {
        super(out);
        if (enc == null)
            throw new NullPointerException("charset encoder");
        se = StreamEncoder.forOutputStreamWriter(out, this, enc);
    }

    /**
     * Returns the name of the character encoding being used by this stream.
     *
     * <p> If the encoding has an historical name then that name is returned;
     * otherwise the encoding's canonical name is returned.
     *
     * <p> If this instance was created with the {@link
     * #OutputStreamWriter(OutputStream, String)} constructor then the returned
     * name, being unique for the encoding, may differ from the name passed to
     * the constructor.  This method may return <tt>null</tt> if the stream has
     * been closed. </p>
     *
     * @return The historical name of this encoding, or possibly
     *         <code>null</code> if the stream has been closed
     *
     * @see java.nio.charset.Charset
     *
     * @revised 1.4
     * @spec JSR-51
     */
    public String getEncoding() {
        return se.getEncoding();
    }

    /**
     * Flushes the output buffer to the underlying byte stream, without flushing
     * the byte stream itself.  This method is non-private only so that it may
     * be invoked by PrintStream.
     */
    void flushBuffer() throws IOException {
        se.flushBuffer();
    }
```

```
    /**
     * Writes a single character.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void write(int c) throws IOException {
        se.write(c);
    }

    /**
     * Writes a portion of an array of characters.
     *
     * @param  cbuf  Buffer of characters
     * @param  off   Offset from which to start writing characters
     * @param  len   Number of characters to write
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void write(char cbuf[], int off, int len) throws IOException {
        se.write(cbuf, off, len);
    }

    /**
     * Writes a portion of a string.
     *
     * @param  str  A String
     * @param  off  Offset from which to start writing characters
     * @param  len  Number of characters to write
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void write(String str, int off, int len) throws IOException {
        se.write(str, off, len);
    }

    /**
     * Flushes the stream.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void flush() throws IOException {
        se.flush();
    }

    public void close() throws IOException {
        se.close();
    }
}
```

PipedInputStream.java

```
/*
 * Copyright (c) 1995, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * A piped input stream should be connected
 * to a piped output stream; the piped  input
 * stream then provides whatever data bytes
 * are written to the piped output  stream.
 * Typically, data is read from a <code>PipedInputStream</code>
 * object by one thread  and data is written
 * to the corresponding <code>PipedOutputStream</code>
 * by some  other thread. Attempting to use
 * both objects from a single thread is not
 * recommended, as it may deadlock the thread.
 * The piped input stream contains a buffer,
 * decoupling read operations from write operations,
 * within limits.
 * A pipe is said to be <a name="BROKEN"> <i>broken</i> </a> if a
 * thread that was providing data bytes to the connected
 * piped output stream is no longer alive.
 *
 * @author  James Gosling
 * @see     java.io.PipedOutputStream
 * @since   JDK1.0
 */
public class PipedInputStream extends InputStream {
    boolean closedByWriter = false;
    volatile boolean closedByReader = false;
    boolean connected = false;

        /* REMIND: identification of the read and write sides needs to be
           more sophisticated.  Either using thread groups (but what about
           pipes within a thread?) or using finalization (but it may be a
           long time until the next GC). */
    Thread readSide;
    Thread writeSide;
```

```
private static final int DEFAULT_PIPE_SIZE = 1024;

/**
 * The default size of the pipe's circular input buffer.
 * @since   JDK1.1
 */
// This used to be a constant before the pipe size was allowed
// to change. This field will continue to be maintained
// for backward compatibility.
protected static final int PIPE_SIZE = DEFAULT_PIPE_SIZE;

/**
 * The circular buffer into which incoming data is placed.
 * @since   JDK1.1
 */
protected byte buffer[];

/**
 * The index of the position in the circular buffer at which the
 * next byte of data will be stored when received from the connected
 * piped output stream. <code>in<0</code> implies the buffer is empty,
 * <code>in==out</code> implies the buffer is full
 * @since   JDK1.1
 */
protected int in = -1;

/**
 * The index of the position in the circular buffer at which the next
 * byte of data will be read by this piped input stream.
 * @since   JDK1.1
 */
protected int out = 0;

/**
 * Creates a <code>PipedInputStream</code> so
 * that it is connected to the piped output
 * stream <code>src</code>. Data bytes written
 * to <code>src</code> will then be  available
 * as input from this stream.
 *
 * @param      src   the stream to connect to.
 * @exception  IOException  if an I/O error occurs.
 */
public PipedInputStream(PipedOutputStream src) throws IOException {
    this(src, DEFAULT_PIPE_SIZE);
}

/**
 * Creates a <code>PipedInputStream</code> so that it is
 * connected to the piped output stream
 * <code>src</code> and uses the specified pipe size for
 * the pipe's buffer.
 * Data bytes written to <code>src</code> will then
 * be available as input from this stream.
 *
 * @param      src   the stream to connect to.
 * @param      pipeSize the size of the pipe's buffer.
 * @exception  IOException  if an I/O error occurs.
 * @exception  IllegalArgumentException if {@code pipeSize <= 0}.
 * @since      1.6
 */
public PipedInputStream(PipedOutputStream src, int pipeSize)
```

```
        throws IOException {
     initPipe(pipeSize);
     connect(src);
}

/**
 * Creates a <code>PipedInputStream</code> so
 * that it is not yet {@linkplain #connect(java.io.PipedOutputStream)
 * connected}.
 * It must be {@linkplain java.io.PipedOutputStream#connect(
 * java.io.PipedInputStream) connected} to a
 * <code>PipedOutputStream</code> before being used.
 */
public PipedInputStream() {
    initPipe(DEFAULT_PIPE_SIZE);
}

/**
 * Creates a <code>PipedInputStream</code> so that it is not yet
 * {@linkplain #connect(java.io.PipedOutputStream) connected} and
 * uses the specified pipe size for the pipe's buffer.
 * It must be {@linkplain java.io.PipedOutputStream#connect(
 * java.io.PipedInputStream)
 * connected} to a <code>PipedOutputStream</code> before being used.
 *
 * @param      pipeSize the size of the pipe's buffer.
 * @exception  IllegalArgumentException if {@code pipeSize <= 0}.
 * @since      1.6
 */
public PipedInputStream(int pipeSize) {
    initPipe(pipeSize);
}

private void initPipe(int pipeSize) {
     if (pipeSize <= 0) {
        throw new IllegalArgumentException("Pipe Size <= 0");
     }
     buffer = new byte[pipeSize];
}

/**
 * Causes this piped input stream to be connected
 * to the piped  output stream <code>src</code>.
 * If this object is already connected to some
 * other piped output  stream, an <code>IOException</code>
 * is thrown.
 * <p>
 * If <code>src</code> is an
 * unconnected piped output stream and <code>snk</code>
 * is an unconnected piped input stream, they
 * may be connected by either the call:
 *
 * <pre><code>snk.connect(src)</code> </pre>
 * <p>
 * or the call:
 *
 * <pre><code>src.connect(snk)</code> </pre>
 * <p>
 * The two calls have the same effect.
 *
 * @param      src   The piped output stream to connect to.
 * @exception  IOException  if an I/O error occurs.
 */
```

```
public void connect(PipedOutputStream src) throws IOException {
    src.connect(this);
}

/**
 * Receives a byte of data.  This method will block if no input is
 * available.
 * @param b the byte being received
 * @exception IOException If the pipe is <a href="#BROKEN"> <code>broken</code></a>,
 *          {@link #connect(java.io.PipedOutputStream) unconnected},
 *          closed, or if an I/O error occurs.
 * @since     JDK1.1
 */
protected synchronized void receive(int b) throws IOException {
    checkStateForReceive();
    writeSide = Thread.currentThread();
    if (in == out)
        awaitSpace();
    if (in < 0) {
        in = 0;
        out = 0;
    }
    buffer[in++] = (byte)(b & 0xFF);
    if (in >= buffer.length) {
        in = 0;
    }
}

/**
 * Receives data into an array of bytes.  This method will
 * block until some input is available.
 * @param b the buffer into which the data is received
 * @param off the start offset of the data
 * @param len the maximum number of bytes received
 * @exception IOException If the pipe is <a href="#BROKEN"> broken</a>,
 *           {@link #connect(java.io.PipedOutputStream) unconnected},
 *           closed,or if an I/O error occurs.
 */
synchronized void receive(byte b[], int off, int len)  throws IOException {
    checkStateForReceive();
    writeSide = Thread.currentThread();
    int bytesToTransfer = len;
    while (bytesToTransfer > 0) {
        if (in == out)
            awaitSpace();
        int nextTransferAmount = 0;
        if (out < in) {
            nextTransferAmount = buffer.length - in;
        } else if (in < out) {
            if (in == -1) {
                in = out = 0;
                nextTransferAmount = buffer.length - in;
            } else {
                nextTransferAmount = out - in;
            }
        }
        if (nextTransferAmount > bytesToTransfer)
            nextTransferAmount = bytesToTransfer;
        assert(nextTransferAmount > 0);
        System.arraycopy(b, off, buffer, in, nextTransferAmount);
        bytesToTransfer -= nextTransferAmount;
        off += nextTransferAmount;
        in += nextTransferAmount;
```

```
        if (in >= buffer.length) {
            in = 0;
        }
    }
}

private void checkStateForReceive() throws IOException {
    if (!connected) {
        throw new IOException("Pipe not connected");
    } else if (closedByWriter || closedByReader) {
        throw new IOException("Pipe closed");
    } else if (readSide != null && !readSide.isAlive()) {
        throw new IOException("Read end dead");
    }
}

private void awaitSpace() throws IOException {
    while (in == out) {
        checkStateForReceive();

        /* full: kick any waiting readers */
        notifyAll();
        try {
            wait(1000);
        } catch (InterruptedException ex) {
            throw new java.io.InterruptedIOException();
        }
    }
}

/**
 * Notifies all waiting threads that the last byte of data has been
 * received.
 */
synchronized void receivedLast() {
    closedByWriter = true;
    notifyAll();
}

/**
 * Reads the next byte of data from this piped input stream. The
 * value byte is returned as an <code>int</code> in the range
 * <code>0</code> to <code>255</code>.
 * This method blocks until input data is available, the end of the
 * stream is detected, or an exception is thrown.
 *
 * @return     the next byte of data, or <code>-1</code> if the end of the
 *             stream is reached.
 * @exception  IOException  if the pipe is
 *           {@link #connect(java.io.PipedOutputStream) unconnected},
 *           <a href="#BROKEN"> <code>broken</code></a>, closed,
 *           or if an I/O error occurs.
 */
public synchronized int read()  throws IOException {
    if (!connected) {
        throw new IOException("Pipe not connected");
    } else if (closedByReader) {
        throw new IOException("Pipe closed");
    } else if (writeSide != null && !writeSide.isAlive()
               && !closedByWriter && (in < 0)) {
        throw new IOException("Write end dead");
    }
```

```
    readSide = Thread.currentThread();
    int trials = 2;
    while (in < 0) {
        if (closedByWriter) {
            /* closed by writer, return EOF */
            return -1;
        }
        if ((writeSide != null) && (!writeSide.isAlive()) && (--trials < 0)) {
            throw new IOException("Pipe broken");
        }
        /* might be a writer waiting */
        notifyAll();
        try {
            wait(1000);
        } catch (InterruptedException ex) {
            throw new java.io.InterruptedIOException();
        }
    }
    int ret = buffer[out++] & 0xFF;
    if (out >= buffer.length) {
        out = 0;
    }
    if (in == out) {
        /* now empty */
        in = -1;
    }

    return ret;
}

/**
 * Reads up to <code>len</code> bytes of data from this piped input
 * stream into an array of bytes. Less than <code>len</code> bytes
 * will be read if the end of the data stream is reached or if
 * <code>len</code> exceeds the pipe's buffer size.
 * If <code>len </code> is zero, then no bytes are read and 0 is returned;
 * otherwise, the method blocks until at least 1 byte of input is
 * available, end of the stream has been detected, or an exception is
 * thrown.
 *
 * @param      b     the buffer into which the data is read.
 * @param      off   the start offset in the destination array <code>b</code>
 * @param      len   the maximum number of bytes read.
 * @return     the total number of bytes read into the buffer, or
 *             <code>-1</code> if there is no more data because the end of
 *             the stream has been reached.
 * @exception  NullPointerException If <code>b</code> is <code>null</code>.
 * @exception  IndexOutOfBoundsException If <code>off</code> is negative,
 * <code>len</code> is negative, or <code>len</code> is greater than
 * <code>b.length - off</code>
 * @exception  IOException if the pipe is <a href="#BROKEN"> <code>broken</code></a>,
 *           {@link #connect(java.io.PipedOutputStream) unconnected},
 *           closed, or if an I/O error occurs.
 */
public synchronized int read(byte b[], int off, int len)  throws IOException {
    if (b == null) {
        throw new NullPointerException();
    } else if (off < 0 || len < 0 || len > b.length - off) {
        throw new IndexOutOfBoundsException();
    } else if (len == 0) {
        return 0;
    }
```

```
    /* possibly wait on the first character */
    int c = read();
    if (c < 0) {
        return -1;
    }
    b[off] = (byte) c;
    int rlen = 1;
    while ((in >= 0) && (len > 1)) {

        int available;

        if (in > out) {
            available = Math.min((buffer.length - out), (in - out));
        } else {
            available = buffer.length - out;
        }

        // A byte is read beforehand outside the loop
        if (available > (len - 1)) {
            available = len - 1;
        }
        System.arraycopy(buffer, out, b, off + rlen, available);
        out += available;
        rlen += available;
        len -= available;

        if (out >= buffer.length) {
            out = 0;
        }
        if (in == out) {
            /* now empty */
            in = -1;
        }
    }
    return rlen;
}

/**
 * Returns the number of bytes that can be read from this input
 * stream without blocking.
 *
 * @return the number of bytes that can be read from this input stream
 *         without blocking, or {@code 0} if this input stream has been
 *         closed by invoking its {@link #close()} method, or if the pipe
 *         is {@link #connect(java.io.PipedOutputStream) unconnected}, or
 *          <a href="#BROKEN"> <code>broken</code></a>.
 *
 * @exception  IOException  if an I/O error occurs.
 * @since   JDK1.0.2
 */
public synchronized int available() throws IOException {
    if(in < 0)
        return 0;
    else if(in == out)
        return buffer.length;
    else if (in > out)
        return in - out;
    else
        return in + buffer.length - out;
}

/**
 * Closes this piped input stream and releases any system resources
```

```
     * associated with the stream.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void close()  throws IOException {
        closedByReader = true;
        synchronized (this) {
            in = -1;
        }
    }
}
```

PipedOutputStream.java

```
/*
 * Copyright (c) 1995, 2006, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.io.*;

/**
 * A piped output stream can be connected to a piped input stream
 * to create a communications pipe. The piped output stream is the
 * sending end of the pipe. Typically, data is written to a
 * <code>PipedOutputStream</code> object by one thread and data is
 * read from the connected <code>PipedInputStream</code> by some
 * other thread. Attempting to use both objects from a single thread
 * is not recommended as it may deadlock the thread.
 * The pipe is said to be <a name=BROKEN> <i>broken</i> </a> if a
 * thread that was reading data bytes from the connected piped input
 * stream is no longer alive.
 *
 * @author  James Gosling
 * @see     java.io.PipedInputStream
 * @since   JDK1.0
 */
public
class PipedOutputStream extends OutputStream {

        /* REMIND: identification of the read and write sides needs to be
           more sophisticated.  Either using thread groups (but what about
           pipes within a thread?) or using finalization (but it may be a
           long time until the next GC). */
    private PipedInputStream sink;

    /**
     * Creates a piped output stream connected to the specified piped
     * input stream. Data bytes written to this stream will then be
     * available as input from <code>snk</code>.
     *
     * @param      snk   The piped input stream to connect to.
```

```
     * @exception  IOException  if an I/O error occurs.
     */
    public PipedOutputStream(PipedInputStream snk)  throws IOException {
        connect(snk);
    }

    /**
     * Creates a piped output stream that is not yet connected to a
     * piped input stream. It must be connected to a piped input stream,
     * either by the receiver or the sender, before being used.
     *
     * @see     java.io.PipedInputStream#connect(java.io.PipedOutputStream)
     * @see     java.io.PipedOutputStream#connect(java.io.PipedInputStream)
     */
    public PipedOutputStream() {
    }

    /**
     * Connects this piped output stream to a receiver. If this object
     * is already connected to some other piped input stream, an
     * <code>IOException</code> is thrown.
     * <p>
     * If <code>snk</code> is an unconnected piped input stream and
     * <code>src</code> is an unconnected piped output stream, they may
     * be connected by either the call:
     * <blockquote><pre>
     * src.connect(snk)</pre></blockquote>
     * or the call:
     * <blockquote><pre>
     * snk.connect(src)</pre></blockquote>
     * The two calls have the same effect.
     *
     * @param      snk   the piped input stream to connect to.
     * @exception  IOException  if an I/O error occurs.
     */
    public synchronized void connect(PipedInputStream snk) throws IOException {
        if (snk == null) {
            throw new NullPointerException();
        } else if (sink != null || snk.connected) {
            throw new IOException("Already connected");
        }
        sink = snk;
        snk.in = -1;
        snk.out = 0;
        snk.connected = true;
    }

    /**
     * Writes the specified <code>byte</code> to the piped output stream.
     * <p>
     * Implements the <code>write</code> method of <code>OutputStream</code>.
     *
     * @param      b   the <code>byte</code> to be written.
     * @exception IOException if the pipe is <a href=#BROKEN> broken</a>,
     *          {@link #connect(java.io.PipedInputStream) unconnected},
     *          closed, or if an I/O error occurs.
     */
    public void write(int b)  throws IOException {
        if (sink == null) {
            throw new IOException("Pipe not connected");
        }
        sink.receive(b);
    }
```

```
    /**
     * Writes <code>len</code> bytes from the specified byte array
     * starting at offset <code>off</code> to this piped output stream.
     * This method blocks until all the bytes are written to the output
     * stream.
     *
     * @param      b     the data.
     * @param      off   the start offset in the data.
     * @param      len   the number of bytes to write.
     * @exception IOException if the pipe is <a href=#BROKEN> broken</a>,
     *          {@link #connect(java.io.PipedInputStream) unconnected},
     *          closed, or if an I/O error occurs.
     */
    public void write(byte b[], int off, int len) throws IOException {
        if (sink == null) {
            throw new IOException("Pipe not connected");
        } else if (b == null) {
            throw new NullPointerException();
        } else if ((off < 0) || (off > b.length) || (len < 0) ||
                   ((off + len) > b.length) || ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return;
        }
        sink.receive(b, off, len);
    }

    /**
     * Flushes this output stream and forces any buffered output bytes
     * to be written out.
     * This will notify any readers that bytes are waiting in the pipe.
     *
     * @exception IOException if an I/O error occurs.
     */
    public synchronized void flush() throws IOException {
        if (sink != null) {
            synchronized (sink) {
                sink.notifyAll();
            }
        }
    }

    /**
     * Closes this piped output stream and releases any system resources
     * associated with this stream. This stream may no longer be used for
     * writing bytes.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void close()  throws IOException {
        if (sink != null) {
            sink.receivedLast();
        }
    }
}
```

PipedReader.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Piped character-input streams.
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class PipedReader extends Reader {
    boolean closedByWriter = false;
    boolean closedByReader = false;
    boolean connected = false;

    /* REMIND: identification of the read and write sides needs to be
       more sophisticated.  Either using thread groups (but what about
       pipes within a thread?) or using finalization (but it may be a
       long time until the next GC). */
    Thread readSide;
    Thread writeSide;

   /**
    * The size of the pipe's circular input buffer.
    */
    private static final int DEFAULT_PIPE_SIZE = 1024;

    /**
     * The circular buffer into which incoming data is placed.
     */
    char buffer[];

    /**
     * The index of the position in the circular buffer at which the
     * next character of data will be stored when received from the connected
```

```
     * piped writer. <code>in<0</code> implies the buffer is empty,
     * <code>in==out</code> implies the buffer is full
     */
    int in = -1;

    /**
     * The index of the position in the circular buffer at which the next
     * character of data will be read by this piped reader.
     */
    int out = 0;

    /**
     * Creates a <code>PipedReader</code> so
     * that it is connected to the piped writer
     * <code>src</code>. Data written to <code>src</code>
     * will then be available as input from this stream.
     *
     * @param      src   the stream to connect to.
     * @exception  IOException  if an I/O error occurs.
     */
    public PipedReader(PipedWriter src) throws IOException {
        this(src, DEFAULT_PIPE_SIZE);
    }

    /**
     * Creates a <code>PipedReader</code> so that it is connected
     * to the piped writer <code>src</code> and uses the specified
     * pipe size for the pipe's buffer. Data written to <code>src</code>
     * will then be  available as input from this stream.
     * @param      src       the stream to connect to.
     * @param      pipeSize  the size of the pipe's buffer.
     * @exception  IOException  if an I/O error occurs.
     * @exception  IllegalArgumentException if {@code pipeSize <= 0}.
     * @since      1.6
     */
    public PipedReader(PipedWriter src, int pipeSize) throws IOException {
        initPipe(pipeSize);
        connect(src);
    }


    /**
     * Creates a <code>PipedReader</code> so
     * that it is not yet {@linkplain #connect(java.io.PipedWriter)
     * connected}. It must be {@linkplain java.io.PipedWriter#connect(
     * java.io.PipedReader) connected} to a <code>PipedWriter</code>
     * before being used.
     */
    public PipedReader() {
        initPipe(DEFAULT_PIPE_SIZE);
    }

    /**
     * Creates a <code>PipedReader</code> so that it is not yet
     * {@link #connect(java.io.PipedWriter) connected} and uses
     * the specified pipe size for the pipe's buffer.
     * It must be  {@linkplain java.io.PipedWriter#connect(
     * java.io.PipedReader) connected} to a <code>PipedWriter</code>
     * before being used.
     *
     * @param   pipeSize the size of the pipe's buffer.
     * @exception  IllegalArgumentException if {@code pipeSize <= 0}.
     * @since      1.6
```

```
     */
    public PipedReader(int pipeSize) {
        initPipe(pipeSize);
    }

    private void initPipe(int pipeSize) {
        if (pipeSize <= 0) {
            throw new IllegalArgumentException("Pipe size <= 0");
        }
        buffer = new char[pipeSize];
    }

    /**
     * Causes this piped reader to be connected
     * to the piped  writer <code>src</code>.
     * If this object is already connected to some
     * other piped writer, an <code>IOException</code>
     * is thrown.
     * <p>
     * If <code>src</code> is an
     * unconnected piped writer and <code>snk</code>
     * is an unconnected piped reader, they
     * may be connected by either the call:
     *
     * <pre><code>snk.connect(src)</code> </pre>
     * <p>
     * or the call:
     *
     * <pre><code>src.connect(snk)</code> </pre>
     * <p>
     * The two calls have the same effect.
     *
     * @param      src   The piped writer to connect to.
     * @exception  IOException  if an I/O error occurs.
     */
    public void connect(PipedWriter src) throws IOException {
        src.connect(this);
    }

    /**
     * Receives a char of data. This method will block if no input is
     * available.
     */
    synchronized void receive(int c) throws IOException {
        if (!connected) {
            throw new IOException("Pipe not connected");
        } else if (closedByWriter || closedByReader) {
            throw new IOException("Pipe closed");
        } else if (readSide != null && !readSide.isAlive()) {
            throw new IOException("Read end dead");
        }

        writeSide = Thread.currentThread();
        while (in == out) {
            if ((readSide != null) && !readSide.isAlive()) {
                throw new IOException("Pipe broken");
            }
            /* full: kick any waiting readers */
            notifyAll();
            try {
                wait(1000);
            } catch (InterruptedException ex) {
                throw new java.io.InterruptedIOException();
```

```
        }
    }
    if (in < 0) {
        in = 0;
        out = 0;
    }
    buffer[in++] = (char) c;
    if (in >= buffer.length) {
        in = 0;
    }
}

/**
 * Receives data into an array of characters.  This method will
 * block until some input is available.
 */
synchronized void receive(char c[], int off, int len)  throws IOException {
    while (--len >= 0) {
        receive(c[off++]);
    }
}

/**
 * Notifies all waiting threads that the last character of data has been
 * received.
 */
synchronized void receivedLast() {
    closedByWriter = true;
    notifyAll();
}

/**
 * Reads the next character of data from this piped stream.
 * If no character is available because the end of the stream
 * has been reached, the value <code>-1</code> is returned.
 * This method blocks until input data is available, the end of
 * the stream is detected, or an exception is thrown.
 *
 * @return     the next character of data, or <code>-1</code> if the end of the
 *             stream is reached.
 * @exception  IOException  if the pipe is
 *          <a href=PipedInputStream.html#BROKEN> <code>broken</code></a>,
 *          {@link #connect(java.io.PipedWriter) unconnected}, closed,
 *          or an I/O error occurs.
 */
public synchronized int read()  throws IOException {
    if (!connected) {
        throw new IOException("Pipe not connected");
    } else if (closedByReader) {
        throw new IOException("Pipe closed");
    } else if (writeSide != null && !writeSide.isAlive()
               && !closedByWriter && (in < 0)) {
        throw new IOException("Write end dead");
    }

    readSide = Thread.currentThread();
    int trials = 2;
    while (in < 0) {
        if (closedByWriter) {
            /* closed by writer, return EOF */
            return -1;
        }
        if ((writeSide != null) && (!writeSide.isAlive()) && (--trials < 0)) {
```

```
            throw new IOException("Pipe broken");
        }
        /* might be a writer waiting */
        notifyAll();
        try {
            wait(1000);
        } catch (InterruptedException ex) {
            throw new java.io.InterruptedIOException();
        }
    }
    int ret = buffer[out++];
    if (out >= buffer.length) {
        out = 0;
    }
    if (in == out) {
        /* now empty */
        in = -1;
    }
    return ret;
}

/**
 * Reads up to <code>len</code> characters of data from this piped
 * stream into an array of characters. Less than <code>len</code> characters
 * will be read if the end of the data stream is reached or if
 * <code>len</code> exceeds the pipe's buffer size. This method
 * blocks until at least one character of input is available.
 *
 * @param      cbuf     the buffer into which the data is read.
 * @param      off   the start offset of the data.
 * @param      len   the maximum number of characters read.
 * @return     the total number of characters read into the buffer, or
 *             <code>-1</code> if there is no more data because the end of
 *             the stream has been reached.
 * @exception  IOException  if the pipe is
 *                 <a href=PipedInputStream.html#BROKEN> <code>broken</code></a>,
 *                 {@link #connect(java.io.PipedWriter) unconnected}, closed,
 *                 or an I/O error occurs.
 */
public synchronized int read(char cbuf[], int off, int len)  throws IOException {
    if (!connected) {
        throw new IOException("Pipe not connected");
    } else if (closedByReader) {
        throw new IOException("Pipe closed");
    } else if (writeSide != null && !writeSide.isAlive()
               && !closedByWriter && (in < 0)) {
        throw new IOException("Write end dead");
    }

    if ((off < 0) || (off > cbuf.length) || (len < 0) ||
        ((off + len) > cbuf.length) || ((off + len) < 0)) {
        throw new IndexOutOfBoundsException();
    } else if (len == 0) {
        return 0;
    }

    /* possibly wait on the first character */
    int c = read();
    if (c < 0) {
        return -1;
    }
    cbuf[off] =  (char)c;
    int rlen = 1;
```

```
        while ((in >= 0) && (--len > 0)) {
            cbuf[off + rlen] = buffer[out++];
            rlen++;
            if (out >= buffer.length) {
                out = 0;
            }
            if (in == out) {
                /* now empty */
                in = -1;
            }
        }
        return rlen;
    }

    /**
     * Tell whether this stream is ready to be read.  A piped character
     * stream is ready if the circular buffer is not empty.
     *
     * @exception  IOException  if the pipe is
     *                  <a href=PipedInputStream.html#BROKEN> <code>broken</code></a>,
     *                  {@link #connect(java.io.PipedWriter) unconnected}, or closed.
     */
    public synchronized boolean ready() throws IOException {
        if (!connected) {
            throw new IOException("Pipe not connected");
        } else if (closedByReader) {
            throw new IOException("Pipe closed");
        } else if (writeSide != null && !writeSide.isAlive()
                   && !closedByWriter && (in < 0)) {
            throw new IOException("Write end dead");
        }
        if (in < 0) {
            return false;
        } else {
            return true;
        }
    }

    /**
     * Closes this piped stream and releases any system resources
     * associated with the stream.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void close()  throws IOException {
        in = -1;
        closedByReader = true;
    }
}
```

PipedWriter.java

```
/*
 * Copyright (c) 1996, 2006, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Piped character-output streams.
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class PipedWriter extends Writer {

    /* REMIND: identification of the read and write sides needs to be
       more sophisticated.  Either using thread groups (but what about
       pipes within a thread?) or using finalization (but it may be a
       long time until the next GC). */
    private PipedReader sink;

    /* This flag records the open status of this particular writer. It
     * is independent of the status flags defined in PipedReader. It is
     * used to do a sanity check on connect.
     */
    private boolean closed = false;

    /**
     * Creates a piped writer connected to the specified piped
     * reader. Data characters written to this stream will then be
     * available as input from <code>snk</code>.
     *
     * @param      snk   The piped reader to connect to.
     * @exception  IOException  if an I/O error occurs.
     */
    public PipedWriter(PipedReader snk)  throws IOException {
        connect(snk);
    }
```

```
/**
 * Creates a piped writer that is not yet connected to a
 * piped reader. It must be connected to a piped reader,
 * either by the receiver or the sender, before being used.
 *
 * @see     java.io.PipedReader#connect(java.io.PipedWriter)
 * @see     java.io.PipedWriter#connect(java.io.PipedReader)
 */
public PipedWriter() {
}

/**
 * Connects this piped writer to a receiver. If this object
 * is already connected to some other piped reader, an
 * <code>IOException</code> is thrown.
 * <p>
 * If <code>snk</code> is an unconnected piped reader and
 * <code>src</code> is an unconnected piped writer, they may
 * be connected by either the call:
 * <blockquote><pre>
 * src.connect(snk)</pre></blockquote>
 * or the call:
 * <blockquote><pre>
 * snk.connect(src)</pre></blockquote>
 * The two calls have the same effect.
 *
 * @param      snk   the piped reader to connect to.
 * @exception  IOException  if an I/O error occurs.
 */
public synchronized void connect(PipedReader snk) throws IOException {
    if (snk == null) {
        throw new NullPointerException();
    } else if (sink != null || snk.connected) {
        throw new IOException("Already connected");
    } else if (snk.closedByReader || closed) {
        throw new IOException("Pipe closed");
    }

    sink = snk;
    snk.in = -1;
    snk.out = 0;
    snk.connected = true;
}

/**
 * Writes the specified <code>char</code> to the piped output stream.
 * If a thread was reading data characters from the connected piped input
 * stream, but the thread is no longer alive, then an
 * <code>IOException</code> is thrown.
 * <p>
 * Implements the <code>write</code> method of <code>Writer</code>.
 *
 * @param      c   the <code>char</code> to be written.
 * @exception  IOException  if the pipe is
 *          <a href=PipedOutputStream.html#BROKEN> <code>broken</code></a>,
 *          {@link #connect(java.io.PipedReader) unconnected}, closed
 *          or an I/O error occurs.
 */
public void write(int c)  throws IOException {
    if (sink == null) {
        throw new IOException("Pipe not connected");
    }
```

```
            sink.receive(c);
        }

        /**
         * Writes <code>len</code> characters from the specified character array
         * starting at offset <code>off</code> to this piped output stream.
         * This method blocks until all the characters are written to the output
         * stream.
         * If a thread was reading data characters from the connected piped input
         * stream, but the thread is no longer alive, then an
         * <code>IOException</code> is thrown.
         *
         * @param      cbuf  the data.
         * @param      off   the start offset in the data.
         * @param      len   the number of characters to write.
         * @exception  IOException  if the pipe is
         *          <a href=PipedOutputStream.html#BROKEN> <code>broken</code></a>,
         *          {@link #connect(java.io.PipedReader) unconnected}, closed
         *          or an I/O error occurs.
         */
        public void write(char cbuf[], int off, int len) throws IOException {
            if (sink == null) {
                throw new IOException("Pipe not connected");
            } else if ((off | len | (off + len) | (cbuf.length - (off + len))) < 0) {
                throw new IndexOutOfBoundsException();
            }
            sink.receive(cbuf, off, len);
        }

        /**
         * Flushes this output stream and forces any buffered output characters
         * to be written out.
         * This will notify any readers that characters are waiting in the pipe.
         *
         * @exception  IOException  if the pipe is closed, or an I/O error occurs.
         */
        public synchronized void flush() throws IOException {
            if (sink != null) {
                if (sink.closedByReader || closed) {
                    throw new IOException("Pipe closed");
                }
                synchronized (sink) {
                    sink.notifyAll();
                }
            }
        }

        /**
         * Closes this piped output stream and releases any system resources
         * associated with this stream. This stream may no longer be used for
         * writing characters.
         *
         * @exception  IOException  if an I/O error occurs.
         */
        public void close()  throws IOException {
            closed = true;
            if (sink != null) {
                sink.receivedLast();
            }
        }
    }
```

PrintStream.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.util.Formatter;
import java.util.Locale;
import java.nio.charset.Charset;
import java.nio.charset.IllegalCharsetNameException;
import java.nio.charset.UnsupportedCharsetException;

/**
 * A <code>PrintStream</code> adds functionality to another output stream,
 * namely the ability to print representations of various data values
 * conveniently.  Two other features are provided as well.  Unlike other output
 * streams, a <code>PrintStream</code> never throws an
 * <code>IOException</code>; instead, exceptional situations merely set an
 * internal flag that can be tested via the <code>checkError</code> method.
 * Optionally, a <code>PrintStream</code> can be created so as to flush
 * automatically; this means that the <code>flush</code> method is
 * automatically invoked after a byte array is written, one of the
 * <code>println</code> methods is invoked, or a newline character or byte
 * (<code>'\n'</code>) is written.
 *
 * <p> All characters printed by a <code>PrintStream</code> are converted into
 * bytes using the platform's default character encoding.  The <code>{@link
 * PrintWriter}</code> class should be used in situations that require writing
 * characters rather than bytes.
 *
 * @author     Frank Yellin
 * @author     Mark Reinhold
 * @since      JDK1.0
 */

public class PrintStream extends FilterOutputStream
    implements Appendable, Closeable
{
```

```
private final boolean autoFlush;
private boolean trouble = false;
private Formatter formatter;

/**
 * Track both the text- and character-output streams, so that their buffers
 * can be flushed without flushing the entire stream.
 */
private BufferedWriter textOut;
private OutputStreamWriter charOut;

/**
 * requireNonNull is explicitly declared here so as not to create an extra
 * dependency on java.util.Objects.requireNonNull. PrintStream is loaded
 * early during system initialization.
 */
private static <T> T requireNonNull(T obj, String message) {
    if (obj == null)
        throw new NullPointerException(message);
    return obj;
}

/**
 * Returns a charset object for the given charset name.
 * @throws NullPointerException          is csn is null
 * @throws UnsupportedEncodingException  if the charset is not supported
 */
private static Charset toCharset(String csn)
    throws UnsupportedEncodingException
{
    requireNonNull(csn, "charsetName");
    try {
        return Charset.forName(csn);
    } catch (IllegalCharsetNameException|UnsupportedCharsetException unused) {
        // UnsupportedEncodingException should be thrown
        throw new UnsupportedEncodingException(csn);
    }
}

/* Private constructors */
private PrintStream(boolean autoFlush, OutputStream out) {
    super(out);
    this.autoFlush = autoFlush;
    this.charOut = new OutputStreamWriter(this);
    this.textOut = new BufferedWriter(charOut);
}

private PrintStream(boolean autoFlush, OutputStream out, Charset charset) {
    super(out);
    this.autoFlush = autoFlush;
    this.charOut = new OutputStreamWriter(this, charset);
    this.textOut = new BufferedWriter(charOut);
}

/* Variant of the private constructor so that the given charset name
 * can be verified before evaluating the OutputStream argument. Used
 * by constructors creating a FileOutputStream that also take a
 * charset name.
 */
private PrintStream(boolean autoFlush, Charset charset, OutputStream out)
    throws UnsupportedEncodingException
{
    this(autoFlush, out, charset);
```

```
}

/**
 * Creates a new print stream.  This stream will not flush automatically.
 *
 * @param  out        The output stream to which values and objects will be
 *                    printed
 *
 * @see java.io.PrintWriter#PrintWriter(java.io.OutputStream)
 */
public PrintStream(OutputStream out) {
    this(out, false);
}

/**
 * Creates a new print stream.
 *
 * @param  out        The output stream to which values and objects will be
 *                    printed
 * @param  autoFlush  A boolean; if true, the output buffer will be flushed
 *                    whenever a byte array is written, one of the
 *                    <code>println</code> methods is invoked, or a newline
 *                    character or byte (<code>'\n'</code>) is written
 *
 * @see java.io.PrintWriter#PrintWriter(java.io.OutputStream, boolean)
 */
public PrintStream(OutputStream out, boolean autoFlush) {
    this(autoFlush, requireNonNull(out, "Null output stream"));
}

/**
 * Creates a new print stream.
 *
 * @param  out        The output stream to which values and objects will be
 *                    printed
 * @param  autoFlush  A boolean; if true, the output buffer will be flushed
 *                    whenever a byte array is written, one of the
 *                    <code>println</code> methods is invoked, or a newline
 *                    character or byte (<code>'\n'</code>) is written
 * @param  encoding   The name of a supported
 *                    <a href="../lang/package-summary.html#charenc">
 *                    character encoding</a>
 *
 * @throws  UnsupportedEncodingException
 *          If the named encoding is not supported
 *
 * @since  1.4
 */
public PrintStream(OutputStream out, boolean autoFlush, String encoding)
    throws UnsupportedEncodingException
{
    this(autoFlush,
         requireNonNull(out, "Null output stream"),
         toCharset(encoding));
}

/**
 * Creates a new print stream, without automatic line flushing, with the
 * specified file name.  This convenience constructor creates
 * the necessary intermediate {@link java.io.OutputStreamWriter
 * OutputStreamWriter}, which will encode characters using the
 * {@linkplain java.nio.charset.Charset#defaultCharset() default charset}
 * for this instance of the Java virtual machine.
```

```
     *
     * @param  fileName
     *         The name of the file to use as the destination of this print
     *         stream.  If the file exists, then it will be truncated to
     *         zero size; otherwise, a new file will be created.  The output
     *         will be written to the file and is buffered.
     *
     * @throws  FileNotFoundException
     *          If the given file object does not denote an existing, writable
     *          regular file and a new regular file of that name cannot be
     *          created, or if some other error occurs while opening or
     *          creating the file
     *
     * @throws  SecurityException
     *          If a security manager is present and {@link
     *          SecurityManager#checkWrite checkWrite(fileName)} denies write
     *          access to the file
     *
     * @since  1.5
     */
    public PrintStream(String fileName) throws FileNotFoundException {
        this(false, new FileOutputStream(fileName));
    }

    /**
     * Creates a new print stream, without automatic line flushing, with the
     * specified file name and charset.  This convenience constructor creates
     * the necessary intermediate {@link java.io.OutputStreamWriter
     * OutputStreamWriter}, which will encode characters using the provided
     * charset.
     *
     * @param  fileName
     *         The name of the file to use as the destination of this print
     *         stream.  If the file exists, then it will be truncated to
     *         zero size; otherwise, a new file will be created.  The output
     *         will be written to the file and is buffered.
     *
     * @param  csn
     *         The name of a supported {@linkplain java.nio.charset.Charset
     *         charset}
     *
     * @throws  FileNotFoundException
     *          If the given file object does not denote an existing, writable
     *          regular file and a new regular file of that name cannot be
     *          created, or if some other error occurs while opening or
     *          creating the file
     *
     * @throws  SecurityException
     *          If a security manager is present and {@link
     *          SecurityManager#checkWrite checkWrite(fileName)} denies write
     *          access to the file
     *
     * @throws  UnsupportedEncodingException
     *          If the named charset is not supported
     *
     * @since  1.5
     */
    public PrintStream(String fileName, String csn)
        throws FileNotFoundException, UnsupportedEncodingException
    {
        // ensure charset is checked before the file is opened
        this(false, toCharset(csn), new FileOutputStream(fileName));
    }
```

```
/**
 * Creates a new print stream, without automatic line flushing, with the
 * specified file.  This convenience constructor creates the necessary
 * intermediate {@link java.io.OutputStreamWriter OutputStreamWriter},
 * which will encode characters using the {@linkplain
 * java.nio.charset.Charset#defaultCharset() default charset} for this
 * instance of the Java virtual machine.
 *
 * @param  file
 *         The file to use as the destination of this print stream.  If the
 *         file exists, then it will be truncated to zero size; otherwise,
 *         a new file will be created.  The output will be written to the
 *         file and is buffered.
 *
 * @throws  FileNotFoundException
 *          If the given file object does not denote an existing, writable
 *          regular file and a new regular file of that name cannot be
 *          created, or if some other error occurs while opening or
 *          creating the file
 *
 * @throws  SecurityException
 *          If a security manager is present and {@link
 *          SecurityManager#checkWrite checkWrite(file.getPath())}
 *          denies write access to the file
 *
 * @since  1.5
 */
public PrintStream(File file) throws FileNotFoundException {
    this(false, new FileOutputStream(file));
}

/**
 * Creates a new print stream, without automatic line flushing, with the
 * specified file and charset.  This convenience constructor creates
 * the necessary intermediate {@link java.io.OutputStreamWriter
 * OutputStreamWriter}, which will encode characters using the provided
 * charset.
 *
 * @param  file
 *         The file to use as the destination of this print stream.  If the
 *         file exists, then it will be truncated to zero size; otherwise,
 *         a new file will be created.  The output will be written to the
 *         file and is buffered.
 *
 * @param  csn
 *         The name of a supported {@linkplain java.nio.charset.Charset
 *         charset}
 *
 * @throws  FileNotFoundException
 *          If the given file object does not denote an existing, writable
 *          regular file and a new regular file of that name cannot be
 *          created, or if some other error occurs while opening or
 *          creating the file
 *
 * @throws  SecurityException
 *          If a security manager is present and {@link
 *          SecurityManager#checkWrite checkWrite(file.getPath())}
 *          denies write access to the file
 *
 * @throws  UnsupportedEncodingException
 *          If the named charset is not supported
 *
```

```
     * @since  1.5
     */
    public PrintStream(File file, String csn)
        throws FileNotFoundException, UnsupportedEncodingException
    {
        // ensure charset is checked before the file is opened
        this(false, toCharset(csn), new FileOutputStream(file));
    }

    /** Check to make sure that the stream has not been closed */
    private void ensureOpen() throws IOException {
        if (out == null)
            throw new IOException("Stream closed");
    }

    /**
     * Flushes the stream.  This is done by writing any buffered output bytes to
     * the underlying output stream and then flushing that stream.
     *
     * @see        java.io.OutputStream#flush()
     */
    public void flush() {
        synchronized (this) {
            try {
                ensureOpen();
                out.flush();
            }
            catch (IOException x) {
                trouble = true;
            }
        }
    }

    private boolean closing = false; /* To avoid recursive closing */

    /**
     * Closes the stream.  This is done by flushing the stream and then closing
     * the underlying output stream.
     *
     * @see        java.io.OutputStream#close()
     */
    public void close() {
        synchronized (this) {
            if (! closing) {
                closing = true;
                try {
                    textOut.close();
                    out.close();
                }
                catch (IOException x) {
                    trouble = true;
                }
                textOut = null;
                charOut = null;
                out = null;
            }
        }
    }

    /**
     * Flushes the stream and checks its error state. The internal error state
     * is set to <code>true</code> when the underlying output stream throws an
     * <code>IOException</code> other than <code>InterruptedIOException</code>,
```

```
     * and when the <code>setError</code> method is invoked.  If an operation
     * on the underlying output stream throws an
     * <code>InterruptedIOException</code>, then the <code>PrintStream</code>
     * converts the exception back into an interrupt by doing:
     * <pre>
     *     Thread.currentThread().interrupt();
     * </pre>
     * or the equivalent.
     *
     * @return <code>true</code> if and only if this stream has encountered an
     *         <code>IOException</code> other than
     *         <code>InterruptedIOException</code>, or the
     *         <code>setError</code> method has been invoked
     */
    public boolean checkError() {
        if (out != null)
            flush();
        if (out instanceof java.io.PrintStream) {
            PrintStream ps = (PrintStream) out;
            return ps.checkError();
        }
        return trouble;
    }

    /**
     * Sets the error state of the stream to <code>true</code>.
     *
     * <p> This method will cause subsequent invocations of {@link
     * #checkError()} to return <tt>true</tt> until {@link
     * #clearError()} is invoked.
     *
     * @since JDK1.1
     */
    protected void setError() {
        trouble = true;
    }

    /**
     * Clears the internal error state of this stream.
     *
     * <p> This method will cause subsequent invocations of {@link
     * #checkError()} to return <tt>false</tt> until another write
     * operation fails and invokes {@link #setError()}.
     *
     * @since 1.6
     */
    protected void clearError() {
        trouble = false;
    }

    /*
     * Exception-catching, synchronized output operations,
     * which also implement the write() methods of OutputStream
     */

    /**
     * Writes the specified byte to this stream.  If the byte is a newline and
     * automatic flushing is enabled then the <code>flush</code> method will be
     * invoked.
     *
     * <p> Note that the byte is written as given; to write a character that
     * will be translated according to the platform's default character
     * encoding, use the <code>print(char)</code> or <code>println(char)</code>
```

```
     * methods.
     *
     * @param  b  The byte to be written
     * @see #print(char)
     * @see #println(char)
     */
    public void write(int b) {
        try {
            synchronized (this) {
                ensureOpen();
                out.write(b);
                if ((b == '\n') && autoFlush)
                    out.flush();
            }
        }
        catch (InterruptedIOException x) {
            Thread.currentThread().interrupt();
        }
        catch (IOException x) {
            trouble = true;
        }
    }

    /**
     * Writes <code>len</code> bytes from the specified byte array starting at
     * offset <code>off</code> to this stream.  If automatic flushing is
     * enabled then the <code>flush</code> method will be invoked.
     *
     * <p> Note that the bytes will be written as given; to write characters
     * that will be translated according to the platform's default character
     * encoding, use the <code>print(char)</code> or <code>println(char)</code>
     * methods.
     *
     * @param  buf   A byte array
     * @param  off   Offset from which to start taking bytes
     * @param  len   Number of bytes to write
     */
    public void write(byte buf[], int off, int len) {
        try {
            synchronized (this) {
                ensureOpen();
                out.write(buf, off, len);
                if (autoFlush)
                    out.flush();
            }
        }
        catch (InterruptedIOException x) {
            Thread.currentThread().interrupt();
        }
        catch (IOException x) {
            trouble = true;
        }
    }

    /*
     * The following private methods on the text- and character-output streams
     * always flush the stream buffers, so that writes to the underlying byte
     * stream occur as promptly as with the original PrintStream.
     */

    private void write(char buf[]) {
        try {
            synchronized (this) {
```

```
            ensureOpen();
            textOut.write(buf);
            textOut.flushBuffer();
            charOut.flushBuffer();
            if (autoFlush) {
                for (int i = 0; i < buf.length; i++)
                    if (buf[i] == '\n')
                        out.flush();
            }
        }
    }
    catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    }
    catch (IOException x) {
        trouble = true;
    }
}

private void write(String s) {
    try {
        synchronized (this) {
            ensureOpen();
            textOut.write(s);
            textOut.flushBuffer();
            charOut.flushBuffer();
            if (autoFlush && (s.indexOf('\n') >= 0))
                out.flush();
        }
    }
    catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    }
    catch (IOException x) {
        trouble = true;
    }
}

private void newLine() {
    try {
        synchronized (this) {
            ensureOpen();
            textOut.newLine();
            textOut.flushBuffer();
            charOut.flushBuffer();
            if (autoFlush)
                out.flush();
        }
    }
    catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    }
    catch (IOException x) {
        trouble = true;
    }
}

/* Methods that do not terminate lines */

/**
 * Prints a boolean value.  The string produced by <code>{@link
 * java.lang.String#valueOf(boolean)}</code> is translated into bytes
 * according to the platform's default character encoding, and these bytes
```

```
     * are written in exactly the manner of the
     * <code>{@link #write(int)}</code> method.
     *
     * @param      b   The <code>boolean</code> to be printed
     */
    public void print(boolean b) {
        write(b ? "true" : "false");
    }

    /**
     * Prints a character.  The character is translated into one or more bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the
     * <code>{@link #write(int)}</code> method.
     *
     * @param      c   The <code>char</code> to be printed
     */
    public void print(char c) {
        write(String.valueOf(c));
    }

    /**
     * Prints an integer.  The string produced by <code>{@link
     * java.lang.String#valueOf(int)}</code> is translated into bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the
     * <code>{@link #write(int)}</code> method.
     *
     * @param      i   The <code>int</code> to be printed
     * @see        java.lang.Integer#toString(int)
     */
    public void print(int i) {
        write(String.valueOf(i));
    }

    /**
     * Prints a long integer.  The string produced by <code>{@link
     * java.lang.String#valueOf(long)}</code> is translated into bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the
     * <code>{@link #write(int)}</code> method.
     *
     * @param      l   The <code>long</code> to be printed
     * @see        java.lang.Long#toString(long)
     */
    public void print(long l) {
        write(String.valueOf(l));
    }

    /**
     * Prints a floating-point number.  The string produced by <code>{@link
     * java.lang.String#valueOf(float)}</code> is translated into bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the
     * <code>{@link #write(int)}</code> method.
     *
     * @param      f   The <code>float</code> to be printed
     * @see        java.lang.Float#toString(float)
     */
    public void print(float f) {
        write(String.valueOf(f));
    }
```

```
/**
 * Prints a double-precision floating-point number.  The string produced by
 * <code>{@link java.lang.String#valueOf(double)}</code> is translated into
 * bytes according to the platform's default character encoding, and these
 * bytes are written in exactly the manner of the <code>{@link
 * #write(int)}</code> method.
 *
 * @param      d   The <code>double</code> to be printed
 * @see        java.lang.Double#toString(double)
 */
public void print(double d) {
    write(String.valueOf(d));
}

/**
 * Prints an array of characters.  The characters are converted into bytes
 * according to the platform's default character encoding, and these bytes
 * are written in exactly the manner of the
 * <code>{@link #write(int)}</code> method.
 *
 * @param      s   The array of chars to be printed
 *
 * @throws  NullPointerException  If <code>s</code> is <code>null</code>
 */
public void print(char s[]) {
    write(s);
}

/**
 * Prints a string.  If the argument is <code>null</code> then the string
 * <code>"null"</code> is printed.  Otherwise, the string's characters are
 * converted into bytes according to the platform's default character
 * encoding, and these bytes are written in exactly the manner of the
 * <code>{@link #write(int)}</code> method.
 *
 * @param      s   The <code>String</code> to be printed
 */
public void print(String s) {
    if (s == null) {
        s = "null";
    }
    write(s);
}

/**
 * Prints an object.  The string produced by the <code>{@link
 * java.lang.String#valueOf(Object)}</code> method is translated into bytes
 * according to the platform's default character encoding, and these bytes
 * are written in exactly the manner of the
 * <code>{@link #write(int)}</code> method.
 *
 * @param      obj   The <code>Object</code> to be printed
 * @see        java.lang.Object#toString()
 */
public void print(Object obj) {
    write(String.valueOf(obj));
}


/* Methods that do terminate lines */

/**
 * Terminates the current line by writing the line separator string.  The
```

```
     * line separator string is defined by the system property
     * <code>line.separator</code>, and is not necessarily a single newline
     * character (<code>'\n'</code>).
     */
    public void println() {
        newLine();
    }

    /**
     * Prints a boolean and then terminate the line.  This method behaves as
     * though it invokes <code>{@link #print(boolean)}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x  The <code>boolean</code> to be printed
     */
    public void println(boolean x) {
        synchronized (this) {
            print(x);
            newLine();
        }
    }

    /**
     * Prints a character and then terminate the line.  This method behaves as
     * though it invokes <code>{@link #print(char)}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x  The <code>char</code> to be printed.
     */
    public void println(char x) {
        synchronized (this) {
            print(x);
            newLine();
        }
    }

    /**
     * Prints an integer and then terminate the line.  This method behaves as
     * though it invokes <code>{@link #print(int)}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x  The <code>int</code> to be printed.
     */
    public void println(int x) {
        synchronized (this) {
            print(x);
            newLine();
        }
    }

    /**
     * Prints a long and then terminate the line.  This method behaves as
     * though it invokes <code>{@link #print(long)}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x  a The <code>long</code> to be printed.
     */
    public void println(long x) {
        synchronized (this) {
            print(x);
            newLine();
        }
    }
```

```
/**
 * Prints a float and then terminate the line.  This method behaves as
 * though it invokes <code>{@link #print(float)}</code> and then
 * <code>{@link #println()}</code>.
 *
 * @param x  The <code>float</code> to be printed.
 */
public void println(float x) {
    synchronized (this) {
        print(x);
        newLine();
    }
}

/**
 * Prints a double and then terminate the line.  This method behaves as
 * though it invokes <code>{@link #print(double)}</code> and then
 * <code>{@link #println()}</code>.
 *
 * @param x  The <code>double</code> to be printed.
 */
public void println(double x) {
    synchronized (this) {
        print(x);
        newLine();
    }
}

/**
 * Prints an array of characters and then terminate the line.  This method
 * behaves as though it invokes <code>{@link #print(char[])}</code> and
 * then <code>{@link #println()}</code>.
 *
 * @param x  an array of chars to print.
 */
public void println(char x[]) {
    synchronized (this) {
        print(x);
        newLine();
    }
}

/**
 * Prints a String and then terminate the line.  This method behaves as
 * though it invokes <code>{@link #print(String)}</code> and then
 * <code>{@link #println()}</code>.
 *
 * @param x  The <code>String</code> to be printed.
 */
public void println(String x) {
    synchronized (this) {
        print(x);
        newLine();
    }
}

/**
 * Prints an Object and then terminate the line.  This method calls
 * at first String.valueOf(x) to get the printed object's string value,
 * then behaves as
 * though it invokes <code>{@link #print(String)}</code> and then
 * <code>{@link #println()}</code>.
```

```
     *
     * @param x  The <code>Object</code> to be printed.
     */
    public void println(Object x) {
        String s = String.valueOf(x);
        synchronized (this) {
            print(s);
            newLine();
        }
    }


    /**
     * A convenience method to write a formatted string to this output stream
     * using the specified format string and arguments.
     *
     * <p> An invocation of this method of the form <tt>out.printf(format,
     * args)</tt> behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.format(format, args) </pre>
     *
     * @param  format
     *         A format string as described in <a
     *         href="../util/Formatter.html#syntax">Format string syntax</a>
     *
     * @param  args
     *         Arguments referenced by the format specifiers in the format
     *         string.  If there are more arguments than format specifiers, the
     *         extra arguments are ignored.  The number of arguments is
     *         variable and may be zero.  The maximum number of arguments is
     *         limited by the maximum dimension of a Java array as defined by
     *         <cite>The Java™ Virtual Machine Specification</cite>.
     *         The behaviour on a
     *         <tt>null</tt> argument depends on the <a
     *         href="../util/Formatter.html#syntax">conversion</a>.
     *
     * @throws  java.util.IllegalFormatException
     *          If a format string contains an illegal syntax, a format
     *          specifier that is incompatible with the given arguments,
     *          insufficient arguments given the format string, or other
     *          illegal conditions.  For specification of all possible
     *          formatting errors, see the <a
     *          href="../util/Formatter.html#detail">Details</a> section of the
     *          formatter class specification.
     *
     * @throws  NullPointerException
     *          If the <tt>format</tt> is <tt>null</tt>
     *
     * @return  This output stream
     *
     * @since  1.5
     */
    public PrintStream printf(String format, Object ... args) {
        return format(format, args);
    }

    /**
     * A convenience method to write a formatted string to this output stream
     * using the specified format string and arguments.
     *
     * <p> An invocation of this method of the form <tt>out.printf(l, format,
     * args)</tt> behaves in exactly the same way as the invocation
```

```
 *
 * <pre>
 *     out.format(l, format, args) </pre>
 *
 * @param  l
 *         The {@linkplain java.util.Locale locale} to apply during
 *         formatting.  If <tt>l</tt> is <tt>null</tt> then no localization
 *         is applied.
 *
 * @param  format
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  java.util.IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a> section of the
 *          formatter class specification.
 *
 * @throws  NullPointerException
 *          If the <tt>format</tt> is <tt>null</tt>
 *
 * @return  This output stream
 *
 * @since  1.5
 */
public PrintStream printf(Locale l, String format, Object ... args) {
    return format(l, format, args);
}

/**
 * Writes a formatted string to this output stream using the specified
 * format string and arguments.
 *
 * <p> The locale always used is the one returned by {@link
 * java.util.Locale#getDefault() Locale.getDefault()}, regardless of any
 * previous invocations of other formatting methods on this object.
 *
 * @param  format
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
```

```
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  java.util.IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a> section of the
 *          formatter class specification.
 *
 * @throws  NullPointerException
 *          If the <tt>format</tt> is <tt>null</tt>
 *
 * @return  This output stream
 *
 * @since  1.5
 */
public PrintStream format(String format, Object ... args) {
    try {
        synchronized (this) {
            ensureOpen();
            if ((formatter == null)
                || (formatter.locale() != Locale.getDefault()))
                formatter = new Formatter((Appendable) this);
            formatter.format(Locale.getDefault(), format, args);
        }
    } catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    } catch (IOException x) {
        trouble = true;
    }
    return this;
}

/**
 * Writes a formatted string to this output stream using the specified
 * format string and arguments.
 *
 * @param  l
 *         The {@linkplain java.util.Locale locale} to apply during
 *         formatting.  If <tt>l</tt> is <tt>null</tt> then no localization
 *         is applied.
 *
 * @param  format
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  java.util.IllegalFormatException
 *          If a format string contains an illegal syntax, a format
```

```
     *          specifier that is incompatible with the given arguments,
     *          insufficient arguments given the format string, or other
     *          illegal conditions.  For specification of all possible
     *          formatting errors, see the <a
     *          href="../util/Formatter.html#detail">Details</a> section of the
     *          formatter class specification.
     *
     * @throws  NullPointerException
     *          If the <tt>format</tt> is <tt>null</tt>
     *
     * @return  This output stream
     *
     * @since  1.5
     */
    public PrintStream format(Locale l, String format, Object ... args) {
        try {
            synchronized (this) {
                ensureOpen();
                if ((formatter == null)
                    || (formatter.locale() != l))
                    formatter = new Formatter(this, l);
                formatter.format(l, format, args);
            }
        } catch (InterruptedIOException x) {
            Thread.currentThread().interrupt();
        } catch (IOException x) {
            trouble = true;
        }
        return this;
    }

    /**
     * Appends the specified character sequence to this output stream.
     *
     * <p> An invocation of this method of the form <tt>out.append(csq)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.print(csq.toString()) </pre>
     *
     * <p> Depending on the specification of <tt>toString</tt> for the
     * character sequence <tt>csq</tt>, the entire sequence may not be
     * appended.  For instance, invoking then <tt>toString</tt> method of a
     * character buffer will return a subsequence whose content depends upon
     * the buffer's position and limit.
     *
     * @param  csq
     *         The character sequence to append.  If <tt>csq</tt> is
     *         <tt>null</tt>, then the four characters <tt>"null"</tt> are
     *         appended to this output stream.
     *
     * @return  This output stream
     *
     * @since  1.5
     */
    public PrintStream append(CharSequence csq) {
        if (csq == null)
            print("null");
        else
            print(csq.toString());
        return this;
    }
```

```
    /**
     * Appends a subsequence of the specified character sequence to this output
     * stream.
     *
     * <p> An invocation of this method of the form <tt>out.append(csq, start,
     * end)</tt> when <tt>csq</tt> is not <tt>null</tt>, behaves in
     * exactly the same way as the invocation
     *
     * <pre>
     *     out.print(csq.subSequence(start, end).toString()) </pre>
     *
     * @param  csq
     *         The character sequence from which a subsequence will be
     *         appended.  If <tt>csq</tt> is <tt>null</tt>, then characters
     *         will be appended as if <tt>csq</tt> contained the four
     *         characters <tt>"null"</tt>.
     *
     * @param  start
     *         The index of the first character in the subsequence
     *
     * @param  end
     *         The index of the character following the last character in the
     *         subsequence
     *
     * @return  This output stream
     *
     * @throws  IndexOutOfBoundsException
     *          If <tt>start</tt> or <tt>end</tt> are negative, <tt>start</tt>
     *          is greater than <tt>end</tt>, or <tt>end</tt> is greater than
     *          <tt>csq.length()</tt>
     *
     * @since  1.5
     */
    public PrintStream append(CharSequence csq, int start, int end) {
        CharSequence cs = (csq == null ? "null" : csq);
        write(cs.subSequence(start, end).toString());
        return this;
    }

    /**
     * Appends the specified character to this output stream.
     *
     * <p> An invocation of this method of the form <tt>out.append(c)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.print(c) </pre>
     *
     * @param  c
     *         The 16-bit character to append
     *
     * @return  This output stream
     *
     * @since  1.5
     */
    public PrintStream append(char c) {
        print(c);
        return this;
    }

}
```

PrintWriter.java

```java
/*
 * Copyright (c) 1996, 2012, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.util.Objects;
import java.util.Formatter;
import java.util.Locale;
import java.nio.charset.Charset;
import java.nio.charset.IllegalCharsetNameException;
import java.nio.charset.UnsupportedCharsetException;

/**
 * Prints formatted representations of objects to a text-output stream.  This
 * class implements all of the <tt>print</tt> methods found in {@link
 * PrintStream}.  It does not contain methods for writing raw bytes, for which
 * a program should use unencoded byte streams.
 *
 * <p> Unlike the {@link PrintStream} class, if automatic flushing is enabled
 * it will be done only when one of the <tt>println</tt>, <tt>printf</tt>, or
 * <tt>format</tt> methods is invoked, rather than whenever a newline character
 * happens to be output.  These methods use the platform's own notion of line
 * separator rather than the newline character.
 *
 * <p> Methods in this class never throw I/O exceptions, although some of its
 * constructors may.  The client may inquire as to whether any errors have
 * occurred by invoking {@link #checkError checkError()}.
 *
 * @author      Frank Yellin
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class PrintWriter extends Writer {

    /**
     * The underlying character-output stream of this
     * <code>PrintWriter</code>.
```

```
 *
 * @since 1.2
 */
protected Writer out;

private final boolean autoFlush;
private boolean trouble = false;
private Formatter formatter;
private PrintStream psOut = null;

/**
 * Line separator string.  This is the value of the line.separator
 * property at the moment that the stream was created.
 */
private final String lineSeparator;

/**
 * Returns a charset object for the given charset name.
 * @throws NullPointerException          is csn is null
 * @throws UnsupportedEncodingException  if the charset is not supported
 */
private static Charset toCharset(String csn)
    throws UnsupportedEncodingException
{
    Objects.requireNonNull(csn, "charsetName");
    try {
        return Charset.forName(csn);
    } catch (IllegalCharsetNameException|UnsupportedCharsetException unused) {
        // UnsupportedEncodingException should be thrown
        throw new UnsupportedEncodingException(csn);
    }
}

/**
 * Creates a new PrintWriter, without automatic line flushing.
 *
 * @param  out        A character-output stream
 */
public PrintWriter (Writer out) {
    this(out, false);
}

/**
 * Creates a new PrintWriter.
 *
 * @param  out        A character-output stream
 * @param  autoFlush  A boolean; if true, the <tt>println</tt>,
 *                    <tt>printf</tt>, or <tt>format</tt> methods will
 *                    flush the output buffer
 */
public PrintWriter(Writer out,
                   boolean autoFlush) {
    super(out);
    this.out = out;
    this.autoFlush = autoFlush;
    lineSeparator = java.security.AccessController.doPrivileged(
        new sun.security.action.GetPropertyAction("line.separator"));
}

/**
 * Creates a new PrintWriter, without automatic line flushing, from an
 * existing OutputStream.  This convenience constructor creates the
 * necessary intermediate OutputStreamWriter, which will convert characters
```

```
     * into bytes using the default character encoding.
     *
     * @param  out        An output stream
     *
     * @see java.io.OutputStreamWriter#OutputStreamWriter(java.io.OutputStream)
     */
    public PrintWriter(OutputStream out) {
        this(out, false);
    }

    /**
     * Creates a new PrintWriter from an existing OutputStream.  This
     * convenience constructor creates the necessary intermediate
     * OutputStreamWriter, which will convert characters into bytes using the
     * default character encoding.
     *
     * @param  out        An output stream
     * @param  autoFlush  A boolean; if true, the <tt>println</tt>,
     *                    <tt>printf</tt>, or <tt>format</tt> methods will
     *                    flush the output buffer
     *
     * @see java.io.OutputStreamWriter#OutputStreamWriter(java.io.OutputStream)
     */
    public PrintWriter(OutputStream out, boolean autoFlush) {
        this(new BufferedWriter(new OutputStreamWriter(out)), autoFlush);

        // save print stream for error propagation
        if (out instanceof java.io.PrintStream) {
            psOut = (PrintStream) out;
        }
    }

    /**
     * Creates a new PrintWriter, without automatic line flushing, with the
     * specified file name.  This convenience constructor creates the necessary
     * intermediate {@link java.io.OutputStreamWriter OutputStreamWriter},
     * which will encode characters using the {@linkplain
     * java.nio.charset.Charset#defaultCharset() default charset} for this
     * instance of the Java virtual machine.
     *
     * @param  fileName
     *         The name of the file to use as the destination of this writer.
     *         If the file exists then it will be truncated to zero size;
     *         otherwise, a new file will be created.  The output will be
     *         written to the file and is buffered.
     *
     * @throws  FileNotFoundException
     *          If the given string does not denote an existing, writable
     *          regular file and a new regular file of that name cannot be
     *          created, or if some other error occurs while opening or
     *          creating the file
     *
     * @throws  SecurityException
     *          If a security manager is present and {@link
     *          SecurityManager#checkWrite checkWrite(fileName)} denies write
     *          access to the file
     *
     * @since  1.5
     */
    public PrintWriter(String fileName) throws FileNotFoundException {
        this(new BufferedWriter(new OutputStreamWriter(new FileOutputStream(fileName))),
             false);
    }
```

```
/* Private constructor */
private PrintWriter(Charset charset, File file)
    throws FileNotFoundException
{
    this(new BufferedWriter(new OutputStreamWriter(new FileOutputStream(file), charset)),
         false);
}

/**
 * Creates a new PrintWriter, without automatic line flushing, with the
 * specified file name and charset.  This convenience constructor creates
 * the necessary intermediate {@link java.io.OutputStreamWriter
 * OutputStreamWriter}, which will encode characters using the provided
 * charset.
 *
 * @param  fileName
 *         The name of the file to use as the destination of this writer.
 *         If the file exists then it will be truncated to zero size;
 *         otherwise, a new file will be created.  The output will be
 *         written to the file and is buffered.
 *
 * @param  csn
 *         The name of a supported {@linkplain java.nio.charset.Charset
 *         charset}
 *
 * @throws  FileNotFoundException
 *          If the given string does not denote an existing, writable
 *          regular file and a new regular file of that name cannot be
 *          created, or if some other error occurs while opening or
 *          creating the file
 *
 * @throws  SecurityException
 *          If a security manager is present and {@link
 *          SecurityManager#checkWrite checkWrite(fileName)} denies write
 *          access to the file
 *
 * @throws  UnsupportedEncodingException
 *          If the named charset is not supported
 *
 * @since  1.5
 */
public PrintWriter(String fileName, String csn)
    throws FileNotFoundException, UnsupportedEncodingException
{
    this(toCharset(csn), new File(fileName));
}

/**
 * Creates a new PrintWriter, without automatic line flushing, with the
 * specified file.  This convenience constructor creates the necessary
 * intermediate {@link java.io.OutputStreamWriter OutputStreamWriter},
 * which will encode characters using the {@linkplain
 * java.nio.charset.Charset#defaultCharset() default charset} for this
 * instance of the Java virtual machine.
 *
 * @param  file
 *         The file to use as the destination of this writer.  If the file
 *         exists then it will be truncated to zero size; otherwise, a new
 *         file will be created.  The output will be written to the file
 *         and is buffered.
 *
 * @throws  FileNotFoundException
```

```
     *          If the given file object does not denote an existing, writable
     *          regular file and a new regular file of that name cannot be
     *          created, or if some other error occurs while opening or
     *          creating the file
     *
     * @throws  SecurityException
     *          If a security manager is present and {@link
     *          SecurityManager#checkWrite checkWrite(file.getPath())}
     *          denies write access to the file
     *
     * @since  1.5
     */
    public PrintWriter(File file) throws FileNotFoundException {
        this(new BufferedWriter(new OutputStreamWriter(new FileOutputStream(file))),
             false);
    }

    /**
     * Creates a new PrintWriter, without automatic line flushing, with the
     * specified file and charset.  This convenience constructor creates the
     * necessary intermediate {@link java.io.OutputStreamWriter
     * OutputStreamWriter}, which will encode characters using the provided
     * charset.
     *
     * @param  file
     *         The file to use as the destination of this writer.  If the file
     *         exists then it will be truncated to zero size; otherwise, a new
     *         file will be created.  The output will be written to the file
     *         and is buffered.
     *
     * @param  csn
     *         The name of a supported {@linkplain java.nio.charset.Charset
     *         charset}
     *
     * @throws  FileNotFoundException
     *          If the given file object does not denote an existing, writable
     *          regular file and a new regular file of that name cannot be
     *          created, or if some other error occurs while opening or
     *          creating the file
     *
     * @throws  SecurityException
     *          If a security manager is present and {@link
     *          SecurityManager#checkWrite checkWrite(file.getPath())}
     *          denies write access to the file
     *
     * @throws  UnsupportedEncodingException
     *          If the named charset is not supported
     *
     * @since  1.5
     */
    public PrintWriter(File file, String csn)
        throws FileNotFoundException, UnsupportedEncodingException
    {
        this(toCharset(csn), file);
    }

    /** Checks to make sure that the stream has not been closed */
    private void ensureOpen() throws IOException {
        if (out == null)
            throw new IOException("Stream closed");
    }

    /**
```

```
     * Flushes the stream.
     * @see #checkError()
     */
    public void flush() {
        try {
            synchronized (lock) {
                ensureOpen();
                out.flush();
            }
        }
        catch (IOException x) {
            trouble = true;
        }
    }

    /**
     * Closes the stream and releases any system resources associated
     * with it. Closing a previously closed stream has no effect.
     *
     * @see #checkError()
     */
    public void close() {
        try {
            synchronized (lock) {
                if (out == null)
                    return;
                out.close();
                out = null;
            }
        }
        catch (IOException x) {
            trouble = true;
        }
    }

    /**
     * Flushes the stream if it's not closed and checks its error state.
     *
     * @return <code>true</code> if the print stream has encountered an error,
     *          either on the underlying output stream or during a format
     *          conversion.
     */
    public boolean checkError() {
        if (out != null) {
            flush();
        }
        if (out instanceof java.io.PrintWriter) {
            PrintWriter pw = (PrintWriter) out;
            return pw.checkError();
        } else if (psOut != null) {
            return psOut.checkError();
        }
        return trouble;
    }

    /**
     * Indicates that an error has occurred.
     *
     * <p> This method will cause subsequent invocations of {@link
     * #checkError()} to return <tt>true</tt> until {@link
     * #clearError()} is invoked.
     */
    protected void setError() {
```

```
    trouble = true;
}

/**
 * Clears the error state of this stream.
 *
 * <p> This method will cause subsequent invocations of {@link
 * #checkError()} to return <tt>false</tt> until another write
 * operation fails and invokes {@link #setError()}.
 *
 * @since 1.6
 */
protected void clearError() {
    trouble = false;
}

/*
 * Exception-catching, synchronized output operations,
 * which also implement the write() methods of Writer
 */

/**
 * Writes a single character.
 * @param c int specifying a character to be written.
 */
public void write(int c) {
    try {
        synchronized (lock) {
            ensureOpen();
            out.write(c);
        }
    }
    catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    }
    catch (IOException x) {
        trouble = true;
    }
}

/**
 * Writes A Portion of an array of characters.
 * @param buf Array of characters
 * @param off Offset from which to start writing characters
 * @param len Number of characters to write
 */
public void write(char buf[], int off, int len) {
    try {
        synchronized (lock) {
            ensureOpen();
            out.write(buf, off, len);
        }
    }
    catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    }
    catch (IOException x) {
        trouble = true;
    }
}

/**
 * Writes an array of characters.  This method cannot be inherited from the
```

```
     * Writer class because it must suppress I/O exceptions.
     * @param buf Array of characters to be written
     */
    public void write(char buf[]) {
        write(buf, 0, buf.length);
    }

    /**
     * Writes a portion of a string.
     * @param s A String
     * @param off Offset from which to start writing characters
     * @param len Number of characters to write
     */
    public void write(String s, int off, int len) {
        try {
            synchronized (lock) {
                ensureOpen();
                out.write(s, off, len);
            }
        }
        catch (InterruptedIOException x) {
            Thread.currentThread().interrupt();
        }
        catch (IOException x) {
            trouble = true;
        }
    }

    /**
     * Writes a string.  This method cannot be inherited from the Writer class
     * because it must suppress I/O exceptions.
     * @param s String to be written
     */
    public void write(String s) {
        write(s, 0, s.length());
    }

    private void newLine() {
        try {
            synchronized (lock) {
                ensureOpen();
                out.write(lineSeparator);
                if (autoFlush)
                    out.flush();
            }
        }
        catch (InterruptedIOException x) {
            Thread.currentThread().interrupt();
        }
        catch (IOException x) {
            trouble = true;
        }
    }

    /* Methods that do not terminate lines */

    /**
     * Prints a boolean value.  The string produced by <code>{@link
     * java.lang.String#valueOf(boolean)}</code> is translated into bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the <code>{@link
     * #write(int)}</code> method.
     *
```

```
     * @param      b   The <code>boolean</code> to be printed
     */
    public void print(boolean b) {
        write(b ? "true" : "false");
    }

    /**
     * Prints a character.  The character is translated into one or more bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the <code>{@link
     * #write(int)}</code> method.
     *
     * @param      c   The <code>char</code> to be printed
     */
    public void print(char c) {
        write(c);
    }

    /**
     * Prints an integer.  The string produced by <code>{@link
     * java.lang.String#valueOf(int)}</code> is translated into bytes according
     * to the platform's default character encoding, and these bytes are
     * written in exactly the manner of the <code>{@link #write(int)}</code>
     * method.
     *
     * @param      i   The <code>int</code> to be printed
     * @see        java.lang.Integer#toString(int)
     */
    public void print(int i) {
        write(String.valueOf(i));
    }

    /**
     * Prints a long integer.  The string produced by <code>{@link
     * java.lang.String#valueOf(long)}</code> is translated into bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the <code>{@link #write(int)}</code>
     * method.
     *
     * @param      l   The <code>long</code> to be printed
     * @see        java.lang.Long#toString(long)
     */
    public void print(long l) {
        write(String.valueOf(l));
    }

    /**
     * Prints a floating-point number.  The string produced by <code>{@link
     * java.lang.String#valueOf(float)}</code> is translated into bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the <code>{@link #write(int)}</code>
     * method.
     *
     * @param      f   The <code>float</code> to be printed
     * @see        java.lang.Float#toString(float)
     */
    public void print(float f) {
        write(String.valueOf(f));
    }

    /**
     * Prints a double-precision floating-point number.  The string produced by
     * <code>{@link java.lang.String#valueOf(double)}</code> is translated into
```

```
     * bytes according to the platform's default character encoding, and these
     * bytes are written in exactly the manner of the <code>{@link
     * #write(int)}</code> method.
     *
     * @param      d   The <code>double</code> to be printed
     * @see        java.lang.Double#toString(double)
     */
    public void print(double d) {
        write(String.valueOf(d));
    }

    /**
     * Prints an array of characters.  The characters are converted into bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the <code>{@link #write(int)}</code>
     * method.
     *
     * @param      s   The array of chars to be printed
     *
     * @throws  NullPointerException  If <code>s</code> is <code>null</code>
     */
    public void print(char s[]) {
        write(s);
    }

    /**
     * Prints a string.  If the argument is <code>null</code> then the string
     * <code>"null"</code> is printed.  Otherwise, the string's characters are
     * converted into bytes according to the platform's default character
     * encoding, and these bytes are written in exactly the manner of the
     * <code>{@link #write(int)}</code> method.
     *
     * @param      s   The <code>String</code> to be printed
     */
    public void print(String s) {
        if (s == null) {
            s = "null";
        }
        write(s);
    }

    /**
     * Prints an object.  The string produced by the <code>{@link
     * java.lang.String#valueOf(Object)}</code> method is translated into bytes
     * according to the platform's default character encoding, and these bytes
     * are written in exactly the manner of the <code>{@link #write(int)}</code>
     * method.
     *
     * @param      obj   The <code>Object</code> to be printed
     * @see        java.lang.Object#toString()
     */
    public void print(Object obj) {
        write(String.valueOf(obj));
    }

    /* Methods that do terminate lines */

    /**
     * Terminates the current line by writing the line separator string.  The
     * line separator string is defined by the system property
     * <code>line.separator</code>, and is not necessarily a single newline
     * character (<code>'\n'</code>).
     */
```

```
public void println() {
    newLine();
}

/**
 * Prints a boolean value and then terminates the line.  This method behaves
 * as though it invokes <code>{@link #print(boolean)}</code> and then
 * <code>{@link #println()}</code>.
 *
 * @param x the <code>boolean</code> value to be printed
 */
public void println(boolean x) {
    synchronized (lock) {
        print(x);
        println();
    }
}

/**
 * Prints a character and then terminates the line.  This method behaves as
 * though it invokes <code>{@link #print(char)}</code> and then <code>{@link
 * #println()}</code>.
 *
 * @param x the <code>char</code> value to be printed
 */
public void println(char x) {
    synchronized (lock) {
        print(x);
        println();
    }
}

/**
 * Prints an integer and then terminates the line.  This method behaves as
 * though it invokes <code>{@link #print(int)}</code> and then <code>{@link
 * #println()}</code>.
 *
 * @param x the <code>int</code> value to be printed
 */
public void println(int x) {
    synchronized (lock) {
        print(x);
        println();
    }
}

/**
 * Prints a long integer and then terminates the line.  This method behaves
 * as though it invokes <code>{@link #print(long)}</code> and then
 * <code>{@link #println()}</code>.
 *
 * @param x the <code>long</code> value to be printed
 */
public void println(long x) {
    synchronized (lock) {
        print(x);
        println();
    }
}

/**
 * Prints a floating-point number and then terminates the line.  This method
 * behaves as though it invokes <code>{@link #print(float)}</code> and then
```

```
     * <code>{@link #println()}</code>.
     *
     * @param x the <code>float</code> value to be printed
     */
    public void println(float x) {
        synchronized (lock) {
            print(x);
            println();
        }
    }

    /**
     * Prints a double-precision floating-point number and then terminates the
     * line.  This method behaves as though it invokes <code>{@link
     * #print(double)}</code> and then <code>{@link #println()}</code>.
     *
     * @param x the <code>double</code> value to be printed
     */
    public void println(double x) {
        synchronized (lock) {
            print(x);
            println();
        }
    }

    /**
     * Prints an array of characters and then terminates the line.  This method
     * behaves as though it invokes <code>{@link #print(char[])}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x the array of <code>char</code> values to be printed
     */
    public void println(char x[]) {
        synchronized (lock) {
            print(x);
            println();
        }
    }

    /**
     * Prints a String and then terminates the line.  This method behaves as
     * though it invokes <code>{@link #print(String)}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x the <code>String</code> value to be printed
     */
    public void println(String x) {
        synchronized (lock) {
            print(x);
            println();
        }
    }

    /**
     * Prints an Object and then terminates the line.  This method calls
     * at first String.valueOf(x) to get the printed object's string value,
     * then behaves as
     * though it invokes <code>{@link #print(String)}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x  The <code>Object</code> to be printed.
     */
    public void println(Object x) {
```

```
    String s = String.valueOf(x);
    synchronized (lock) {
        print(s);
        println();
    }
}

/**
 * A convenience method to write a formatted string to this writer using
 * the specified format string and arguments.  If automatic flushing is
 * enabled, calls to this method will flush the output buffer.
 *
 * <p> An invocation of this method of the form <tt>out.printf(format,
 * args)</tt> behaves in exactly the same way as the invocation
 *
 * <pre>
 *     out.format(format, args) </pre>
 *
 * @param  format
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>.
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  java.util.IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a> section of the
 *          formatter class specification.
 *
 * @throws  NullPointerException
 *          If the <tt>format</tt> is <tt>null</tt>
 *
 * @return  This writer
 *
 * @since  1.5
 */
public PrintWriter printf(String format, Object ... args) {
    return format(format, args);
}

/**
 * A convenience method to write a formatted string to this writer using
 * the specified format string and arguments.  If automatic flushing is
 * enabled, calls to this method will flush the output buffer.
 *
 * <p> An invocation of this method of the form <tt>out.printf(l, format,
 * args)</tt> behaves in exactly the same way as the invocation
 *
 * <pre>
 *     out.format(l, format, args) </pre>
```

```
 *
 * @param  l
 *         The {@linkplain java.util.Locale locale} to apply during
 *         formatting.  If <tt>l</tt> is <tt>null</tt> then no localization
 *         is applied.
 *
 * @param  format
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>.
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  java.util.IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a> section of the
 *          formatter class specification.
 *
 * @throws  NullPointerException
 *          If the <tt>format</tt> is <tt>null</tt>
 *
 * @return  This writer
 *
 * @since  1.5
 */
public PrintWriter printf(Locale l, String format, Object ... args) {
    return format(l, format, args);
}

/**
 * Writes a formatted string to this writer using the specified format
 * string and arguments.  If automatic flushing is enabled, calls to this
 * method will flush the output buffer.
 *
 * <p> The locale always used is the one returned by {@link
 * java.util.Locale#getDefault() Locale.getDefault()}, regardless of any
 * previous invocations of other formatting methods on this object.
 *
 * @param  format
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>.
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
```

```
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  java.util.IllegalFormatException
 *          If a format string contains an illegal syntax, a format
 *          specifier that is incompatible with the given arguments,
 *          insufficient arguments given the format string, or other
 *          illegal conditions.  For specification of all possible
 *          formatting errors, see the <a
 *          href="../util/Formatter.html#detail">Details</a> section of the
 *          Formatter class specification.
 *
 * @throws  NullPointerException
 *          If the <tt>format</tt> is <tt>null</tt>
 *
 * @return  This writer
 *
 * @since  1.5
 */
public PrintWriter format(String format, Object ... args) {
    try {
        synchronized (lock) {
            ensureOpen();
            if ((formatter == null)
                || (formatter.locale() != Locale.getDefault()))
                formatter = new Formatter(this);
            formatter.format(Locale.getDefault(), format, args);
            if (autoFlush)
                out.flush();
        }
    } catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    } catch (IOException x) {
        trouble = true;
    }
    return this;
}

/**
 * Writes a formatted string to this writer using the specified format
 * string and arguments.  If automatic flushing is enabled, calls to this
 * method will flush the output buffer.
 *
 * @param  l
 *         The {@linkplain java.util.Locale locale} to apply during
 *         formatting.  If <tt>l</tt> is <tt>null</tt> then no localization
 *         is applied.
 *
 * @param  format
 *         A format string as described in <a
 *         href="../util/Formatter.html#syntax">Format string syntax</a>.
 *
 * @param  args
 *         Arguments referenced by the format specifiers in the format
 *         string.  If there are more arguments than format specifiers, the
 *         extra arguments are ignored.  The number of arguments is
 *         variable and may be zero.  The maximum number of arguments is
 *         limited by the maximum dimension of a Java array as defined by
 *         <cite>The Java™ Virtual Machine Specification</cite>.
 *         The behaviour on a
 *         <tt>null</tt> argument depends on the <a
 *         href="../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws  java.util.IllegalFormatException
```

```
     *          If a format string contains an illegal syntax, a format
     *          specifier that is incompatible with the given arguments,
     *          insufficient arguments given the format string, or other
     *          illegal conditions.  For specification of all possible
     *          formatting errors, see the <a
     *          href="../util/Formatter.html#detail">Details</a> section of the
     *          formatter class specification.
     *
     * @throws  NullPointerException
     *          If the <tt>format</tt> is <tt>null</tt>
     *
     * @return  This writer
     *
     * @since  1.5
     */
    public PrintWriter format(Locale l, String format, Object ... args) {
        try {
            synchronized (lock) {
                ensureOpen();
                if ((formatter == null) || (formatter.locale() != l))
                    formatter = new Formatter(this, l);
                formatter.format(l, format, args);
                if (autoFlush)
                    out.flush();
            }
        } catch (InterruptedIOException x) {
            Thread.currentThread().interrupt();
        } catch (IOException x) {
            trouble = true;
        }
        return this;
    }

    /**
     * Appends the specified character sequence to this writer.
     *
     * <p> An invocation of this method of the form <tt>out.append(csq)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.write(csq.toString()) </pre>
     *
     * <p> Depending on the specification of <tt>toString</tt> for the
     * character sequence <tt>csq</tt>, the entire sequence may not be
     * appended. For instance, invoking the <tt>toString</tt> method of a
     * character buffer will return a subsequence whose content depends upon
     * the buffer's position and limit.
     *
     * @param  csq
     *         The character sequence to append.  If <tt>csq</tt> is
     *         <tt>null</tt>, then the four characters <tt>"null"</tt> are
     *         appended to this writer.
     *
     * @return  This writer
     *
     * @since  1.5
     */
    public PrintWriter append(CharSequence csq) {
        if (csq == null)
            write("null");
        else
            write(csq.toString());
        return this;
```

```
    }

    /**
     * Appends a subsequence of the specified character sequence to this writer.
     *
     * <p> An invocation of this method of the form <tt>out.append(csq, start,
     * end)</tt> when <tt>csq</tt> is not <tt>null</tt>, behaves in
     * exactly the same way as the invocation
     *
     * <pre>
     *     out.write(csq.subSequence(start, end).toString()) </pre>
     *
     * @param  csq
     *         The character sequence from which a subsequence will be
     *         appended.  If <tt>csq</tt> is <tt>null</tt>, then characters
     *         will be appended as if <tt>csq</tt> contained the four
     *         characters <tt>"null"</tt>.
     *
     * @param  start
     *         The index of the first character in the subsequence
     *
     * @param  end
     *         The index of the character following the last character in the
     *         subsequence
     *
     * @return  This writer
     *
     * @throws  IndexOutOfBoundsException
     *          If <tt>start</tt> or <tt>end</tt> are negative, <tt>start</tt>
     *          is greater than <tt>end</tt>, or <tt>end</tt> is greater than
     *          <tt>csq.length()</tt>
     *
     * @since  1.5
     */
    public PrintWriter append(CharSequence csq, int start, int end) {
        CharSequence cs = (csq == null ? "null" : csq);
        write(cs.subSequence(start, end).toString());
        return this;
    }

    /**
     * Appends the specified character to this writer.
     *
     * <p> An invocation of this method of the form <tt>out.append(c)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.write(c) </pre>
     *
     * @param  c
     *         The 16-bit character to append
     *
     * @return  This writer
     *
     * @since 1.5
     */
    public PrintWriter append(char c) {
        write(c);
        return this;
    }
}
```

PushbackInputStream.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * A <code>PushbackInputStream</code> adds
 * functionality to another input stream, namely
 * the  ability to "push back" or "unread"
 * one byte. This is useful in situations where
 * it is  convenient for a fragment of code
 * to read an indefinite number of data bytes
 * that  are delimited by a particular byte
 * value; after reading the terminating byte,
 * the  code fragment can "unread" it, so that
 * the next read operation on the input stream
 * will reread the byte that was pushed back.
 * For example, bytes representing the  characters
 * constituting an identifier might be terminated
 * by a byte representing an  operator character;
 * a method whose job is to read just an identifier
 * can read until it  sees the operator and
 * then push the operator back to be re-read.
 *
 * @author  David Connelly
 * @author  Jonathan Payne
 * @since   JDK1.0
 */
public
class PushbackInputStream extends FilterInputStream {
    /**
     * The pushback buffer.
     * @since   JDK1.1
     */
    protected byte[] buf;

    /**
     * The position within the pushback buffer from which the next byte will
```

```
     * be read.  When the buffer is empty, <code>pos</code> is equal to
     * <code>buf.length</code>; when the buffer is full, <code>pos</code> is
     * equal to zero.
     *
     * @since   JDK1.1
     */
    protected int pos;

    /**
     * Check to make sure that this stream has not been closed
     */
    private void ensureOpen() throws IOException {
        if (in == null)
            throw new IOException("Stream closed");
    }

    /**
     * Creates a <code>PushbackInputStream</code>
     * with a pushback buffer of the specified <code>size</code>,
     * and saves its  argument, the input stream
     * <code>in</code>, for later use. Initially,
     * there is no pushed-back byte  (the field
     * <code>pushBack</code> is initialized to
     * <code>-1</code>).
     *
     * @param  in    the input stream from which bytes will be read.
     * @param  size  the size of the pushback buffer.
     * @exception IllegalArgumentException if {@code size <= 0}
     * @since  JDK1.1
     */
    public PushbackInputStream(InputStream in, int size) {
        super(in);
        if (size <= 0) {
            throw new IllegalArgumentException("size <= 0");
        }
        this.buf = new byte[size];
        this.pos = size;
    }

    /**
     * Creates a <code>PushbackInputStream</code>
     * and saves its  argument, the input stream
     * <code>in</code>, for later use. Initially,
     * there is no pushed-back byte  (the field
     * <code>pushBack</code> is initialized to
     * <code>-1</code>).
     *
     * @param   in   the input stream from which bytes will be read.
     */
    public PushbackInputStream(InputStream in) {
        this(in, 1);
    }

    /**
     * Reads the next byte of data from this input stream. The value
     * byte is returned as an <code>int</code> in the range
     * <code>0</code> to <code>255</code>. If no byte is available
     * because the end of the stream has been reached, the value
     * <code>-1</code> is returned. This method blocks until input data
     * is available, the end of the stream is detected, or an exception
     * is thrown.
     *
     * <p> This method returns the most recently pushed-back byte, if there is
```

```
     * one, and otherwise calls the <code>read</code> method of its underlying
     * input stream and returns whatever value that method returns.
     *
     * @return     the next byte of data, or <code>-1</code> if the end of the
     *             stream has been reached.
     * @exception  IOException  if this input stream has been closed by
     *             invoking its {@link #close()} method,
     *             or an I/O error occurs.
     * @see        java.io.InputStream#read()
     */
    public int read() throws IOException {
        ensureOpen();
        if (pos < buf.length) {
            return buf[pos++] & 0xff;
        }
        return super.read();
    }

    /**
     * Reads up to <code>len</code> bytes of data from this input stream into
     * an array of bytes.  This method first reads any pushed-back bytes; after
     * that, if fewer than <code>len</code> bytes have been read then it
     * reads from the underlying input stream. If <code>len</code> is not zero, the method
     * blocks until at least 1 byte of input is available; otherwise, no
     * bytes are read and <code>0</code> is returned.
     *
     * @param      b     the buffer into which the data is read.
     * @param      off   the start offset in the destination array <code>b</code>
     * @param      len   the maximum number of bytes read.
     * @return     the total number of bytes read into the buffer, or
     *             <code>-1</code> if there is no more data because the end of
     *             the stream has been reached.
     * @exception  NullPointerException If <code>b</code> is <code>null</code>.
     * @exception  IndexOutOfBoundsException If <code>off</code> is negative,
     * <code>len</code> is negative, or <code>len</code> is greater than
     * <code>b.length - off</code>
     * @exception  IOException  if this input stream has been closed by
     *             invoking its {@link #close()} method,
     *             or an I/O error occurs.
     * @see        java.io.InputStream#read(byte[], int, int)
     */
    public int read(byte[] b, int off, int len) throws IOException {
        ensureOpen();
        if (b == null) {
            throw new NullPointerException();
        } else if (off < 0 || len < 0 || len > b.length - off) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return 0;
        }

        int avail = buf.length - pos;
        if (avail > 0) {
            if (len < avail) {
                avail = len;
            }
            System.arraycopy(buf, pos, b, off, avail);
            pos += avail;
            off += avail;
            len -= avail;
        }
        if (len > 0) {
            len = super.read(b, off, len);
```

```
        if (len == -1) {
            return avail == 0 ? -1 : avail;
        }
        return avail + len;
    }
    return avail;
}

/**
 * Pushes back a byte by copying it to the front of the pushback buffer.
 * After this method returns, the next byte to be read will have the value
 * <code>(byte)b</code>.
 *
 * @param      b   the <code>int</code> value whose low-order
 *                 byte is to be pushed back.
 * @exception IOException If there is not enough room in the pushback
 *            buffer for the byte, or this input stream has been closed by
 *            invoking its {@link #close()} method.
 */
public void unread(int b) throws IOException {
    ensureOpen();
    if (pos == 0) {
        throw new IOException("Push back buffer is full");
    }
    buf[--pos] = (byte)b;
}

/**
 * Pushes back a portion of an array of bytes by copying it to the front
 * of the pushback buffer.  After this method returns, the next byte to be
 * read will have the value <code>b[off]</code>, the byte after that will
 * have the value <code>b[off+1]</code>, and so forth.
 *
 * @param b the byte array to push back.
 * @param off the start offset of the data.
 * @param len the number of bytes to push back.
 * @exception IOException If there is not enough room in the pushback
 *            buffer for the specified number of bytes,
 *            or this input stream has been closed by
 *            invoking its {@link #close()} method.
 * @since     JDK1.1
 */
public void unread(byte[] b, int off, int len) throws IOException {
    ensureOpen();
    if (len > pos) {
        throw new IOException("Push back buffer is full");
    }
    pos -= len;
    System.arraycopy(b, off, buf, pos, len);
}

/**
 * Pushes back an array of bytes by copying it to the front of the
 * pushback buffer.  After this method returns, the next byte to be read
 * will have the value <code>b[0]</code>, the byte after that will have the
 * value <code>b[1]</code>, and so forth.
 *
 * @param b the byte array to push back
 * @exception IOException If there is not enough room in the pushback
 *            buffer for the specified number of bytes,
 *            or this input stream has been closed by
 *            invoking its {@link #close()} method.
 * @since     JDK1.1
```

```
 */
public void unread(byte[] b) throws IOException {
    unread(b, 0, b.length);
}

/**
 * Returns an estimate of the number of bytes that can be read (or
 * skipped over) from this input stream without blocking by the next
 * invocation of a method for this input stream. The next invocation might be
 * the same thread or another thread.  A single read or skip of this
 * many bytes will not block, but may read or skip fewer bytes.
 *
 * <p> The method returns the sum of the number of bytes that have been
 * pushed back and the value returned by {@link
 * java.io.FilterInputStream#available available}.
 *
 * @return     the number of bytes that can be read (or skipped over) from
 *             the input stream without blocking.
 * @exception  IOException  if this input stream has been closed by
 *             invoking its {@link #close()} method,
 *             or an I/O error occurs.
 * @see        java.io.FilterInputStream#in
 * @see        java.io.InputStream#available()
 */
public int available() throws IOException {
    ensureOpen();
    int n = buf.length - pos;
    int avail = super.available();
    return n > (Integer.MAX_VALUE - avail)
                ? Integer.MAX_VALUE
                : n + avail;
}

/**
 * Skips over and discards <code>n</code> bytes of data from this
 * input stream. The <code>skip</code> method may, for a variety of
 * reasons, end up skipping over some smaller number of bytes,
 * possibly zero.  If <code>n</code> is negative, no bytes are skipped.
 *
 * <p> The <code>skip</code> method of <code>PushbackInputStream</code>
 * first skips over the bytes in the pushback buffer, if any.  It then
 * calls the <code>skip</code> method of the underlying input stream if
 * more bytes need to be skipped.  The actual number of bytes skipped
 * is returned.
 *
 * @param      n  {@inheritDoc}
 * @return     {@inheritDoc}
 * @exception  IOException  if the stream does not support seek,
 *             or the stream has been closed by
 *             invoking its {@link #close()} method,
 *             or an I/O error occurs.
 * @see        java.io.FilterInputStream#in
 * @see        java.io.InputStream#skip(long n)
 * @since      1.2
 */
public long skip(long n) throws IOException {
    ensureOpen();
    if (n <= 0) {
        return 0;
    }

    long pskip = buf.length - pos;
    if (pskip > 0) {
```

```
        if (n < pskip) {
            pskip = n;
        }
        pos += pskip;
        n -= pskip;
    }
    if (n > 0) {
        pskip += super.skip(n);
    }
    return pskip;
}

/**
 * Tests if this input stream supports the <code>mark</code> and
 * <code>reset</code> methods, which it does not.
 *
 * @return   <code>false</code>, since this class does not support the
 *           <code>mark</code> and <code>reset</code> methods.
 * @see     java.io.InputStream#mark(int)
 * @see     java.io.InputStream#reset()
 */
public boolean markSupported() {
    return false;
}

/**
 * Marks the current position in this input stream.
 *
 * <p> The <code>mark</code> method of <code>PushbackInputStream</code>
 * does nothing.
 *
 * @param   readlimit   the maximum limit of bytes that can be read before
 *                      the mark position becomes invalid.
 * @see     java.io.InputStream#reset()
 */
public synchronized void mark(int readlimit) {
}

/**
 * Repositions this stream to the position at the time the
 * <code>mark</code> method was last called on this input stream.
 *
 * <p> The method <code>reset</code> for class
 * <code>PushbackInputStream</code> does nothing except throw an
 * <code>IOException</code>.
 *
 * @exception  IOException  if this method is invoked.
 * @see     java.io.InputStream#mark(int)
 * @see     java.io.IOException
 */
public synchronized void reset() throws IOException {
    throw new IOException("mark/reset not supported");
}

/**
 * Closes this input stream and releases any system resources
 * associated with the stream.
 * Once the stream has been closed, further read(), unread(),
 * available(), reset(), or skip() invocations will throw an IOException.
 * Closing a previously closed stream has no effect.
 *
 * @exception  IOException  if an I/O error occurs.
 */
```

```
    public synchronized void close() throws IOException {
        if (in == null)
            return;
        in.close();
        in = null;
        buf = null;
    }
}
```

PushbackReader.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * A character-stream reader that allows characters to be pushed back into the
 * stream.
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class PushbackReader extends FilterReader {

    /** Pushback buffer */
    private char[] buf;

    /** Current position in buffer */
    private int pos;

    /**
     * Creates a new pushback reader with a pushback buffer of the given size.
     *
     * @param   in   The reader from which characters will be read
     * @param   size The size of the pushback buffer
     * @exception IllegalArgumentException if {@code size <= 0}
     */
    public PushbackReader(Reader in, int size) {
        super(in);
        if (size <= 0) {
            throw new IllegalArgumentException("size <= 0");
        }
        this.buf = new char[size];
        this.pos = size;
    }
```

```
/**
 * Creates a new pushback reader with a one-character pushback buffer.
 *
 * @param   in  The reader from which characters will be read
 */
public PushbackReader(Reader in) {
    this(in, 1);
}

/** Checks to make sure that the stream has not been closed. */
private void ensureOpen() throws IOException {
    if (buf == null)
        throw new IOException("Stream closed");
}

/**
 * Reads a single character.
 *
 * @return     The character read, or -1 if the end of the stream has been
 *             reached
 *
 * @exception  IOException  If an I/O error occurs
 */
public int read() throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (pos < buf.length)
            return buf[pos++];
        else
            return super.read();
    }
}

/**
 * Reads characters into a portion of an array.
 *
 * @param      cbuf  Destination buffer
 * @param      off   Offset at which to start writing characters
 * @param      len   Maximum number of characters to read
 *
 * @return     The number of characters read, or -1 if the end of the
 *             stream has been reached
 *
 * @exception  IOException  If an I/O error occurs
 */
public int read(char cbuf[], int off, int len) throws IOException {
    synchronized (lock) {
        ensureOpen();
        try {
            if (len <= 0) {
                if (len < 0) {
                    throw new IndexOutOfBoundsException();
                } else if ((off < 0) || (off > cbuf.length)) {
                    throw new IndexOutOfBoundsException();
                }
                return 0;
            }
            int avail = buf.length - pos;
            if (avail > 0) {
                if (len < avail)
                    avail = len;
                System.arraycopy(buf, pos, cbuf, off, avail);
                pos += avail;
```

```
                off += avail;
                len -= avail;
            }
            if (len > 0) {
                len = super.read(cbuf, off, len);
                if (len == -1) {
                    return (avail == 0) ? -1 : avail;
                }
                return avail + len;
            }
            return avail;
        } catch (ArrayIndexOutOfBoundsException e) {
            throw new IndexOutOfBoundsException();
        }
    }
}

/**
 * Pushes back a single character by copying it to the front of the
 * pushback buffer. After this method returns, the next character to be read
 * will have the value <code>(char)c</code>.
 *
 * @param  c  The int value representing a character to be pushed back
 *
 * @exception  IOException  If the pushback buffer is full,
 *                          or if some other I/O error occurs
 */
public void unread(int c) throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (pos == 0)
            throw new IOException("Pushback buffer overflow");
        buf[--pos] = (char) c;
    }
}

/**
 * Pushes back a portion of an array of characters by copying it to the
 * front of the pushback buffer.  After this method returns, the next
 * character to be read will have the value <code>cbuf[off]</code>, the
 * character after that will have the value <code>cbuf[off+1]</code>, and
 * so forth.
 *
 * @param  cbuf  Character array
 * @param  off   Offset of first character to push back
 * @param  len   Number of characters to push back
 *
 * @exception  IOException  If there is insufficient room in the pushback
 *                          buffer, or if some other I/O error occurs
 */
public void unread(char cbuf[], int off, int len) throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (len > pos)
            throw new IOException("Pushback buffer overflow");
        pos -= len;
        System.arraycopy(cbuf, off, buf, pos, len);
    }
}

/**
 * Pushes back an array of characters by copying it to the front of the
 * pushback buffer.  After this method returns, the next character to be
```

```
     * read will have the value <code>cbuf[0]</code>, the character after that
     * will have the value <code>cbuf[1]</code>, and so forth.
     *
     * @param  cbuf  Character array to push back
     *
     * @exception  IOException  If there is insufficient room in the pushback
     *                          buffer, or if some other I/O error occurs
     */
    public void unread(char cbuf[]) throws IOException {
        unread(cbuf, 0, cbuf.length);
    }

    /**
     * Tells whether this stream is ready to be read.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public boolean ready() throws IOException {
        synchronized (lock) {
            ensureOpen();
            return (pos < buf.length) || super.ready();
        }
    }

    /**
     * Marks the present position in the stream. The <code>mark</code>
     * for class <code>PushbackReader</code> always throws an exception.
     *
     * @exception  IOException  Always, since mark is not supported
     */
    public void mark(int readAheadLimit) throws IOException {
        throw new IOException("mark/reset not supported");
    }

    /**
     * Resets the stream. The <code>reset</code> method of
     * <code>PushbackReader</code> always throws an exception.
     *
     * @exception  IOException  Always, since reset is not supported
     */
    public void reset() throws IOException {
        throw new IOException("mark/reset not supported");
    }

    /**
     * Tells whether this stream supports the mark() operation, which it does
     * not.
     */
    public boolean markSupported() {
        return false;
    }

    /**
     * Closes the stream and releases any system resources associated with
     * it. Once the stream has been closed, further read(),
     * unread(), ready(), or skip() invocations will throw an IOException.
     * Closing a previously closed stream has no effect.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public void close() throws IOException {
        super.close();
        buf = null;
```

```
    }

    /**
     * Skips characters.  This method will block until some characters are
     * available, an I/O error occurs, or the end of the stream is reached.
     *
     * @param  n  The number of characters to skip
     *
     * @return    The number of characters actually skipped
     *
     * @exception  IllegalArgumentException  If <code>n</code> is negative.
     * @exception  IOException  If an I/O error occurs
     */
    public long skip(long n) throws IOException {
        if (n < 0L)
            throw new IllegalArgumentException("skip value is negative");
        synchronized (lock) {
            ensureOpen();
            int avail = buf.length - pos;
            if (avail > 0) {
                if (n <= avail) {
                    pos += n;
                    return n;
                } else {
                    pos = buf.length;
                    n -= avail;
                }
            }
            return avail + super.skip(n);
        }
    }
}
```

RandomAccessFile.java

```
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.nio.channels.FileChannel;
import sun.nio.ch.FileChannelImpl;


/**
 * Instances of this class support both reading and writing to a
 * random access file. A random access file behaves like a large
 * array of bytes stored in the file system. There is a kind of cursor,
 * or index into the implied array, called the <em>file pointer</em>;
 * input operations read bytes starting at the file pointer and advance
 * the file pointer past the bytes read. If the random access file is
 * created in read/write mode, then output operations are also available;
 * output operations write bytes starting at the file pointer and advance
 * the file pointer past the bytes written. Output operations that write
 * past the current end of the implied array cause the array to be
 * extended. The file pointer can be read by the
 * {@code getFilePointer} method and set by the {@code seek}
 * method.
 * <p>
 * It is generally true of all the reading routines in this class that
 * if end-of-file is reached before the desired number of bytes has been
 * read, an {@code EOFException} (which is a kind of
 * {@code IOException}) is thrown. If any byte cannot be read for
 * any reason other than end-of-file, an {@code IOException} other
 * than {@code EOFException} is thrown. In particular, an
 * {@code IOException} may be thrown if the stream has been closed.
 *
 * @author  unascribed
 * @since   JDK1.0
 */

public class RandomAccessFile implements DataOutput, DataInput, Closeable {
```

```
private FileDescriptor fd;
private FileChannel channel = null;
private boolean rw;

/**
 * The path of the referenced file
 * (null if the stream is created with a file descriptor)
 */
private final String path;

private Object closeLock = new Object();
private volatile boolean closed = false;

private static final int O_RDONLY = 1;
private static final int O_RDWR =   2;
private static final int O_SYNC =   4;
private static final int O_DSYNC =  8;

/**
 * Creates a random access file stream to read from, and optionally
 * to write to, a file with the specified name. A new
 * {@link FileDescriptor} object is created to represent the
 * connection to the file.
 *
 * <p> The <tt>mode</tt> argument specifies the access mode with which the
 * file is to be opened.  The permitted values and their meanings are as
 * specified for the <a
 * href="#mode"><tt>RandomAccessFile(File,String)</tt></a> constructor.
 *
 * <p>
 * If there is a security manager, its {@code checkRead} method
 * is called with the {@code name} argument
 * as its argument to see if read access to the file is allowed.
 * If the mode allows writing, the security manager's
 * {@code checkWrite} method
 * is also called with the {@code name} argument
 * as its argument to see if write access to the file is allowed.
 *
 * @param      name   the system-dependent filename
 * @param      mode   the access <a href="#mode">mode</a>
 * @exception  IllegalArgumentException  if the mode argument is not equal
 *               to one of <tt>"r"</tt>, <tt>"rw"</tt>, <tt>"rws"</tt>, or
 *               <tt>"rwd"</tt>
 * @exception FileNotFoundException
 *            if the mode is <tt>"r"</tt> but the given string does not
 *            denote an existing regular file, or if the mode begins with
 *            <tt>"rw"</tt> but the given string does not denote an
 *            existing, writable regular file and a new regular file of
 *            that name cannot be created, or if some other error occurs
 *            while opening or creating the file
 * @exception  SecurityException         if a security manager exists and its
 *               {@code checkRead} method denies read access to the file
 *               or the mode is "rw" and the security manager's
 *               {@code checkWrite} method denies write access to the file
 * @see        java.lang.SecurityException
 * @see        java.lang.SecurityManager#checkRead(java.lang.String)
 * @see        java.lang.SecurityManager#checkWrite(java.lang.String)
 * @revised 1.4
 * @spec JSR-51
 */
public RandomAccessFile(String name, String mode)
    throws FileNotFoundException
{
```

```
    this(name != null ? new File(name) : null, mode);
}

/**
 * Creates a random access file stream to read from, and optionally to
 * write to, the file specified by the {@link File} argument.  A new {@link
 * FileDescriptor} object is created to represent this file connection.
 *
 * <p>The <a name="mode"><tt>mode</tt></a> argument specifies the access mode
 * in which the file is to be opened.  The permitted values and their
 * meanings are:
 *
 * <table summary="Access mode permitted values and meanings">
 * <tr><th align="left">Value</th><th align="left">Meaning</th></tr>
 * <tr><td valign="top"><tt>"r"</tt></td>
 *     <td> Open for reading only.  Invoking any of the <tt>write</tt>
 *     methods of the resulting object will cause an {@link
 *     java.io.IOException} to be thrown. </td></tr>
 * <tr><td valign="top"><tt>"rw"</tt></td>
 *     <td> Open for reading and writing.  If the file does not already
 *     exist then an attempt will be made to create it. </td></tr>
 * <tr><td valign="top"><tt>"rws"</tt></td>
 *     <td> Open for reading and writing, as with <tt>"rw"</tt>, and also
 *     require that every update to the file's content or metadata be
 *     written synchronously to the underlying storage device.  </td></tr>
 * <tr><td valign="top"><tt>"rwd"  </tt></td>
 *     <td> Open for reading and writing, as with <tt>"rw"</tt>, and also
 *     require that every update to the file's content be written
 *     synchronously to the underlying storage device. </td></tr>
 * </table>
 *
 * The <tt>"rws"</tt> and <tt>"rwd"</tt> modes work much like the {@link
 * java.nio.channels.FileChannel#force(boolean) force(boolean)} method of
 * the {@link java.nio.channels.FileChannel} class, passing arguments of
 * <tt>true</tt> and <tt>false</tt>, respectively, except that they always
 * apply to every I/O operation and are therefore often more efficient.  If
 * the file resides on a local storage device then when an invocation of a
 * method of this class returns it is guaranteed that all changes made to
 * the file by that invocation will have been written to that device.  This
 * is useful for ensuring that critical information is not lost in the
 * event of a system crash.  If the file does not reside on a local device
 * then no such guarantee is made.
 *
 * <p>The <tt>"rwd"</tt> mode can be used to reduce the number of I/O
 * operations performed.  Using <tt>"rwd"</tt> only requires updates to the
 * file's content to be written to storage; using <tt>"rws"</tt> requires
 * updates to both the file's content and its metadata to be written, which
 * generally requires at least one more low-level I/O operation.
 *
 * <p>If there is a security manager, its {@code checkRead} method is
 * called with the pathname of the {@code file} argument as its
 * argument to see if read access to the file is allowed.  If the mode
 * allows writing, the security manager's {@code checkWrite} method is
 * also called with the path argument to see if write access to the file is
 * allowed.
 *
 * @param      file   the file object
 * @param      mode   the access mode, as described
 *                    <a href="#mode">above</a>
 * @exception  IllegalArgumentException  if the mode argument is not equal
 *               to one of <tt>"r"</tt>, <tt>"rw"</tt>, <tt>"rws"</tt>, or
 *               <tt>"rwd"</tt>
 * @exception FileNotFoundException
```

```
     *            if the mode is <tt>"r"</tt> but the given file object does
     *            not denote an existing regular file, or if the mode begins
     *            with <tt>"rw"</tt> but the given file object does not denote
     *            an existing, writable regular file and a new regular file of
     *            that name cannot be created, or if some other error occurs
     *            while opening or creating the file
     * @exception  SecurityException         if a security manager exists and its
     *               {@code checkRead} method denies read access to the file
     *               or the mode is "rw" and the security manager's
     *               {@code checkWrite} method denies write access to the file
     * @see        java.lang.SecurityManager#checkRead(java.lang.String)
     * @see        java.lang.SecurityManager#checkWrite(java.lang.String)
     * @see        java.nio.channels.FileChannel#force(boolean)
     * @revised 1.4
     * @spec JSR-51
     */
    public RandomAccessFile(File file, String mode)
        throws FileNotFoundException
    {
        String name = (file != null ? file.getPath() : null);
        int imode = -1;
        if (mode.equals("r"))
            imode = O_RDONLY;
        else if (mode.startsWith("rw")) {
            imode = O_RDWR;
            rw = true;
            if (mode.length() > 2) {
                if (mode.equals("rws"))
                    imode |= O_SYNC;
                else if (mode.equals("rwd"))
                    imode |= O_DSYNC;
                else
                    imode = -1;
            }
        }
        if (imode < 0)
            throw new IllegalArgumentException("Illegal mode \"" + mode
                                               + "\" must be one of "
                                               + "\"r\", \"rw\", \"rws\","
                                               + " or \"rwd\"");
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkRead(name);
            if (rw) {
                security.checkWrite(name);
            }
        }
        if (name == null) {
            throw new NullPointerException();
        }
        if (file.isInvalid()) {
            throw new FileNotFoundException("Invalid file path");
        }
        fd = new FileDescriptor();
        fd.attach(this);
        path = name;
        open(name, imode);
    }

    /**
     * Returns the opaque file descriptor object associated with this
     * stream.
     *
```

```
     * @return     the file descriptor object associated with this stream.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FileDescriptor
     */
    public final FileDescriptor getFD() throws IOException {
        if (fd != null) {
            return fd;
        }
        throw new IOException();
    }

    /**
     * Returns the unique {@link java.nio.channels.FileChannel FileChannel}
     * object associated with this file.
     *
     * <p> The {@link java.nio.channels.FileChannel#position()
     * position} of the returned channel will always be equal to
     * this object's file-pointer offset as returned by the {@link
     * #getFilePointer getFilePointer} method.  Changing this object's
     * file-pointer offset, whether explicitly or by reading or writing bytes,
     * will change the position of the channel, and vice versa.  Changing the
     * file's length via this object will change the length seen via the file
     * channel, and vice versa.
     *
     * @return  the file channel associated with this file
     *
     * @since 1.4
     * @spec JSR-51
     */
    public final FileChannel getChannel() {
        synchronized (this) {
            if (channel == null) {
                channel = FileChannelImpl.open(fd, path, true, rw, this);
            }
            return channel;
        }
    }

    /**
     * Opens a file and returns the file descriptor.  The file is
     * opened in read-write mode if the O_RDWR bit in {@code mode}
     * is true, else the file is opened as read-only.
     * If the {@code name} refers to a directory, an IOException
     * is thrown.
     *
     * @param name the name of the file
     * @param mode the mode flags, a combination of the O_ constants
     *             defined above
     */
    private native void open(String name, int mode)
        throws FileNotFoundException;

    // 'Read' primitives

    /**
     * Reads a byte of data from this file. The byte is returned as an
     * integer in the range 0 to 255 ({@code 0x00-0x0ff}). This
     * method blocks if no input is yet available.
     * <p>
     * Although {@code RandomAccessFile} is not a subclass of
     * {@code InputStream}, this method behaves in exactly the same
     * way as the {@link InputStream#read()} method of
     * {@code InputStream}.
```

```
     *
     * @return     the next byte of data, or {@code -1} if the end of the
     *             file has been reached.
     * @exception  IOException  if an I/O error occurs. Not thrown if
     *                          end-of-file has been reached.
     */
    public int read() throws IOException {
        return read0();
    }

    private native int read0() throws IOException;

    /**
     * Reads a sub array as a sequence of bytes.
     * @param b the buffer into which the data is read.
     * @param off the start offset of the data.
     * @param len the number of bytes to read.
     * @exception IOException If an I/O error has occurred.
     */
    private native int readBytes(byte b[], int off, int len) throws IOException;

    /**
     * Reads up to {@code len} bytes of data from this file into an
     * array of bytes. This method blocks until at least one byte of input
     * is available.
     * <p>
     * Although {@code RandomAccessFile} is not a subclass of
     * {@code InputStream}, this method behaves in exactly the
     * same way as the {@link InputStream#read(byte[], int, int)} method of
     * {@code InputStream}.
     *
     * @param      b     the buffer into which the data is read.
     * @param      off   the start offset in array {@code b}
     *                   at which the data is written.
     * @param      len   the maximum number of bytes read.
     * @return     the total number of bytes read into the buffer, or
     *             {@code -1} if there is no more data because the end of
     *             the file has been reached.
     * @exception  IOException If the first byte cannot be read for any reason
     * other than end of file, or if the random access file has been closed, or if
     * some other I/O error occurs.
     * @exception  NullPointerException If {@code b} is {@code null}.
     * @exception  IndexOutOfBoundsException If {@code off} is negative,
     * {@code len} is negative, or {@code len} is greater than
     * {@code b.length - off}
     */
    public int read(byte b[], int off, int len) throws IOException {
        return readBytes(b, off, len);
    }

    /**
     * Reads up to {@code b.length} bytes of data from this file
     * into an array of bytes. This method blocks until at least one byte
     * of input is available.
     * <p>
     * Although {@code RandomAccessFile} is not a subclass of
     * {@code InputStream}, this method behaves in exactly the
     * same way as the {@link InputStream#read(byte[])} method of
     * {@code InputStream}.
     *
     * @param      b   the buffer into which the data is read.
     * @return     the total number of bytes read into the buffer, or
     *             {@code -1} if there is no more data because the end of
```

```
 *             this file has been reached.
 * @exception  IOException If the first byte cannot be read for any reason
 * other than end of file, or if the random access file has been closed, or if
 * some other I/O error occurs.
 * @exception  NullPointerException If {@code b} is {@code null}.
 */
public int read(byte b[]) throws IOException {
    return readBytes(b, 0, b.length);
}

/**
 * Reads {@code b.length} bytes from this file into the byte
 * array, starting at the current file pointer. This method reads
 * repeatedly from the file until the requested number of bytes are
 * read. This method blocks until the requested number of bytes are
 * read, the end of the stream is detected, or an exception is thrown.
 *
 * @param      b   the buffer into which the data is read.
 * @exception  EOFException  if this file reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
public final void readFully(byte b[]) throws IOException {
    readFully(b, 0, b.length);
}

/**
 * Reads exactly {@code len} bytes from this file into the byte
 * array, starting at the current file pointer. This method reads
 * repeatedly from the file until the requested number of bytes are
 * read. This method blocks until the requested number of bytes are
 * read, the end of the stream is detected, or an exception is thrown.
 *
 * @param      b     the buffer into which the data is read.
 * @param      off   the start offset of the data.
 * @param      len   the number of bytes to read.
 * @exception  EOFException  if this file reaches the end before reading
 *               all the bytes.
 * @exception  IOException   if an I/O error occurs.
 */
public final void readFully(byte b[], int off, int len) throws IOException {
    int n = 0;
    do {
        int count = this.read(b, off + n, len - n);
        if (count < 0)
            throw new EOFException();
        n += count;
    } while (n < len);
}

/**
 * Attempts to skip over {@code n} bytes of input discarding the
 * skipped bytes.
 * <p>
 *
 * This method may skip over some smaller number of bytes, possibly zero.
 * This may result from any of a number of conditions; reaching end of
 * file before {@code n} bytes have been skipped is only one
 * possibility. This method never throws an {@code EOFException}.
 * The actual number of bytes skipped is returned.  If {@code n}
 * is negative, no bytes are skipped.
 *
 * @param      n   the number of bytes to be skipped.
```

```
     * @return     the actual number of bytes skipped.
     * @exception  IOException  if an I/O error occurs.
     */
    public int skipBytes(int n) throws IOException {
        long pos;
        long len;
        long newpos;

        if (n <= 0) {
            return 0;
        }
        pos = getFilePointer();
        len = length();
        newpos = pos + n;
        if (newpos > len) {
            newpos = len;
        }
        seek(newpos);

        /* return the actual number of bytes skipped */
        return (int) (newpos - pos);
    }

    // 'Write' primitives

    /**
     * Writes the specified byte to this file. The write starts at
     * the current file pointer.
     *
     * @param      b   the {@code byte} to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public void write(int b) throws IOException {
        write0(b);
    }

    private native void write0(int b) throws IOException;

    /**
     * Writes a sub array as a sequence of bytes.
     * @param b the data to be written
     * @param off the start offset in the data
     * @param len the number of bytes that are written
     * @exception IOException If an I/O error has occurred.
     */
    private native void writeBytes(byte b[], int off, int len) throws IOException;

    /**
     * Writes {@code b.length} bytes from the specified byte array
     * to this file, starting at the current file pointer.
     *
     * @param      b   the data.
     * @exception  IOException  if an I/O error occurs.
     */
    public void write(byte b[]) throws IOException {
        writeBytes(b, 0, b.length);
    }

    /**
     * Writes {@code len} bytes from the specified byte array
     * starting at offset {@code off} to this file.
     *
     * @param      b     the data.
```

```
     * @param      off   the start offset in the data.
     * @param      len   the number of bytes to write.
     * @exception  IOException  if an I/O error occurs.
     */
    public void write(byte b[], int off, int len) throws IOException {
        writeBytes(b, off, len);
    }

    // 'Random access' stuff

    /**
     * Returns the current offset in this file.
     *
     * @return     the offset from the beginning of the file, in bytes,
     *             at which the next read or write occurs.
     * @exception  IOException  if an I/O error occurs.
     */
    public native long getFilePointer() throws IOException;

    /**
     * Sets the file-pointer offset, measured from the beginning of this
     * file, at which the next read or write occurs.  The offset may be
     * set beyond the end of the file. Setting the offset beyond the end
     * of the file does not change the file length.  The file length will
     * change only by writing after the offset has been set beyond the end
     * of the file.
     *
     * @param      pos   the offset position, measured in bytes from the
     *                   beginning of the file, at which to set the file
     *                   pointer.
     * @exception  IOException  if {@code pos} is less than
     *                          {@code 0} or if an I/O error occurs.
     */
    public void seek(long pos) throws IOException {
        if (pos < 0) {
            throw new IOException("Negative seek offset");
        } else {
            seek0(pos);
        }
    }

    private native void seek0(long pos) throws IOException;

    /**
     * Returns the length of this file.
     *
     * @return     the length of this file, measured in bytes.
     * @exception  IOException  if an I/O error occurs.
     */
    public native long length() throws IOException;

    /**
     * Sets the length of this file.
     *
     * <p> If the present length of the file as returned by the
     * {@code length} method is greater than the {@code newLength}
     * argument then the file will be truncated.  In this case, if the file
     * offset as returned by the {@code getFilePointer} method is greater
     * than {@code newLength} then after this method returns the offset
     * will be equal to {@code newLength}.
     *
     * <p> If the present length of the file as returned by the
     * {@code length} method is smaller than the {@code newLength}
```

```
     * argument then the file will be extended.  In this case, the contents of
     * the extended portion of the file are not defined.
     *
     * @param      newLength    The desired length of the file
     * @exception  IOException  If an I/O error occurs
     * @since      1.2
     */
    public native void setLength(long newLength) throws IOException;

    /**
     * Closes this random access file stream and releases any system
     * resources associated with the stream. A closed random access
     * file cannot perform input or output operations and cannot be
     * reopened.
     *
     * <p> If this file has an associated channel then the channel is closed
     * as well.
     *
     * @exception  IOException  if an I/O error occurs.
     *
     * @revised 1.4
     * @spec JSR-51
     */
    public void close() throws IOException {
        synchronized (closeLock) {
            if (closed) {
                return;
            }
            closed = true;
        }
        if (channel != null) {
            channel.close();
        }

        fd.closeAll(new Closeable() {
            public void close() throws IOException {
               close0();
           }
        });
    }

    //
    //  Some "reading/writing Java data types" methods stolen from
    //  DataInputStream and DataOutputStream.
    //

    /**
     * Reads a {@code boolean} from this file. This method reads a
     * single byte from the file, starting at the current file pointer.
     * A value of {@code 0} represents
     * {@code false}. Any other value represents {@code true}.
     * This method blocks until the byte is read, the end of the stream
     * is detected, or an exception is thrown.
     *
     * @return     the {@code boolean} value read.
     * @exception  EOFException  if this file has reached the end.
     * @exception  IOException   if an I/O error occurs.
     */
    public final boolean readBoolean() throws IOException {
        int ch = this.read();
        if (ch < 0)
            throw new EOFException();
        return (ch != 0);
```

```
}

/**
 * Reads a signed eight-bit value from this file. This method reads a
 * byte from the file, starting from the current file pointer.
 * If the byte read is {@code b}, where
 * <code>0 <= b <= 255</code>,
 * then the result is:
 * <blockquote><pre>
 *     (byte)(b)
 * </pre></blockquote>
 * <p>
 * This method blocks until the byte is read, the end of the stream
 * is detected, or an exception is thrown.
 *
 * @return     the next byte of this file as a signed eight-bit
 *             {@code byte}.
 * @exception  EOFException  if this file has reached the end.
 * @exception  IOException   if an I/O error occurs.
 */
public final byte readByte() throws IOException {
    int ch = this.read();
    if (ch < 0)
        throw new EOFException();
    return (byte)(ch);
}

/**
 * Reads an unsigned eight-bit number from this file. This method reads
 * a byte from this file, starting at the current file pointer,
 * and returns that byte.
 * <p>
 * This method blocks until the byte is read, the end of the stream
 * is detected, or an exception is thrown.
 *
 * @return     the next byte of this file, interpreted as an unsigned
 *             eight-bit number.
 * @exception  EOFException  if this file has reached the end.
 * @exception  IOException   if an I/O error occurs.
 */
public final int readUnsignedByte() throws IOException {
    int ch = this.read();
    if (ch < 0)
        throw new EOFException();
    return ch;
}

/**
 * Reads a signed 16-bit number from this file. The method reads two
 * bytes from this file, starting at the current file pointer.
 * If the two bytes read, in order, are
 * {@code b1} and {@code b2}, where each of the two values is
 * between {@code 0} and {@code 255}, inclusive, then the
 * result is equal to:
 * <blockquote><pre>
 *     (short)((b1 << 8) | b2)
 * </pre></blockquote>
 * <p>
 * This method blocks until the two bytes are read, the end of the
 * stream is detected, or an exception is thrown.
 *
 * @return     the next two bytes of this file, interpreted as a signed
 *             16-bit number.
```

```
     * @exception  EOFException  if this file reaches the end before reading
     *               two bytes.
     * @exception  IOException   if an I/O error occurs.
     */
    public final short readShort() throws IOException {
        int ch1 = this.read();
        int ch2 = this.read();
        if ((ch1 | ch2) < 0)
            throw new EOFException();
        return (short)((ch1 << 8) + (ch2 << 0));
    }

    /**
     * Reads an unsigned 16-bit number from this file. This method reads
     * two bytes from the file, starting at the current file pointer.
     * If the bytes read, in order, are
     * {@code b1} and {@code b2}, where
     * <code>0 <= b1, b2 <= 255</code>,
     * then the result is equal to:
     * <blockquote><pre>
     *     (b1 << 8) | b2
     * </pre></blockquote>
     * <p>
     * This method blocks until the two bytes are read, the end of the
     * stream is detected, or an exception is thrown.
     *
     * @return     the next two bytes of this file, interpreted as an unsigned
     *             16-bit integer.
     * @exception  EOFException  if this file reaches the end before reading
     *               two bytes.
     * @exception  IOException   if an I/O error occurs.
     */
    public final int readUnsignedShort() throws IOException {
        int ch1 = this.read();
        int ch2 = this.read();
        if ((ch1 | ch2) < 0)
            throw new EOFException();
        return (ch1 << 8) + (ch2 << 0);
    }

    /**
     * Reads a character from this file. This method reads two
     * bytes from the file, starting at the current file pointer.
     * If the bytes read, in order, are
     * {@code b1} and {@code b2}, where
     * <code>0 <= b1, b2 <= 255</code>,
     * then the result is equal to:
     * <blockquote><pre>
     *     (char)((b1 << 8) | b2)
     * </pre></blockquote>
     * <p>
     * This method blocks until the two bytes are read, the end of the
     * stream is detected, or an exception is thrown.
     *
     * @return     the next two bytes of this file, interpreted as a
     *                  {@code char}.
     * @exception  EOFException  if this file reaches the end before reading
     *               two bytes.
     * @exception  IOException   if an I/O error occurs.
     */
    public final char readChar() throws IOException {
        int ch1 = this.read();
        int ch2 = this.read();
```

```
    if ((ch1 | ch2) < 0)
        throw new EOFException();
    return (char)((ch1 << 8) + (ch2 << 0));
}

/**
 * Reads a signed 32-bit integer from this file. This method reads 4
 * bytes from the file, starting at the current file pointer.
 * If the bytes read, in order, are {@code b1},
 * {@code b2}, {@code b3}, and {@code b4}, where
 * <code>0 <= b1, b2, b3, b4 <= 255</code>,
 * then the result is equal to:
 * <blockquote><pre>
 *     (b1 << 24) | (b2 << 16) + (b3 << 8) + b4
 * </pre></blockquote>
 * <p>
 * This method blocks until the four bytes are read, the end of the
 * stream is detected, or an exception is thrown.
 *
 * @return     the next four bytes of this file, interpreted as an
 *             {@code int}.
 * @exception  EOFException  if this file reaches the end before reading
 *               four bytes.
 * @exception  IOException   if an I/O error occurs.
 */
public final int readInt() throws IOException {
    int ch1 = this.read();
    int ch2 = this.read();
    int ch3 = this.read();
    int ch4 = this.read();
    if ((ch1 | ch2 | ch3 | ch4) < 0)
        throw new EOFException();
    return ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4 << 0));
}

/**
 * Reads a signed 64-bit integer from this file. This method reads eight
 * bytes from the file, starting at the current file pointer.
 * If the bytes read, in order, are
 * {@code b1}, {@code b2}, {@code b3},
 * {@code b4}, {@code b5}, {@code b6},
 * {@code b7}, and {@code b8,} where:
 * <blockquote><pre>
 *     0 <= b1, b2, b3, b4, b5, b6, b7, b8 <=255,
 * </pre></blockquote>
 * <p>
 * then the result is equal to:
 * <blockquote><pre>
 *     ((long)b1 << 56) + ((long)b2 << 48)
 *     + ((long)b3 << 40) + ((long)b4 << 32)
 *     + ((long)b5 << 24) + ((long)b6 << 16)
 *     + ((long)b7 << 8) + b8
 * </pre></blockquote>
 * <p>
 * This method blocks until the eight bytes are read, the end of the
 * stream is detected, or an exception is thrown.
 *
 * @return     the next eight bytes of this file, interpreted as a
 *             {@code long}.
 * @exception  EOFException  if this file reaches the end before reading
 *               eight bytes.
 * @exception  IOException   if an I/O error occurs.
 */
```

```
public final long readLong() throws IOException {
    return ((long)(readInt()) << 32) + (readInt() & 0xFFFFFFFFL);
}

/**
 * Reads a {@code float} from this file. This method reads an
 * {@code int} value, starting at the current file pointer,
 * as if by the {@code readInt} method
 * and then converts that {@code int} to a {@code float}
 * using the {@code intBitsToFloat} method in class
 * {@code Float}.
 * <p>
 * This method blocks until the four bytes are read, the end of the
 * stream is detected, or an exception is thrown.
 *
 * @return     the next four bytes of this file, interpreted as a
 *             {@code float}.
 * @exception  EOFException  if this file reaches the end before reading
 *             four bytes.
 * @exception  IOException   if an I/O error occurs.
 * @see        java.io.RandomAccessFile#readInt()
 * @see        java.lang.Float#intBitsToFloat(int)
 */
public final float readFloat() throws IOException {
    return Float.intBitsToFloat(readInt());
}

/**
 * Reads a {@code double} from this file. This method reads a
 * {@code long} value, starting at the current file pointer,
 * as if by the {@code readLong} method
 * and then converts that {@code long} to a {@code double}
 * using the {@code longBitsToDouble} method in
 * class {@code Double}.
 * <p>
 * This method blocks until the eight bytes are read, the end of the
 * stream is detected, or an exception is thrown.
 *
 * @return     the next eight bytes of this file, interpreted as a
 *             {@code double}.
 * @exception  EOFException  if this file reaches the end before reading
 *             eight bytes.
 * @exception  IOException   if an I/O error occurs.
 * @see        java.io.RandomAccessFile#readLong()
 * @see        java.lang.Double#longBitsToDouble(long)
 */
public final double readDouble() throws IOException {
    return Double.longBitsToDouble(readLong());
}

/**
 * Reads the next line of text from this file.  This method successively
 * reads bytes from the file, starting at the current file pointer,
 * until it reaches a line terminator or the end
 * of the file.  Each byte is converted into a character by taking the
 * byte's value for the lower eight bits of the character and setting the
 * high eight bits of the character to zero.  This method does not,
 * therefore, support the full Unicode character set.
 *
 * <p> A line of text is terminated by a carriage-return character
 * ({@code '\u005Cr'}), a newline character ({@code '\u005Cn'}), a
 * carriage-return character immediately followed by a newline character,
 * or the end of the file.  Line-terminating characters are discarded and
```

```
 * are not included as part of the string returned.
 *
 * <p> This method blocks until a newline character is read, a carriage
 * return and the byte following it are read (to see if it is a newline),
 * the end of the file is reached, or an exception is thrown.
 *
 * @return     the next line of text from this file, or null if end
 *             of file is encountered before even one byte is read.
 * @exception  IOException  if an I/O error occurs.
 */

public final String readLine() throws IOException {
    StringBuffer input = new StringBuffer();
    int c = -1;
    boolean eol = false;

    while (!eol) {
        switch (c = read()) {
        case -1:
        case '\n':
            eol = true;
            break;
        case '\r':
            eol = true;
            long cur = getFilePointer();
            if ((read()) != '\n') {
                seek(cur);
            }
            break;
        default:
            input.append((char)c);
            break;
        }
    }

    if ((c == -1) && (input.length() == 0)) {
        return null;
    }
    return input.toString();
}

/**
 * Reads in a string from this file. The string has been encoded
 * using a
 * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
 * format.
 * <p>
 * The first two bytes are read, starting from the current file
 * pointer, as if by
 * {@code readUnsignedShort}. This value gives the number of
 * following bytes that are in the encoded string, not
 * the length of the resulting string. The following bytes are then
 * interpreted as bytes encoding characters in the modified UTF-8 format
 * and are converted into characters.
 * <p>
 * This method blocks until all the bytes are read, the end of the
 * stream is detected, or an exception is thrown.
 *
 * @return     a Unicode string.
 * @exception  EOFException            if this file reaches the end before
 *               reading all the bytes.
 * @exception  IOException             if an I/O error occurs.
 * @exception  UTFDataFormatException  if the bytes do not represent
```

```
     *               valid modified UTF-8 encoding of a Unicode string.
     * @see        java.io.RandomAccessFile#readUnsignedShort()
     */
    public final String readUTF() throws IOException {
        return DataInputStream.readUTF(this);
    }

    /**
     * Writes a {@code boolean} to the file as a one-byte value. The
     * value {@code true} is written out as the value
     * {@code (byte)1}; the value {@code false} is written out
     * as the value {@code (byte)0}. The write starts at
     * the current position of the file pointer.
     *
     * @param      v   a {@code boolean} value to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public final void writeBoolean(boolean v) throws IOException {
        write(v ? 1 : 0);
        //written++;
    }

    /**
     * Writes a {@code byte} to the file as a one-byte value. The
     * write starts at the current position of the file pointer.
     *
     * @param      v   a {@code byte} value to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public final void writeByte(int v) throws IOException {
        write(v);
        //written++;
    }

    /**
     * Writes a {@code short} to the file as two bytes, high byte first.
     * The write starts at the current position of the file pointer.
     *
     * @param      v   a {@code short} to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public final void writeShort(int v) throws IOException {
        write((v >>> 8) & 0xFF);
        write((v >>> 0) & 0xFF);
        //written += 2;
    }

    /**
     * Writes a {@code char} to the file as a two-byte value, high
     * byte first. The write starts at the current position of the
     * file pointer.
     *
     * @param      v   a {@code char} value to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public final void writeChar(int v) throws IOException {
        write((v >>> 8) & 0xFF);
        write((v >>> 0) & 0xFF);
        //written += 2;
    }

    /**
     * Writes an {@code int} to the file as four bytes, high byte first.
```

```
     * The write starts at the current position of the file pointer.
     *
     * @param      v   an {@code int} to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public final void writeInt(int v) throws IOException {
        write((v >>> 24) & 0xFF);
        write((v >>> 16) & 0xFF);
        write((v >>>  8) & 0xFF);
        write((v >>>  0) & 0xFF);
        //written += 4;
    }

    /**
     * Writes a {@code long} to the file as eight bytes, high byte first.
     * The write starts at the current position of the file pointer.
     *
     * @param      v   a {@code long} to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public final void writeLong(long v) throws IOException {
        write((int)(v >>> 56) & 0xFF);
        write((int)(v >>> 48) & 0xFF);
        write((int)(v >>> 40) & 0xFF);
        write((int)(v >>> 32) & 0xFF);
        write((int)(v >>> 24) & 0xFF);
        write((int)(v >>> 16) & 0xFF);
        write((int)(v >>>  8) & 0xFF);
        write((int)(v >>>  0) & 0xFF);
        //written += 8;
    }

    /**
     * Converts the float argument to an {@code int} using the
     * {@code floatToIntBits} method in class {@code Float},
     * and then writes that {@code int} value to the file as a
     * four-byte quantity, high byte first. The write starts at the
     * current position of the file pointer.
     *
     * @param      v   a {@code float} value to be written.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.lang.Float#floatToIntBits(float)
     */
    public final void writeFloat(float v) throws IOException {
        writeInt(Float.floatToIntBits(v));
    }

    /**
     * Converts the double argument to a {@code long} using the
     * {@code doubleToLongBits} method in class {@code Double},
     * and then writes that {@code long} value to the file as an
     * eight-byte quantity, high byte first. The write starts at the current
     * position of the file pointer.
     *
     * @param      v   a {@code double} value to be written.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.lang.Double#doubleToLongBits(double)
     */
    public final void writeDouble(double v) throws IOException {
        writeLong(Double.doubleToLongBits(v));
    }

    /**
```

```
     * Writes the string to the file as a sequence of bytes. Each
     * character in the string is written out, in sequence, by discarding
     * its high eight bits. The write starts at the current position of
     * the file pointer.
     *
     * @param      s   a string of bytes to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    @SuppressWarnings("deprecation")
    public final void writeBytes(String s) throws IOException {
        int len = s.length();
        byte[] b = new byte[len];
        s.getBytes(0, len, b, 0);
        writeBytes(b, 0, len);
    }

    /**
     * Writes a string to the file as a sequence of characters. Each
     * character is written to the data output stream as if by the
     * {@code writeChar} method. The write starts at the current
     * position of the file pointer.
     *
     * @param      s   a {@code String} value to be written.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.RandomAccessFile#writeChar(int)
     */
    public final void writeChars(String s) throws IOException {
        int clen = s.length();
        int blen = 2*clen;
        byte[] b = new byte[blen];
        char[] c = new char[clen];
        s.getChars(0, clen, c, 0);
        for (int i = 0, j = 0; i < clen; i++) {
            b[j++] = (byte)(c[i] >>> 8);
            b[j++] = (byte)(c[i] >>> 0);
        }
        writeBytes(b, 0, blen);
    }

    /**
     * Writes a string to the file using
     * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
     * encoding in a machine-independent manner.
     * <p>
     * First, two bytes are written to the file, starting at the
     * current file pointer, as if by the
     * {@code writeShort} method giving the number of bytes to
     * follow. This value is the number of bytes actually written out,
     * not the length of the string. Following the length, each character
     * of the string is output, in sequence, using the modified UTF-8 encoding
     * for each character.
     *
     * @param      str   a string to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public final void writeUTF(String str) throws IOException {
        DataOutputStream.writeUTF(str, this);
    }

    private static native void initIDs();

    private native void close0() throws IOException;
```

```
    static {
        initIDs();
    }
}
```

Reader.java

```
/*
 * Copyright (c) 1996, 2012, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Abstract class for reading character streams.  The only methods that a
 * subclass must implement are read(char[], int, int) and close().  Most
 * subclasses, however, will override some of the methods defined here in order
 * to provide higher efficiency, additional functionality, or both.
 *
 *
 * @see BufferedReader
 * @see   LineNumberReader
 * @see CharArrayReader
 * @see InputStreamReader
 * @see   FileReader
 * @see FilterReader
 * @see   PushbackReader
 * @see PipedReader
 * @see StringReader
 * @see Writer
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public abstract class Reader implements Readable, Closeable {

    /**
     * The object used to synchronize operations on this stream.  For
     * efficiency, a character-stream object may use an object other than
     * itself to protect critical sections.  A subclass should therefore use
     * the object in this field rather than <tt>this</tt> or a synchronized
     * method.
     */
    protected Object lock;
```

```
/**
 * Creates a new character-stream reader whose critical sections will
 * synchronize on the reader itself.
 */
protected Reader() {
    this.lock = this;
}

/**
 * Creates a new character-stream reader whose critical sections will
 * synchronize on the given object.
 *
 * @param lock  The Object to synchronize on.
 */
protected Reader(Object lock) {
    if (lock == null) {
        throw new NullPointerException();
    }
    this.lock = lock;
}

/**
 * Attempts to read characters into the specified character buffer.
 * The buffer is used as a repository of characters as-is: the only
 * changes made are the results of a put operation. No flipping or
 * rewinding of the buffer is performed.
 *
 * @param target the buffer to read characters into
 * @return The number of characters added to the buffer, or
 *         -1 if this source of characters is at its end
 * @throws IOException if an I/O error occurs
 * @throws NullPointerException if target is null
 * @throws java.nio.ReadOnlyBufferException if target is a read only buffer
 * @since 1.5
 */
public int read(java.nio.CharBuffer target) throws IOException {
    int len = target.remaining();
    char[] cbuf = new char[len];
    int n = read(cbuf, 0, len);
    if (n > 0)
        target.put(cbuf, 0, n);
    return n;
}

/**
 * Reads a single character.  This method will block until a character is
 * available, an I/O error occurs, or the end of the stream is reached.
 *
 * <p> Subclasses that intend to support efficient single-character input
 * should override this method.
 *
 * @return     The character read, as an integer in the range 0 to 65535
 *             (<tt>0x00-0xffff</tt>), or -1 if the end of the stream has
 *             been reached
 *
 * @exception  IOException  If an I/O error occurs
 */
public int read() throws IOException {
    char cb[] = new char[1];
    if (read(cb, 0, 1) == -1)
        return -1;
    else
```

```
        return cb[0];
}

/**
 * Reads characters into an array.  This method will block until some input
 * is available, an I/O error occurs, or the end of the stream is reached.
 *
 * @param       cbuf  Destination buffer
 *
 * @return      The number of characters read, or -1
 *              if the end of the stream
 *              has been reached
 *
 * @exception   IOException  If an I/O error occurs
 */
public int read(char cbuf[]) throws IOException {
    return read(cbuf, 0, cbuf.length);
}

/**
 * Reads characters into a portion of an array.  This method will block
 * until some input is available, an I/O error occurs, or the end of the
 * stream is reached.
 *
 * @param      cbuf  Destination buffer
 * @param      off   Offset at which to start storing characters
 * @param      len   Maximum number of characters to read
 *
 * @return     The number of characters read, or -1 if the end of the
 *             stream has been reached
 *
 * @exception  IOException  If an I/O error occurs
 */
abstract public int read(char cbuf[], int off, int len) throws IOException;

/** Maximum skip-buffer size */
private static final int maxSkipBufferSize = 8192;

/** Skip buffer, null until allocated */
private char skipBuffer[] = null;

/**
 * Skips characters.  This method will block until some characters are
 * available, an I/O error occurs, or the end of the stream is reached.
 *
 * @param  n  The number of characters to skip
 *
 * @return    The number of characters actually skipped
 *
 * @exception  IllegalArgumentException  If <code>n</code> is negative.
 * @exception  IOException  If an I/O error occurs
 */
public long skip(long n) throws IOException {
    if (n < 0L)
        throw new IllegalArgumentException("skip value is negative");
    int nn = (int) Math.min(n, maxSkipBufferSize);
    synchronized (lock) {
        if ((skipBuffer == null) || (skipBuffer.length < nn))
            skipBuffer = new char[nn];
        long r = n;
        while (r > 0) {
            int nc = read(skipBuffer, 0, (int)Math.min(r, nn));
            if (nc == -1)
```

```
                break;
            r -= nc;
        }
        return n - r;
    }
}

/**
 * Tells whether this stream is ready to be read.
 *
 * @return True if the next read() is guaranteed not to block for input,
 * false otherwise.  Note that returning false does not guarantee that the
 * next read will block.
 *
 * @exception  IOException  If an I/O error occurs
 */
public boolean ready() throws IOException {
    return false;
}

/**
 * Tells whether this stream supports the mark() operation. The default
 * implementation always returns false. Subclasses should override this
 * method.
 *
 * @return true if and only if this stream supports the mark operation.
 */
public boolean markSupported() {
    return false;
}

/**
 * Marks the present position in the stream.  Subsequent calls to reset()
 * will attempt to reposition the stream to this point.  Not all
 * character-input streams support the mark() operation.
 *
 * @param  readAheadLimit  Limit on the number of characters that may be
 *                         read while still preserving the mark.  After
 *                         reading this many characters, attempting to
 *                         reset the stream may fail.
 *
 * @exception  IOException  If the stream does not support mark(),
 *                          or if some other I/O error occurs
 */
public void mark(int readAheadLimit) throws IOException {
    throw new IOException("mark() not supported");
}

/**
 * Resets the stream.  If the stream has been marked, then attempt to
 * reposition it at the mark.  If the stream has not been marked, then
 * attempt to reset it in some way appropriate to the particular stream,
 * for example by repositioning it to its starting point.  Not all
 * character-input streams support the reset() operation, and some support
 * reset() without supporting mark().
 *
 * @exception  IOException  If the stream has not been marked,
 *                          or if the mark has been invalidated,
 *                          or if the stream does not support reset(),
 *                          or if some other I/O error occurs
 */
public void reset() throws IOException {
    throw new IOException("reset() not supported");
```

```
    }

    /**
     * Closes the stream and releases any system resources associated with
     * it.  Once the stream has been closed, further read(), ready(),
     * mark(), reset(), or skip() invocations will throw an IOException.
     * Closing a previously closed stream has no effect.
     *
     * @exception  IOException  If an I/O error occurs
     */
     abstract public void close() throws IOException;

}
```

SequenceInputStream.java

```java
/*
 * Copyright (c) 1994, 2011, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.io.InputStream;
import java.util.Enumeration;
import java.util.Vector;

/**
 * A <code>SequenceInputStream</code> represents
 * the logical concatenation of other input
 * streams. It starts out with an ordered
 * collection of input streams and reads from
 * the first one until end of file is reached,
 * whereupon it reads from the second one,
 * and so on, until end of file is reached
 * on the last of the contained input streams.
 *
 * @author  Author van Hoff
 * @since   JDK1.0
 */
public
class SequenceInputStream extends InputStream {
    Enumeration<? extends InputStream> e;
    InputStream in;

    /**
     * Initializes a newly created <code>SequenceInputStream</code>
     * by remembering the argument, which must
     * be an <code>Enumeration</code>  that produces
     * objects whose run-time type is <code>InputStream</code>.
     * The input streams that are  produced by
     * the enumeration will be read, in order,
     * to provide the bytes to be read  from this
     * <code>SequenceInputStream</code>. After
     * each input stream from the enumeration
     * is exhausted, it is closed by calling its
```

```
     * <code>close</code> method.
     *
     * @param   e   an enumeration of input streams.
     * @see     java.util.Enumeration
     */
    public SequenceInputStream(Enumeration<? extends InputStream> e) {
        this.e = e;
        try {
            nextStream();
        } catch (IOException ex) {
            // This should never happen
            throw new Error("panic");
        }
    }

    /**
     * Initializes a newly
     * created <code>SequenceInputStream</code>
     * by remembering the two arguments, which
     * will be read in order, first <code>s1</code>
     * and then <code>s2</code>, to provide the
     * bytes to be read from this <code>SequenceInputStream</code>.
     *
     * @param   s1   the first input stream to read.
     * @param   s2   the second input stream to read.
     */
    public SequenceInputStream(InputStream s1, InputStream s2) {
        Vector<InputStream> v = new Vector<>(2);

        v.addElement(s1);
        v.addElement(s2);
        e = v.elements();
        try {
            nextStream();
        } catch (IOException ex) {
            // This should never happen
            throw new Error("panic");
        }
    }

    /**
     *  Continues reading in the next stream if an EOF is reached.
     */
    final void nextStream() throws IOException {
        if (in != null) {
            in.close();
        }

        if (e.hasMoreElements()) {
            in = (InputStream) e.nextElement();
            if (in == null)
                throw new NullPointerException();
        }
        else in = null;

    }

    /**
     * Returns an estimate of the number of bytes that can be read (or
     * skipped over) from the current underlying input stream without
     * blocking by the next invocation of a method for the current
     * underlying input stream. The next invocation might be
     * the same thread or another thread.  A single read or skip of this
```

```
     * many bytes will not block, but may read or skip fewer bytes.
     * <p>
     * This method simply calls {@code available} of the current underlying
     * input stream and returns the result.
     *
     * @return an estimate of the number of bytes that can be read (or
     *         skipped over) from the current underlying input stream
     *         without blocking or {@code 0} if this input stream
     *         has been closed by invoking its {@link #close()} method
     * @exception  IOException  if an I/O error occurs.
     *
     * @since   JDK1.1
     */
    public int available() throws IOException {
        if(in == null) {
            return 0; // no way to signal EOF from available()
        }
        return in.available();
    }

    /**
     * Reads the next byte of data from this input stream. The byte is
     * returned as an <code>int</code> in the range <code>0</code> to
     * <code>255</code>. If no byte is available because the end of the
     * stream has been reached, the value <code>-1</code> is returned.
     * This method blocks until input data is available, the end of the
     * stream is detected, or an exception is thrown.
     * <p>
     * This method
     * tries to read one character from the current substream. If it
     * reaches the end of the stream, it calls the <code>close</code>
     * method of the current substream and begins reading from the next
     * substream.
     *
     * @return     the next byte of data, or <code>-1</code> if the end of the
     *             stream is reached.
     * @exception  IOException  if an I/O error occurs.
     */
    public int read() throws IOException {
        if (in == null) {
            return -1;
        }
        int c = in.read();
        if (c == -1) {
            nextStream();
            return read();
        }
        return c;
    }

    /**
     * Reads up to <code>len</code> bytes of data from this input stream
     * into an array of bytes.  If <code>len</code> is not zero, the method
     * blocks until at least 1 byte of input is available; otherwise, no
     * bytes are read and <code>0</code> is returned.
     * <p>
     * The <code>read</code> method of <code>SequenceInputStream</code>
     * tries to read the data from the current substream. If it fails to
     * read any characters because the substream has reached the end of
     * the stream, it calls the <code>close</code> method of the current
     * substream and begins reading from the next substream.
     *
     * @param      b     the buffer into which the data is read.
```

```
     * @param      off   the start offset in array <code>b</code>
     *                   at which the data is written.
     * @param      len   the maximum number of bytes read.
     * @return     int   the number of bytes read.
     * @exception  NullPointerException If <code>b</code> is <code>null</code>.
     * @exception  IndexOutOfBoundsException If <code>off</code> is negative,
     * <code>len</code> is negative, or <code>len</code> is greater than
     * <code>b.length - off</code>
     * @exception  IOException  if an I/O error occurs.
     */
    public int read(byte b[], int off, int len) throws IOException {
        if (in == null) {
            return -1;
        } else if (b == null) {
            throw new NullPointerException();
        } else if (off < 0 || len < 0 || len > b.length - off) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return 0;
        }

        int n = in.read(b, off, len);
        if (n <= 0) {
            nextStream();
            return read(b, off, len);
        }
        return n;
    }

    /**
     * Closes this input stream and releases any system resources
     * associated with the stream.
     * A closed <code>SequenceInputStream</code>
     * cannot  perform input operations and cannot
     * be reopened.
     * <p>
     * If this stream was created
     * from an enumeration, all remaining elements
     * are requested from the enumeration and closed
     * before the <code>close</code> method returns.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void close() throws IOException {
        do {
            nextStream();
        } while (in != null);
    }
}
```

SerialCallbackContext.java

```
/*
 * Copyright (c) 2006, 2012, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Context during upcalls from object stream to class-defined
 * readObject/writeObject methods.
 * Holds object currently being deserialized and descriptor for current class.
 *
 * This context keeps track of the thread it was constructed on, and allows
 * only a single call of defaultReadObject, readFields, defaultWriteObject
 * or writeFields which must be invoked on the same thread before the class's
 * readObject/writeObject method has returned.
 * If not set to the current thread, the getObj method throws NotActiveException.
 */
final class SerialCallbackContext {
    private final Object obj;
    private final ObjectStreamClass desc;
    /**
     * Thread this context is in use by.
     * As this only works in one thread, we do not need to worry about thread-safety.
     */
    private Thread thread;

    public SerialCallbackContext(Object obj, ObjectStreamClass desc) {
        this.obj = obj;
        this.desc = desc;
        this.thread = Thread.currentThread();
    }

    public Object getObj() throws NotActiveException {
        checkAndSetUsed();
        return obj;
    }

    public ObjectStreamClass getDesc() {
        return desc;
```

```
    }

    private void checkAndSetUsed() throws NotActiveException {
        if (thread != Thread.currentThread()) {
             throw new NotActiveException(
              "not in readObject invocation or fields already read");
        }
        thread = null;
    }

    public void setUsed() {
        thread = null;
    }
}
```

Serializable.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Serializability of a class is enabled by the class implementing the
 * java.io.Serializable interface. Classes that do not implement this
 * interface will not have any of their state serialized or
 * deserialized.  All subtypes of a serializable class are themselves
 * serializable.  The serialization interface has no methods or fields
 * and serves only to identify the semantics of being serializable. <p>
 *
 * To allow subtypes of non-serializable classes to be serialized, the
 * subtype may assume responsibility for saving and restoring the
 * state of the supertype's public, protected, and (if accessible)
 * package fields.  The subtype may assume this responsibility only if
 * the class it extends has an accessible no-arg constructor to
 * initialize the class's state.  It is an error to declare a class
 * Serializable if this is not the case.  The error will be detected at
 * runtime. <p>
 *
 * During deserialization, the fields of non-serializable classes will
 * be initialized using the public or protected no-arg constructor of
 * the class.  A no-arg constructor must be accessible to the subclass
 * that is serializable.  The fields of serializable subclasses will
 * be restored from the stream. <p>
 *
 * When traversing a graph, an object may be encountered that does not
 * support the Serializable interface. In this case the
 * NotSerializableException will be thrown and will identify the class
 * of the non-serializable object. <p>
 *
 * Classes that require special handling during the serialization and
 * deserialization process must implement special methods with these exact
 * signatures:
 *
 * <PRE>
```

```
* private void writeObject(java.io.ObjectOutputStream out)
*     throws IOException
* private void readObject(java.io.ObjectInputStream in)
*     throws IOException, ClassNotFoundException;
* private void readObjectNoData()
*     throws ObjectStreamException;
* </PRE>
*
* <p>The writeObject method is responsible for writing the state of the
* object for its particular class so that the corresponding
* readObject method can restore it.  The default mechanism for saving
* the Object's fields can be invoked by calling
* out.defaultWriteObject. The method does not need to concern
* itself with the state belonging to its superclasses or subclasses.
* State is saved by writing the individual fields to the
* ObjectOutputStream using the writeObject method or by using the
* methods for primitive data types supported by DataOutput.
*
* <p>The readObject method is responsible for reading from the stream and
* restoring the classes fields. It may call in.defaultReadObject to invoke
* the default mechanism for restoring the object's non-static and
* non-transient fields.  The defaultReadObject method uses information in
* the stream to assign the fields of the object saved in the stream with the
* correspondingly named fields in the current object.  This handles the case
* when the class has evolved to add new fields. The method does not need to
* concern itself with the state belonging to its superclasses or subclasses.
* State is saved by writing the individual fields to the
* ObjectOutputStream using the writeObject method or by using the
* methods for primitive data types supported by DataOutput.
*
* <p>The readObjectNoData method is responsible for initializing the state of
* the object for its particular class in the event that the serialization
* stream does not list the given class as a superclass of the object being
* deserialized.  This may occur in cases where the receiving party uses a
* different version of the deserialized instance's class than the sending
* party, and the receiver's version extends classes that are not extended by
* the sender's version.  This may also occur if the serialization stream has
* been tampered; hence, readObjectNoData is useful for initializing
* deserialized objects properly despite a "hostile" or incomplete source
* stream.
*
* <p>Serializable classes that need to designate an alternative object to be
* used when writing an object to the stream should implement this
* special method with the exact signature:
*
* <PRE>
* ANY-ACCESS-MODIFIER Object writeReplace() throws ObjectStreamException;
* </PRE><p>
*
* This writeReplace method is invoked by serialization if the method
* exists and it would be accessible from a method defined within the
* class of the object being serialized. Thus, the method can have private,
* protected and package-private access. Subclass access to this method
* follows java accessibility rules. <p>
*
* Classes that need to designate a replacement when an instance of it
* is read from the stream should implement this special method with the
* exact signature.
*
* <PRE>
* ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;
* </PRE><p>
*
```

```
 * This readResolve method follows the same invocation rules and
 * accessibility rules as writeReplace.<p>
 *
 * The serialization runtime associates with each serializable class a version
 * number, called a serialVersionUID, which is used during deserialization to
 * verify that the sender and receiver of a serialized object have loaded
 * classes for that object that are compatible with respect to serialization.
 * If the receiver has loaded a class for the object that has a different
 * serialVersionUID than that of the corresponding sender's class, then
 * deserialization will result in an {@link InvalidClassException}.  A
 * serializable class can declare its own serialVersionUID explicitly by
 * declaring a field named <code>"serialVersionUID"</code> that must be static,
 * final, and of type <code>long</code>:
 *
 * <PRE>
 * ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;
 * </PRE>
 *
 * If a serializable class does not explicitly declare a serialVersionUID, then
 * the serialization runtime will calculate a default serialVersionUID value
 * for that class based on various aspects of the class, as described in the
 * Java(TM) Object Serialization Specification.  However, it is <em>strongly
 * recommended</em> that all serializable classes explicitly declare
 * serialVersionUID values, since the default serialVersionUID computation is
 * highly sensitive to class details that may vary depending on compiler
 * implementations, and can thus result in unexpected
 * <code>InvalidClassException</code>s during deserialization.  Therefore, to
 * guarantee a consistent serialVersionUID value across different java compiler
 * implementations, a serializable class must declare an explicit
 * serialVersionUID value.  It is also strongly advised that explicit
 * serialVersionUID declarations use the <code>private</code> modifier where
 * possible, since such declarations apply only to the immediately declaring
 * class--serialVersionUID fields are not useful as inherited members. Array
 * classes cannot declare an explicit serialVersionUID, so they always have
 * the default computed value, but the requirement for matching
 * serialVersionUID values is waived for array classes.
 *
 * @author  unascribed
 * @see java.io.ObjectOutputStream
 * @see java.io.ObjectInputStream
 * @see java.io.ObjectOutput
 * @see java.io.ObjectInput
 * @see java.io.Externalizable
 * @since   JDK1.1
 */
public interface Serializable {
}
```

SerializablePermission.java

```
/*
 * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.security.*;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.StringTokenizer;

/**
 * This class is for Serializable permissions. A SerializablePermission
 * contains a name (also referred to as a "target name") but
 * no actions list; you either have the named permission
 * or you don't.
 *
 * <P>
 * The target name is the name of the Serializable permission (see below).
 *
 * <P>
 * The following table lists all the possible SerializablePermission target names,
 * and for each provides a description of what the permission allows
 * and a discussion of the risks of granting code the permission.
 *
 * <table border=1 cellpadding=5 summary="Permission target name, what the permission allows, and associated
risks">
 * <tr>
 * <th>Permission Target Name</th>
 * <th>What the Permission Allows</th>
 * <th>Risks of Allowing this Permission</th>
 * </tr>
 *
 * <tr>
 *   <td>enableSubclassImplementation</td>
 *   <td>Subclass implementation of ObjectOutputStream or ObjectInputStream
 * to override the default serialization or deserialization, respectively,
 * of objects</td>
 *   <td>Code can use this to serialize or
```

```
 * deserialize classes in a purposefully malfeasant manner. For example,
 * during serialization, malicious code can use this to
 * purposefully store confidential private field data in a way easily accessible
 * to attackers. Or, during deserialization it could, for example, deserialize
 * a class with all its private fields zeroed out.</td>
 * </tr>
 *
 * <tr>
 *   <td>enableSubstitution</td>
 *   <td>Substitution of one object for another during
 * serialization or deserialization</td>
 *   <td>This is dangerous because malicious code
 * can replace the actual object with one which has incorrect or
 * malignant data.</td>
 * </tr>
 *
 * </table>
 *
 * @see java.security.BasicPermission
 * @see java.security.Permission
 * @see java.security.Permissions
 * @see java.security.PermissionCollection
 * @see java.lang.SecurityManager
 *
 *
 * @author Joe Fialli
 * @since 1.2
 */

/* code was borrowed originally from java.lang.RuntimePermission. */

public final class SerializablePermission extends BasicPermission {

    private static final long serialVersionUID = 8537212141160296410L;

    /**
     * @serial
     */
    private String actions;

    /**
     * Creates a new SerializablePermission with the specified name.
     * The name is the symbolic name of the SerializablePermission, such as
     * "enableSubstitution", etc.
     *
     * @param name the name of the SerializablePermission.
     *
     * @throws NullPointerException if <code>name</code> is <code>null</code>.
     * @throws IllegalArgumentException if <code>name</code> is empty.
     */
    public SerializablePermission(String name)
    {
        super(name);
    }

    /**
     * Creates a new SerializablePermission object with the specified name.
     * The name is the symbolic name of the SerializablePermission, and the
     * actions String is currently unused and should be null.
     *
     * @param name the name of the SerializablePermission.
     * @param actions currently unused and must be set to null
     *
```

```
     * @throws NullPointerException if <code>name</code> is <code>null</code>.
     * @throws IllegalArgumentException if <code>name</code> is empty.
     */

    public SerializablePermission(String name, String actions)
    {
        super(name, actions);
    }
}
```

StreamCorruptedException.java

```java
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Thrown when control information that was read from an object stream
 * violates internal consistency checks.
 *
 * @author  unascribed
 * @since   JDK1.1
 */
public class StreamCorruptedException extends ObjectStreamException {

    private static final long serialVersionUID = 8983558202217591746L;

    /**
     * Create a StreamCorruptedException and list a reason why thrown.
     *
     * @param reason  String describing the reason for the exception.
     */
    public StreamCorruptedException(String reason) {
        super(reason);
    }

    /**
     * Create a StreamCorruptedException and list no reason why thrown.
     */
    public StreamCorruptedException() {
        super();
    }
}
```

StreamTokenizer.java

```
/*
 * Copyright (c) 1995, 2012, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.util.Arrays;

/**
 * The {@code StreamTokenizer} class takes an input stream and
 * parses it into "tokens", allowing the tokens to be
 * read one at a time. The parsing process is controlled by a table
 * and a number of flags that can be set to various states. The
 * stream tokenizer can recognize identifiers, numbers, quoted
 * strings, and various comment styles.
 * <p>
 * Each byte read from the input stream is regarded as a character
 * in the range {@code '\u005Cu0000'} through {@code '\u005Cu00FF'}.
 * The character value is used to look up five possible attributes of
 * the character: <i>white space</i>, <i>alphabetic</i>,
 * <i>numeric</i>, <i>string quote</i>, and <i>comment character</i>.
 * Each character can have zero or more of these attributes.
 * <p>
 * In addition, an instance has four flags. These flags indicate:
 * <ul>
 * <li>Whether line terminators are to be returned as tokens or treated
 *     as white space that merely separates tokens.
 * <li>Whether C-style comments are to be recognized and skipped.
 * <li>Whether C++-style comments are to be recognized and skipped.
 * <li>Whether the characters of identifiers are converted to lowercase.
 * </ul>
 * <p>
 * A typical application first constructs an instance of this class,
 * sets up the syntax tables, and then repeatedly loops calling the
 * {@code nextToken} method in each iteration of the loop until
 * it returns the value {@code TT_EOF}.
 *
 * @author  James Gosling
 * @see     java.io.StreamTokenizer#nextToken()
```

```
 * @see     java.io.StreamTokenizer#TT_EOF
 * @since   JDK1.0
 */

public class StreamTokenizer {

    /* Only one of these will be non-null */
    private Reader reader = null;
    private InputStream input = null;

    private char buf[] = new char[20];

    /**
     * The next character to be considered by the nextToken method.  May also
     * be NEED_CHAR to indicate that a new character should be read, or SKIP_LF
     * to indicate that a new character should be read and, if it is a '\n'
     * character, it should be discarded and a second new character should be
     * read.
     */
    private int peekc = NEED_CHAR;

    private static final int NEED_CHAR = Integer.MAX_VALUE;
    private static final int SKIP_LF = Integer.MAX_VALUE - 1;

    private boolean pushedBack;
    private boolean forceLower;
    /** The line number of the last token read */
    private int LINENO = 1;

    private boolean eolIsSignificantP = false;
    private boolean slashSlashCommentsP = false;
    private boolean slashStarCommentsP = false;

    private byte ctype[] = new byte[256];
    private static final byte CT_WHITESPACE = 1;
    private static final byte CT_DIGIT = 2;
    private static final byte CT_ALPHA = 4;
    private static final byte CT_QUOTE = 8;
    private static final byte CT_COMMENT = 16;

    /**
     * After a call to the {@code nextToken} method, this field
     * contains the type of the token just read. For a single character
     * token, its value is the single character, converted to an integer.
     * For a quoted string token, its value is the quote character.
     * Otherwise, its value is one of the following:
     * <ul>
     * <li>{@code TT_WORD} indicates that the token is a word.
     * <li>{@code TT_NUMBER} indicates that the token is a number.
     * <li>{@code TT_EOL} indicates that the end of line has been read.
     *     The field can only have this value if the
     *     {@code eolIsSignificant} method has been called with the
     *     argument {@code true}.
     * <li>{@code TT_EOF} indicates that the end of the input stream
     *     has been reached.
     * </ul>
     * <p>
     * The initial value of this field is -4.
     *
     * @see     java.io.StreamTokenizer#eolIsSignificant(boolean)
     * @see     java.io.StreamTokenizer#nextToken()
     * @see     java.io.StreamTokenizer#quoteChar(int)
     * @see     java.io.StreamTokenizer#TT_EOF
```

```
 * @see     java.io.StreamTokenizer#TT_EOL
 * @see     java.io.StreamTokenizer#TT_NUMBER
 * @see     java.io.StreamTokenizer#TT_WORD
 */
public int ttype = TT_NOTHING;

/**
 * A constant indicating that the end of the stream has been read.
 */
public static final int TT_EOF = -1;

/**
 * A constant indicating that the end of the line has been read.
 */
public static final int TT_EOL = '\n';

/**
 * A constant indicating that a number token has been read.
 */
public static final int TT_NUMBER = -2;

/**
 * A constant indicating that a word token has been read.
 */
public static final int TT_WORD = -3;

/* A constant indicating that no token has been read, used for
 * initializing ttype.  FIXME This could be made public and
 * made available as the part of the API in a future release.
 */
private static final int TT_NOTHING = -4;

/**
 * If the current token is a word token, this field contains a
 * string giving the characters of the word token. When the current
 * token is a quoted string token, this field contains the body of
 * the string.
 * <p>
 * The current token is a word when the value of the
 * {@code ttype} field is {@code TT_WORD}. The current token is
 * a quoted string token when the value of the {@code ttype} field is
 * a quote character.
 * <p>
 * The initial value of this field is null.
 *
 * @see     java.io.StreamTokenizer#quoteChar(int)
 * @see     java.io.StreamTokenizer#TT_WORD
 * @see     java.io.StreamTokenizer#ttype
 */
public String sval;

/**
 * If the current token is a number, this field contains the value
 * of that number. The current token is a number when the value of
 * the {@code ttype} field is {@code TT_NUMBER}.
 * <p>
 * The initial value of this field is 0.0.
 *
 * @see     java.io.StreamTokenizer#TT_NUMBER
 * @see     java.io.StreamTokenizer#ttype
 */
public double nval;
```

```
/** Private constructor that initializes everything except the streams. */
private StreamTokenizer() {
    wordChars('a', 'z');
    wordChars('A', 'Z');
    wordChars(128 + 32, 255);
    whitespaceChars(0, ' ');
    commentChar('/');
    quoteChar('"');
    quoteChar('\'');
    parseNumbers();
}

/**
 * Creates a stream tokenizer that parses the specified input
 * stream. The stream tokenizer is initialized to the following
 * default state:
 * <ul>
 * <li>All byte values {@code 'A'} through {@code 'Z'},
 *     {@code 'a'} through {@code 'z'}, and
 *     {@code '\u005Cu00A0'} through {@code '\u005Cu00FF'} are
 *     considered to be alphabetic.
 * <li>All byte values {@code '\u005Cu0000'} through
 *     {@code '\u005Cu0020'} are considered to be white space.
 * <li>{@code '/'} is a comment character.
 * <li>Single quote {@code '\u005C''} and double quote {@code '"'}
 *     are string quote characters.
 * <li>Numbers are parsed.
 * <li>Ends of lines are treated as white space, not as separate tokens.
 * <li>C-style and C++-style comments are not recognized.
 * </ul>
 *
 * @deprecated As of JDK version 1.1, the preferred way to tokenize an
 * input stream is to convert it into a character stream, for example:
 * <blockquote><pre>
 *   Reader r = new BufferedReader(new InputStreamReader(is));
 *   StreamTokenizer st = new StreamTokenizer(r);
 * </pre></blockquote>
 *
 * @param      is        an input stream.
 * @see        java.io.BufferedReader
 * @see        java.io.InputStreamReader
 * @see        java.io.StreamTokenizer#StreamTokenizer(java.io.Reader)
 */
@Deprecated
public StreamTokenizer(InputStream is) {
    this();
    if (is == null) {
        throw new NullPointerException();
    }
    input = is;
}

/**
 * Create a tokenizer that parses the given character stream.
 *
 * @param r  a Reader object providing the input stream.
 * @since   JDK1.1
 */
public StreamTokenizer(Reader r) {
    this();
    if (r == null) {
        throw new NullPointerException();
    }
```

```
    reader = r;
}

/**
 * Resets this tokenizer's syntax table so that all characters are
 * "ordinary." See the {@code ordinaryChar} method
 * for more information on a character being ordinary.
 *
 * @see     java.io.StreamTokenizer#ordinaryChar(int)
 */
public void resetSyntax() {
    for (int i = ctype.length; --i >= 0;)
        ctype[i] = 0;
}

/**
 * Specifies that all characters <i>c</i> in the range
 * <code>low <= <i>c</i> <= high</code>
 * are word constituents. A word token consists of a word constituent
 * followed by zero or more word constituents or number constituents.
 *
 * @param   low   the low end of the range.
 * @param   hi    the high end of the range.
 */
public void wordChars(int low, int hi) {
    if (low < 0)
        low = 0;
    if (hi >= ctype.length)
        hi = ctype.length - 1;
    while (low <= hi)
        ctype[low++] |= CT_ALPHA;
}

/**
 * Specifies that all characters <i>c</i> in the range
 * <code>low <= <i>c</i> <= high</code>
 * are white space characters. White space characters serve only to
 * separate tokens in the input stream.
 *
 * <p>Any other attribute settings for the characters in the specified
 * range are cleared.
 *
 * @param   low   the low end of the range.
 * @param   hi    the high end of the range.
 */
public void whitespaceChars(int low, int hi) {
    if (low < 0)
        low = 0;
    if (hi >= ctype.length)
        hi = ctype.length - 1;
    while (low <= hi)
        ctype[low++] = CT_WHITESPACE;
}

/**
 * Specifies that all characters <i>c</i> in the range
 * <code>low <= <i>c</i> <= high</code>
 * are "ordinary" in this tokenizer. See the
 * {@code ordinaryChar} method for more information on a
 * character being ordinary.
 *
 * @param   low   the low end of the range.
 * @param   hi    the high end of the range.
```

```
     * @see     java.io.StreamTokenizer#ordinaryChar(int)
     */
    public void ordinaryChars(int low, int hi) {
        if (low < 0)
            low = 0;
        if (hi >= ctype.length)
            hi = ctype.length - 1;
        while (low <= hi)
            ctype[low++] = 0;
    }

    /**
     * Specifies that the character argument is "ordinary"
     * in this tokenizer. It removes any special significance the
     * character has as a comment character, word component, string
     * delimiter, white space, or number character. When such a character
     * is encountered by the parser, the parser treats it as a
     * single-character token and sets {@code ttype} field to the
     * character value.
     *
     * <p>Making a line terminator character "ordinary" may interfere
     * with the ability of a {@code StreamTokenizer} to count
     * lines. The {@code lineno} method may no longer reflect
     * the presence of such terminator characters in its line count.
     *
     * @param   ch   the character.
     * @see     java.io.StreamTokenizer#ttype
     */
    public void ordinaryChar(int ch) {
        if (ch >= 0 && ch < ctype.length)
            ctype[ch] = 0;
    }

    /**
     * Specified that the character argument starts a single-line
     * comment. All characters from the comment character to the end of
     * the line are ignored by this stream tokenizer.
     *
     * <p>Any other attribute settings for the specified character are cleared.
     *
     * @param   ch   the character.
     */
    public void commentChar(int ch) {
        if (ch >= 0 && ch < ctype.length)
            ctype[ch] = CT_COMMENT;
    }

    /**
     * Specifies that matching pairs of this character delimit string
     * constants in this tokenizer.
     * <p>
     * When the {@code nextToken} method encounters a string
     * constant, the {@code ttype} field is set to the string
     * delimiter and the {@code sval} field is set to the body of
     * the string.
     * <p>
     * If a string quote character is encountered, then a string is
     * recognized, consisting of all characters after (but not including)
     * the string quote character, up to (but not including) the next
     * occurrence of that same string quote character, or a line
     * terminator, or end of file. The usual escape sequences such as
     * {@code "\u005Cn"} and {@code "\u005Ct"} are recognized and
     * converted to single characters as the string is parsed.
```

```
 *
 * <p>Any other attribute settings for the specified character are cleared.
 *
 * @param   ch   the character.
 * @see     java.io.StreamTokenizer#nextToken()
 * @see     java.io.StreamTokenizer#sval
 * @see     java.io.StreamTokenizer#ttype
 */
public void quoteChar(int ch) {
    if (ch >= 0 && ch < ctype.length)
        ctype[ch] = CT_QUOTE;
}

/**
 * Specifies that numbers should be parsed by this tokenizer. The
 * syntax table of this tokenizer is modified so that each of the twelve
 * characters:
 * <blockquote><pre>
 *      0 1 2 3 4 5 6 7 8 9 . -
 * </pre></blockquote>
 * <p>
 * has the "numeric" attribute.
 * <p>
 * When the parser encounters a word token that has the format of a
 * double precision floating-point number, it treats the token as a
 * number rather than a word, by setting the {@code ttype}
 * field to the value {@code TT_NUMBER} and putting the numeric
 * value of the token into the {@code nval} field.
 *
 * @see     java.io.StreamTokenizer#nval
 * @see     java.io.StreamTokenizer#TT_NUMBER
 * @see     java.io.StreamTokenizer#ttype
 */
public void parseNumbers() {
    for (int i = '0'; i <= '9'; i++)
        ctype[i] |= CT_DIGIT;
    ctype['.'] |= CT_DIGIT;
    ctype['-'] |= CT_DIGIT;
}

/**
 * Determines whether or not ends of line are treated as tokens.
 * If the flag argument is true, this tokenizer treats end of lines
 * as tokens; the {@code nextToken} method returns
 * {@code TT_EOL} and also sets the {@code ttype} field to
 * this value when an end of line is read.
 * <p>
 * A line is a sequence of characters ending with either a
 * carriage-return character ({@code '\u005Cr'}) or a newline
 * character ({@code '\u005Cn'}). In addition, a carriage-return
 * character followed immediately by a newline character is treated
 * as a single end-of-line token.
 * <p>
 * If the {@code flag} is false, end-of-line characters are
 * treated as white space and serve only to separate tokens.
 *
 * @param   flag   {@code true} indicates that end-of-line characters
 *                 are separate tokens; {@code false} indicates that
 *                 end-of-line characters are white space.
 * @see     java.io.StreamTokenizer#nextToken()
 * @see     java.io.StreamTokenizer#ttype
 * @see     java.io.StreamTokenizer#TT_EOL
 */
```

```
public void eolIsSignificant(boolean flag) {
    eolIsSignificantP = flag;
}

/**
 * Determines whether or not the tokenizer recognizes C-style comments.
 * If the flag argument is {@code true}, this stream tokenizer
 * recognizes C-style comments. All text between successive
 * occurrences of {@code /*} and <code>*/</code> are discarded.
 * <p>
 * If the flag argument is {@code false}, then C-style comments
 * are not treated specially.
 *
 * @param   flag   {@code true} indicates to recognize and ignore
 *                 C-style comments.
 */
public void slashStarComments(boolean flag) {
    slashStarCommentsP = flag;
}

/**
 * Determines whether or not the tokenizer recognizes C++-style comments.
 * If the flag argument is {@code true}, this stream tokenizer
 * recognizes C++-style comments. Any occurrence of two consecutive
 * slash characters ({@code '/'}) is treated as the beginning of
 * a comment that extends to the end of the line.
 * <p>
 * If the flag argument is {@code false}, then C++-style
 * comments are not treated specially.
 *
 * @param   flag   {@code true} indicates to recognize and ignore
 *                 C++-style comments.
 */
public void slashSlashComments(boolean flag) {
    slashSlashCommentsP = flag;
}

/**
 * Determines whether or not word token are automatically lowercased.
 * If the flag argument is {@code true}, then the value in the
 * {@code sval} field is lowercased whenever a word token is
 * returned (the {@code ttype} field has the
 * value {@code TT_WORD} by the {@code nextToken} method
 * of this tokenizer.
 * <p>
 * If the flag argument is {@code false}, then the
 * {@code sval} field is not modified.
 *
 * @param   fl   {@code true} indicates that all word tokens should
 *               be lowercased.
 * @see     java.io.StreamTokenizer#nextToken()
 * @see     java.io.StreamTokenizer#ttype
 * @see     java.io.StreamTokenizer#TT_WORD
 */
public void lowerCaseMode(boolean fl) {
    forceLower = fl;
}

/** Read the next character */
private int read() throws IOException {
    if (reader != null)
        return reader.read();
    else if (input != null)
```

```
        return input.read();
    else
        throw new IllegalStateException();
}

/**
 * Parses the next token from the input stream of this tokenizer.
 * The type of the next token is returned in the {@code ttype}
 * field. Additional information about the token may be in the
 * {@code nval} field or the {@code sval} field of this
 * tokenizer.
 * <p>
 * Typical clients of this
 * class first set up the syntax tables and then sit in a loop
 * calling nextToken to parse successive tokens until TT_EOF
 * is returned.
 *
 * @return     the value of the {@code ttype} field.
 * @exception  IOException  if an I/O error occurs.
 * @see        java.io.StreamTokenizer#nval
 * @see        java.io.StreamTokenizer#sval
 * @see        java.io.StreamTokenizer#ttype
 */
public int nextToken() throws IOException {
    if (pushedBack) {
        pushedBack = false;
        return ttype;
    }
    byte ct[] = ctype;
    sval = null;

    int c = peekc;
    if (c < 0)
        c = NEED_CHAR;
    if (c == SKIP_LF) {
        c = read();
        if (c < 0)
            return ttype = TT_EOF;
        if (c == '\n')
            c = NEED_CHAR;
    }
    if (c == NEED_CHAR) {
        c = read();
        if (c < 0)
            return ttype = TT_EOF;
    }
    ttype = c;              /* Just to be safe */

    /* Set peekc so that the next invocation of nextToken will read
     * another character unless peekc is reset in this invocation
     */
    peekc = NEED_CHAR;

    int ctype = c < 256 ? ct[c] : CT_ALPHA;
    while ((ctype & CT_WHITESPACE) != 0) {
        if (c == '\r') {
            LINENO++;
            if (eolIsSignificantP) {
                peekc = SKIP_LF;
                return ttype = TT_EOL;
            }
            c = read();
            if (c == '\n')
```

```
            c = read();
        } else {
            if (c == '\n') {
                LINENO++;
                if (eolIsSignificantP) {
                    return ttype = TT_EOL;
                }
            }
            c = read();
        }
        if (c < 0)
            return ttype = TT_EOF;
        ctype = c < 256 ? ct[c] : CT_ALPHA;
    }

    if ((ctype & CT_DIGIT) != 0) {
        boolean neg = false;
        if (c == '-') {
            c = read();
            if (c != '.' && (c < '0' || c > '9')) {
                peekc = c;
                return ttype = '-';
            }
            neg = true;
        }
        double v = 0;
        int decexp = 0;
        int seendot = 0;
        while (true) {
            if (c == '.' && seendot == 0)
                seendot = 1;
            else if ('0' <= c && c <= '9') {
                v = v * 10 + (c - '0');
                decexp += seendot;
            } else
                break;
            c = read();
        }
        peekc = c;
        if (decexp != 0) {
            double denom = 10;
            decexp--;
            while (decexp > 0) {
                denom *= 10;
                decexp--;
            }
            /* Do one division of a likely-to-be-more-accurate number */
            v = v / denom;
        }
        nval = neg ? -v : v;
        return ttype = TT_NUMBER;
    }

    if ((ctype & CT_ALPHA) != 0) {
        int i = 0;
        do {
            if (i >= buf.length) {
                buf = Arrays.copyOf(buf, buf.length * 2);
            }
            buf[i++] = (char) c;
            c = read();
            ctype = c < 0 ? CT_WHITESPACE : c < 256 ? ct[c] : CT_ALPHA;
        } while ((ctype & (CT_ALPHA | CT_DIGIT)) != 0);
```

```
    peekc = c;
    sval = String.copyValueOf(buf, 0, i);
    if (forceLower)
        sval = sval.toLowerCase();
    return ttype = TT_WORD;
}

if ((ctype & CT_QUOTE) != 0) {
    ttype = c;
    int i = 0;
    /* Invariants (because \Octal needs a lookahead):
     *   (i)  c contains char value
     *   (ii) d contains the lookahead
     */
    int d = read();
    while (d >= 0 && d != ttype && d != '\n' && d != '\r') {
        if (d == '\\') {
            c = read();
            int first = c;   /* To allow \377, but not \477 */
            if (c >= '0' && c <= '7') {
                c = c - '0';
                int c2 = read();
                if ('0' <= c2 && c2 <= '7') {
                    c = (c << 3) + (c2 - '0');
                    c2 = read();
                    if ('0' <= c2 && c2 <= '7' && first <= '3') {
                        c = (c << 3) + (c2 - '0');
                        d = read();
                    } else
                        d = c2;
                } else
                  d = c2;
            } else {
                switch (c) {
                case 'a':
                    c = 0x7;
                    break;
                case 'b':
                    c = '\b';
                    break;
                case 'f':
                    c = 0xC;
                    break;
                case 'n':
                    c = '\n';
                    break;
                case 'r':
                    c = '\r';
                    break;
                case 't':
                    c = '\t';
                    break;
                case 'v':
                    c = 0xB;
                    break;
                }
                d = read();
            }
        } else {
            c = d;
            d = read();
        }
        if (i >= buf.length) {
```

```
                buf = Arrays.copyOf(buf, buf.length * 2);
            }
            buf[i++] = (char)c;
        }

        /* If we broke out of the loop because we found a matching quote
         * character then arrange to read a new character next time
         * around; otherwise, save the character.
         */
        peekc = (d == ttype) ? NEED_CHAR : d;

        sval = String.copyValueOf(buf, 0, i);
        return ttype;
    }

    if (c == '/' && (slashSlashCommentsP || slashStarCommentsP)) {
        c = read();
        if (c == '*' && slashStarCommentsP) {
            int prevc = 0;
            while ((c = read()) != '/' || prevc != '*') {
                if (c == '\r') {
                    LINENO++;
                    c = read();
                    if (c == '\n') {
                        c = read();
                    }
                } else {
                    if (c == '\n') {
                        LINENO++;
                        c = read();
                    }
                }
                if (c < 0)
                    return ttype = TT_EOF;
                prevc = c;
            }
            return nextToken();
        } else if (c == '/' && slashSlashCommentsP) {
            while ((c = read()) != '\n' && c != '\r' && c >= 0);
            peekc = c;
            return nextToken();
        } else {
            /* Now see if it is still a single line comment */
            if ((ct['/'] & CT_COMMENT) != 0) {
                while ((c = read()) != '\n' && c != '\r' && c >= 0);
                peekc = c;
                return nextToken();
            } else {
                peekc = c;
                return ttype = '/';
            }
        }
    }

    if ((ctype & CT_COMMENT) != 0) {
        while ((c = read()) != '\n' && c != '\r' && c >= 0);
        peekc = c;
        return nextToken();
    }

    return ttype = c;
}
```

```
/**
 * Causes the next call to the {@code nextToken} method of this
 * tokenizer to return the current value in the {@code ttype}
 * field, and not to modify the value in the {@code nval} or
 * {@code sval} field.
 *
 * @see     java.io.StreamTokenizer#nextToken()
 * @see     java.io.StreamTokenizer#nval
 * @see     java.io.StreamTokenizer#sval
 * @see     java.io.StreamTokenizer#ttype
 */
public void pushBack() {
    if (ttype != TT_NOTHING)   /* No-op if nextToken() not called */
        pushedBack = true;
}

/**
 * Return the current line number.
 *
 * @return  the current line number of this stream tokenizer.
 */
public int lineno() {
    return LINENO;
}

/**
 * Returns the string representation of the current stream token and
 * the line number it occurs on.
 *
 * <p>The precise string returned is unspecified, although the following
 * example can be considered typical:
 *
 * <blockquote><pre>Token['a'], line 10</pre></blockquote>
 *
 * @return  a string representation of the token
 * @see     java.io.StreamTokenizer#nval
 * @see     java.io.StreamTokenizer#sval
 * @see     java.io.StreamTokenizer#ttype
 */
public String toString() {
    String ret;
    switch (ttype) {
      case TT_EOF:
        ret = "EOF";
        break;
      case TT_EOL:
        ret = "EOL";
        break;
      case TT_WORD:
        ret = sval;
        break;
      case TT_NUMBER:
        ret = "n=" + nval;
        break;
      case TT_NOTHING:
        ret = "NOTHING";
        break;
      default: {
            /*
             * ttype is the first character of either a quoted string or
             * is an ordinary character. ttype can definitely not be less
             * than 0, since those are reserved values used in the previous
             * case statements
```

```
                 */
                if (ttype < 256 &&
                    ((ctype[ttype] & CT_QUOTE) != 0)) {
                    ret = sval;
                    break;
                }

                char s[] = new char[3];
                s[0] = s[2] = '\'';
                s[1] = (char) ttype;
                ret = new String(s);
                break;
            }
        }
        return "Token[" + ret + "], line " + LINENO;
    }

}
```

StringBufferInputStream.java

```java
/*
 * Copyright (c) 1995, 2004, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * This class allows an application to create an input stream in
 * which the bytes read are supplied by the contents of a string.
 * Applications can also read bytes from a byte array by using a
 * <code>ByteArrayInputStream</code>.
 * <p>
 * Only the low eight bits of each character in the string are used by
 * this class.
 *
 * @author     Arthur van Hoff
 * @see        java.io.ByteArrayInputStream
 * @see        java.io.StringReader
 * @since      JDK1.0
 * @deprecated This class does not properly convert characters into bytes.  As
 *             of JDK 1.1, the preferred way to create a stream from a
 *             string is via the <code>StringReader</code> class.
 */
@Deprecated
public
class StringBufferInputStream extends InputStream {
    /**
     * The string from which bytes are read.
     */
    protected String buffer;

    /**
     * The index of the next character to read from the input stream buffer.
     *
     * @see        java.io.StringBufferInputStream#buffer
     */
    protected int pos;

    /**
```

```
     * The number of valid characters in the input stream buffer.
     *
     * @see        java.io.StringBufferInputStream#buffer
     */
    protected int count;

    /**
     * Creates a string input stream to read data from the specified string.
     *
     * @param      s   the underlying input buffer.
     */
    public StringBufferInputStream(String s) {
        this.buffer = s;
        count = s.length();
    }

    /**
     * Reads the next byte of data from this input stream. The value
     * byte is returned as an <code>int</code> in the range
     * <code>0</code> to <code>255</code>. If no byte is available
     * because the end of the stream has been reached, the value
     * <code>-1</code> is returned.
     * <p>
     * The <code>read</code> method of
     * <code>StringBufferInputStream</code> cannot block. It returns the
     * low eight bits of the next character in this input stream's buffer.
     *
     * @return     the next byte of data, or <code>-1</code> if the end of the
     *             stream is reached.
     */
    public synchronized int read() {
        return (pos < count) ? (buffer.charAt(pos++) & 0xFF) : -1;
    }

    /**
     * Reads up to <code>len</code> bytes of data from this input stream
     * into an array of bytes.
     * <p>
     * The <code>read</code> method of
     * <code>StringBufferInputStream</code> cannot block. It copies the
     * low eight bits from the characters in this input stream's buffer into
     * the byte array argument.
     *
     * @param      b     the buffer into which the data is read.
     * @param      off   the start offset of the data.
     * @param      len   the maximum number of bytes read.
     * @return     the total number of bytes read into the buffer, or
     *             <code>-1</code> if there is no more data because the end of
     *             the stream has been reached.
     */
    public synchronized int read(byte b[], int off, int len) {
        if (b == null) {
            throw new NullPointerException();
        } else if ((off < 0) || (off > b.length) || (len < 0) ||
                   ((off + len) > b.length) || ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
        }
        if (pos >= count) {
            return -1;
        }
        if (pos + len > count) {
            len = count - pos;
        }
```

```
        if (len <= 0) {
            return 0;
        }
        String  s = buffer;
        int cnt = len;
        while (--cnt >= 0) {
            b[off++] = (byte)s.charAt(pos++);
        }

        return len;
    }

    /**
     * Skips <code>n</code> bytes of input from this input stream. Fewer
     * bytes might be skipped if the end of the input stream is reached.
     *
     * @param      n   the number of bytes to be skipped.
     * @return     the actual number of bytes skipped.
     */
    public synchronized long skip(long n) {
        if (n < 0) {
            return 0;
        }
        if (n > count - pos) {
            n = count - pos;
        }
        pos += n;
        return n;
    }

    /**
     * Returns the number of bytes that can be read from the input
     * stream without blocking.
     *
     * @return     the value of <code>count - pos</code>, which is the
     *             number of bytes remaining to be read from the input buffer.
     */
    public synchronized int available() {
        return count - pos;
    }

    /**
     * Resets the input stream to begin reading from the first character
     * of this input stream's underlying buffer.
     */
    public synchronized void reset() {
        pos = 0;
    }
}
```

StringReader.java

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * A character stream whose source is a string.
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class StringReader extends Reader {

    private String str;
    private int length;
    private int next = 0;
    private int mark = 0;

    /**
     * Creates a new string reader.
     *
     * @param s  String providing the character stream.
     */
    public StringReader(String s) {
        this.str = s;
        this.length = s.length();
    }

    /** Check to make sure that the stream has not been closed */
    private void ensureOpen() throws IOException {
        if (str == null)
            throw new IOException("Stream closed");
    }

    /**
     * Reads a single character.
```

```
     *
     * @return     The character read, or -1 if the end of the stream has been
     *             reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read() throws IOException {
        synchronized (lock) {
            ensureOpen();
            if (next >= length)
                return -1;
            return str.charAt(next++);
        }
    }

    /**
     * Reads characters into a portion of an array.
     *
     * @param      cbuf  Destination buffer
     * @param      off   Offset at which to start writing characters
     * @param      len   Maximum number of characters to read
     *
     * @return     The number of characters read, or -1 if the end of the
     *             stream has been reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read(char cbuf[], int off, int len) throws IOException {
        synchronized (lock) {
            ensureOpen();
            if ((off < 0) || (off > cbuf.length) || (len < 0) ||
                ((off + len) > cbuf.length) || ((off + len) < 0)) {
                throw new IndexOutOfBoundsException();
            } else if (len == 0) {
                return 0;
            }
            if (next >= length)
                return -1;
            int n = Math.min(length - next, len);
            str.getChars(next, next + n, cbuf, off);
            next += n;
            return n;
        }
    }

    /**
     * Skips the specified number of characters in the stream. Returns
     * the number of characters that were skipped.
     *
     * <p>The <code>ns</code> parameter may be negative, even though the
     * <code>skip</code> method of the {@link Reader} superclass throws
     * an exception in this case. Negative values of <code>ns</code> cause the
     * stream to skip backwards. Negative return values indicate a skip
     * backwards. It is not possible to skip backwards past the beginning of
     * the string.
     *
     * <p>If the entire string has been read or skipped, then this method has
     * no effect and always returns 0.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public long skip(long ns) throws IOException {
        synchronized (lock) {
```

```
        ensureOpen();
        if (next >= length)
            return 0;
        // Bound skip by beginning and end of the source
        long n = Math.min(length - next, ns);
        n = Math.max(-next, n);
        next += n;
        return n;
    }
}

/**
 * Tells whether this stream is ready to be read.
 *
 * @return True if the next read() is guaranteed not to block for input
 *
 * @exception  IOException  If the stream is closed
 */
public boolean ready() throws IOException {
    synchronized (lock) {
    ensureOpen();
    return true;
    }
}

/**
 * Tells whether this stream supports the mark() operation, which it does.
 */
public boolean markSupported() {
    return true;
}

/**
 * Marks the present position in the stream.  Subsequent calls to reset()
 * will reposition the stream to this point.
 *
 * @param  readAheadLimit  Limit on the number of characters that may be
 *                         read while still preserving the mark.  Because
 *                         the stream's input comes from a string, there
 *                         is no actual limit, so this argument must not
 *                         be negative, but is otherwise ignored.
 *
 * @exception  IllegalArgumentException  If {@code readAheadLimit < 0}
 * @exception  IOException  If an I/O error occurs
 */
public void mark(int readAheadLimit) throws IOException {
    if (readAheadLimit < 0){
        throw new IllegalArgumentException("Read-ahead limit < 0");
    }
    synchronized (lock) {
        ensureOpen();
        mark = next;
    }
}

/**
 * Resets the stream to the most recent mark, or to the beginning of the
 * string if it has never been marked.
 *
 * @exception  IOException  If an I/O error occurs
 */
public void reset() throws IOException {
    synchronized (lock) {
```

```
            ensureOpen();
            next = mark;
        }
    }

    /**
     * Closes the stream and releases any system resources associated with
     * it. Once the stream has been closed, further read(),
     * ready(), mark(), or reset() invocations will throw an IOException.
     * Closing a previously closed stream has no effect.
     */
    public void close() {
        str = null;
    }
}
```

StringWriter.java

```
/*
 * Copyright (c) 1996, 2004, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * A character stream that collects its output in a string buffer, which can
 * then be used to construct a string.
 * <p>
 * Closing a <tt>StringWriter</tt> has no effect. The methods in this class
 * can be called after the stream has been closed without generating an
 * <tt>IOException</tt>.
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class StringWriter extends Writer {

    private StringBuffer buf;

    /**
     * Create a new string writer using the default initial string-buffer
     * size.
     */
    public StringWriter() {
        buf = new StringBuffer();
        lock = buf;
    }

    /**
     * Create a new string writer using the specified initial string-buffer
     * size.
     *
     * @param initialSize
     *        The number of <tt>char</tt> values that will fit into this buffer
     *        before it is automatically expanded
```

```
     *
     * @throws IllegalArgumentException
     *         If <tt>initialSize</tt> is negative
     */
    public StringWriter(int initialSize) {
        if (initialSize < 0) {
            throw new IllegalArgumentException("Negative buffer size");
        }
        buf = new StringBuffer(initialSize);
        lock = buf;
    }

    /**
     * Write a single character.
     */
    public void write(int c) {
        buf.append((char) c);
    }

    /**
     * Write a portion of an array of characters.
     *
     * @param  cbuf  Array of characters
     * @param  off   Offset from which to start writing characters
     * @param  len   Number of characters to write
     */
    public void write(char cbuf[], int off, int len) {
        if ((off < 0) || (off > cbuf.length) || (len < 0) ||
            ((off + len) > cbuf.length) || ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return;
        }
        buf.append(cbuf, off, len);
    }

    /**
     * Write a string.
     */
    public void write(String str) {
        buf.append(str);
    }

    /**
     * Write a portion of a string.
     *
     * @param  str  String to be written
     * @param  off  Offset from which to start writing characters
     * @param  len  Number of characters to write
     */
    public void write(String str, int off, int len)  {
        buf.append(str.substring(off, off + len));
    }

    /**
     * Appends the specified character sequence to this writer.
     *
     * <p> An invocation of this method of the form <tt>out.append(csq)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.write(csq.toString()) </pre>
     *
```

```
     * <p> Depending on the specification of <tt>toString</tt> for the
     * character sequence <tt>csq</tt>, the entire sequence may not be
     * appended. For instance, invoking the <tt>toString</tt> method of a
     * character buffer will return a subsequence whose content depends upon
     * the buffer's position and limit.
     *
     * @param  csq
     *         The character sequence to append.  If <tt>csq</tt> is
     *         <tt>null</tt>, then the four characters <tt>"null"</tt> are
     *         appended to this writer.
     *
     * @return  This writer
     *
     * @since  1.5
     */
    public StringWriter append(CharSequence csq) {
        if (csq == null)
            write("null");
        else
            write(csq.toString());
        return this;
    }

    /**
     * Appends a subsequence of the specified character sequence to this writer.
     *
     * <p> An invocation of this method of the form <tt>out.append(csq, start,
     * end)</tt> when <tt>csq</tt> is not <tt>null</tt>, behaves in
     * exactly the same way as the invocation
     *
     * <pre>
     *     out.write(csq.subSequence(start, end).toString()) </pre>
     *
     * @param  csq
     *         The character sequence from which a subsequence will be
     *         appended.  If <tt>csq</tt> is <tt>null</tt>, then characters
     *         will be appended as if <tt>csq</tt> contained the four
     *         characters <tt>"null"</tt>.
     *
     * @param  start
     *         The index of the first character in the subsequence
     *
     * @param  end
     *         The index of the character following the last character in the
     *         subsequence
     *
     * @return  This writer
     *
     * @throws  IndexOutOfBoundsException
     *          If <tt>start</tt> or <tt>end</tt> are negative, <tt>start</tt>
     *          is greater than <tt>end</tt>, or <tt>end</tt> is greater than
     *          <tt>csq.length()</tt>
     *
     * @since  1.5
     */
    public StringWriter append(CharSequence csq, int start, int end) {
        CharSequence cs = (csq == null ? "null" : csq);
        write(cs.subSequence(start, end).toString());
        return this;
    }

    /**
     * Appends the specified character to this writer.
```

```
     *
     * <p> An invocation of this method of the form <tt>out.append(c)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.write(c) </pre>
     *
     * @param  c
     *         The 16-bit character to append
     *
     * @return  This writer
     *
     * @since 1.5
     */
    public StringWriter append(char c) {
        write(c);
        return this;
    }

    /**
     * Return the buffer's current value as a string.
     */
    public String toString() {
        return buf.toString();
    }

    /**
     * Return the string buffer itself.
     *
     * @return StringBuffer holding the current buffer value.
     */
    public StringBuffer getBuffer() {
        return buf;
    }

    /**
     * Flush the stream.
     */
    public void flush() {
    }

    /**
     * Closing a <tt>StringWriter</tt> has no effect. The methods in this
     * class can be called after the stream has been closed without generating
     * an <tt>IOException</tt>.
     */
    public void close() throws IOException {
    }

}
```

SyncFailedException.java

```
/*
 * Copyright (c) 1996, 2008, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Signals that a sync operation has failed.
 *
 * @author  Ken Arnold
 * @see     java.io.FileDescriptor#sync
 * @see     java.io.IOException
 * @since   JDK1.1
 */
public class SyncFailedException extends IOException {
    private static final long serialVersionUID = -2353342684412443330L;

    /**
     * Constructs an SyncFailedException with a detail message.
     * A detail message is a String that describes this particular exception.
     *
     * @param desc  a String describing the exception.
     */
    public SyncFailedException(String desc) {
        super(desc);
    }
}
```

UncheckedIOException.java

```
/*
 * Copyright (c) 2012, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
package java.io;

import java.util.Objects;

/**
 * Wraps an {@link IOException} with an unchecked exception.
 *
 * @since   1.8
 */
public class UncheckedIOException extends RuntimeException {
    private static final long serialVersionUID = -8134305061645241065L;

    /**
     * Constructs an instance of this class.
     *
     * @param   message
     *          the detail message, can be null
     * @param   cause
     *          the {@code IOException}
     *
     * @throws  NullPointerException
     *          if the cause is {@code null}
     */
    public UncheckedIOException(String message, IOException cause) {
        super(message, Objects.requireNonNull(cause));
    }

    /**
     * Constructs an instance of this class.
     *
     * @param   cause
     *          the {@code IOException}
     *
     * @throws  NullPointerException
     *          if the cause is {@code null}
     */
```

```
    public UncheckedIOException(IOException cause) {
        super(Objects.requireNonNull(cause));
    }

    /**
     * Returns the cause of this exception.
     *
     * @return  the {@code IOException} which is the cause of this exception.
     */
    @Override
    public IOException getCause() {
        return (IOException) super.getCause();
    }

    /**
     * Called to read the object from a stream.
     *
     * @throws  InvalidObjectException
     *          if the object is invalid or has a cause that is not
     *          an {@code IOException}
     */
    private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException
    {
        s.defaultReadObject();
        Throwable cause = super.getCause();
        if (!(cause instanceof IOException))
            throw new InvalidObjectException("Cause must be an IOException");
    }
}
```

UnsupportedEncodingException.java

```java
/*
 * Copyright (c) 1996, 2008, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
package java.io;

/**
 * The Character Encoding is not supported.
 *
 * @author  Asmus Freytag
 * @since   JDK1.1
 */
public class UnsupportedEncodingException
    extends IOException
{
    private static final long serialVersionUID = -4274276298326136670L;

    /**
     * Constructs an UnsupportedEncodingException without a detail message.
     */
    public UnsupportedEncodingException() {
        super();
    }

    /**
     * Constructs an UnsupportedEncodingException with a detail message.
     * @param s Describes the reason for the exception.
     */
    public UnsupportedEncodingException(String s) {
        super(s);
    }
}
```

UTFDataFormatException.java

```
/*
 * Copyright (c) 1995, 2008, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Signals that a malformed string in
 * <a href="DataInput.html#modified-utf-8">modified UTF-8</a>
 * format has been read in a data
 * input stream or by any class that implements the data input
 * interface.
 * See the
 * <a href="DataInput.html#modified-utf-8"><code>DataInput</code></a>
 * class description for the format in
 * which modified UTF-8 strings are read and written.
 *
 * @author  Frank Yellin
 * @see     java.io.DataInput
 * @see     java.io.DataInputStream#readUTF(java.io.DataInput)
 * @see     java.io.IOException
 * @since   JDK1.0
 */
public
class UTFDataFormatException extends IOException {
    private static final long serialVersionUID = 420743449228280612L;

    /**
     * Constructs a <code>UTFDataFormatException</code> with
     * <code>null</code> as its error detail message.
     */
    public UTFDataFormatException() {
        super();
    }

    /**
     * Constructs a <code>UTFDataFormatException</code> with the
     * specified detail message. The string <code>s</code> can be
     * retrieved later by the
```

```
     * <code>{@link java.lang.Throwable#getMessage}</code>
     * method of class <code>java.lang.Throwable</code>.
     *
     * @param   s   the detail message.
     */
    public UTFDataFormatException(String s) {
        super(s);
    }
}
```

WinNTFileSystem.java

```
/*
 * Copyright (c) 2001, 2012, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.security.AccessController;
import java.util.Locale;
import sun.security.action.GetPropertyAction;

/**
 * Unicode-aware FileSystem for Windows NT/2000.
 *
 * @author Konstantin Kladko
 * @since 1.4
 */
class WinNTFileSystem extends FileSystem {

    private final char slash;
    private final char altSlash;
    private final char semicolon;

    public WinNTFileSystem() {
        slash = AccessController.doPrivileged(
            new GetPropertyAction("file.separator")).charAt(0);
        semicolon = AccessController.doPrivileged(
            new GetPropertyAction("path.separator")).charAt(0);
        altSlash = (this.slash == '\\') ? '/' : '\\';
    }

    private boolean isSlash(char c) {
        return (c == '\\') || (c == '/');
    }

    private boolean isLetter(char c) {
        return ((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z'));
    }

    private String slashify(String p) {
```

```
    if ((p.length() > 0) && (p.charAt(0) != slash)) return slash + p;
    else return p;
}

/* -- Normalization and construction -- */

@Override
public char getSeparator() {
    return slash;
}

@Override
public char getPathSeparator() {
    return semicolon;
}

/* Check that the given pathname is normal.  If not, invoke the real
   normalizer on the part of the pathname that requires normalization.
   This way we iterate through the whole pathname string only once. */
@Override
public String normalize(String path) {
    int n = path.length();
    char slash = this.slash;
    char altSlash = this.altSlash;
    char prev = 0;
    for (int i = 0; i < n; i++) {
        char c = path.charAt(i);
        if (c == altSlash)
            return normalize(path, n, (prev == slash) ? i - 1 : i);
        if ((c == slash) && (prev == slash) && (i > 1))
            return normalize(path, n, i - 1);
        if ((c == ':') && (i > 1))
            return normalize(path, n, 0);
        prev = c;
    }
    if (prev == slash) return normalize(path, n, n - 1);
    return path;
}

/* Normalize the given pathname, whose length is len, starting at the given
   offset; everything before this offset is already normal. */
private String normalize(String path, int len, int off) {
    if (len == 0) return path;
    if (off < 3) off = 0;   /* Avoid fencepost cases with UNC pathnames */
    int src;
    char slash = this.slash;
    StringBuffer sb = new StringBuffer(len);

    if (off == 0) {
        /* Complete normalization, including prefix */
        src = normalizePrefix(path, len, sb);
    } else {
        /* Partial normalization */
        src = off;
        sb.append(path.substring(0, off));
    }

    /* Remove redundant slashes from the remainder of the path, forcing all
       slashes into the preferred slash */
    while (src < len) {
        char c = path.charAt(src++);
        if (isSlash(c)) {
            while ((src < len) && isSlash(path.charAt(src))) src++;
```

```
            if (src == len) {
                /* Check for trailing separator */
                int sn = sb.length();
                if ((sn == 2) && (sb.charAt(1) == ':')) {
                    /* "z:\\" */
                    sb.append(slash);
                    break;
                }
                if (sn == 0) {
                    /* "\\" */
                    sb.append(slash);
                    break;
                }
                if ((sn == 1) && (isSlash(sb.charAt(0)))) {
                    /* "\\\\" is not collapsed to "\\" because "\\\\" marks
                       the beginning of a UNC pathname.  Even though it is
                       not, by itself, a valid UNC pathname, we leave it as
                       is in order to be consistent with the win32 APIs,
                       which treat this case as an invalid UNC pathname
                       rather than as an alias for the root directory of
                       the current drive. */
                    sb.append(slash);
                    break;
                }
                /* Path does not denote a root directory, so do not append
                   trailing slash */
                break;
            } else {
                sb.append(slash);
            }
        } else {
            sb.append(c);
        }
    }

    String rv = sb.toString();
    return rv;
}

/* A normal Win32 pathname contains no duplicate slashes, except possibly
   for a UNC prefix, and does not end with a slash.  It may be the empty
   string.  Normalized Win32 pathnames have the convenient property that
   the length of the prefix almost uniquely identifies the type of the path
   and whether it is absolute or relative:
       0  relative to both drive and directory
       1  drive-relative (begins with '\\')
       2  absolute UNC (if first char is '\\'),
            else directory-relative (has form "z:foo")
       3  absolute local pathname (begins with "z:\\")
 */
private int normalizePrefix(String path, int len, StringBuffer sb) {
    int src = 0;
    while ((src < len) && isSlash(path.charAt(src))) src++;
    char c;
    if ((len - src >= 2)
        && isLetter(c = path.charAt(src))
        && path.charAt(src + 1) == ':') {
        /* Remove leading slashes if followed by drive specifier.
           This hack is necessary to support file URLs containing drive
           specifiers (e.g., "file://c:/path").  As a side effect,
           "/c:/path" can be used as an alternative to "c:/path". */
        sb.append(c);
        sb.append(':');
```

```
            src += 2;
        } else {
            src = 0;
            if ((len >= 2)
                && isSlash(path.charAt(0))
                && isSlash(path.charAt(1))) {
                /* UNC pathname: Retain first slash; leave src pointed at
                   second slash so that further slashes will be collapsed
                   into the second slash.  The result will be a pathname
                   beginning with "\\\\" followed (most likely) by a host
                   name. */
                src = 1;
                sb.append(slash);
            }
        }
        return src;
    }

    @Override
    public int prefixLength(String path) {
        char slash = this.slash;
        int n = path.length();
        if (n == 0) return 0;
        char c0 = path.charAt(0);
        char c1 = (n > 1) ? path.charAt(1) : 0;
        if (c0 == slash) {
            if (c1 == slash) return 2;  /* Absolute UNC pathname "\\\\foo" */
            return 1;                   /* Drive-relative "\\foo" */
        }
        if (isLetter(c0) && (c1 == ':')) {
            if ((n > 2) && (path.charAt(2) == slash))
                return 3;               /* Absolute local pathname "z:\\foo" */
            return 2;                   /* Directory-relative "z:foo" */
        }
        return 0;                       /* Completely relative */
    }

    @Override
    public String resolve(String parent, String child) {
        int pn = parent.length();
        if (pn == 0) return child;
        int cn = child.length();
        if (cn == 0) return parent;

        String c = child;
        int childStart = 0;
        int parentEnd = pn;

        if ((cn > 1) && (c.charAt(0) == slash)) {
            if (c.charAt(1) == slash) {
                /* Drop prefix when child is a UNC pathname */
                childStart = 2;
            } else {
                /* Drop prefix when child is drive-relative */
                childStart = 1;

            }
            if (cn == childStart) { // Child is double slash
                if (parent.charAt(pn - 1) == slash)
                    return parent.substring(0, pn - 1);
                return parent;
            }
        }
```

```
    if (parent.charAt(pn - 1) == slash)
        parentEnd--;

    int strlen = parentEnd + cn - childStart;
    char[] theChars = null;
    if (child.charAt(childStart) == slash) {
        theChars = new char[strlen];
        parent.getChars(0, parentEnd, theChars, 0);
        child.getChars(childStart, cn, theChars, parentEnd);
    } else {
        theChars = new char[strlen + 1];
        parent.getChars(0, parentEnd, theChars, 0);
        theChars[parentEnd] = slash;
        child.getChars(childStart, cn, theChars, parentEnd + 1);
    }
    return new String(theChars);
}

@Override
public String getDefaultParent() {
    return ("" + slash);
}

@Override
public String fromURIPath(String path) {
    String p = path;
    if ((p.length() > 2) && (p.charAt(2) == ':')) {
        // "/c:/foo" --> "c:/foo"
        p = p.substring(1);
        // "c:/foo/" --> "c:/foo", but "c:/" --> "c:/"
        if ((p.length() > 3) && p.endsWith("/"))
            p = p.substring(0, p.length() - 1);
    } else if ((p.length() > 1) && p.endsWith("/")) {
        // "/foo/" --> "/foo"
        p = p.substring(0, p.length() - 1);
    }
    return p;
}

/* -- Path operations -- */

@Override
public boolean isAbsolute(File f) {
    int pl = f.getPrefixLength();
    return (((pl == 2) && (f.getPath().charAt(0) == slash))
            || (pl == 3));
}

@Override
public String resolve(File f) {
    String path = f.getPath();
    int pl = f.getPrefixLength();
    if ((pl == 2) && (path.charAt(0) == slash))
        return path;                        /* UNC */
    if (pl == 3)
        return path;                        /* Absolute local */
    if (pl == 0)
        return getUserPath() + slashify(path); /* Completely relative */
    if (pl == 1) {                          /* Drive-relative */
        String up = getUserPath();
        String ud = getDrive(up);
        if (ud != null) return ud + path;
```

```
            return up + path;                   /* User dir is a UNC path */
        }
        if (pl == 2) {                          /* Directory-relative */
            String up = getUserPath();
            String ud = getDrive(up);
            if ((ud != null) && path.startsWith(ud))
                return up + slashify(path.substring(2));
            char drive = path.charAt(0);
            String dir = getDriveDirectory(drive);
            String np;
            if (dir != null) {
                /* When resolving a directory-relative path that refers to a
                   drive other than the current drive, insist that the caller
                   have read permission on the result */
                String p = drive + (':' + dir + slashify(path.substring(2)));
                SecurityManager security = System.getSecurityManager();
                try {
                    if (security != null) security.checkRead(p);
                } catch (SecurityException x) {
                    /* Don't disclose the drive's directory in the exception */
                    throw new SecurityException("Cannot resolve path " + path);
                }
                return p;
            }
            return drive + ":" + slashify(path.substring(2)); /* fake it */
        }
        throw new InternalError("Unresolvable path: " + path);
    }

    private String getUserPath() {
        /* For both compatibility and security,
           we must look this up every time */
        return normalize(System.getProperty("user.dir"));
    }

    private String getDrive(String path) {
        int pl = prefixLength(path);
        return (pl == 3) ? path.substring(0, 2) : null;
    }

    private static String[] driveDirCache = new String[26];

    private static int driveIndex(char d) {
        if ((d >= 'a') && (d <= 'z')) return d - 'a';
        if ((d >= 'A') && (d <= 'Z')) return d - 'A';
        return -1;
    }

    private native String getDriveDirectory(int drive);

    private String getDriveDirectory(char drive) {
        int i = driveIndex(drive);
        if (i < 0) return null;
        String s = driveDirCache[i];
        if (s != null) return s;
        s = getDriveDirectory(i + 1);
        driveDirCache[i] = s;
        return s;
    }

    // Caches for canonicalization results to improve startup performance.
    // The first cache handles repeated canonicalizations of the same path
    // name. The prefix cache handles repeated canonicalizations within the
```

```
// same directory, and must not create results differing from the true
// canonicalization algorithm in canonicalize_md.c. For this reason the
// prefix cache is conservative and is not used for complex path names.
private ExpiringCache cache       = new ExpiringCache();
private ExpiringCache prefixCache = new ExpiringCache();

@Override
public String canonicalize(String path) throws IOException {
    // If path is a drive letter only then skip canonicalization
    int len = path.length();
    if ((len == 2) &&
        (isLetter(path.charAt(0))) &&
        (path.charAt(1) == ':')) {
        char c = path.charAt(0);
        if ((c >= 'A') && (c <= 'Z'))
            return path;
        return "" + ((char) (c-32)) + ':';
    } else if ((len == 3) &&
               (isLetter(path.charAt(0))) &&
               (path.charAt(1) == ':') &&
               (path.charAt(2) == '\\')) {
        char c = path.charAt(0);
        if ((c >= 'A') && (c <= 'Z'))
            return path;
        return "" + ((char) (c-32)) + ':' + '\\';
    }
    if (!useCanonCaches) {
        return canonicalize0(path);
    } else {
        String res = cache.get(path);
        if (res == null) {
            String dir = null;
            String resDir = null;
            if (useCanonPrefixCache) {
                dir = parentOrNull(path);
                if (dir != null) {
                    resDir = prefixCache.get(dir);
                    if (resDir != null) {
                        /*
                         * Hit only in prefix cache; full path is canonical,
                         * but we need to get the canonical name of the file
                         * in this directory to get the appropriate
                         * capitalization
                         */
                        String filename = path.substring(1 + dir.length());
                        res = canonicalizeWithPrefix(resDir, filename);
                        cache.put(dir + File.separatorChar + filename, res);
                    }
                }
            }
            if (res == null) {
                res = canonicalize0(path);
                cache.put(path, res);
                if (useCanonPrefixCache && dir != null) {
                    resDir = parentOrNull(res);
                    if (resDir != null) {
                        File f = new File(res);
                        if (f.exists() && !f.isDirectory()) {
                            prefixCache.put(dir, resDir);
                        }
                    }
                }
            }
```

```
        }
        return res;
    }
}

private native String canonicalize0(String path)
        throws IOException;

private String canonicalizeWithPrefix(String canonicalPrefix,
        String filename) throws IOException
{
    return canonicalizeWithPrefix0(canonicalPrefix,
            canonicalPrefix + File.separatorChar + filename);
}

// Run the canonicalization operation assuming that the prefix
// (everything up to the last filename) is canonical; just gets
// the canonical name of the last element of the path
private native String canonicalizeWithPrefix0(String canonicalPrefix,
        String pathWithCanonicalPrefix)
        throws IOException;

// Best-effort attempt to get parent of this path; used for
// optimization of filename canonicalization. This must return null for
// any cases where the code in canonicalize_md.c would throw an
// exception or otherwise deal with non-simple pathnames like handling
// of "." and "..". It may conservatively return null in other
// situations as well. Returning null will cause the underlying
// (expensive) canonicalization routine to be called.
private static String parentOrNull(String path) {
    if (path == null) return null;
    char sep = File.separatorChar;
    char altSep = '/';
    int last = path.length() - 1;
    int idx = last;
    int adjacentDots = 0;
    int nonDotCount = 0;
    while (idx > 0) {
        char c = path.charAt(idx);
        if (c == '.') {
            if (++adjacentDots >= 2) {
                // Punt on pathnames containing . and ..
                return null;
            }
            if (nonDotCount == 0) {
                // Punt on pathnames ending in a .
                return null;
            }
        } else if (c == sep) {
            if (adjacentDots == 1 && nonDotCount == 0) {
                // Punt on pathnames containing . and ..
                return null;
            }
            if (idx == 0 ||
                idx >= last - 1 ||
                path.charAt(idx - 1) == sep ||
                path.charAt(idx - 1) == altSep) {
                // Punt on pathnames containing adjacent slashes
                // toward the end
                return null;
            }
            return path.substring(0, idx);
        } else if (c == altSep) {
```

```
            // Punt on pathnames containing both backward and
            // forward slashes
            return null;
        } else if (c == '*' || c == '?') {
            // Punt on pathnames containing wildcards
            return null;
        } else {
            ++nonDotCount;
            adjacentDots = 0;
        }
        --idx;
    }
    return null;
}

/* -- Attribute accessors -- */

@Override
public native int getBooleanAttributes(File f);

@Override
public native boolean checkAccess(File f, int access);

@Override
public native long getLastModifiedTime(File f);

@Override
public native long getLength(File f);

@Override
public native boolean setPermission(File f, int access, boolean enable,
        boolean owneronly);

/* -- File operations -- */

@Override
public native boolean createFileExclusively(String path)
        throws IOException;

@Override
public native String[] list(File f);

@Override
public native boolean createDirectory(File f);

@Override
public native boolean setLastModifiedTime(File f, long time);

@Override
public native boolean setReadOnly(File f);

@Override
public boolean delete(File f) {
    // Keep canonicalization caches in sync after file deletion
    // and renaming operations. Could be more clever than this
    // (i.e., only remove/update affected entries) but probably
    // not worth it since these entries expire after 30 seconds
    // anyway.
    cache.clear();
    prefixCache.clear();
    return delete0(f);
}
```

```
private native boolean delete0(File f);

@Override
public boolean rename(File f1, File f2) {
    // Keep canonicalization caches in sync after file deletion
    // and renaming operations. Could be more clever than this
    // (i.e., only remove/update affected entries) but probably
    // not worth it since these entries expire after 30 seconds
    // anyway.
    cache.clear();
    prefixCache.clear();
    return rename0(f1, f2);
}

private native boolean rename0(File f1, File f2);

/* -- Filesystem interface -- */

@Override
public File[] listRoots() {
    int ds = listRoots0();
    int n = 0;
    for (int i = 0; i < 26; i++) {
        if (((ds >> i) & 1) != 0) {
            if (!access((char)('A' + i) + ":" + slash))
                ds &= ~(1 << i);
            else
                n++;
        }
    }
    File[] fs = new File[n];
    int j = 0;
    char slash = this.slash;
    for (int i = 0; i < 26; i++) {
        if (((ds >> i) & 1) != 0)
            fs[j++] = new File((char)('A' + i) + ":" + slash);
    }
    return fs;
}

private static native int listRoots0();

private boolean access(String path) {
    try {
        SecurityManager security = System.getSecurityManager();
        if (security != null) security.checkRead(path);
        return true;
    } catch (SecurityException x) {
        return false;
    }
}

/* -- Disk usage -- */

@Override
public long getSpace(File f, int t) {
    if (f.exists()) {
        return getSpace0(f, t);
    }
    return 0;
}

private native long getSpace0(File f, int t);
```

```
    /* -- Basic infrastructure -- */

    @Override
    public int compare(File f1, File f2) {
        return f1.getPath().compareToIgnoreCase(f2.getPath());
    }

    @Override
    public int hashCode(File f) {
        /* Could make this more efficient: String.hashCodeIgnoreCase */
        return f.getPath().toLowerCase(Locale.ENGLISH).hashCode() ^ 1234321;
    }

    private static native void initIDs();

    static {
        initIDs();
    }
}
```

WriteAbortedException.java

```
/*
 * Copyright (c) 1996, 2005, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

/**
 * Signals that one of the ObjectStreamExceptions was thrown during a
 * write operation.  Thrown during a read operation when one of the
 * ObjectStreamExceptions was thrown during a write operation.  The
 * exception that terminated the write can be found in the detail
 * field. The stream is reset to it's initial state and all references
 * to objects already deserialized are discarded.
 *
 * <p>As of release 1.4, this exception has been retrofitted to conform to
 * the general purpose exception-chaining mechanism.  The "exception causing
 * the abort" that is provided at construction time and
 * accessed via the public {@link #detail} field is now known as the
 * <i>cause</i>, and may be accessed via the {@link Throwable#getCause()}
 * method, as well as the aforementioned "legacy field."
 *
 * @author  unascribed
 * @since   JDK1.1
 */
public class WriteAbortedException extends ObjectStreamException {
    private static final long serialVersionUID = -3326426625597282442L;

    /**
     * Exception that was caught while writing the ObjectStream.
     *
     * <p>This field predates the general-purpose exception chaining facility.
     * The {@link Throwable#getCause()} method is now the preferred means of
     * obtaining this information.
     *
     * @serial
     */
    public Exception detail;

    /**
```

```
     * Constructs a WriteAbortedException with a string describing
     * the exception and the exception causing the abort.
     * @param s   String describing the exception.
     * @param ex  Exception causing the abort.
     */
    public WriteAbortedException(String s, Exception ex) {
        super(s);
        initCause(null);  // Disallow subsequent initCause
        detail = ex;
    }

    /**
     * Produce the message and include the message from the nested
     * exception, if there is one.
     */
    public String getMessage() {
        if (detail == null)
            return super.getMessage();
        else
            return super.getMessage() + "; " + detail.toString();
    }

    /**
     * Returns the exception that terminated the operation (the <i>cause</i>).
     *
     * @return  the exception that terminated the operation (the <i>cause</i>),
     *          which may be null.
     * @since   1.4
     */
    public Throwable getCause() {
        return detail;
    }
}
```

Writer.java

```java
/*
 * Copyright (c) 1996, 2011, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Abstract class for writing to character streams.  The only methods that a
 * subclass must implement are write(char[], int, int), flush(), and close().
 * Most subclasses, however, will override some of the methods defined here in
 * order to provide higher efficiency, additional functionality, or both.
 *
 * @see Writer
 * @see   BufferedWriter
 * @see   CharArrayWriter
 * @see   FilterWriter
 * @see   OutputStreamWriter
 * @see     FileWriter
 * @see   PipedWriter
 * @see   PrintWriter
 * @see   StringWriter
 * @see Reader
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public abstract class Writer implements Appendable, Closeable, Flushable {

    /**
     * Temporary buffer used to hold writes of strings and single characters
     */
    private char[] writeBuffer;

    /**
     * Size of writeBuffer, must be >= 1
     */
    private static final int WRITE_BUFFER_SIZE = 1024;
```

```
/**
 * The object used to synchronize operations on this stream.  For
 * efficiency, a character-stream object may use an object other than
 * itself to protect critical sections.  A subclass should therefore use
 * the object in this field rather than <tt>this</tt> or a synchronized
 * method.
 */
protected Object lock;

/**
 * Creates a new character-stream writer whose critical sections will
 * synchronize on the writer itself.
 */
protected Writer() {
    this.lock = this;
}

/**
 * Creates a new character-stream writer whose critical sections will
 * synchronize on the given object.
 *
 * @param  lock
 *         Object to synchronize on
 */
protected Writer(Object lock) {
    if (lock == null) {
        throw new NullPointerException();
    }
    this.lock = lock;
}

/**
 * Writes a single character.  The character to be written is contained in
 * the 16 low-order bits of the given integer value; the 16 high-order bits
 * are ignored.
 *
 * <p> Subclasses that intend to support efficient single-character output
 * should override this method.
 *
 * @param  c
 *         int specifying a character to be written
 *
 * @throws  IOException
 *          If an I/O error occurs
 */
public void write(int c) throws IOException {
    synchronized (lock) {
        if (writeBuffer == null){
            writeBuffer = new char[WRITE_BUFFER_SIZE];
        }
        writeBuffer[0] = (char) c;
        write(writeBuffer, 0, 1);
    }
}

/**
 * Writes an array of characters.
 *
 * @param  cbuf
 *         Array of characters to be written
 *
 * @throws  IOException
```

```
     *          If an I/O error occurs
     */
    public void write(char cbuf[]) throws IOException {
        write(cbuf, 0, cbuf.length);
    }

    /**
     * Writes a portion of an array of characters.
     *
     * @param  cbuf
     *         Array of characters
     *
     * @param  off
     *         Offset from which to start writing characters
     *
     * @param  len
     *         Number of characters to write
     *
     * @throws  IOException
     *          If an I/O error occurs
     */
    abstract public void write(char cbuf[], int off, int len) throws IOException;

    /**
     * Writes a string.
     *
     * @param  str
     *         String to be written
     *
     * @throws  IOException
     *          If an I/O error occurs
     */
    public void write(String str) throws IOException {
        write(str, 0, str.length());
    }

    /**
     * Writes a portion of a string.
     *
     * @param  str
     *         A String
     *
     * @param  off
     *         Offset from which to start writing characters
     *
     * @param  len
     *         Number of characters to write
     *
     * @throws  IndexOutOfBoundsException
     *          If <tt>off</tt> is negative, or <tt>len</tt> is negative,
     *          or <tt>off+len</tt> is negative or greater than the length
     *          of the given string
     *
     * @throws  IOException
     *          If an I/O error occurs
     */
    public void write(String str, int off, int len) throws IOException {
        synchronized (lock) {
            char cbuf[];
            if (len <= WRITE_BUFFER_SIZE) {
                if (writeBuffer == null) {
                    writeBuffer = new char[WRITE_BUFFER_SIZE];
                }
```

```
            cbuf = writeBuffer;
        } else {    // Don't permanently allocate very large buffers.
            cbuf = new char[len];
        }
        str.getChars(off, (off + len), cbuf, 0);
        write(cbuf, 0, len);
    }
}

/**
 * Appends the specified character sequence to this writer.
 *
 * <p> An invocation of this method of the form <tt>out.append(csq)</tt>
 * behaves in exactly the same way as the invocation
 *
 * <pre>
 *     out.write(csq.toString()) </pre>
 *
 * <p> Depending on the specification of <tt>toString</tt> for the
 * character sequence <tt>csq</tt>, the entire sequence may not be
 * appended. For instance, invoking the <tt>toString</tt> method of a
 * character buffer will return a subsequence whose content depends upon
 * the buffer's position and limit.
 *
 * @param  csq
 *         The character sequence to append.  If <tt>csq</tt> is
 *         <tt>null</tt>, then the four characters <tt>"null"</tt> are
 *         appended to this writer.
 *
 * @return  This writer
 *
 * @throws  IOException
 *          If an I/O error occurs
 *
 * @since  1.5
 */
public Writer append(CharSequence csq) throws IOException {
    if (csq == null)
        write("null");
    else
        write(csq.toString());
    return this;
}

/**
 * Appends a subsequence of the specified character sequence to this writer.
 * <tt>Appendable</tt>.
 *
 * <p> An invocation of this method of the form <tt>out.append(csq, start,
 * end)</tt> when <tt>csq</tt> is not <tt>null</tt> behaves in exactly the
 * same way as the invocation
 *
 * <pre>
 *     out.write(csq.subSequence(start, end).toString()) </pre>
 *
 * @param  csq
 *         The character sequence from which a subsequence will be
 *         appended.  If <tt>csq</tt> is <tt>null</tt>, then characters
 *         will be appended as if <tt>csq</tt> contained the four
 *         characters <tt>"null"</tt>.
 *
 * @param  start
 *         The index of the first character in the subsequence
```

```
     *
     * @param  end
     *         The index of the character following the last character in the
     *         subsequence
     *
     * @return  This writer
     *
     * @throws  IndexOutOfBoundsException
     *          If <tt>start</tt> or <tt>end</tt> are negative, <tt>start</tt>
     *          is greater than <tt>end</tt>, or <tt>end</tt> is greater than
     *          <tt>csq.length()</tt>
     *
     * @throws  IOException
     *          If an I/O error occurs
     *
     * @since  1.5
     */
    public Writer append(CharSequence csq, int start, int end) throws IOException {
        CharSequence cs = (csq == null ? "null" : csq);
        write(cs.subSequence(start, end).toString());
        return this;
    }

    /**
     * Appends the specified character to this writer.
     *
     * <p> An invocation of this method of the form <tt>out.append(c)</tt>
     * behaves in exactly the same way as the invocation
     *
     * <pre>
     *     out.write(c) </pre>
     *
     * @param  c
     *         The 16-bit character to append
     *
     * @return  This writer
     *
     * @throws  IOException
     *          If an I/O error occurs
     *
     * @since 1.5
     */
    public Writer append(char c) throws IOException {
        write(c);
        return this;
    }

    /**
     * Flushes the stream.  If the stream has saved any characters from the
     * various write() methods in a buffer, write them immediately to their
     * intended destination.  Then, if that destination is another character or
     * byte stream, flush it.  Thus one flush() invocation will flush all the
     * buffers in a chain of Writers and OutputStreams.
     *
     * <p> If the intended destination of this stream is an abstraction provided
     * by the underlying operating system, for example a file, then flushing the
     * stream guarantees only that bytes previously written to the stream are
     * passed to the operating system for writing; it does not guarantee that
     * they are actually written to a physical device such as a disk drive.
     *
     * @throws  IOException
     *          If an I/O error occurs
     */
```

```
    abstract public void flush() throws IOException;

    /**
     * Closes the stream, flushing it first. Once the stream has been closed,
     * further write() or flush() invocations will cause an IOException to be
     * thrown. Closing a previously closed stream has no effect.
     *
     * @throws  IOException
     *          If an I/O error occurs
     */
    abstract public void close() throws IOException;

}
```