abstract class InputStream {
    // Nasko: the actual class in the Java libraries has additional
    // code - for example, to handle things like "end of data in input
    // stream", "I/O error", etc. For simplicity, I removed that code.

    // --- int read(): Reads the next byte of data from the input
    // stream. A subclass must provide an implementation of this
    // method. Returns the next byte of data.
    public abstract byte read();

    // --- int read(byte[]): Reads some number of bytes from the input
    // stream and stores them into the buffer array b. The number of
    // bytes actually read is returned as an integer. If the length of
    // b is zero, then no bytes are read. Otherwise, the first byte
    // read is stored into element b[0], the next one into b[1], and
    // so on. Parameters: b - the buffer into which the data is read.
    public void read(byte[] ba) {
        if (ba.length == 0) return;
        ba[0] = this.read();
        // ... some code here
    }

    // --- int skip(int): Skips over and discards n bytes of data from
    // this input stream. The skip method of InputStream creates a
    // byte array and then repeatedly reads into it until n bytes have
    // been read. Subclasses are encouraged to provide a more
    // efficient implementation of this method.
    public void skip(int n) {
        byte[] data = new byte[n]; // create a new array of n bytes
        this.read(data);
    }

    // --- void close(): Closes this input stream and releases any
    // system resources associated with the stream. The close method
    // of InputStream does nothing.
    public void close() {
        // ... some code here
    }
}

// -----------------------------------------

class FileInputStream extends InputStream {
    public FileInputStream(String name) {
        SecurityManager security = System.getSecurityManager();
        if (security != null) { // checks if file reading is allowed
            security.checkRead(name);
        }
        this.fd = new FileDescriptor();
        this.open(name);
    }

    public byte read() {
        byte x = this.buf[pos];
        this.pos++;
        return x;
    }

    public void read(byte[] ba) {
        // ... some code here
    }

    public void skip(int n) {
        // ... some code here
    }

    public void close() {
        // ... some code here
    }

    private FileDescriptor fd;
}

// --------------------------------------------------------
// Implementation of an input stream using a byte array. A
// ByteArrayInputStream 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 read method.

class ByteArrayInputStream extends InputStream {
    private byte[] buf;
    private int pos;
    private int count;

    public ByteArrayInputStream(byte[] ba) {
        this.buf = ba;
        this.pos = 0;
        this.count = ba.length;
    }

    public byte read() {
        byte x = this.buf[pos];
        this.pos++;
        return x;
    }

    public void read(byte[] ba) {
        // ... some code here
    }

    public void skip(int n) {
        // ... some code here
    }

    public void close() {
        // ... some code here
    }

    private FileDescriptor fd;

    // --------------------------------------------------------
    // System is a library class with a static method getSecurityManager()
    // SecurityManager security = System.getSecurityManager();
    // if (security != null) {
    //     security.checkRead(name); // checks if file reading is allowed
    // }
    // this.fd = new FileDescriptor();
    // this.open(name);
    // }
    // private void open(String name) {
    //     // ... some code here
    // }
    // public byte read() {
    //     // ... some code here
    // }
// --- void skip(int): Skips n bytes of input from this input
// stream. [Nasko: note that here we could have used the method
// from the superclass; however, for efficiency reasons we can use
// a specialized version that works better given the specific
// structure of this subclass.]
public void skip(int n) {
    this.pos = this.pos + n;
}

// ------------------------------------------
class FilterInputStream extends InputStream {

protected InputStream in;

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

public byte read() { return this.in.read(); }

public void read(byte[] ba) { this.in.read(ba); }

public void skip(int n) { this.in.skip(n); }

public void close() { this.in.close(); }
}

// -----------------------------------------------------------------
// A PushbackInputStream 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
// characters; 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.

class PushbackInputStream extends FilterInputStream {

    // Pushback buffer
    private byte[] buf;

    // The position within the pushback buffer from which the next
    // byte will be read. When the buffer is empty, pos is equal to
    // buf.length; when the buffer is full, pos is equal to zero.
    private int pos;

    // Constructor: Creates a PushbackInputStream with a pushback
    // buffer of the specified size, and saves its argument, the input
    // stream in, for later use. Initially, there is no pushed-back
    // byte. Parameters: i - &...