Chapter3Sending and Receiving Data

3.1 Encoding InformationLet’s first consider the question of how simple values such as ints, longs, chars, and Strings can be sent and received via sockets. We have seen that bytes of information can be transmitted through a socket by writing them to an OutputStream (associated with a Socket) or encapsulating them in a DatagramPacket (which is then sent via a DatagramSocket). However, the only data types to which these operations can be applied are bytes and arrays of bytes. As a strongly typed language, Java requires that other types—int, String, and so on—be explicitly converted to byte arrays.

*3.1.1 Primitive Integers*  
As we have already seen, TCP and UDP sockets give us the ability to send and receive sequences (arrays) of bytes, i.e., integer values in the range 0–255. Using that ability, we can encode the values of other (larger) primitive integer types. However, the sender and receiver have to agree on several things first. One is the *size* (in bytes) of each integer to be transmitted. For example,  
an int value in a Java program is represented as a 32-bit quantity. We can therefore transmit the value of any variable or constant of type int using four bytes. Values of type short, on the other hand, are represented using 16 bits and so only require two bytes to transmit, while longs are 64 bits or eight bytes. Let’s consider how we would encode a sequence of four integer values: a byte, a short, an int, and a long, in that order, for transmission from sender to receiver. We need a total of 15 bytes: the first contains the value of the byte, the next two contain the value of the short, the next four encode the value of the int, and the last eight bytes contain the long value

The main point is that for any multibyte integer quantity, the sender and receiver need to agree on whether big-endian or little-endian order will be used.

One last detail on which the sender and receiver must agree: whether the numbers transmitted will be *signed* or *unsigned*. The four primitive integer types in Java are all signed; values are stored in *two’s-complement* representation, which is the usual way of representing signed numbers. When dealing with signed *k*-bit numbers, the two’s-complement representation of the negative integer *-n*, 1 *≤ n ≤* 2*k-*1, is the binary value of 2*k - n*. The non-negative integer *p*, 0 *≤ p ≤* 2*k-*1 *-* 1, is encoded simply by the *k*-bit binary value of *p*. Thus, given *k* bits, we can represent values in the range *-*2*k-*1 through 2*k-*1 *-* 1 using two’s-complement. Note that the most significant bit (msb) tells whether the value is positive (msb = 0) or negative (msb = 1). On the other hand, a *k*-bit, *unsigned* integer can encode values in the range 0 through 2*k –* 1 directly.

*3.1.2 Strings and Text*  
Old-fashioned *text*—strings of printable (displayable) characters—is perhaps the most common way to represent information. Text is convenient because humans are accustomed to dealing with all kinds of information represented as strings of characters in books, newspapers, and on computer displays. Thus, once we know how to encode text for transmission, we can send almost any other kind of data: first represent it as text, then encode the text. Obviously we can represent numbers and boolean values as Strings

To better understand what’s going on, we first need to consider that text is made up of symbols or *characters*. In fact every String instance corresponds to a sequence (array) of *characters* (type char[ ]). A char value in Java is represented internally as an integer.

A mapping between a set of symbols and a set of integers is called a *coded character set*. You may have heard of the coded character set known as *ASCII* —*American Standard Code for Information Interchange*. ASCII maps the letters of the English alphabet, digits, punctuation and some other special (non-printable) symbols to integers between 0 and 127. It has been used  
for data transmission since the 1960s, and is used extensively in application protocols such as HTTP (the protocol used for the World Wide Web), even today. However, because it omits symbols used by many languages other than English, it is less than ideal for developing applications and protocols designed to function in today’s global economy. Java therefore uses an international standard coded character set called *Unicode* to represent values of type char and String. Unicode maps symbols from “most of the languages and symbol systems of the world” to integers between 0 and 65,535, and is much better  
suited for internationalized programs

So sender and receiver have to agree on a mapping from symbols to integers in order to communicate using text messages. Is that all they need to agree on? It depends. For a small set of characters with no integer value larger than 255, nothing more is needed because each character can be encoded as a single byte. For a code that may use larger integer values that require more than a single byte to represent, there is more than one way to encode those values on the wire. Thus, sender and receiver need to agree on how those integers will be represented as byte sequences—that is, an *encoding scheme*. The combination of a coded character set and a character encoding scheme is called a *charset*. It is possible to define your own charset, but there is hardly ever a reason to do so. A large number of different *standardized* charsets are in use around the world

*3.1.3 Bit-Diddling: Encoding Booleans*  
*Bitmaps* are a very compact way to encode boolean information, which is often used in protocols. The idea of a bitmap is that each of the bits of an integer type can encode one boolean value— typically with 0 representing false, and 1 representing true. To be able to manipulate bitmaps, you need to know how to set and clear individual bits using Java’s “bit-diddling” operations. A  
*mask* is an integer value that has one or more specific bits set to 1, and all others cleared (i.e.,  
0).

3.2 Composing I/O StreamsJava’s stream classes can be composed to provide powerful capabilities. For example, we can wrap the OutputStream of a Socket instance in a BufferedOutputStream instance to improve performance by buffering bytes temporarily and flushing them to the underlying channel all at once. We can then wrap that instance in a DataOutputStream to send primitive data types.

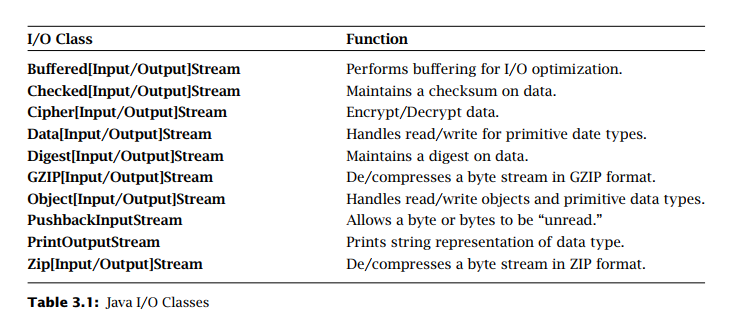


Figure 3.1 demonstrates this composition. Here, we write our primitive data values, one by one, to DataOutputStream, which writes the binary data to BufferedOutputStream, which buffers the data from the three writes and then writes once to the socket OutputStream, which controls writing to the network. We create a corresponding composition for the InputStream on the other  
endpoint to efficiently receive primitive data types.

**3.3 Framing and Parsing**Converting data to wire format is, of course, only half the story; the original information must be recovered at the receiver from the transmitted sequence of bytes. Application protocols typically deal with discrete messages, which are viewed as collections of fields. *Framing* refers to the problem of enabling the receiver to locate the beginning and end of a message. Whether information is encoded as text, as multibyte binary numbers, or as some combination of the two, the application protocol must specify how the receiver of a message can determine when it has received all of the message.

If a receiver tries to receive more bytes from the socket than were in the message, one of two things can happen. If no other message is in the channel, the receiver will block and be prevented from processing the message; if the sender is also blocked waiting for a reply, the result will be *deadlock*. On the other hand, if another message is in the channel, the receiver may read some or all of it as part of the first message, leading to protocol errors. Therefore framing is an important consideration when using TCP sockets.

Two general techniques enable a receiver to unambiguously find the end of the  
message:  
*Delimiter-based*: The end of the message is indicated by a *unique marker*, an explicit byte sequence that the sender transmits immediately following the data. The marker must be known not to occur in the data.  
*Explicit length*: The variable-length field or message is preceded by a (fixed-size) length  
field that tells how many bytes it contains. A special case of the delimiter-based method can be used for the last message sent on a TCP connection: the sender simply closes the sending side of the connection (using shutdownOutput() or close()) after sending the message. After the receiver reads the last byte of the message, it receives an end-of-stream indication (i.e., read() returns *-*1), and thus can tell that it has reached the end of the message.

The delimiter-based approach is often used with messages encoded as text: A particular character or sequence of characters is defined to mark the end of the message. The receiver simply scans the input (as characters) looking for the delimiter sequence; it returns the character string preceding the delimiter. The drawback is that *the message itself must not contain the delimiter*, otherwise the receiver will find the end of the message prematurely. With a delimiterbased framing method, the sender is responsible for ensuring that this precondition is satisfied. Fortunately so-called *stuffing* techniques allow delimiters that occur naturally in the message to be modified so the receiver will not recognize them as such; as it scans for the delimiter, it also recognizes the modified delimiters and restores them in the output message so it matches the original. The downside of such techniques is that *both* sender and receiver have to scan the message.

The length-based approach is simpler, but requires a known upper bound on the size of the message. The sender first determines the length of the message, encodes it as an integer, and prefixes the result to the message. The upper bound on the message length determines the number of bytes required to encode the length: one byte if messages always contain fewer than 256 bytes, two bytes if they are always shorter than 65,536 bytes, and so on.

**3.4 Java-Specific Encodings**When you use sockets, generally either you are building the programs on both ends of the communication channel—in which case you also have complete control over the protocol—or you are communicating using a *given* protocol, which you have to implement. When you know that (i) both ends of the communication will be implemented in Java, and (ii) you have complete  
control over the protocol, you can make use of Java’s built-in facilities

3.5 Constructing and Parsing Protocol Messages

*3.5.1 Text-Based Representation*  
We first present a version in which messages are encoded as text. The protocol specifies that the text be encoded using the US-ASCII charset. The message begins with a so-called “magicstring”—a sequence of characters that allows a recipient to quickly recognize the message as a Voting protocol message, as opposed to random garbage that happened to arrive over the network. The Vote/Inquiry boolean is encoded with the character ‘v’ for a vote or ‘i’ for an inquiry. The message’s status as a response is indicated by the presence of the character ‘R’. Then comes the candidate ID, followed by the vote count, both encoded as decimal string

*3.5.2 Binary Representation*  
Next we present a different way to encode the Voting protocol message. In contrast with the text-based format, the binary format uses fixed-size messages. Each message begins with a one-byte field that contains the “magic” value 010101 in its high-order six bits. This little bit of redundancy provides the receiver with a small degree of assurance that it is receiving a proper voting message. The two low-order bits of the first byte encode the two booleans. The second byte of the message always contains zeros, and the third and fourth bytes contain the candidateID. The final eight bytes of a response message (only) contain the vote count.

*3.5.3 Sending and Receiving*  
Sending a message over a stream is as simple as creating it, calling toWire(), adding appropriate framing information, and writing it. Receiving, of course, does things in the opposite order. This approach applies to TCP; in UDP explicit framing is not necessary, because message boundaries are preserved. To demonstrate this, consider a vote server that 1) maintains a mapping of candidate IDs to number of votes, 2) counts submitted votes, and 3) responds to inquiries and votes with the current count for the specified candidate. We begin by implementing a service for use by vote servers.