CSE 686 Internet Programming

Week 4: Sending/Receiving Data

Edmund Yu, PhD
Associate Teaching Professor
esyu@syr.edu

February 7, 2018

Based on **TCP/IP Sockets in Java**, Kenneth L. Calvert & Michael J. Donahoo, 2008, Chapter 3

TCP Client/Server Interaction (Revisited)

```
OutputStream out = sock.getOutputStream();
out.write(byteBuffer);
```

```
InputStream in = clntSock.getInputStream();
recvMsgSize = in.read(byteBuffer);
```

Client

1. Create a TCP socket
2. **Communicate**
3. Close the connection

Server

1. Create a TCP socket
2. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Protocols Revisited

- We use **sockets** to provide information to, or use information provided by, another program. (a.k.a. communications, or I/O)

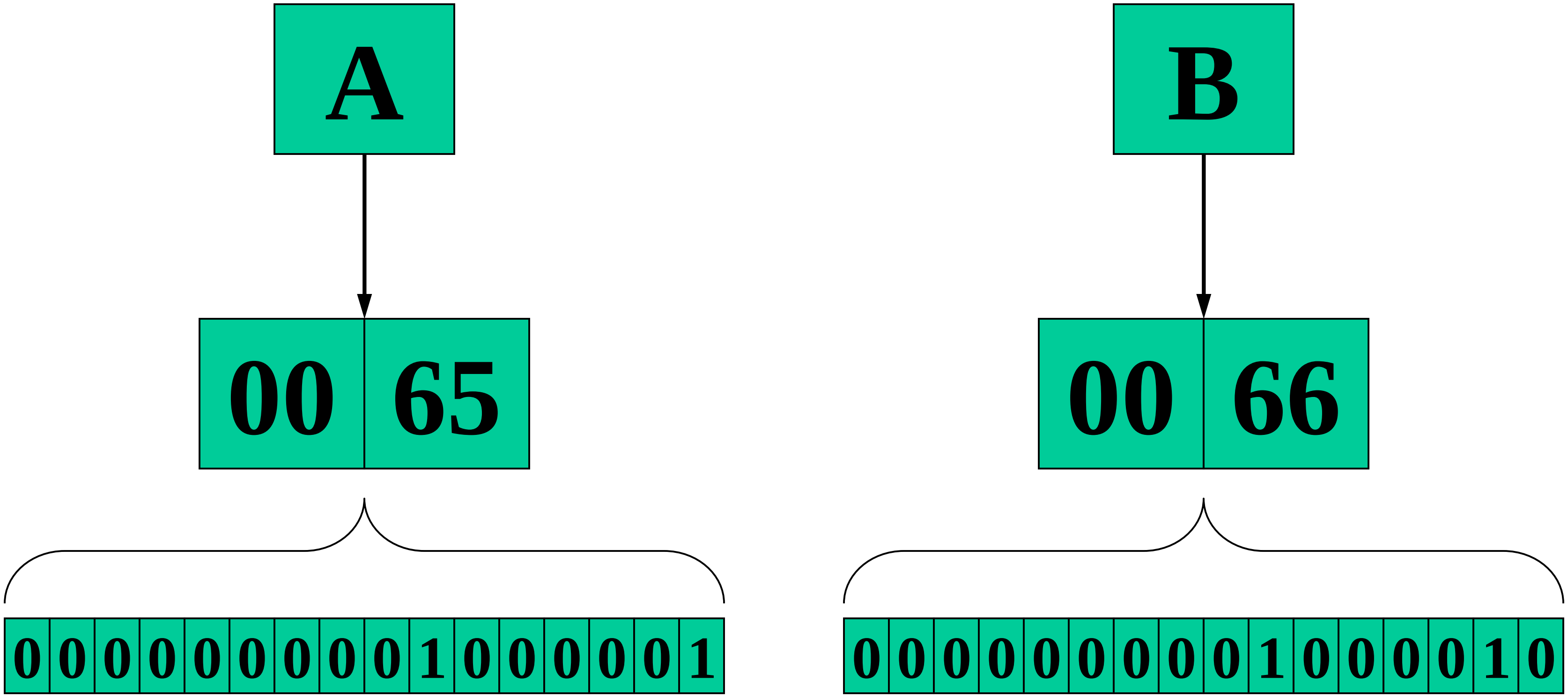
- Any programs, including web servers and clients, that exchange information must agree on: → **protocol**
  1. How that information will be encoded
  2. Which program should send what information when
  3. How the information received affects the behavior of the program.

Protocols Revisited

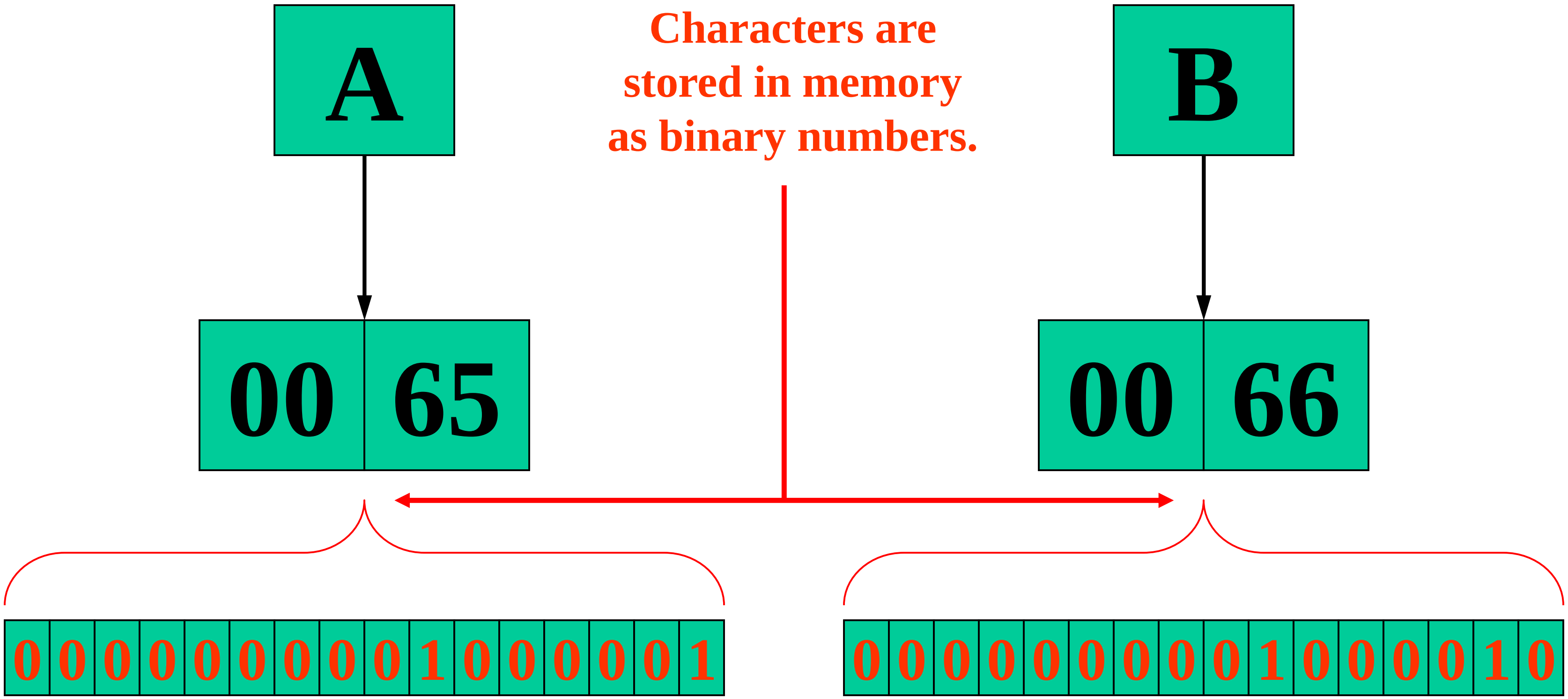
- We use **sockets** to provide information to, or use information provided by, another program. (a.k.a. communications, or I/O)

- Any programs, including web servers and clients, that exchange information must agree on: → **protocol**
  1. How that information will be encoded
  2. Which program should send what information when
  3. How the information received affects the behavior of the program.

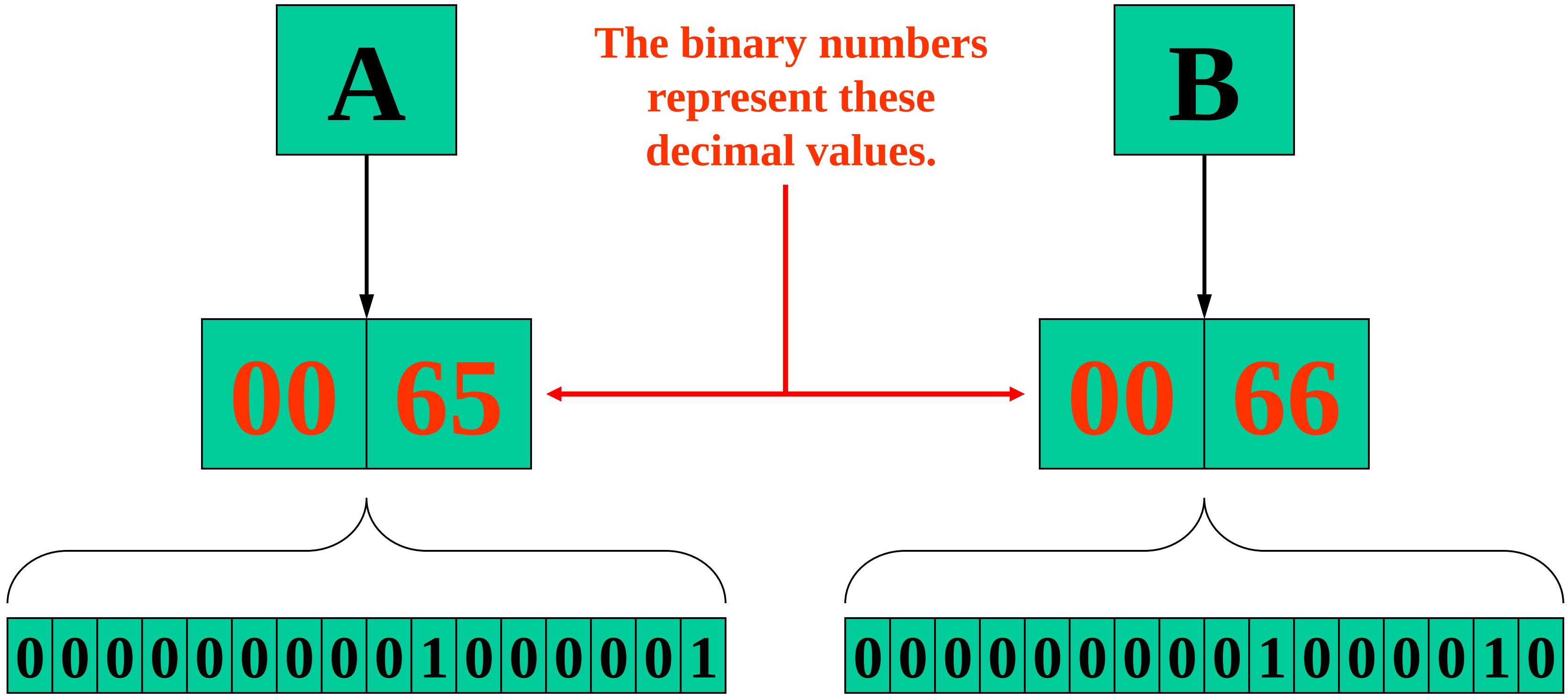
Unicode: An Example



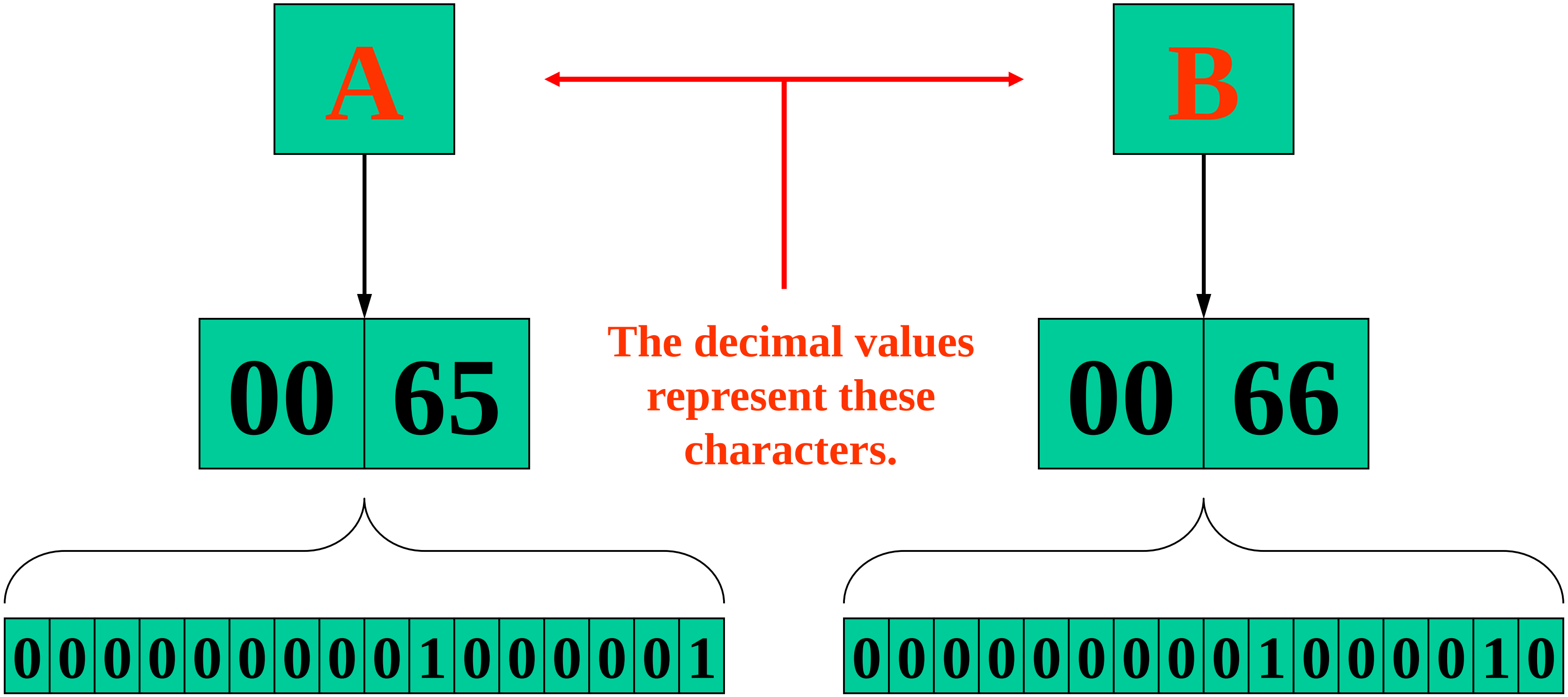
Unicode: An Example



Unicode: An Example



Unicode: An Example



Encoding

- First, let's take a look at how simple values, such as **int**s, **long**s, **char**s, and **String**s, 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**, or reading them from an **InputStream**.
  - Unfortunately, the only data types to which these operations can be applied are arrays of bytes.
    - All other types have to be explicitly converted to byte arrays.
  - But, fortunately, Java has built-in facilities to help with such conversions, such as the **getBytes()** method in **TCPEchoClient.java**, which converts the characters in a String instance to bytes.

TCPEchoClient

```
public class TCPEchoClient {
  public static void main(String[] args) throws IOException {

    if ((args.length < 2) || (args.length > 3))  // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");

    String server = args[0];       // Server name or IP address
    // Convert argument String to bytes using the default character encoding
    byte[] data = args[1].getBytes();

    Int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;

    // Create socket that is connected to server on specified port
    Socket socket = new Socket(server, servPort);
    System.out.println("Connected to server...sending echo string");
    …
```

http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

String (Java Platform SE 7)

boolean	**equals**(**Object** anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(**String** anotherString) Compares this `String` to another `String`, ignoring case considerations.
static **String**	**format**(**Locale** l, **String** format, **Object**... args) Returns a formatted string using the specified locale, format string, and arguments.
static **String**	**format**(**String** format, **Object**... args) Returns a formatted string using the specified format string and arguments.
byte[]	**getBytes**() Encodes this `String` into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
byte[]	**getBytes**(**Charset** charset) Encodes this `String` into a sequence of bytes using the given **charset**, storing the result into a new byte array.
void	**getBytes**(int srcBegin, int srcEnd, byte[] dst, int dstBegin) **Deprecated.** *This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the* **getBytes()** *method, which uses the platform's default charset.*
byte[]	**getBytes**(**String** charsetName) Encodes this `String` into a sequence of bytes using the named charset, storing the result into a new byte array.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
int	**hashCode**() Returns a hash code for this string.
int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
int	**indexOf**(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	**indexOf**(**String** str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(**String** str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
String	**intern**() Returns a canonical representation for the string object.
boolean	**isEmpty**() Returns `true` if, and only if, **length()** is 0.

Integers

- For encoding integers, we could use TCP sockets' ability to send and receive arrays of bytes, using InputStream and OutputStream:

  public InputStream **getInputStream()** throws IOException

  public OutputStream **getOutputStream()** throws IOException

  public void **shutdownInput()** throws IOException

  public void **shutdownOutput()** throws IOException

- Using that ability, we can encode the values of larger integer types (**short**, **int**, and **long**) using **byte**

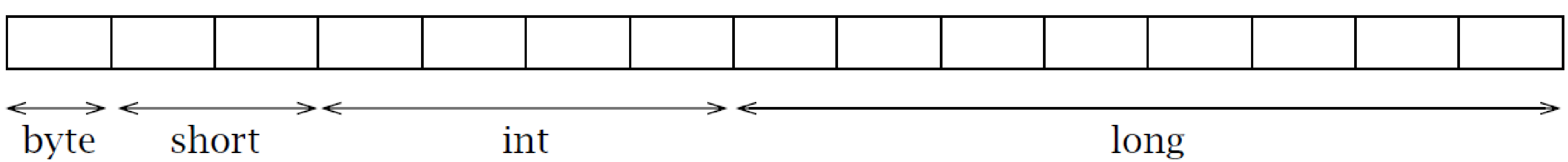
Integers: Size

- ❖ However, the sender and receiver have to agree on 3 things first:

1. The **size** (in bytes) of each integer to be transmitted
   - ❖ In Java, we need to agree to transmit the value of any **int** using four bytes, any **short** using 2 bytes, any **long** using 8 bytes

Sending Integers: An Example

- How would we 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?



Integers: Order

2. The **order** to send the bytes in
   - There are two obvious choices:
     - Start at the right end of the integer, with the least significant byte (so-called little-endian order)
     - Start at the left end, with the most significant byte (so-called big-endian order). (Java default)
   - Fortunately, the ordering of bits within bytes is handled in a standard way.

Primitive Integers: An Example

- 123456787654321**L** → 0x0000704885F926B1
- If we transmit the bytes in big-endian order, the sequence of (decimal) byte values will look like this:

0	0	112	72	133	249	38	177

order of transmission

If we transmit them in little-endian order, the sequence will be:

177	38	249	133	72	112	0	0

order of transmission

Integers: Signs

3. Whether the numbers transmitted will be signed or unsigned

- ❖ The four primitive integer types in Java are all signed
- ❖ Their 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 $-n$, $1 \leq n \leq 2^{k-1}$, is the binary value of $2^{k-1} - n$.
- ❖ The non-negative integer p, $0 \leq p \leq 2^{k-1} - 1$, 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).

Integers: Signs

- On the other hand, a k-bit, unsigned integer can encode values in the range 0 through $2k - 1$
  - For example, the 32-bit value 0xffffffff (the all-ones value) when interpreted as a signed, two's complement integer represents -1; when interpreted as an unsigned integer, it represents 4,294,967,295.

- Java does not support unsigned integer types, so encoding and decoding unsigned numbers in Java requires a little more care.

http://cs.ecs.baylor.edu/~donahoo/practical/JavaSockets2/code/

cs.ecs.baylor.edu

File Edit View Favorites Tools Help

```
public class BruteForceCoding {
  private static byte byteVal = 101; // one hundred and one
  private static short shortVal = 10001; // ten thousand and one
  private static int intVal = 100000001; // one hundred million and one
  private static long longVal = 1000000000001L;// one trillion and one

  private final static int BSIZE = Byte.SIZE / Byte.SIZE;
  private final static int SSIZE = Short.SIZE / Byte.SIZE;
  private final static int ISIZE = Integer.SIZE / Byte.SIZE;
  private final static int LSIZE = Long.SIZE / Byte.SIZE;

  private final static int BYTEMASK = 0xFF; // 8 bits

  public static String byteArrayToDecimalString(byte[] bArray) {
    StringBuilder rtn = new StringBuilder();
    for (byte b : bArray) {
      rtn.append(b & BYTEMASK).append(" ");
    }
    return rtn.toString();
  }

  // Warning:  Untested preconditions (e.g., 0 <= size <= 8)
  public static int encodeIntBigEndian(byte[] dst, long val, int offset, int size) {
    for (int i = 0; i < size; i++) {
      dst[offset++] = (byte) (val >> ((size - i - 1) * Byte.SIZE));
    }
    return offset;
  }

  // Warning:  Untested preconditions (e.g., 0 <= size <= 8)
```

<u>Textbook #1, pp 42-43 (Optional)</u>

```
Socket socket = new Socket(server, port);
DataOutputStream out = new DataOutputStream(
    new BufferedOutputStream(socket.getOutputStream()));
```

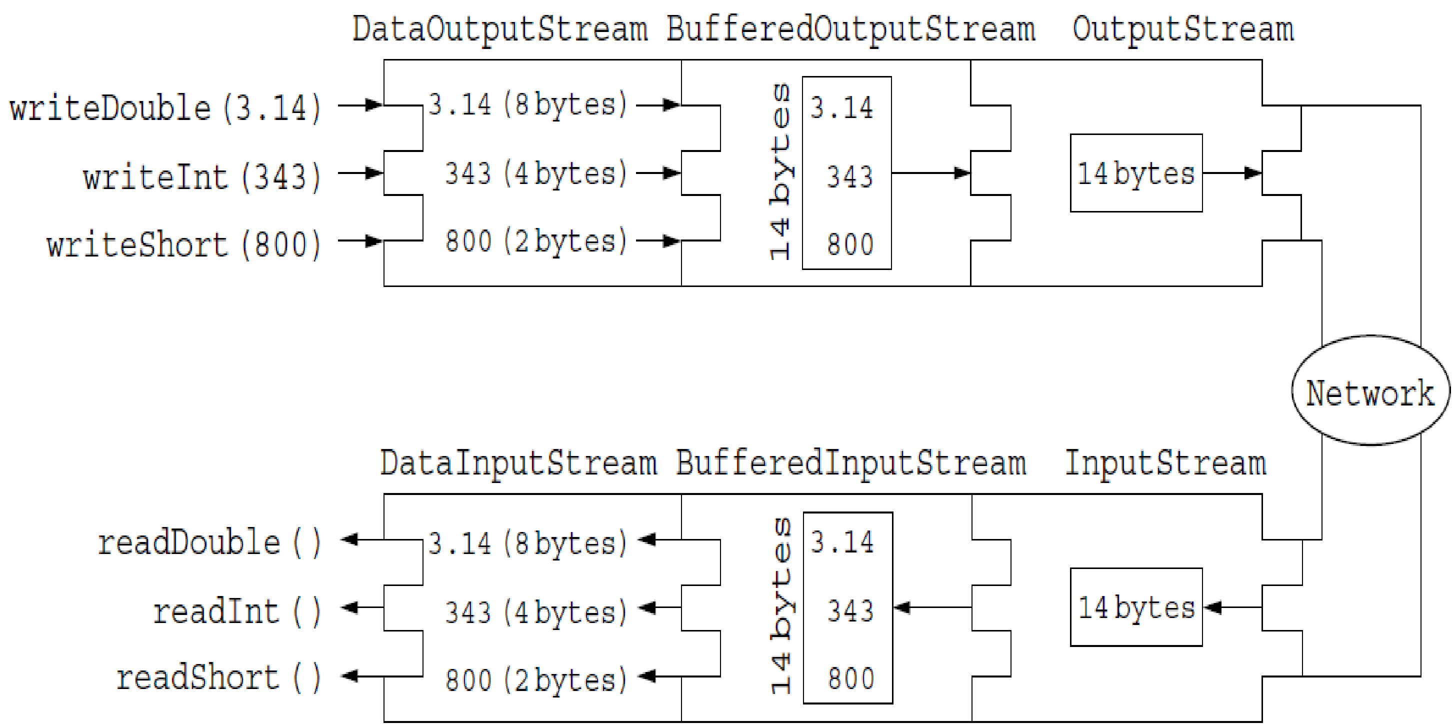


Figure 3.1: Stream composition.

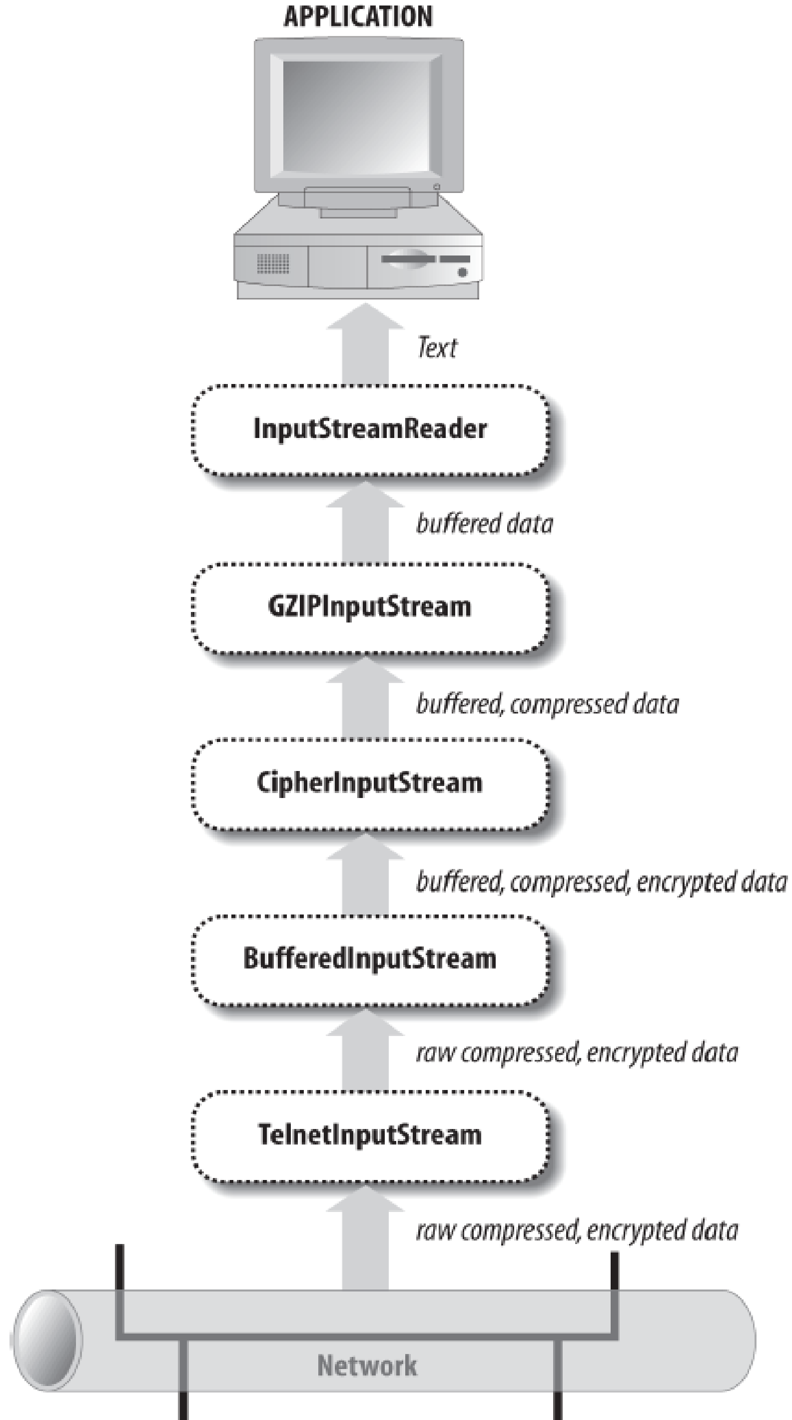
Composing I/O Streams

- Java's stream classes can be composed to provide powerful capabilities. (**chaining**)
- 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.
- We would code this composition as follows:

```
Socket socket = new Socket(server, port);
DataOutputStream out = new DataOutputStream(
        new BufferedOutputStream(socket.getOutputStream()));
```

I/O Class	Function
Buffered[Input/Output]Stream	Performs buffering for I/O optimization.
Checked[Input/Output]Stream	Maintains a checksum on data.
Cipher[Input/Output]Stream	Encrypt/Decrypt data.
Data[Input/Output]Stream	Handles read/write for primitive date types.
Digest[Input/Output]Stream	Maintains a digest on data.
GZIP[Input/Output]Stream	De/compresses a byte stream in GZIP format.
Object[Input/Output]Stream	Handles read/write objects and primitive data types.
PushbackInputStream	Allows a byte or bytes to be "unread."
PrintOutputStream	Prints string representation of data type.
Zip[Input/Output]Stream	De/compresses a byte stream in ZIP format.

Table 3.1: Java I/O Classes



APPLICATION
Text
InputStreamReader
buffered data
GZIPInputStream
buffered, compressed data
CipherInputStream
buffered, compressed, encrypted data
BufferedInputStream
raw compressed, encrypted data
TelnetInputStream
raw compressed, encrypted data
Network

ORACLE Java Documentation

The Java™ Tutorials

Lesson: Basic I/O

This lesson covers the Java platform classes used for basic I/O. It first focuses on *I/O Streams*, a powerful concept that greatly simplifies I/O operations. The lesson also looks at serialization, which lets a program write whole objects out to streams and read them back again. Then the lesson looks at file I/O and file system operations, including random access files.

Most of the classes covered in the `I/O Streams` section are in the `java.io` package. Most of the classes covered in the `File I/O` section are in the `java.nio.file` package.

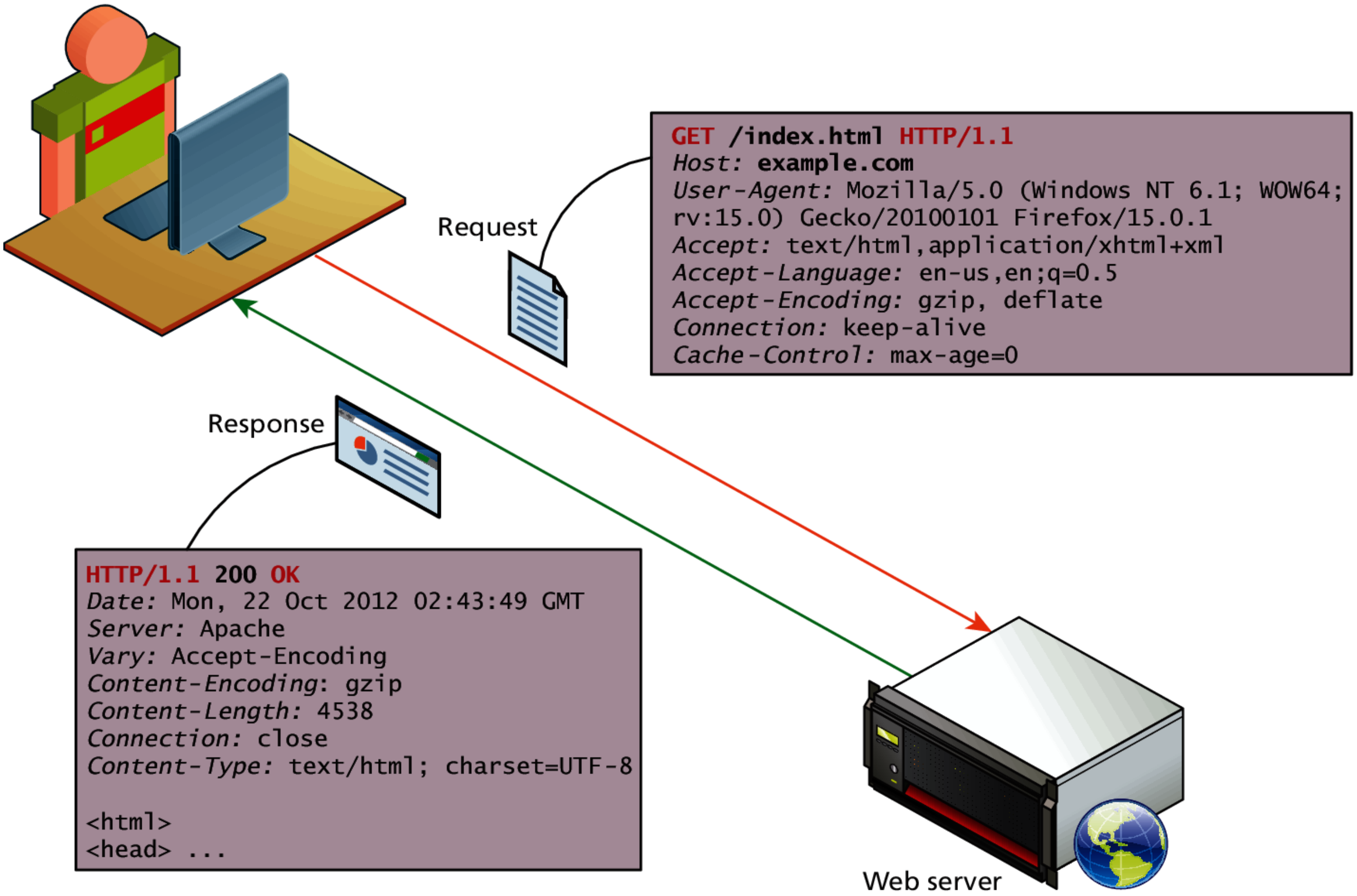
I/O Streams

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and `String` values.

Encoding Texts

- Text (a string of characters) is perhaps the most common way to represent information. (see next slide)
- Once we know how to encode text for transmission, we can send almost any other kind of data:
  - First <u>represent it as text</u>
    - Obviously we can represent numbers and boolean values as Strings:
    - Eg. "123478962", "6.02e23", "true", "false".
  - Then <u>encode the text</u>.
    - A string can be converted to a byte array by calling the **getBytes()** method used in TCPEchoClient

HTTP (HyperText Transfer Protocol)



Encoding Texts

- A mapping between a set of symbols and a set of integers is called a **coded character set**.
- The best example is 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.
  - 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.

Encoding Texts: ASCII

ASCII Table - ASCII and

ascii-table.com/

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

:: move the mouse over the table to see details >>
:: click to select the character...

ASCII / American Standard Code for Information Interchange

ASCII is the standard code used for information interchange and communication between data processing systems, including Internet.

The ASCII character set (or ASCII table) initially contained 128 7-bit coded characters including alphabetic, numeric, control and graphic characters. It has since been extended to include system and country specific characters (Unicode™ standard).

ASCII is the U.S. version of International Reference Alphabet (IRA) No.5 (formerly known as International Alphabet No.5, or "IA5") as specified in ITU-T Recommendation T.50.

:: more character tables

ASCII code

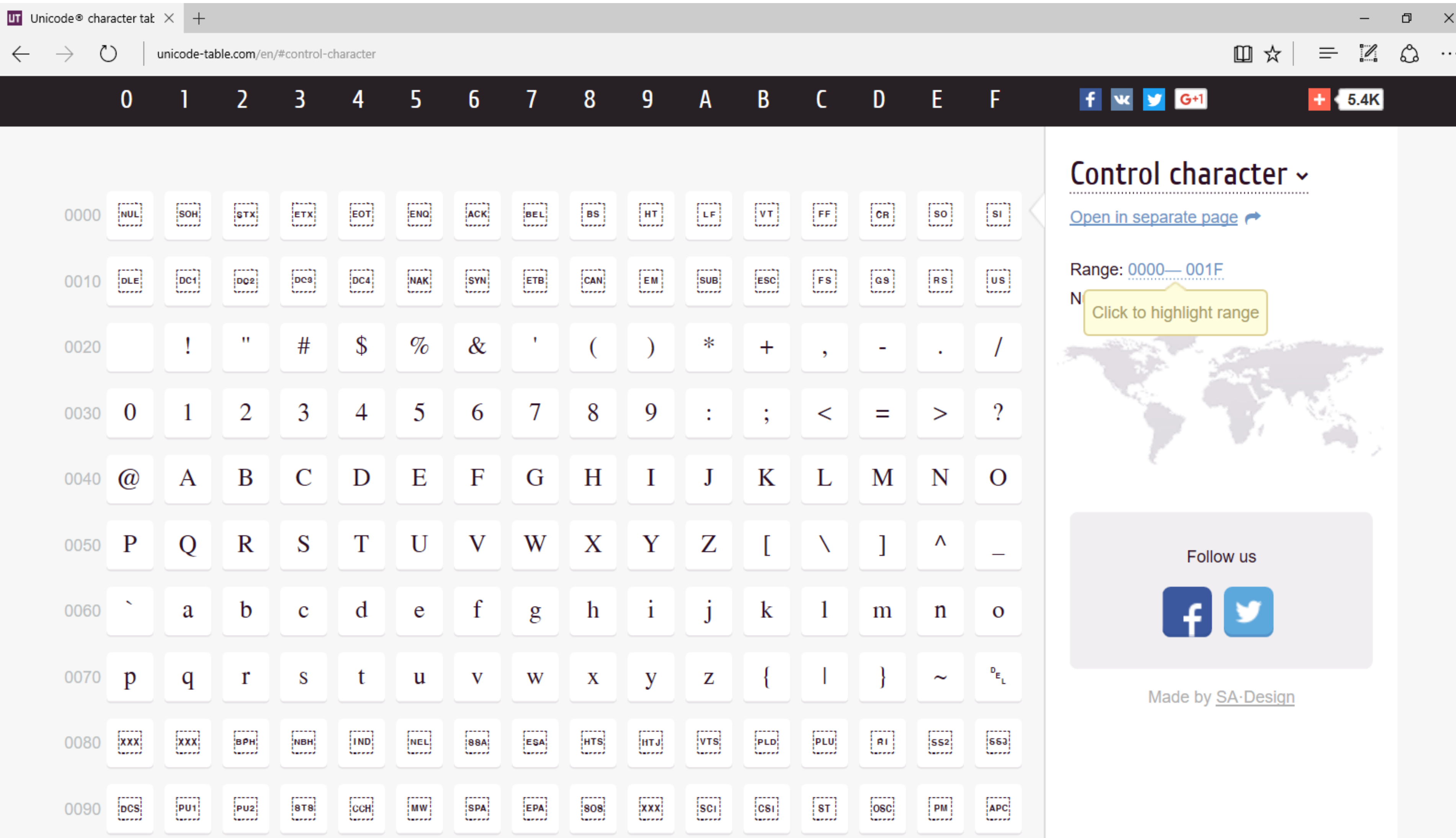
Name	Null
Character	[Ctrl] @
Abbreviation	NUL
Decimal	0
Hexadecimal	00
Octal	00
Binary	00000000

:: more info

Encoding Texts

- 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.
  - Unicode includes ASCII: each symbol defined by ASCII maps to the same integer in Unicode as it does in ASCII.
  - This provides a degree of backward compatibility between ASCII and Unicode.

Encoding Texts: Unicode



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0010	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0020		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
0080	XXX	XXX	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
0090	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	XXX	SCI	CSI	ST	OSC	PM	APC

http://unicode-table.com/

Encoding Texts: Unicode

Script	Range of code values
Arabic	U+0600–U+06FF
Basic Latin	U+0000–U+007F
Bengali (India)	U+0980–U+09FF
Cherokee (Native America)	U+13A0–U+13FF
CJK Unified Ideographs (East Asia)	U+4E00–U+9FFF
Cyrillic (Russia and Eastern Europe)	U+0400–U+04FF
Ethiopic	U+1200–U+137F
Greek	U+0370–U+03FF
Hangul Jamo (Korea)	U+1100–U+11FF
Hebrew	U+0590–U+05FF
Hiragana (Japan)	U+3040–U+309F
Khmer (Cambodia)	U+1780–U+17FF
Lao (Laos)	U+0E80–U+0EFF
Mongolian	U+1800–U+18AF
Myanmar	U+1000–U+109F
Ogham (Ireland)	U+1680–U+169F
Runic (Germany and Scandinavia)	U+16A0–U+16FF
Sinhala (Sri Lanka)	U+0D80–U+0DFF
Telugu (India)	U+0C00–U+0C7F
Thai	U+0E00–U+0E7F

Encoding Texts

- The sender and the receiver have to agree on a mapping from symbols to integers in order to communicate using text messages.
- The sender and the receiver also need to agree on how those integers will be represented as byte sequences - that is the **encoding scheme** that we just discussed.
- The combination of a coded character set and a character encoding scheme is called a **charset** (see RFC 2278).
- A large number of standardized charsets are in use around the world.
- Java provides support for the use of arbitrary charsets, and every implementation is required to support at least the following: ASCII, ISO-8859-1, **UTF-8**, UTF-16BE, UTF-16LE, etc.

Encoding Booleans: Bit-Diddling

- **Bitmaps** are a very compact way to encode boolean information, which is often used in protocols. (See next slide for an example.)
  - The idea of a bitmap is that each of the bits of an integer type can encode one boolean value:
    - 0 → false; 1 → true.
  - We'll deal mainly with int-sized bitmaps (32 bits), but everything here applies to other integer types as well.

Textbook #1, pp 61-64

Binary Encoding

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

/* Wire Format
 *                                 1   1   1   1   1   1
 *   0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 * |     Magic             |Flags|       ZERO                    |
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 * |                   Candidate ID                                |
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 * |                                                               |
 * |         Vote Count (only in response)                         |
 * |                                                               |
 * |                                                               |
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 */
public class VoteMsgBinCoder implements VoteMsgCoder {

  // manifest constants for encoding
  public static final int MIN_WIRE_LENGTH = 4;
  public static final int MAX_WIRE_LENGTH = 16;
  public static final int MAGIC = 0x5400;
  public static final int MAGIC_MASK = 0xfc00;
  public static final int MAGIC_SHIFT = 8;
  public static final int RESPONSE_FLAG = 0x0200;
  public static final int INQUIRE_FLAG =  0x0100;
```

Masks

- **Masks** are used to manipulate bitmaps.
- Let's number the bits of a value of type **int** from 0 to 31, as usual, where bit 0 is the least significant bit.
  - Bit 5 is represented by 32, bit 12 by 4096, etc.
- Some example masks:

```
final int BIT5 = (1 << 5);
final int BIT7 = 0x80;
final int BITS2AND3 = 12; // 8+4
int bitmap = 1234567;
```

Bit-Diddling

- To **set** a particular bit in an int variable, combine it with the mask for that bit using the bitwise-OR operation (|):

```
int bitmap = 1234567;
bitmap |= BIT5;
// bit 5 is now one
```

- To **clear** a particular bit, bitwise-AND it with the bitwise complement of the mask for that bit. The bitwise-AND operation in Java is &, while the bitwise-complement operator is ~.

```
bitmap &= ~BIT7;
// bit 7 is now zero
```

Encoding Booleans: Bit-Diddling

- You can set and clear multiple bits at once by OR-ing together the corresponding masks:

```
// clear bits 2, 3 and 5
bitmap &= ~(BITS2AND3|BIT5);
```

- To test whether a bit is set, compare the result of the bitwise-AND of the mask and the value with zero:

```
boolean bit6Set = (bitmap & (1<< 6)) != 0;
```

The Framing Problem

Framing and Parsing

- How do we recover the original information, included in a message sent by the sender, at the receiver end ? (see next slide for a simple example)
  - The first things we must know are the beginning and end of that message

- **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 multi-byte binary numbers, or as a combination of the two, the application protocol must specify how the receiver can determine when it has received <u>all of the message</u>.

```
Socket socket = new Socket(server, port);
DataOutputStream out = new DataOutputStream(
    new BufferedOutputStream(socket.getOutputStream()));
```

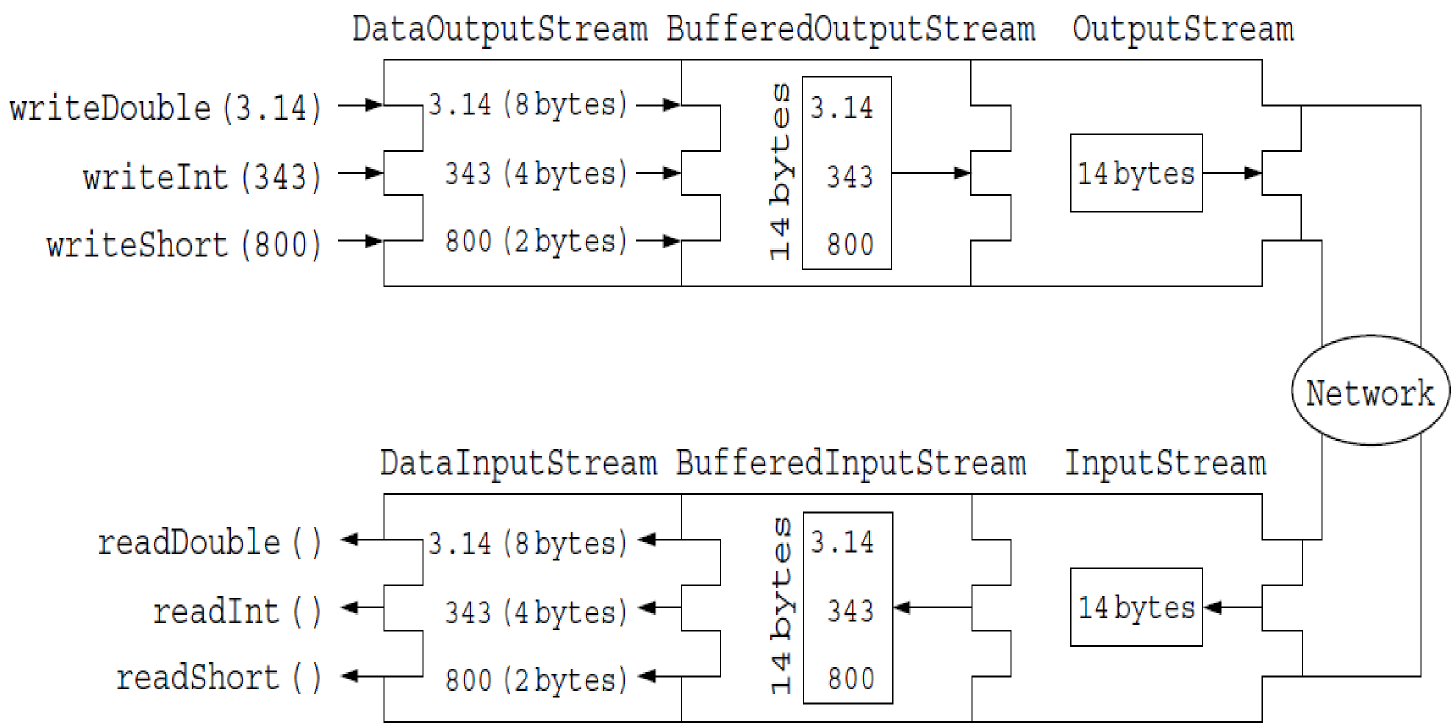


Figure 3.1: Stream composition.

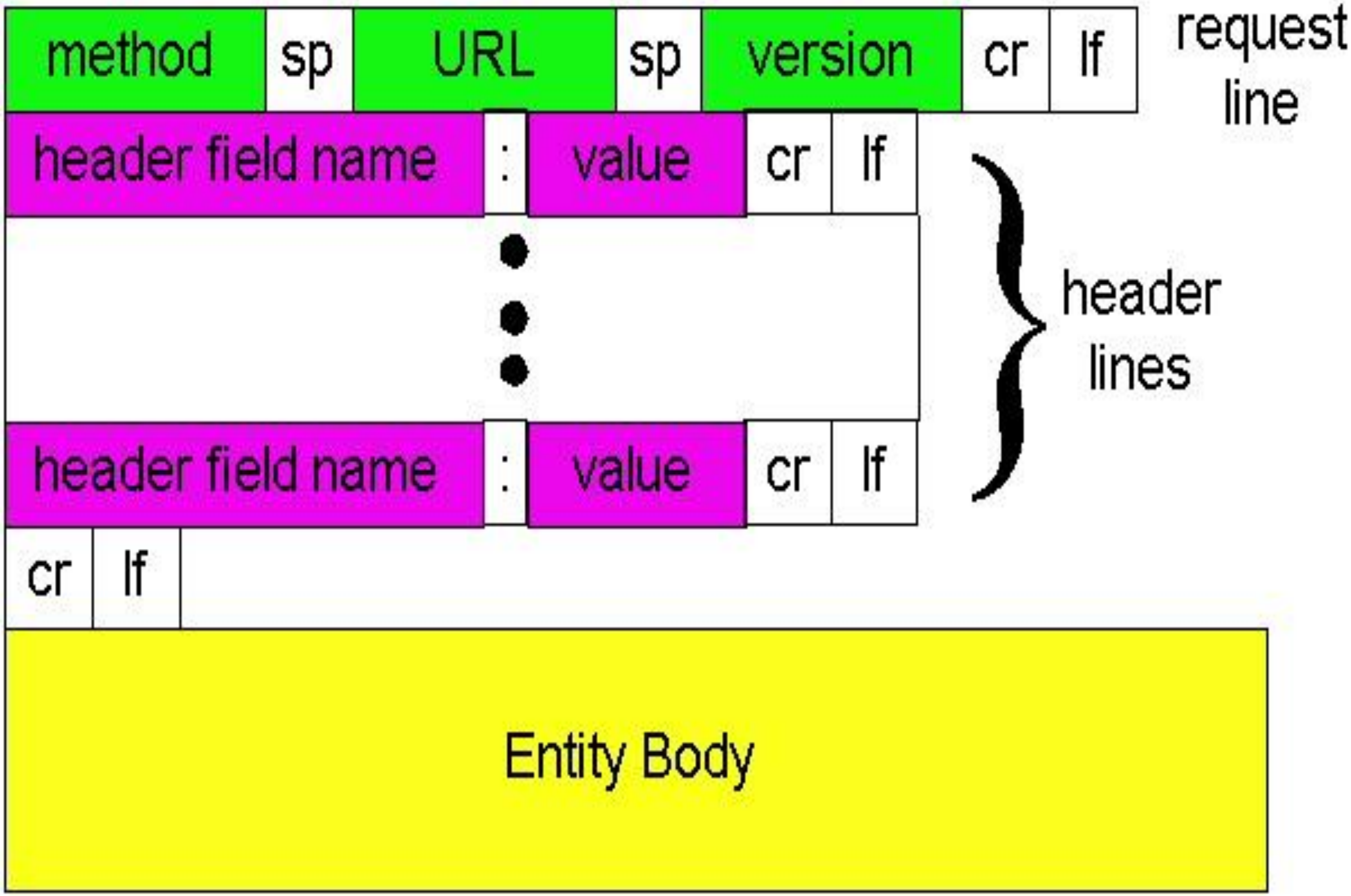
Framing and Parsing

- Recall that application protocols typically deal with discrete messages, which are viewed as collections of fields. (next slide)

- If the fields in a message all have fixed sizes and the message is made up of a fixed number of fields, then the size of the message is known in advance, and the receiver can simply read a fixed amount of bytes into a **byte[ ]** buffer.
  - This technique was used in **TCPEchoClient.java**, where we knew the number of bytes to expect from the server.

- However, when the message can vary in length, such as some, arbitrary text strings, we do not know beforehand how many bytes to read.

HTTP Request Message Format



Framing and Parsing

- 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**.
  - If there is another message 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 very important when using TCP sockets.

Framing and Parsing

- 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.

The Delimiter-Based Approach

- 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
  - Once the delimiter is found, it returns the character string preceding the delimiter.

The Delimiter-Based Approach

- A special case of the delimiter-based method can be used for the last message sent on a TCP connection:
  - The sender can simply close 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).

The Delimiter Approach: Drawback

- The drawback:
  - The message itself must not contain the delimiter, otherwise the receiver will find the end of the message prematurely.
  - The sender is responsible for ensuring that this precondition is satisfied.

Stuffing

- The solution: Use the **stuffing** techniques
  - They allow delimiters that occur naturally in the message to be modified, so the receiver will not recognize them as such.
  - As the receiver scans for the delimiter, it also recognizes the modified delimiters and restores them in the output message so it matches the original.
  - Downside: both sender and receiver have to scan the message.

The Length-Based Approach

- The length-based approach:
  - Is simpler than the delimiter approach, 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 it to the message.
  - The upper bound on the message length determines the number of bytes required to encode the length:
    - 1 byte if messages always contain fewer than 256 bytes; 2 bytes if they are always shorter than 65,536 bytes, etc.

The Framer Interface

- The interface **Framer** from the textbook demonstrates these 2 techniques.
  - This interface has two (abstract) methods:
    - **frameMsg()** adds framing information and outputs a given message to a given stream → Output
    - **nextMsg()** scans a given stream, extracting the next message.

      → Input

http://cs.ecs.baylor.edu/~donahoo/practical/JavaSockets2/code/

cs.ecs.baylor.edu

File Edit View Favorites Tools Help

```
import java.io.IOException;
import java.io.OutputStream;

public interface Framer {
  void frameMsg(byte[] message, OutputStream out) throws IOException;
  byte[] nextMsg() throws IOException;
}
```

Textbook #1, p. 51

DelimFramer

- The class **DelimFramer.java** implements delimiter-based framing using the “newline” character ("\n", byte value 10 decimal) as the delimiter.
  - Its **frameMsg()** method does not do stuffing, but simply throws an exception if the byte sequence to be framed contains the delimiter.
  - Its **nextMsg()** method scans the stream until it reads the delimiter, then returns everything up to the delimiter; null is returned if the stream is empty.
    - If some bytes of a message are accumulated and the stream ends without finding a delimiter, an exception is thrown to indicate a framing error.

Textbook #1, pp 51-52

```java
import java.io.ByteArrayOutputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class DelimFramer implements Framer {

  private InputStream in;          // data source
  private static final byte DELIMITER = '\n'; // message delimiter

  public DelimFramer(InputStream in) {
    this.in = in;
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    // ensure that the message does not contain the delimiter
    for (byte b : message) {
      if (b == DELIMITER) {
        throw new IOException("Message contains delimiter");
      }
    }
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;
```

The constructor

The input stream from which messages are to be extracted is given as an argument.

Textbook #1, pp 51-52

```
import java.io.ByteArrayOutputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class DelimFramer implements Framer {

  private InputStream in;         // data source
  private static final byte DELIMITER = '\n'; // message delimiter

  public DelimFramer(InputStream in) {
    this.in = in;
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    // ensure that the message does not contain the delimiter
    for (byte b : message) {
      if (b == DELIMITER) {
        throw new IOException("Message contains delimiter");
      }
    }
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;
```

This (enhanced) for loop checks that the given message does not contain the delimiter

Textbook #1, pp 51-52

```java
import java.io.ByteArrayOutputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class DelimFramer implements Framer {

  private InputStream in;          // data source
  private static final byte DELIMITER = '\n'; // message delimiter

  public DelimFramer(InputStream in) {
    this.in = in;
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    // ensure that the message does not contain the delimiter
    for (byte b : message) {
      if (b == DELIMITER) {
        throw new IOException("Message contains delimiter");
      }
    }
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;
```

If it does, throw an exception.

Textbook #1, pp 51-52

```java
import java.io.ByteArrayOutputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class DelimFramer implements Framer {

  private InputStream in;         // data source
  private static final byte DELIMITER = '\n'; // message delimiter

  public DelimFramer(InputStream in) {
    this.in = in;
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    // ensure that the message does not contain the delimiter
    for (byte b : message) {
      if (b == DELIMITER) {
        throw new IOException("Message contains delimiter");
      }
    }
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;
```

This block of statements writes the **framed** message to the OutputStream

```java
import java.io.ByteArrayOutputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
```

Textbook #1, pp 51-52

```java
public class DelimFramer implements Framer {

  private InputStream in;          // data source
  private static final byte DELIMITER = '\n'; // message delimiter

  public DelimFramer(InputStream in) {
    this.in = in;
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    // ensure that the message does not contain the delimiter
    for (byte b : message) {
      if (b == DELIMITER) {
        throw new IOException("Message contains delimiter");
      }
    }
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;
```

This statement writes the message to the OutputStream

Textbook #1, pp 51-52

```
import java.io.ByteArrayOutputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class DelimFramer implements Framer {

  private InputStream in;        // data source
  private static final byte DELIMITER = '\n'; // message delimiter

  public DelimFramer(InputStream in) {
    this.in = in;
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    // ensure that the message does not contain the delimiter
    for (byte b : message) {
      if (b == DELIMITER) {
        throw new IOException("Message contains delimiter");
      }
    }
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;
```

This statement writes the **delimiter** to the OutputStream

Textbook #1, pp 51-52

```java
import java.io.ByteArrayOutputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class DelimFramer implements Framer {

  private InputStream in;        // data source
  private static final byte DELIMITER = '\n'; // message delimiter

  public DelimFramer(InputStream in) {
    this.in = in;
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    // ensure that the message does not contain the delimiter
    for (byte b : message) {
      if (b == DELIMITER) {
        throw new IOException("Message contains delimiter");
      }
    }
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;
```

Don’t forget to flush()

The Framer Interface

- The interface **Framer** from the textbook demonstrates these 2 techniques.
  - This interface has two (abstract) methods:
    - **frameMsg()** adds framing information and outputs a given message to a given stream → Output
    - **nextMsg()** scans a given stream, extracting the next message.

      → Input

```
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;

    // fetch bytes until find delimiter
    while ((nextByte = in.read()) != DELIMITER) {
      if (nextByte == -1) { // end of stream?
        if (messageBuffer.size() == 0) { // if no byte read
          return null;
        } else { // if bytes followed by end of stream: framing error
          throw new EOFException("Non-empty message without delimiter");
        }
      }
      messageBuffer.write(nextByte); // write byte to buffer
    }

    return messageBuffer.toByteArray();
  }
}
```

Textbook #1, pp 51-52

Extracts the next message from the **InputStream in**

```java
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;

    // fetch bytes until find delimiter
    while ((nextByte = in.read()) != DELIMITER) {
      if (nextByte == -1) { // end of stream?
        if (messageBuffer.size() == 0) { // if no byte read
          return null;
        } else { // if bytes followed by end of stream: framing error
          throw new EOFException("Non-empty message without delimiter");
        }
      }
      messageBuffer.write(nextByte); // write byte to buffer
    }

    return messageBuffer.toByteArray();
  }
}
```

Textbook #1, pp 51-52

This method uses **ByteArrayOutoutStream** in order to use the **size()** method

```
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;

    // fetch bytes until find delimiter
    while ((nextByte = in.read()) != DELIMITER) {
      if (nextByte == -1) { // end of stream?
        if (messageBuffer.size() == 0) { // if no byte read
          return null;
        } else { // if bytes followed by end of stream: framing error
          throw new EOFException("Non-empty message without delimiter");
        }
      }
      messageBuffer.write(nextByte); // write byte to buffer
    }

    return messageBuffer.toByteArray();
  }
}
```

Textbook #1, pp 51-52
Download from the book site

This while loop scans for the **delimiter**

Textbook #1, pp 51-52
Download from the book site

```
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;

    // fetch bytes until find delimiter
    while ((nextByte = in.read()) != DELIMITER) {
      if (nextByte == -1) { // end of stream?
        if (messageBuffer.size() == 0) { // if no byte read
          return null;
        } else { // if bytes followed by end of stream: framing error
          throw new EOFException("Non-empty message without delimiter");
        }
      }
      messageBuffer.write(nextByte); // write byte to buffer
    }

    return messageBuffer.toByteArray();
  }
}
```

End-of-stream is reached

```
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;

    // fetch bytes until find delimiter
    while ((nextByte = in.read()) != DELIMITER) {
      if (nextByte == -1) { // end of stream?
        if (messageBuffer.size() == 0) { // if no byte read
          return null;
        } else { // if bytes followed by end of stream: framing error
          throw new EOFException("Non-empty message without delimiter");
        }
      }
      messageBuffer.write(nextByte); // write byte to buffer
    }

    return messageBuffer.toByteArray();
  }
}
```

Textbook #1, pp 51-52
Download from the book site

If the message is empty, that's OK

Textbook #1, pp 51-52
Downloadable from the book site

```
    out.write(message);
    out.write(DELIMITER);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
    ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;

    // fetch bytes until find delimiter
    while ((nextByte = in.read()) != DELIMITER) {
      if (nextByte == -1) { // end of stream?
        if (messageBuffer.size() == 0) { // if no byte read
          return null;
        } else { // if bytes followed by end of stream: framing error
          throw new EOFException("Non-empty message without delimiter");
        }
      }
      messageBuffer.write(nextByte); // write byte to buffer
    }

    return messageBuffer.toByteArray();
  }
}
```

Otherwise, it's a protocol error

LengthFramer

- The class **LengthFramer.java** implements length-based framing for messages up to 65,535 ($2^{16} - 1$) bytes in length.
  - The sender determines the length of the given message and writes it to the output stream as a two-byte, big-endian integer, followed by the complete message.
  - On the receiving side, we use a **DataInputStream** to read the length as an integer
  - The **readFully()** method blocks until the given array is completely full, which is exactly what we need here.
  - Note:
    - The sender does not have to inspect the content of the message being framed
    - It needs only to check that the message does not exceed the length limit.

Textbook #1, pp 53-54

Download from the book site

```
import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class LengthFramer implements Framer {
  public static final int MAXMESSAGELENGTH = 65535;
  public static final int BYTEMASK = 0xff;
  public static final int SHORTMASK = 0xffff;
  public static final int BYTESHIFT = 8;

  private DataInputStream in; // wrapper for data I/O

  public LengthFramer(InputStream in) throws IOException {
    this.in = new DataInputStream(in);
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    if (message.length > MAXMESSAGELENGTH) {
      throw new IOException("message too long");
    }
    // write length prefix
    out.write((message.length >> BYTESHIFT) & BYTEMASK);
    out.write(message.length & BYTEMASK);
    // write message
    out.write(message);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
```

Take the **InputStream** and wrap it in a **DataInputStream**

Textbook #1, pp 53-54
Download from the book site

```
import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class LengthFramer implements Framer {
  public static final int MAXMESSAGELENGTH = 65535;
  public static final int BYTEMASK = 0xff;
  public static final int SHORTMASK = 0xffff;
  public static final int BYTESHIFT = 8;

  private DataInputStream in; // wrapper for data I/O

  public LengthFramer(InputStream in) throws IOException {
    this.in = new DataInputStream(in);
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    if (message.length > MAXMESSAGELENGTH) {
      throw new IOException("message too long");
    }
    // write length prefix
    out.write((message.length >> BYTESHIFT) & BYTEMASK);
    out.write(message.length & BYTEMASK);
    // write message
    out.write(message);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
```

This block writes the message prefixed by its **length** (as an **unsigned short**).

Textbook #1, pp 53-54
Download from the book site

```
import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class LengthFramer implements Framer {
  public static final int MAXMESSAGELENGTH = 65535;
  public static final int BYTEMASK = 0xff;
  public static final int SHORTMASK = 0xffff;
  public static final int BYTESHIFT = 8;

  private DataInputStream in; // wrapper for data I/O

  public LengthFramer(InputStream in) throws IOException {
    this.in = new DataInputStream(in);
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    if (message.length > MAXMESSAGELENGTH) {
      throw new IOException("message too long");
    }
    // write length prefix
    out.write((message.length >> BYTESHIFT) & BYTEMASK);
    out.write(message.length & BYTEMASK);
    // write message
    out.write(message);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
```

Get the higher order byte

Textbook #1, pp 53-54

Download from the book site

```
import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class LengthFramer implements Framer {
  public static final int MAXMESSAGELENGTH = 65535;
  public static final int BYTEMASK = 0xff;
  public static final int SHORTMASK = 0xffff;
  public static final int BYTESHIFT = 8;

  private DataInputStream in; // wrapper for data I/O

  public LengthFramer(InputStream in) throws IOException {
    this.in = new DataInputStream(in);
  }

  public void frameMsg(byte[] message, OutputStream out) throws IOException {
    if (message.length > MAXMESSAGELENGTH) {
      throw new IOException("message too long");
    }
    // write length prefix
    out.write((message.length >> BYTESHIFT) & BYTEMASK);
    out.write(message.length & BYTEMASK);
    // write message
    out.write(message);
    out.flush();
  }

  public byte[] nextMsg() throws IOException {
```

Get the lower order byte

Textbook #1, pp 53-54
Download from the book site

```
public void frameMsg(byte[] message, OutputStream out) throws IOException {
  if (message.length > MAXMESSAGELENGTH) {
    throw new IOException("message too long");
  }
  // write length prefix
  out.write((message.length >> BYTESHIFT) & BYTEMASK);
  out.write(message.length & BYTEMASK);
  // write message
  out.write(message);
  out.flush();
}

public byte[] nextMsg() throws IOException {
  int length;
  try {
    length = in.readUnsignedShort(); // read 2 bytes
  } catch (EOFException e) { // no (or 1 byte) message
    return null;
  }
  // 0 <= length <= 65535
  byte[] msg = new byte[length];
  in.readFully(msg); // if exception, it's a framing error.
  return msg;
}
}
```

The **readUnsignedShort()** method reads two bytes (the message length), interprets them as a **big-endian integer**, and returns their value as an int.

```java
public void frameMsg(byte[] message, OutputStream out) throws IOException {
  if (message.length > MAXMESSAGELENGTH) {
    throw new IOException("message too long");
  }
  // write length prefix
  out.write((message.length >> BYTESHIFT) & BYTEMASK);
  out.write(message.length & BYTEMASK);
  // write message
  out.write(message);
  out.flush();
}

public byte[] nextMsg() throws IOException {
  int length;
  try {
    length = in.readUnsignedShort(); // read 2 bytes
  } catch (EOFException e) { // no (or 1 byte) message
    return null;
  }
  // 0 <= length <= 65535
  byte[] msg = new byte[length];
  in.readFully(msg); // if exception, it's a framing error.
  return msg;
}
}
```

Textbook #1, pp 53-54

Download from the book site

The **readfully()** method blocks until enough bytes to fill the given array have been returned.

Constructing/Parsing Protocol Messages

- We will close this chapter with a simple example to illustrate some techniques you might use to implement a protocol specified by someone else.
  - A simple “**voting**” (eVoting) protocol in this case, as shown in Figure 3.2 (next slide):

A client sends a **request** message to a server

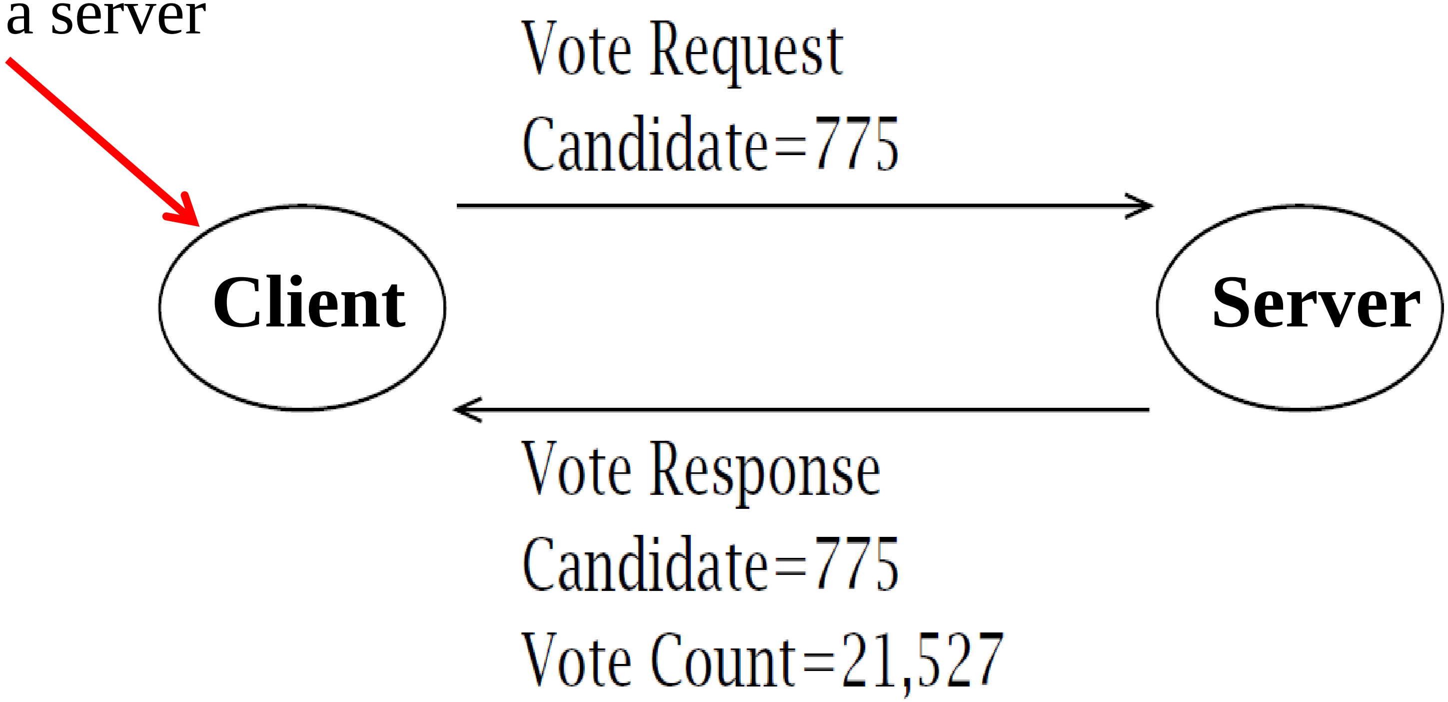


Figure 3.2: Voting protocol.

The message contains a **candidate ID**, which is an integer between 0 and 1000.

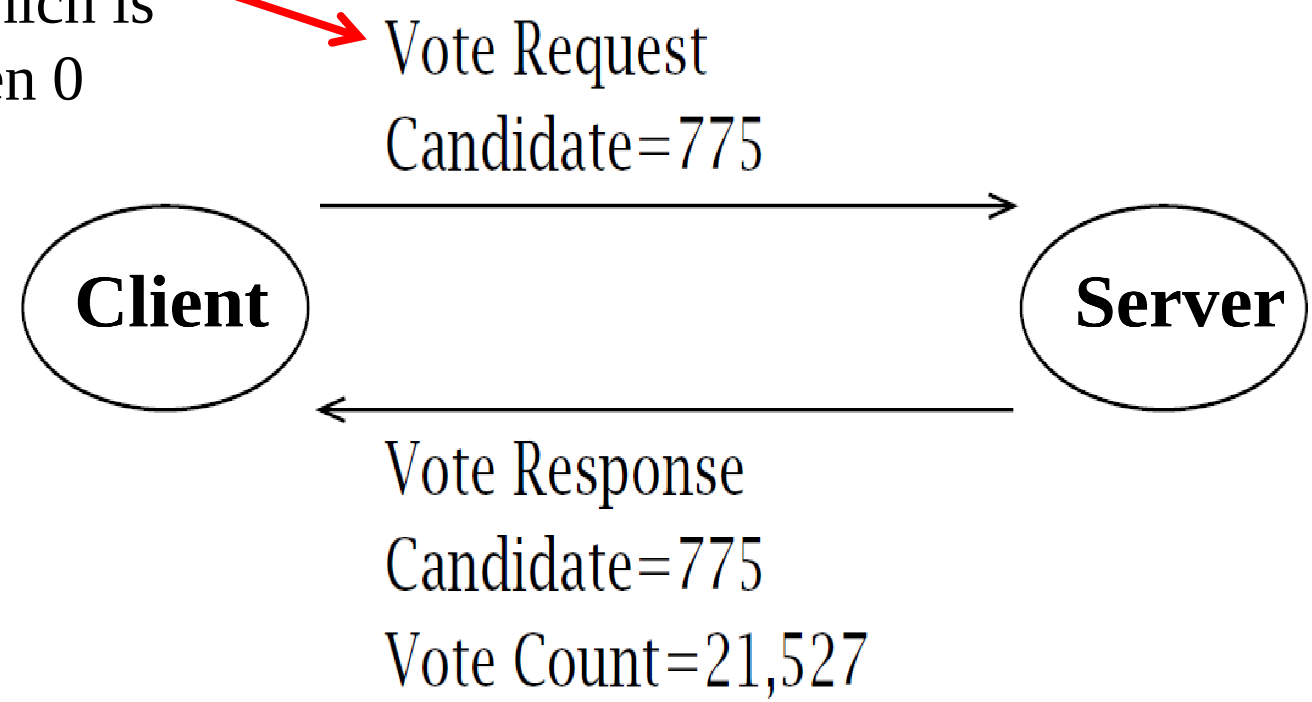


Figure 3.2: Voting protocol.

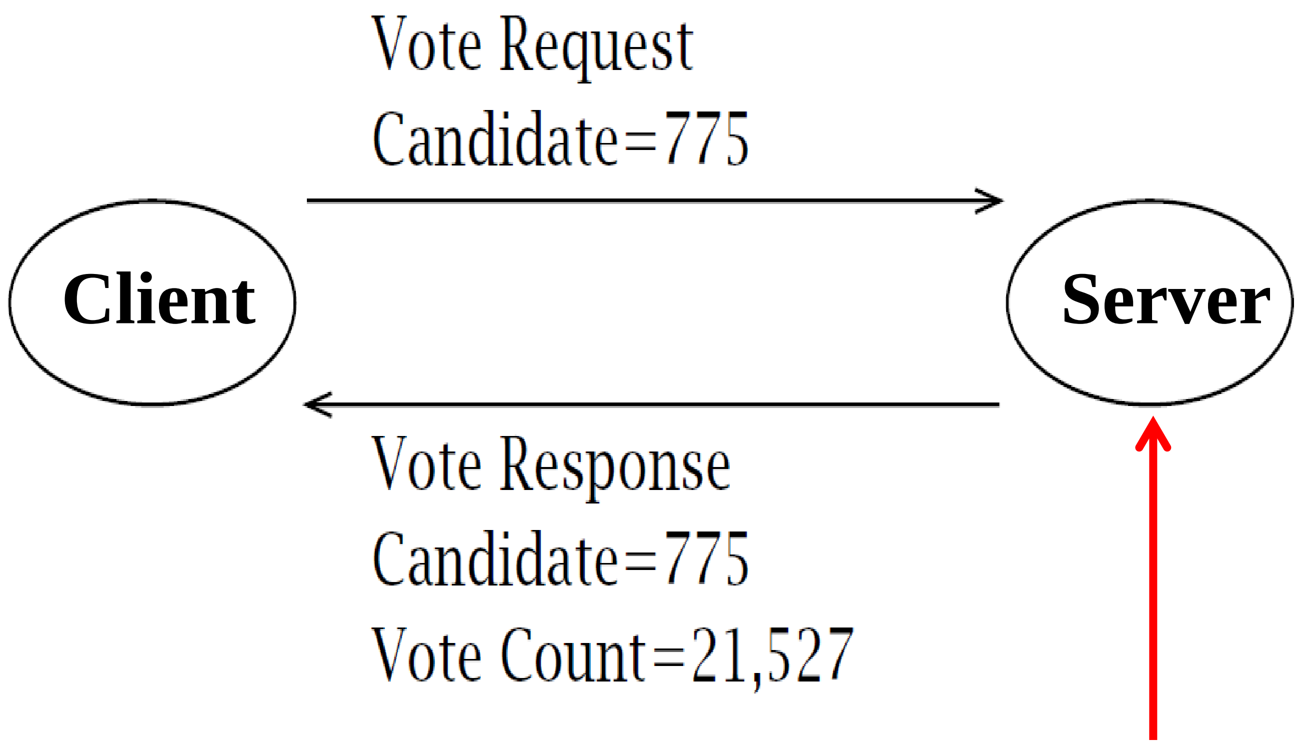


Figure 3.2: Voting protocol.

The server responds by sending a **response** message back to the client

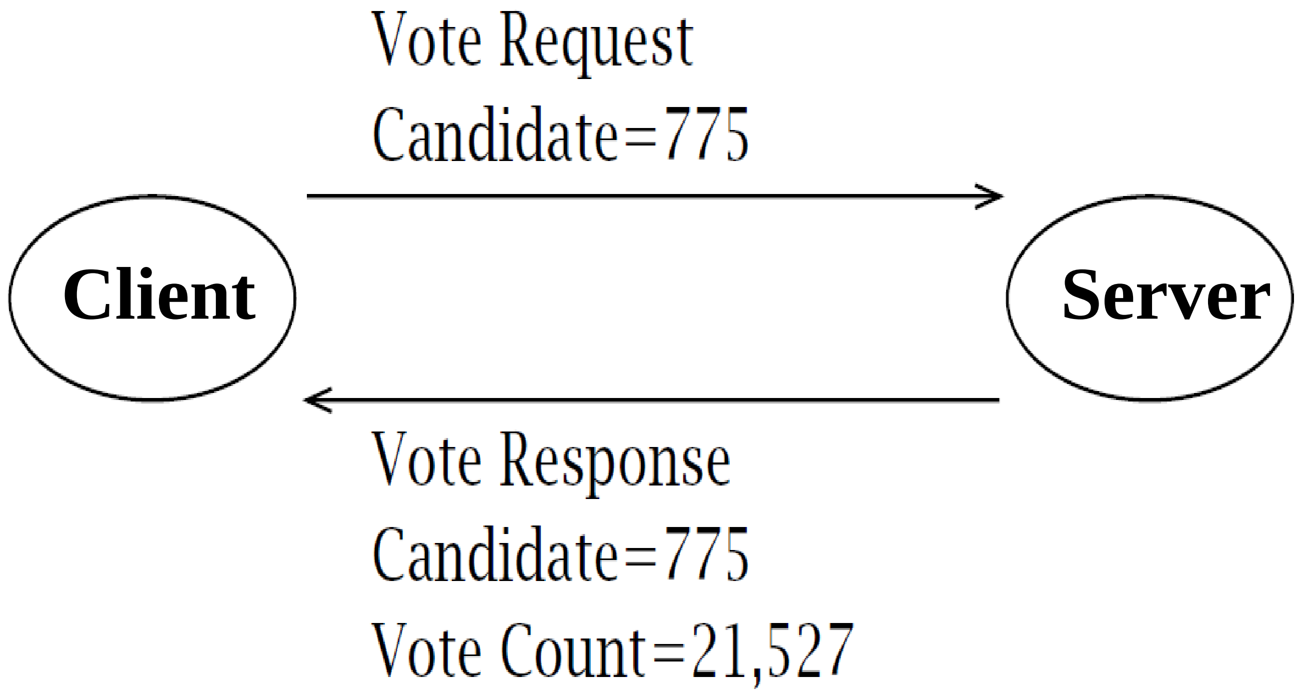


Figure 3.2: Voting protocol.

The response message contains the same **candidate ID**, plus a **VoteCount** (a long integer).

Requests

- In this "voting" protocol, 2 types of requests are supported:
  - An **inquiry** asks the server how many votes have been cast for the given candidate.
    - The server sends back a response message containing the original candidate ID and the vote total (as of the time the request was received) for that candidate.
  - A **voting** request actually casts a vote for the indicated candidate.
    - The server again responds with a message containing the candidate ID and the vote total, which includes the vote just cast.

Messages

- In implementing a protocol, it is helpful to define a class (**the message class**) to contain:
  - The information carried in a message
  - The methods for manipulating the fields of the message.

The Vote Message Class

- For our simple Voting example, the messages sent by client and server are very similar.
  - The only difference is that the messages sent by the server contain the vote count and a flag indicating that they are responses (not requests).
  - See Figure 3.2

VoteMsg.java

- The VoteMsg.java class shows the basic information in each message:
  - A boolean **isInquiry**, which is true if the requested transaction is an inquiry (and false if it is an actual vote)
  - A boolean **isResponse** indicating whether the message is a response (sent by the server) or request
  - An integer **candidateID** that identifies the candidate
  - A long **voteCount** indicating the vote total for the requested candidate
  - The class maintains the following invariants among the fields:
    - **candidateID** is in the range 0–1000.
    - **voteCount** is only nonzero in response messages (isResponse is true).
    - **voteCount** is non-negative.

```java
public class VoteMsg {
  private boolean isInquiry; // true if inquiry; false if vote
  private boolean isResponse;// true if response from server
  private int candidateID;   // in [0,1000]
  private long voteCount;    // nonzero only in response

  public static final int MAX_CANDIDATE_ID = 1000;

  public VoteMsg(boolean isResponse, boolean isInquiry, int candidateID, long voteCount)
      throws IllegalArgumentException {
    // check invariants
    if (voteCount != 0 && !isResponse) {
      throw new IllegalArgumentException("Request vote count must be zero");
    }
    if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
      throw new IllegalArgumentException("Bad Candidate ID: " + candidateID);
    }
    if (voteCount < 0) {
      throw new IllegalArgumentException("Total must be >= zero");
    }
    this.candidateID = candidateID;
    this.isResponse = isResponse;
    this.isInquiry = isInquiry;
    this.voteCount = voteCount;
  }

  public void setInquiry(boolean isInquiry) {
    this.isInquiry = isInquiry;
  }

  public void setResponse(boolean isResponse) {
```

Textbook #1, pp 56-58

Encoding/Decoding

❖ Now that we have a Java representation of a vote message, we need some way to **encode** and **decode** according to the protocol:

```
public interface VoteMsgCoder {
    byte[] toWire(VoteMsg msg) throws IOException;
    VoteMsg fromWire(byte[] input) throws IOException;
}
```

❖ The **toWire**() method <u>converts</u> the vote message to a sequence of bytes according to a particular protocol

❖ The **fromWire()** method <u>parses</u> a given sequence of bytes according to the same protocol and constructs an instance of the message class.

```java
import java.io.IOException;

public interface VoteMsgCoder {
  byte[] toWire(VoteMsg msg) throws IOException;
  VoteMsg fromWire(byte[] input) throws IOException;
}
```

Textbook #1, p. 58

Encoding/Decoding

- To illustrate the different methods of encoding information, we have two implementations of **VoteMsgCoder**:
  - **Text-based encoding**
  - **Binary encoding**

Text-Based Encoding

- The Vote protocol specifies that the text be encoded using the **ASCII**.
  - The message begins with a so-called "**magic string**"
    - A magic string is a sequence of characters that allows a recipient to quickly recognize the message as a Voting protocol message.
  - 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 strings.
  - The VoteMsgTextCoder provides a text-based encoding of VoteMsg.

```java
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

public class VoteMsgTextCoder implements VoteMsgCoder {
  /*
   * Wire Format "VOTEPROTO" <"v" | "i"> [<RESPFLAG>] <CANDIDATE> [<VOTECNT>]
   * Charset is fixed by the wire format.
   */

  // Manifest constants for encoding
  public static final String MAGIC = "Voting";
  public static final String VOTESTR = "v";
  public static final String INQSTR = "i";
  public static final String RESPONSESTR = "R";

  public static final String CHARSETNAME = "US-ASCII";
  public static final String DELIMSTR = " ";
  public static final int MAX_WIRE_LENGTH = 2000;

  public byte[] toWire(VoteMsg msg) throws IOException {
    String msgString = MAGIC + DELIMSTR + (msg.isInquiry() ? INQSTR : VOTESTR)
        + DELIMSTR + (msg.isResponse() ? RESPONSESTR + DELIMSTR : "")
        + Integer.toString(msg.getCandidateID()) + DELIMSTR
        + Long.toString(msg.getVoteCount());
    byte data[] = msgString.getBytes(CHARSETNAME);
    return data;
  }

  public VoteMsg fromWire(byte[] message) throws IOException {
```

Textbook #1, pp 59-60

Text-Based Encoding, cont.

- The **toWire**() method simply constructs a string containing all the fields of the message, separated by white space. (See previous slide)

- The **fromWire**() method first looks for the "magic string"
  - If it is not the first thing in the message, it throws an exception.
    - This illustrates a very important point about implementing protocols: never assume anything about any input from the network. (**Defensive programming**)
  - If the expected magic string is present, it gets the fields, token by token, using **Scanner**. (See next slides)
  - Note that the number of fields in the message depends on whether it is a request (sent by the client) or response (sent by the server).
  - **fromWire()** throws an exception if the input ends prematurely or is otherwise malformed.

The **fromWire**() method first looks for the "magic string"

```
  public VoteMsg fromWire(byte[] message) throws IOException {
    ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
    Scanner s = new Scanner(new InputStreamReader(msgStream, CHARSETNAME));
    boolean isInquiry;
    boolean isResponse;
    int candidateID;
    long voteCount;
    String token;

    try {
      token = s.next();
      if (!token.equals(MAGIC)) {
        throw new IOException("Bad magic string: " + token);
      }
      token = s.next();
      if (token.equals(VOTESTR)) {
        isInquiry = false;
      } else if (!token.equals(INQSTR)) {
        throw new IOException("Bad vote/inq indicator: " + token);
      } else {
        isInquiry = true;
      }

      token = s.next();
      if (token.equals(RESPONSESTR)) {
        isResponse = true;
        token = s.next();
      } else {
        isResponse = false;
      }
      // Current token is candidateID
      // Note: isResponse now valid
      candidateID = Integer.parseInt(token);
      if (isResponse) {
        token = s.next();
        voteCount = Long.parseLong(token);
      } else {
        voteCount = 0;
      }
    } catch (IOException ioe) {
      throw new IOException("Parse error...");
    }
    return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
  }
}
```

```java
  public VoteMsg fromWire(byte[] message) throws IOException {
    ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
    Scanner s = new Scanner(new InputStreamReader(msgStream, CHARSETNAME));
    boolean isInquiry;
    boolean isResponse;
    int candidateID;
    long voteCount;
    String token;

    try {
      token = s.next();
      if (!token.equals(MAGIC)) {
        throw new IOException("Bad magic string: " + token);
      }
      token = s.next();
      if (token.equals(VOTESTR)) {
        isInquiry = false;
      } else if (!token.equals(INQSTR)) {
        throw new IOException("Bad vote/inq indicator: " + token);
      } else {
        isInquiry = true;
      }

      token = s.next();
      if (token.equals(RESPONSESTR)) {
        isResponse = true;
        token = s.next();
      } else {
        isResponse = false;
      }
      // Current token is candidateID
      // Note: isResponse now valid
      candidateID = Integer.parseInt(token);
      if (isResponse) {
        token = s.next();
        voteCount = Long.parseLong(token);
      } else {
        voteCount = 0;
      }
    } catch (IOException ioe) {
      throw new IOException("Parse error...");
    }
    return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
  }
}
```

If it is not the first thing in the message, it throws an exception.

```
public VoteMsg fromWire(byte[] message) throws IOException {
  ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
  Scanner s = new Scanner(new InputStreamReader(msgStream, CHARSETNAME));
  boolean isInquiry;
  boolean isResponse;
  int candidateID;
  long voteCount;
  String token;

  try {
    token = s.next();
    if (!token.equals(MAGIC)) {
      throw new IOException("Bad magic string: " + token);
    }
    token = s.next();
    if (token.equals(VOTESTR)) {
      isInquiry = false;
    } else if (!token.equals(INQSTR)) {
      throw new IOException("Bad vote/inq indicator: " + token);
    } else {
      isInquiry = true;
    }

    token = s.next();
    if (token.equals(RESPONSESTR)) {
      isResponse = true;
      token = s.next();
    } else {
      isResponse = false;
    }
    // Current token is candidateID
    // Note: isResponse now valid
    candidateID = Integer.parseInt(token);
    if (isResponse) {
      token = s.next();
      voteCount = Long.parseLong(token);
    } else {
      voteCount = 0;
    }
  } catch (IOException ioe) {
    throw new IOException("Parse error...");
  }
  return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
 }
}
```

If the expected magic string is present, it gets the fields, token by token, using **Scanner**.

```java
  public VoteMsg fromWire(byte[] message) throws IOException {
    ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
    Scanner s = new Scanner(new InputStreamReader(msgStream, CHARSETNAME));
    boolean isInquiry;
    boolean isResponse;
    int candidateID;
    long voteCount;
    String token;

    try {
      token = s.next();
      if (!token.equals(MAGIC)) {
        throw new IOException("Bad magic string: " + token);
      }
      token = s.next();
      if (token.equals(VOTESTR)) {
        isInquiry = false;
      } else if (!token.equals(INQSTR)) {
        throw new IOException("Bad vote/inq indicator: " + token);
      } else {
        isInquiry = true;
      }

      token = s.next();
      if (token.equals(RESPONSESTR)) {
        isResponse = true;
        token = s.next();
      } else {
        isResponse = false;
      }
      // Current token is candidateID
      // Note: isResponse now valid
      candidateID = Integer.parseInt(token);
      if (isResponse) {
        token = s.next();
        voteCount = Long.parseLong(token);
      } else {
        voteCount = 0;
      }
    } catch (IOException ioe) {
      throw new IOException("Parse error...");
    }
    return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
  }
}
```

If the expected magic string is present, it gets the fields, token by token, using **Scanner**.

Binary Encoding

- In contrast with text-based encoding, binary encoding uses fixed-size messages. (See next slide)
  - Each message begins with a one-byte field that contains the "magic number" **010101** in its high-order six bits.
  - The two low-order bits of the first byte encode the two booleans: one for **inquiry/vote**, one for **request/response**.
  - 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**.

Textbook #1, pp 61-64

Binary Encoding

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

/* Wire Format
 *                                          1   1   1   1   1   1
 *   0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 * |     Magic             |Flags|          ZERO                   |
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 * |                   Candidate ID                                |
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 * |                                                               |
 * |          Vote Count (only in response)                        |
 * |                                                               |
 * |                                                               |
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 */
public class VoteMsgBinCoder implements VoteMsgCoder {

  // manifest constants for encoding
  public static final int MIN_WIRE_LENGTH = 4;
  public static final int MAX_WIRE_LENGTH = 16;
  public static final int MAGIC = 0x5400;
  public static final int MAGIC_MASK = 0xfc00;
  public static final int MAGIC_SHIFT = 8;
  public static final int RESPONSE_FLAG = 0x0200;
  public static final int INQUIRE_FLAG =  0x0100;
```

Binary Encoding, cont.

- As before, we create a **ByteArrayOutputStream** and wrap it in a **DataOutputStream** to receive the result.

- The encoding method takes advantage of the fact that the high-order two bytes of a valid **candidateID** are always zero.

```
// We know the candidate ID will fit in a short: it's > 0 && < 1000
out.writeShort((short) msg.getCandidateID());
```

- Note also the use of bitwise-or operations to encode the booleans using a single bit each:

Sending & Receiving

- Sending a message over a stream is simple:
  1. Create the message
  2. Call **toWire()**
  3. Adding appropriate framing information, and
  4. Write it to the stream.

- How about receiving a message over a stream?

Sending & Receiving

- To demonstrate this process, we can create a vote server that
  1. Maintains a mapping of candidate IDs to number of votes
  2. Counts submitted votes
  3. Responds to inquiries and votes with the current count for the specified candidate (aka **request handling**)
- We begin by implementing a service for use by vote servers:
  - See VoteService.java

Textbook #1, pp 63-64
Downloadable from the book site

```
import java.util.HashMap;
import java.util.Map;

public class VoteService {

  // Map of candidates to number of votes
  private Map<Integer, Long> results = new HashMap<Integer, Long>();

  public VoteMsg handleRequest(VoteMsg msg) {
    if (msg.isResponse()) { // If response, just send it back
      return msg;
    }
    msg.setResponse(true); // Make message a response
    // Get candidate ID and vote count
    int candidate = msg.getCandidateID();
    Long count = results.get(candidate);
    if (count == null) {
      count = 0L; // Candidate does not exist
    }
    if (!msg.isInquiry()) {
      results.put(candidate, ++count); // If vote, increment count
    }
    msg.setVoteCount(count);
    return msg;
  }
}
```

Create map of candidate ID to vote count For inquiries, the given candidate ID is used to look up the candidate's vote count in the map. For votes, the incremented vote count is stored back in the map.

```java
import java.util.HashMap;
import java.util.Map;

public class VoteService {

  // Map of candidates to number of votes
  private Map<Integer, Long> results = new HashMap<Integer, Long>();

  public VoteMsg handleRequest(VoteMsg msg) {
    if (msg.isResponse()) { // If response, just send it back
      return msg;
    }
    msg.setResponse(true); // Make message a response
    // Get candidate ID and vote count
    int candidate = msg.getCandidateID();
    Long count = results.get(candidate);
    if (count == null) {
      count = 0L; // Candidate does not exist
    }
    if (!msg.isInquiry()) {
      results.put(candidate, ++count); // If vote, increment count
    }
    msg.setVoteCount(count);
    return msg;
  }
}
```

java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

---

```
public interface Map<K,V>
```

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.

java.util

Class HashMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>

Direct Known Subclasses:

LinkedHashMap, PrinterStateReasons

```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Hash table based implementation of the `Map` interface. This implementation provides all of the optional map operations, and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (`get` and `put`), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the `HashMap` instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

```java
import java.util.HashMap;
import java.util.Map;

public class VoteService {

  // Map of candidates to number of votes
  private Map<Integer, Long> results = new HashMap<Integer, Long>();

  public VoteMsg handleRequest(VoteMsg msg) {
    if (msg.isResponse()) { // If response, just send it back
      return msg;
    }
    msg.setResponse(true); // Make message a response
    // Get candidate ID and vote count
    int candidate = msg.getCandidateID();
    Long count = results.get(candidate);
    if (count == null) {
      count = 0L; // Candidate does not exist
    }
    if (!msg.isInquiry()) {
      results.put(candidate, ++count); // If vote, increment count
    }
    msg.setVoteCount(count);
    return msg;
  }
}
```

If the message is already a response, we send it back without processing/modification. Otherwise we set the response flag.

Otherwise we set the response flag.

```java
import java.util.HashMap;
import java.util.Map;

public class VoteService {

  // Map of candidates to number of votes
  private Map<Integer, Long> results = new HashMap<Integer, Long>();

  public VoteMsg handleRequest(VoteMsg msg) {
    if (msg.isResponse()) { // If response, just send it back
      return msg;
    }
    msg.setResponse(true); // Make message a response
    // Get candidate ID and vote count
    int candidate = msg.getCandidateID();
    Long count = results.get(candidate);
    if (count == null) {
      count = 0L; // Candidate does not exist
    }
    if (!msg.isInquiry()) {
      results.put(candidate, ++count); // If vote, increment count
    }
    msg.setVoteCount(count);
    return msg;
  }
}
```

Find the candidate by ID in the map and fetch the vote count.

If the candidate ID does not already exist in the map, set the count to 0.

Method Summary

Methods

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map.
Object	**clone**() Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.
boolean	**containsKey**(Object key) Returns `true` if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns `true` if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
V	**get**(Object key) Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
boolean	**isEmpty**() Returns `true` if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map.
void	**putAll**(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map.
V	**remove**(Object key)

```java
import java.util.HashMap;
import java.util.Map;

public class VoteService {

  // Map of candidates to number of votes
  private Map<Integer, Long> results = new HashMap<Integer, Long>();

  public VoteMsg handleRequest(VoteMsg msg) {
    if (msg.isResponse()) { // If response, just send it back
      return msg;
    }
    msg.setResponse(true); // Make message a response
    // Get candidate ID and vote count
    int candidate = msg.getCandidateID();
    Long count = results.get(candidate);
    if (count == null) {
      count = 0L; // Candidate does not exist
    }
    if (!msg.isInquiry()) {
      results.put(candidate, ++count); // If vote, increment count
    }
    msg.setVoteCount(count);
    return msg;
  }
}
```

If the candidate did not previously exist, this creates a new mapping; otherwise, it simply modifies an existing mapping.

Method Summary

Methods

Modifier and Type	Method and Description
void	clear() Removes all of the mappings from this map.
Object	clone() Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() Returns a Set view of the keys contained in this map.
V	put(K key, V value) Associates the specified value with the specified key in this map.
void	putAll(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map.
V	remove(Object key) Removes the mapping for the specified key from this map if present.

The TCP Vote Server

- The server repeatedly accepts a new client connection and uses the **VoteService** to generate responses to the client vote messages.

Textbook #1, pp 66-67

The server repeatedly accepts a new client connection and uses the **VoteService** to generate responses

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class VoteServerTCP {

  public static void main(String args[]) throws Exception {

    if (args.length != 1) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port>");
    }

    int port = Integer.parseInt(args[0]); // Receiving Port

    ServerSocket servSock = new ServerSocket(port);
    // Change Bin to Text on both client and server for different encoding
    VoteMsgCoder coder = new VoteMsgBinCoder();
    VoteService service = new VoteService();

    while (true) {
      Socket clntSock = servSock.accept();
      System.out.println("Handling client at " + clntSock.getRemoteSocketAddress());
      // Change Length to Delim for a different framing strategy
      Framer framer = new LengthFramer(clntSock.getInputStream());
      try {
        byte[] req;
        while ((req = framer.nextMsg()) != null) {
          System.out.println("Received message (" + req.length + " bytes)");
          VoteMsg responseMsg = service.handleRequest(coder.fromWire(req));
          framer.frameMsg(coder.toWire(responseMsg), clntSock.getOutputStream());
        }
```

Establish coder and vote service for server

Textbook #1, pp 66-67
Downloadable from the book site

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class VoteServerTCP {

  public static void main(String args[]) throws Exception {

    if (args.length != 1) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port>");
    }

    int port = Integer.parseInt(args[0]); // Receiving Port

    ServerSocket servSock = new ServerSocket(port);
    // Change Bin to Text on both client and server for different encoding
    VoteMsgCoder coder = new VoteMsgBinCoder();
    VoteService service = new VoteService();

    while (true) {
      Socket clntSock = servSock.accept();
      System.out.println("Handling client at " + clntSock.getRemoteSocketAddress());
      // Change Length to Delim for a different framing strategy
      Framer framer = new LengthFramer(clntSock.getInputStream());
      try {
        byte[] req;
        while ((req = framer.nextMsg()) != null) {
          System.out.println("Received message (" + req.length + " bytes)");
          VoteMsg responseMsg = service.handleRequest(coder.fromWire(req));
          framer.frameMsg(coder.toWire(responseMsg), clntSock.getOutputStream());
        }
```

We could easily switch to using delimiter-based framing and/or text encoding simply by changing them to **VoteMsgTextCoder** and **DelimFramer**, respectively.

The TCP Vote Server

```
// Fetch and decode messages from client
while ((req = framer.nextMsg()) != null) {
   // Repeatedly extracts next message from framer until it
   // returns null, indicating an end of messages.

   System.out.println("Received message (" + req.length + " bytes)");

   // Pass the decoded message to the voting service for handling
   VoteMsg responseMsg = service.handleRequest(coder.fromWire(req));

   // Encode, frame, and send the returned response message
   framer.frameMsg(coder.toWire(responseMsg),
                                                   clntSock.getOutputStream());
}
```

The TCP Voting Client

- Next we show how to implement a TCP voting client that connects over a TCP socket to the voting server, sends an inquiry followed by a vote, and then receives the inquiry and vote responses.

http://cs.ecs.baylor.edu/~donahoo/practical/JavaSockets2/code/ cs.ecs.baylor.edu

Textbook #1, pp 64-66

```
import java.io.OutputStream;
import java.net.Socket;

public class VoteClientTCP {

  public static final int CANDIDATEID = 888;

  public static void main(String args[]) throws Exception {

    if (args.length != 2) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Server> <Port>");
    }

    String destAddr = args[0]; // Destination address
    int destPort = Integer.parseInt(args[1]); // Destination port

    Socket sock = new Socket(destAddr, destPort);
    OutputStream out = sock.getOutputStream();

    // Change Bin to Text for a different framing strategy
    VoteMsgCoder coder = new VoteMsgBinCoder();
    // Change Length to Delim for a different encoding strategy
    Framer framer = new LengthFramer(sock.getInputStream());

    // Create an inquiry request (2nd arg = true)
    VoteMsg msg = new VoteMsg(false, true, CANDIDATEID, 0);
    byte[] encodedMsg = coder.toWire(msg);

    // Send request
    System.out.println("Sending Inquiry (" + encodedMsg.length + " bytes): "
    System.out.println(msg);
```

The TCP Voting Client

```
// Change Bin to Text for a different framing strategy
VoteMsgCoder coder = new VoteMsgBinCoder();
// Change Length to Delim for a different encoding strategy
Framer framer = new LengthFramer(sock.getInputStream());
```

- We encode/decode our vote messages using a coder, a binary encoder in this case.
- For framing, we use the LengthFramer, which prefixes each message with a length.
- Note that we could easily switch to using delimiter-based framing and/or text encoding simply by changing the concrete types of VoteMsgCoder and Framer to **VoteMsgTextCoder** and **DelimFramer**, respectively.

The TCP Voting Client: Sending

```
// Create an inquiry request (2nd arg = true)
VoteMsg msg = new VoteMsg(false, true, CANDIDATEID, 0);
byte[] encodedMsg = coder.toWire(msg);
// Send request
System.out.println("Sending Inquiry (" + encodedMsg.length + " bytes): ");
System.out.println(msg);
framer.frameMsg(encodedMsg, out);
// Now send a vote request
msg.setInquiry(false);
encodedMsg = coder.toWire(msg);
System.out.println("Sending Vote (" + encodedMsg.length + " bytes): ");
framer.frameMsg(encodedMsg, out);
```

- Create, encode, frame and send an inquiry, followed by a vote message for the same candidate.

The TCP Voting Client: Receiving

```
// Receive inquiry response
encodedMsg = framer.nextMsg();
msg = coder.fromWire(encodedMsg);
System.out.println("Received Response (" + encodedMsg.length + " bytes): ");
System.out.println(msg);
// Receive vote response
msg = coder.fromWire(framer.nextMsg());
System.out.println("Received Response (" + encodedMsg.length + " bytes): ");
System.out.println(msg);
```

- The **nextMsg()** method returns the contents of the next encoded message, which we parse/decode via fromWire().

Reading Assignment

- Textbook #1: **TCP/IP Sockets in Java**, Chapter 3: Sending/Receiving Data
  - BruteForceCoding.java (pp. 42-45), VoteClientUDP.java & VoteServerUDP.java (pp. 68-70) are optional.