CSE 686 Internet Programming

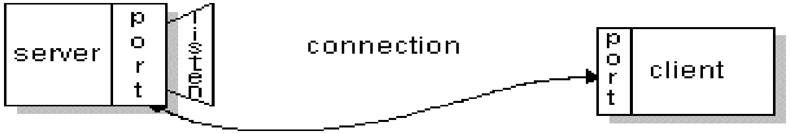
Weeks 3, 4: Sockets

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

January 31, February 5, 2018

TCP Sockets

- In client-server applications
  - The server provides services
  - The client uses those services



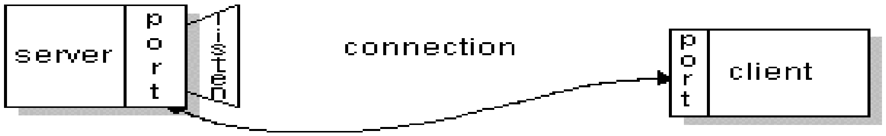
- The communication between them must be reliable
  - No data can be dropped
  - Data must arrive on the client side in the same order in which the server sent it.
- TCP (but not UPD) provides such a reliable communication channel
  - The client and the server establish a connection between them
  - Each program binds a **socket** to its end of the connection.
  - To communicate, the client and the server each reads from and writes to the socket.

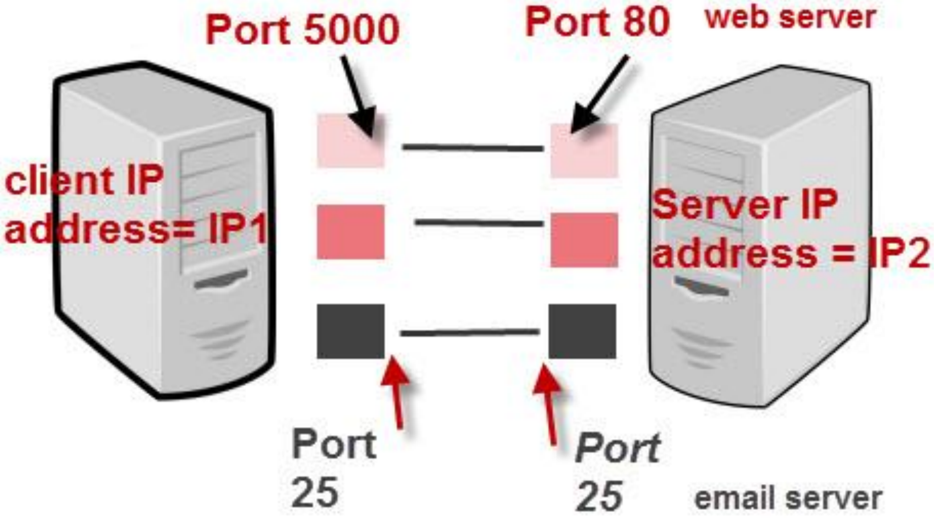
TCP Sockets: Definitions

- A **socket** is one endpoint of a two-way communication between two programs running on the network.

- An **endpoint** is a combination of an **IP address** and a **port number**.

- Every TCP connection can be uniquely identified by its two endpoints.





IP Address + Port number = Socket

TCP/IP Ports And Sockets

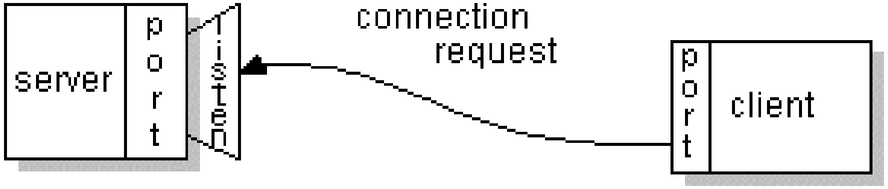
TCP Sockets: Definitions

- The java.net package provides two TCP socket classes:
  - The **Socket** class that implements the client side of the connection
  - The **ServerSocket** class that implements the server side of the connection

TCP Sockets: Client vs. Server

- Normally a **server** runs on a specific computer and has a **socket** (server socket) that is bound to a specific **port**.
  - The server **listens** to the socket for a client to make a connection request.
- If a client wants to connect to a server, it needs to know the **hostname** of the machine on which the server is running, and the **port number** on which the server is listening.
  - To make a connection request, the client tries to rendezvous with the server on the server's machine and port.
  - The client also needs to identify itself to the server so it **binds** to a **local port number** that it will use during this connection. (This is usually assigned by the system -

    See next slide)

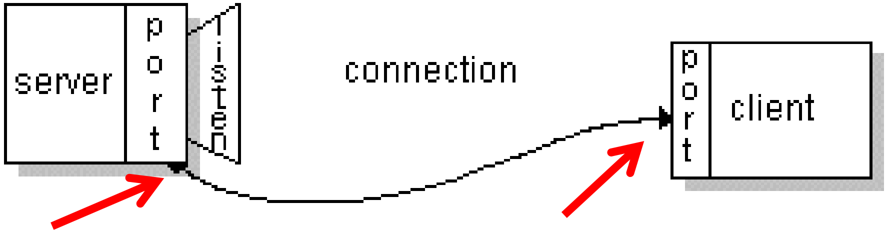


2.4 Exercises (Textbook #1)

1. For **TCPEchoServer.java**, we explicitly specify the port to the socket in the constructor. We said that a socket must have a port for communication, yet we do not specify a port in **TCPEchoClient.java**. How is the echo client’s socket assigned a port? (Page 38)

TCP Sockets: Client vs. Server, cont.

- If everything goes well, the server **accepts** the connection.
  - Upon acceptance, the server creates a **new socket** bound to the same local port, and also has the new socket's **remote endpoint** set to the address and port of the client.



  - It needs this new socket so that it can continue to listen to the original socket (server socket) for connection requests while tending to the needs of the newly connected client.
- On the client side:
  - If the connection is accepted, that means a socket is successfully created, and so the client can use it to communicate with the server.
- The client and server can now communicate by writing to or reading from their sockets.

TCP Client/Server Interaction

Server starts by getting ready to receive client connections…

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

TCP Client/Server Interaction

ServerSocket servSock = new ServerSocket(servPort);

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

TCP Client/Server Interaction

```
for (;;) {
            Socket clntSock =   servSock.accept();
```

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

TCP Client/Server Interaction

Server is now blocked waiting for connection from a client

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

TCP Client/Server Interaction

Later, a client decides to talk to the server…

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

TCP Client/Server Interaction

```
Socket sock = new Socket(server, servPort);
```

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

TCP Client/Server Interaction

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

TCP Client/Server Interaction

Socket clntSock = servSock.accept(); // back to server side

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

TCP Client/Server Interaction

```
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

TCP Client/Server Interaction

```
close(sock); // client side
close(clntSock); // server side
```

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

Assignment #1: Due February 9

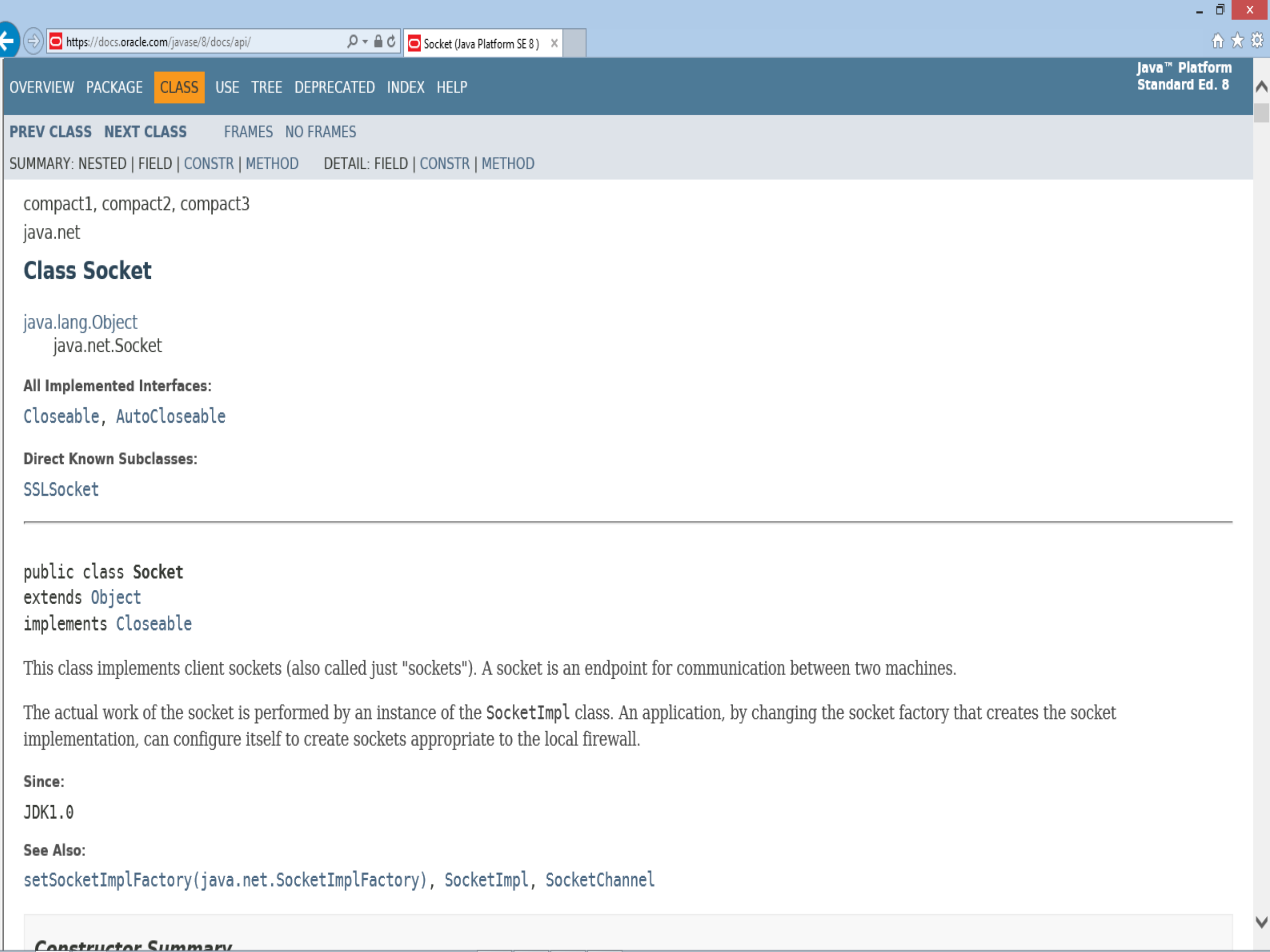
1. Read Chapter 2 of Textbook #1: TCP/IP Sockets in Java, but you may skip Section 2.3 UDP Sockets.
2. Fully understand how TCPEchoClient.java & TCPEchoServer.java work.
3. Answer Question #4 of Section 2.4 Exercises (on page 38), but ignore the part that involves UDPEchoServer.java. (**1 point**)
4. Based on your answers to Step 3, or more specifically, your answers to "Examine the server examples (TCPEchoServer.java in our case) and list anything you can think of that a client might do to cause it to give poor service to other clients," revise TCPEchoClient.java to create such a client, which I call a **bad client**. Your **bad client** should attempt to cause the server to provide poor services to other clients, in at least 2 different ways. ***(2 points, 1 for each bad behavior)***

Assignment #1: Due February 9

5. Based on your answers to Step 3, or more specifically, your answers to "Suggest improvements to fix the problems that you find," revise TCPEchoServer.java to incorporate those improvements. You may call this improved server a **bullet-proof** server. ***(1 points, 1 for each improvement)***
6. Submit i) your answers to Step 3, ii) the revised client program (the bad client) resulting from Step 4, and iii) the improved server program (the bullet-proof server) resulting from Step 5, via Blackboard.
7. Please compile and test your Java programs before submitting them to Blackboard.
8. Please bring your questions about any aspect of this assignment to the class, if possible.

TCP Sockets in Java

- The java.net.**Socket** class is Java's fundamental class for performing client-side TCP operations.
- Other client-oriented classes that make TCP network connections such as **URL**, **URLConnection**, and Applet all ultimately end up invoking the methods of this class.
- This class itself uses native code to communicate with the local TCP stack of the host operating system.


https://docs.oracle.com/javase/8/docs/api/
Socket (Java Platform SE 8)
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java™ Platform
Standard Ed. 8
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.net

Class Socket

java.lang.Object
    java.net.Socket

All Implemented Interfaces:

`Closeable`, `AutoCloseable`

Direct Known Subclasses:

`SSLSocket`

```
public class Socket
extends Object
implements Closeable
```

This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

The actual work of the socket is performed by an instance of the `SocketImpl` class. An application, by changing the socket factory that creates the socket implementation, can configure itself to create sockets appropriate to the local firewall.

Since:

`JDK1.0`

See Also:

`setSocketImplFactory(java.net.SocketImplFactory)`, `SocketImpl`, `SocketChannel`

Constructor Summary

Basic Constructors

- Each Socket constructor specifies the **host** and the **port** to connect to.
- Hosts may be specified as an InetAddress or a String.

public Socket(String **host**, int **port**) throws UnknownHostException,
IOException

public Socket(InetAddress **host**, int **port**) throws IOException

  - Ports are specified as integer values from 1 to 65535

- These constructors connect the socket, that is, an active network connection is established to the remote host.
- If the connection can't be opened for whatever reason, the constructor throws an **IOException** or an **UnknownHostException**.

An Example

```
try {
        Socket toSU = new Socket("www.syr.edu", 80);
        // send and receive data...
} catch (UnknownHostException ex) { // DNS can’t resolve the hostname
        System.err.println(ex);
} catch (IOException ex) { // socket can’t be opened
        System.err.println(ex);
}
```

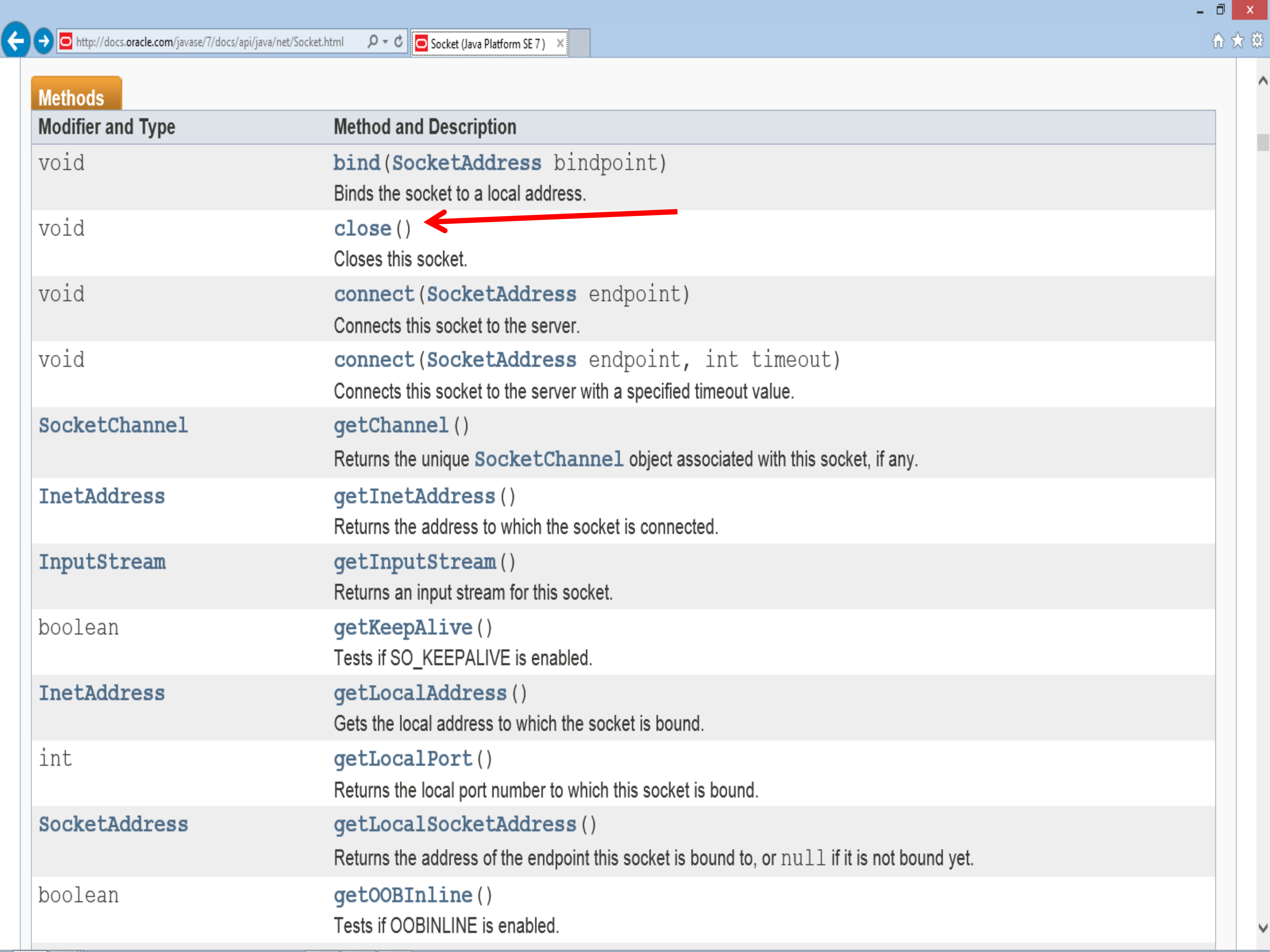
- The reasons that the socket can’t be opened may include:
  - Wrong port number
  - Routing problems preventing your packets from reaching their destination.
  - Hotel WiFi service blocking you for not paying $14.95

SocketTester.java

```
import java.net.*;
import java.io.*;

public class SocketTester {
        public static void main(String[] args) {
                try {
                        Socket toSU = new Socket("www.syr.edu", 80);
                        // send and receive data...
                        System.out.println(toSU); // uses toString()
                } catch (UnknownHostException ex) { // DNS can’t resolve the hostname
                        System.err.println(ex);
                } catch (IOException ex) { // socket can’t be opened
                        System.err.println(ex);
                }
        }
}
```

Using Socket() to Scan Ports

```
import java.net.*;
import java.io.*;
public class LowPortScanner { // From Textbook #2
  public static void main(String[] args) {
    String host = args.length > 0 ? args[0] : "localhost";
    for (int i = 1; i < 1024; i++) {
      try {
        Socket s = new Socket(host, i);
        System.out.println("There is a server on port " + i + " of " + host);
        s.close();
      } catch (UnknownHostException ex) {
        System.err.println(ex);
        break;
      } catch (IOException ex) {
        // no server on this port
      }
    }
  }
}
```


http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html
Socket (Java Platform SE 7)

Methods

Modifier and Type	Method and Description
void	bind(SocketAddress bindpoint) Binds the socket to a local address.
void	close() Closes this socket.
void	connect(SocketAddress endpoint) Connects this socket to the server.
void	connect(SocketAddress endpoint, int timeout) Connects this socket to the server with a specified timeout value.
SocketChannel	getChannel() Returns the unique SocketChannel object associated with this socket, if any.
InetAddress	getInetAddress() Returns the address to which the socket is connected.
InputStream	getInputStream() Returns an input stream for this socket.
boolean	getKeepAlive() Tests if SO_KEEPALIVE is enabled.
InetAddress	getLocalAddress() Gets the local address to which the socket is bound.
int	getLocalPort() Returns the local port number to which this socket is bound.
SocketAddress	getLocalSocketAddress() Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.
boolean	getOOBInline() Tests if OOBINLINE is enabled.

Creating Unconnected Sockets

- Three constructors create unconnected sockets.
- These provide more control over exactly how the underlying socket behaves, for instance by choosing a different proxy server or an encryption scheme:

```
public Socket()
public Socket(Proxy proxy) // Choose a different proxy server
protected Socket(SocketImpl impl) // Use your own socket implementation
```

- Ignore the last 2 for now.

Creating Unconnected Sockets

- If you give no arguments to the Socket constructor, it has nowhere to connect to:
- You can connect later by passing a SocketAddress to one of the connect() methods:

```
try { // revise SocketTester.java to do this
        Socket socket = new Socket();
        // fill in socket options
        SocketAddress toSU = new InetSocketAddress(“www.syr.edu”, 80);
        socket.connect(address);
        // work with the sockets...
} catch (IOException ex) {
        System.err.println(ex);
}
```

The SocketAddress Class

- The **SocketAddress** class represents a Socket Address.
- The primary purpose of the SocketAddress class is to provide a convenient way for storing socket connection information such as the **IP address** and **port** that can be reused to create new sockets, even after the original socket is disconnected and garbage collected.
- To this end, the Socket class offers two methods that return SocketAddress objects
  - **getRemoteSocketAddress**() returns the address of the system being connected to
  - **getLocalSocketAddress()** returns the address from which the connection is made

SocketAddress: An Example

```
import java.io.IOException;
import java.net.*;

public class SATester {
        public static void main(String[] args) {
                try {
                        Socket socket = new Socket("www.syr.edu", 80);
                        SocketAddress su = socket.getRemoteSocketAddress();
                        System.out.println(su); // uses toString()
                        SocketAddress me = socket.getLocalSocketAddress();
                        System.out.println(me); // uses toString()
                        socket.close();

                        // later on..,
                        Socket socket2 = new Socket(); // unonnected socket
                        socket2.connect(su);
                } catch (UnknownHostException ex) { System.err.println(ex);
                } catch (IOException ex) { System.err.println(ex); }
        }
}
```

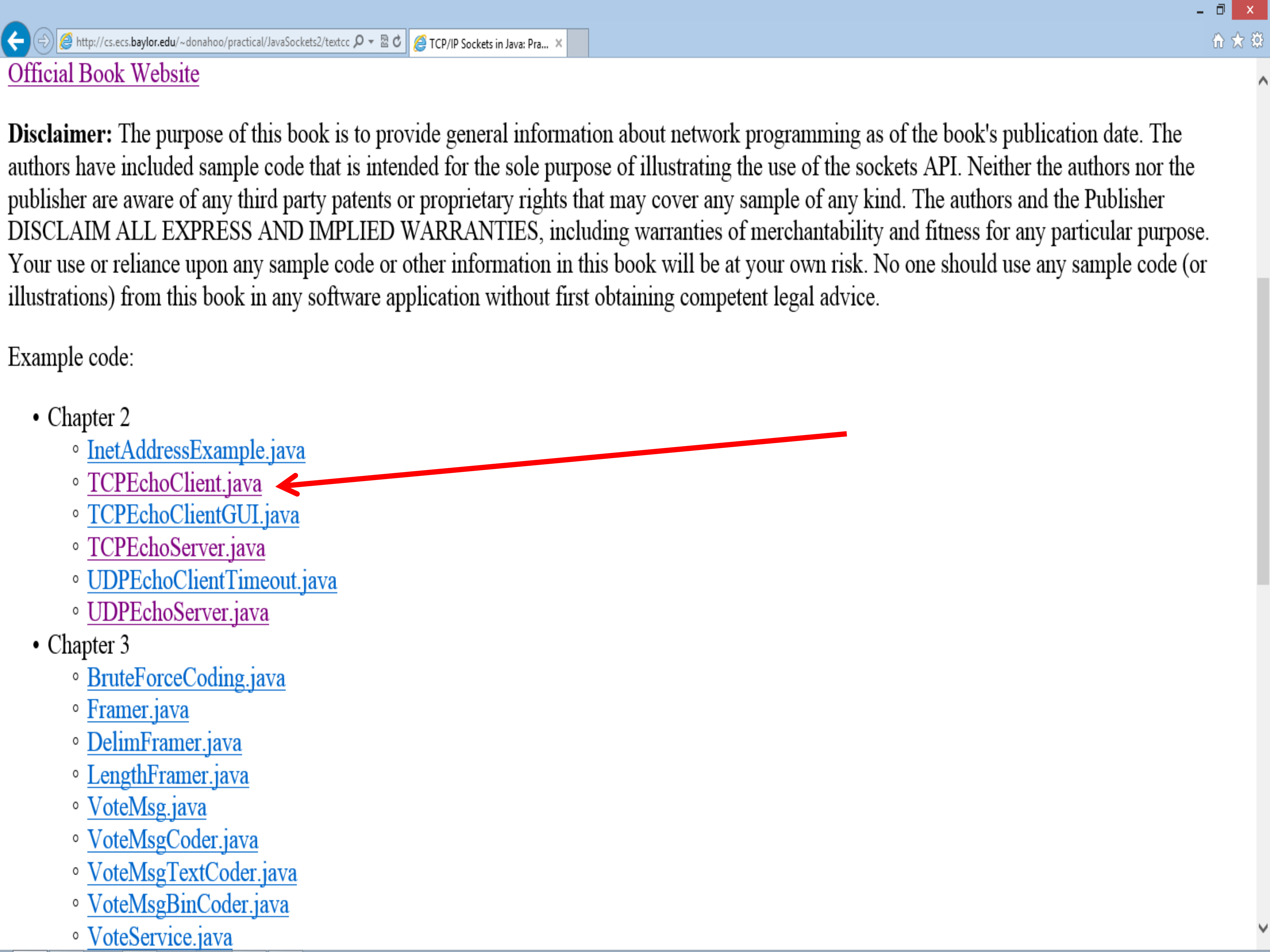
The InetSocketAddress Class

- The **InetSocketAddress** class (which is the one and only known subclass of **SocketAddress**) is usually created with a host and a port (for clients) or just a port (for servers):

```
public InetSocketAddress(InetAddress address, int port)
public InetSocketAddress(String host, int port)
public InetSocketAddress(int port)
```

Polymorphism Revisited

```
try { // revise SocketTester.java to do this
        Socket socket = new Socket();
        // fill in socket options
        SocketAddress toSU = new
                InetSocketAddress(“www.syr.edu”, 80);
        socket.connect(address);
        // work with the sockets...
} catch (IOException ex) {
        System.err.println(ex);
}
```


http://cs.ecs.baylor.edu/~donahoo/practical/JavaSockets2/textcc
TCP/IP Sockets in Java: Pra...

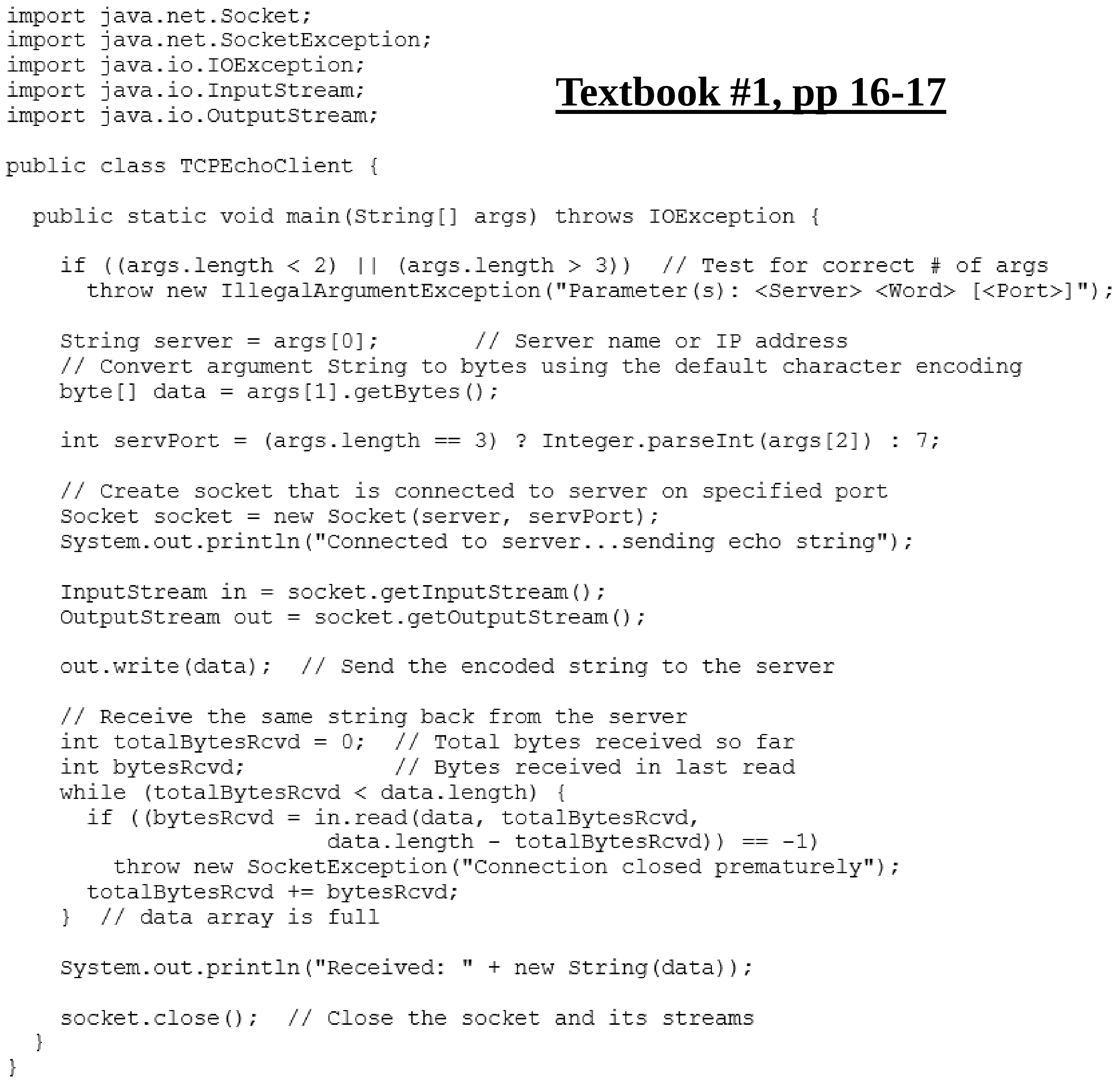
Official Book Website

Disclaimer: The purpose of this book is to provide general information about network programming as of the book's publication date. The authors have included sample code that is intended for the sole purpose of illustrating the use of the sockets API. Neither the authors nor the publisher are aware of any third party patents or proprietary rights that may cover any sample of any kind. The authors and the Publisher DISCLAIM ALL EXPRESS AND IMPLIED WARRANTIES, including warranties of merchantability and fitness for any particular purpose. Your use or reliance upon any sample code or other information in this book will be at your own risk. No one should use any sample code (or illustrations) from this book in any software application without first obtaining competent legal advice.

Example code:

- Chapter 2
  - InetAddressExample.java
  - TCPEchoClient.java
  - TCPEchoClientGUI.java
  - TCPEchoServer.java
  - UDPEchoClientTimeout.java
  - UDPEchoServer.java
- Chapter 3
  - BruteForceCoding.java
  - Framer.java
  - DelimFramer.java
  - LengthFramer.java
  - VoteMsg.java
  - VoteMsgCoder.java
  - VoteMsgTextCoder.java
  - VoteMsgBinCoder.java
  - VoteService.java

Textbook #1, pp 16-17

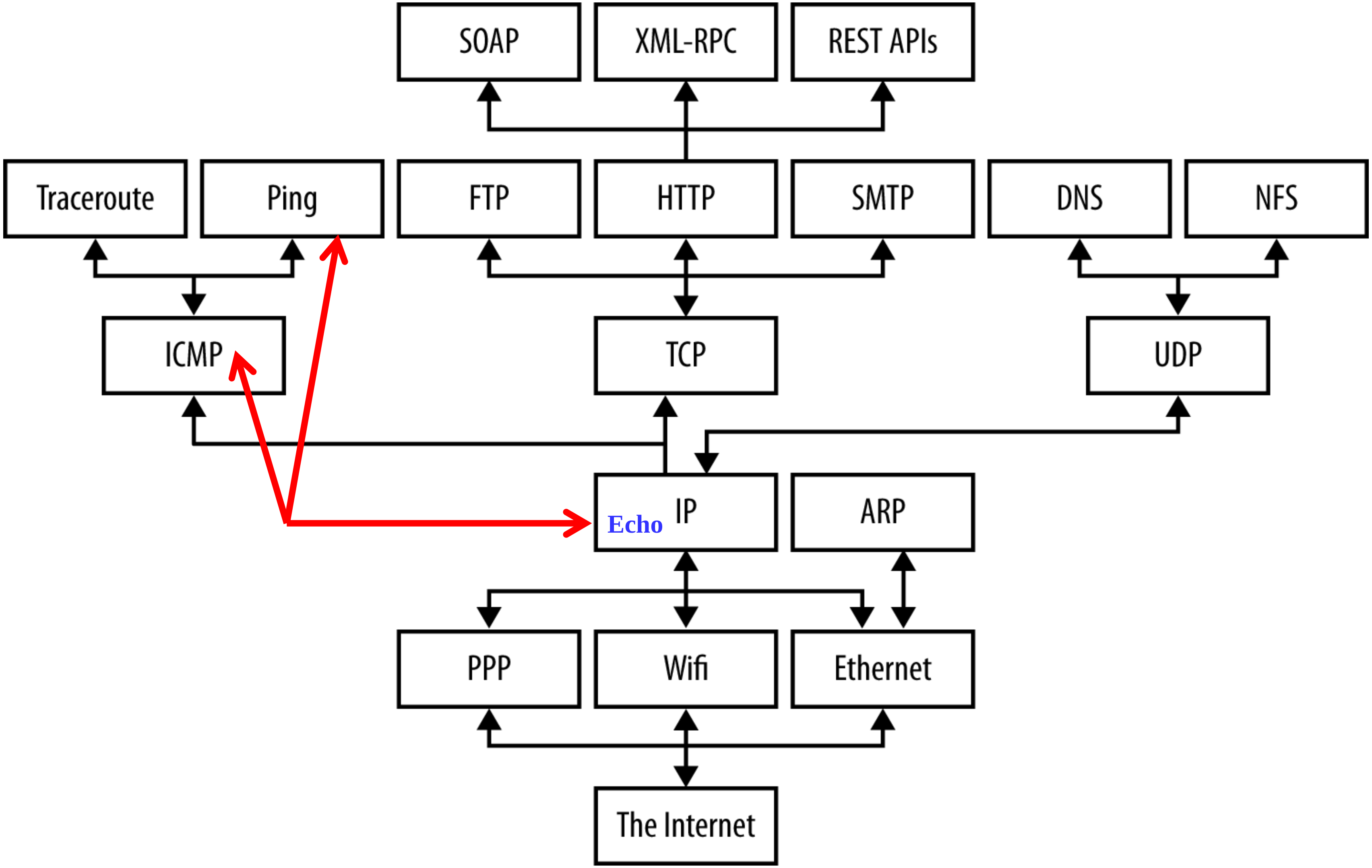


```
import java.net.Socket;
import java.net.SocketException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

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");

    InputStream in = socket.getInputStream();
    OutputStream out = socket.getOutputStream();

    out.write(data);  // Send the encoded string to the server

    // Receive the same string back from the server
    int totalBytesRcvd = 0;  // Total bytes received so far
    int bytesRcvd;           // Bytes received in last read
    while (totalBytesRcvd < data.length) {
      if ((bytesRcvd = in.read(data, totalBytesRcvd,
                        data.length - totalBytesRcvd)) == -1)
        throw new SocketException("Connection closed prematurely");
      totalBytesRcvd += bytesRcvd;
    }  // data array is full

    System.out.println("Received: " + new String(data));

    socket.close();  // Close the socket and its streams
  }
}
```

The Echo Protocol

- The **Echo Protocol** is a service in the Internet Protocol Suite.
  - It was originally proposed for testing and measurement of round-trip times in IP networks.
    - The server sends back an identical copy of the data it received.
  - A host may connect to a server that supports the Echo Protocol using TCP or UDP on the well-known port number **7**.
  - The functionality of the Echo Protocol was superseded by the echo request of the **Internet Control Message Protocol** and the corresponding **ping** utility.

Protocols (Revisited)



TCPEchoClient

- The TCPEchoClient goes through three steps:

1. Construct an instance of Socket: The constructor establishes a TCP connection to the specified remote host and port.

```
// 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");
```

2. Communicate using the socket’s I/O streams: A connected instance of Socket contains an InputStream and OutputStream that can be used just like any other Java I/O stream

3. Close the connection using the close() method of Socket:

```
socket.close(); // Close the socket and its streams
```

TCP Client/Server Interaction

```
Socket sock = new Socket(server, servPort);
```

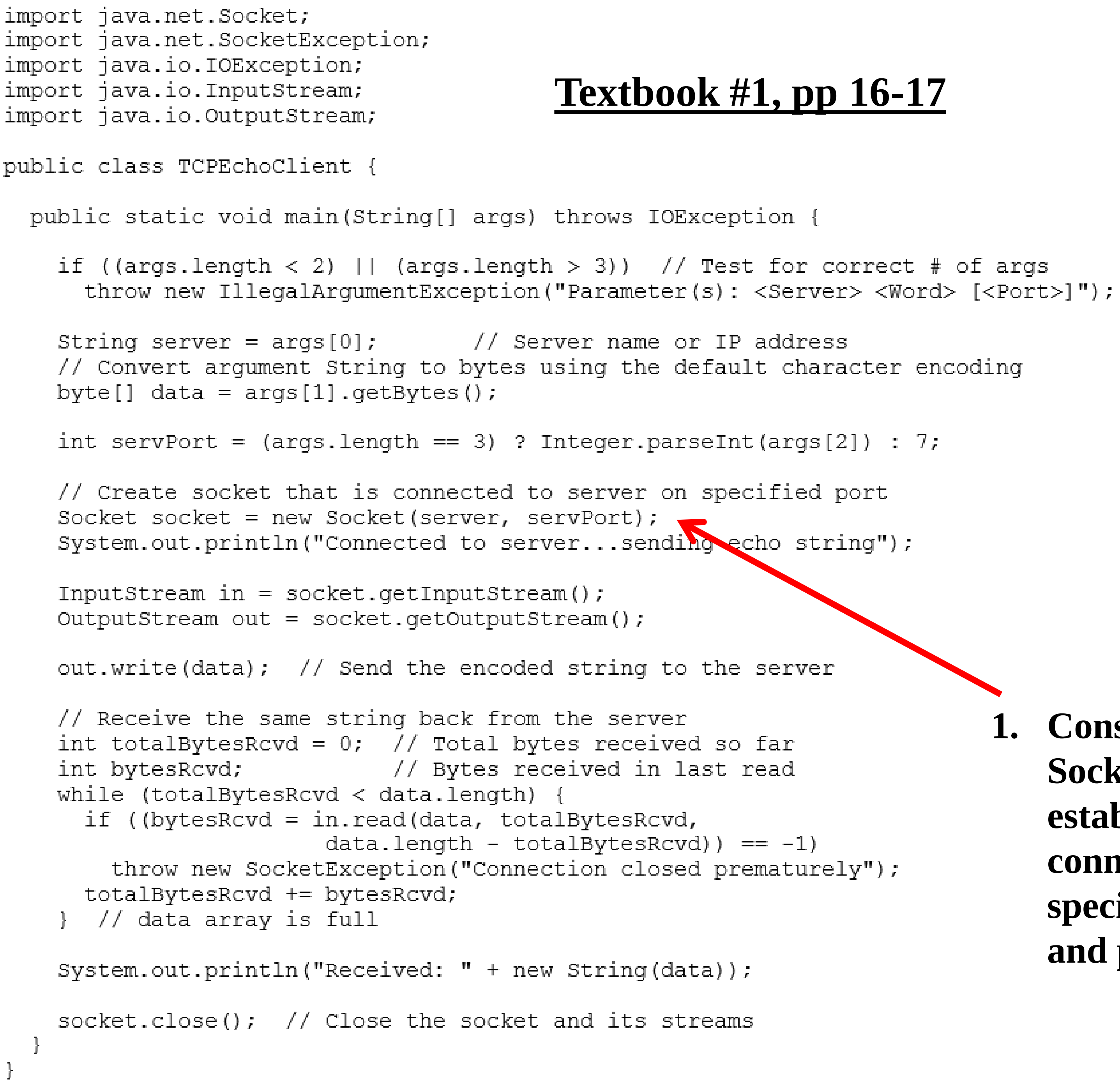
Client

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

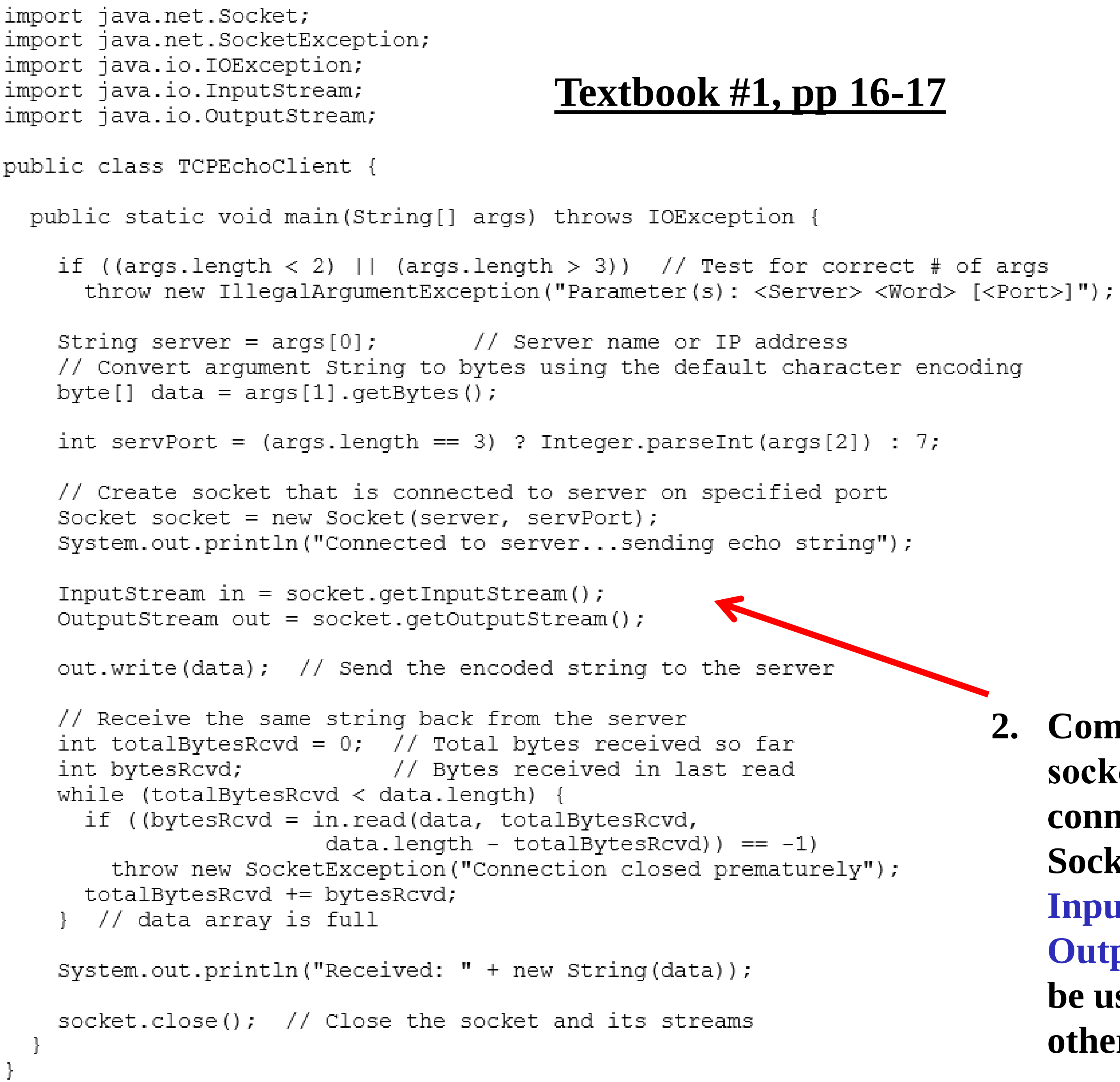
Textbook #1, pp 16-17



```
import java.net.Socket;
import java.net.SocketException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

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");

    InputStream in = socket.getInputStream();
    OutputStream out = socket.getOutputStream();

    out.write(data);  // Send the encoded string to the server

    // Receive the same string back from the server
    int totalBytesRcvd = 0;  // Total bytes received so far
    int bytesRcvd;           // Bytes received in last read
    while (totalBytesRcvd < data.length) {
      if ((bytesRcvd = in.read(data, totalBytesRcvd,
                      data.length - totalBytesRcvd)) == -1)
        throw new SocketException("Connection closed prematurely");
      totalBytesRcvd += bytesRcvd;
    }  // data array is full

    System.out.println("Received: " + new String(data));

    socket.close();  // Close the socket and its streams
  }
}
```

1. **Construct an instance of Socket: The constructor establishes a TCP connection to the specified remote host and port.**

Textbook #1, pp 16-17



```
import java.net.Socket;
import java.net.SocketException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

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");

    InputStream in = socket.getInputStream();
    OutputStream out = socket.getOutputStream();

    out.write(data);  // Send the encoded string to the server

    // Receive the same string back from the server
    int totalBytesRcvd = 0;  // Total bytes received so far
    int bytesRcvd;           // Bytes received in last read
    while (totalBytesRcvd < data.length) {
      if ((bytesRcvd = in.read(data, totalBytesRcvd,
                        data.length - totalBytesRcvd)) == -1)
        throw new SocketException("Connection closed prematurely");
      totalBytesRcvd += bytesRcvd;
    }  // data array is full

    System.out.println("Received: " + new String(data));

    socket.close();  // Close the socket and its streams
  }
}
```

2. Communicate using the socket's I/O streams: A connected instance of Socket contains an InputStream and OutputStream that can be used just like any other Java I/O stream

The Socket Class: IO Operations

public InputStream **getInputStream()** throws IOException

public OutputStream **getOutputStream()** throws IOException

- Communication with the remote system takes place via the associated I/O streams, which are obtained through the **getInputStream**() and **getOutputStream()** methods.

The Socket Class: IO Operations

public void **shutdownInput()** throws IOException

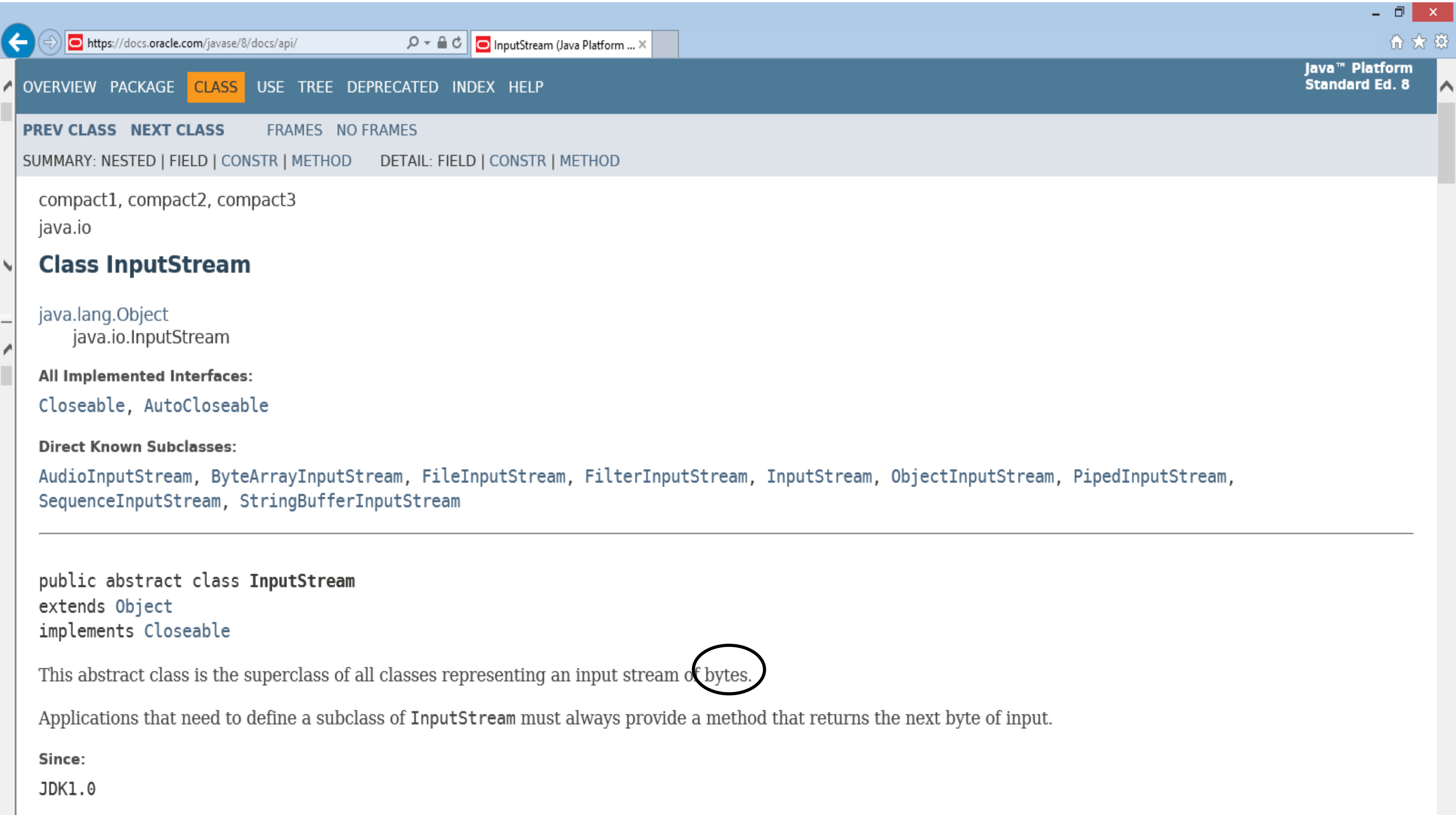
public void **shutdownOutput()** throws IOException

- The **shutdownInput()** method closes the input side of a TCP stream.
  - Any unread data is silently discarded, including data buffered by the socket, data in transit, and data arriving in the future.
  - Any subsequent attempt to read from the socket will cause an exception to be thrown.
- The **shutdownOutput()** method has a similar effect on the output stream, but the implementation will attempt to ensure that any data already written to the socket’s output stream is delivered to the other end.

The InputStream Class

- Java’s basic input class is java.io.InputStream:

  public abstract class **InputStream.**


https://docs.oracle.com/javase/8/docs/api/
InputStream (Java Platform ...
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java™ Platform Standard Ed. 8
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.io

Class InputStream

java.lang.Object
    java.io.InputStream

All Implemented Interfaces:

Closeable, AutoCloseable

Direct Known Subclasses:

AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, InputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream

```
public abstract class InputStream
extends Object
implements Closeable
```

This abstract class is the superclass of all classes representing an input stream of bytes.

Applications that need to define a subclass of `InputStream` must always provide a method that returns the next byte of input.

Since:

JDK1.0

The InputStream Class

- This class provides the fundamental methods needed to read data as **raw bytes**.
  - The read methods are:

    public abstract int **read**() throws IOException

    public int **read**(byte[] input) throws IOException

    public int **read**(byte[] input, int offset, int length)
    throws IOException

  - Other supporting method:

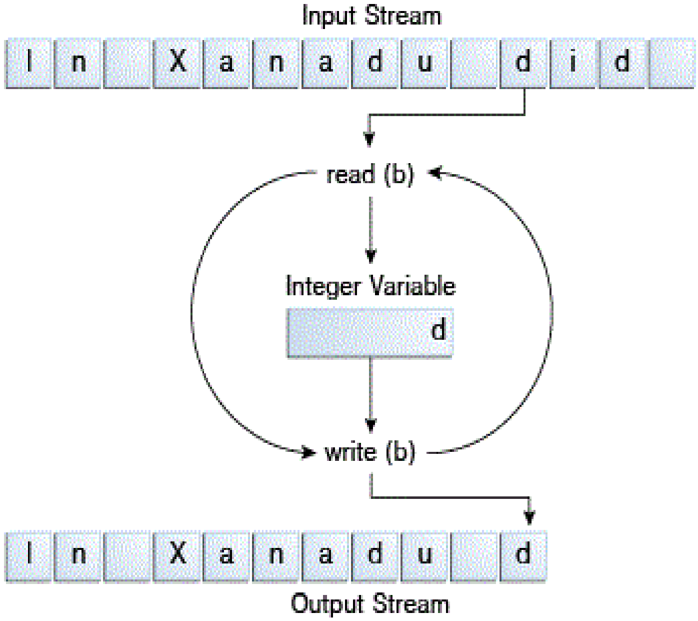
    public int **available**() throws IOException

    public long **skip**(long n) throws IOException

    public void **close**() throws IOException

The InputStream Class

public abstract int read()



- The basic method of InputStream is the **noargs** read() method.
- This method reads a single byte of data from the input stream's source and returns it as an **int** from 0 to 255.
- End of stream is signified by returning –1.
- The read() method **waits and blocks** execution of any code that follows it until a byte of data is available and ready to be read.
- The read() method is declared abstract because subclasses need to change it to handle their particular medium.
  - For example, a **FileInputStream,** which is its one of its subclasses**,** can implement this method that copies the byte from a file.

An Simple read() Example

```
 // read 10 bytes from an InputStream
byte[] input = new byte[10];

for (int i = 0; i < input.length; i++) {
        int b = in.read(); // in is an InputStream object
        if (b == -1) break;
        input[i] = (byte) b;
}
```

The InputStream Class

public int **read**(byte[] input)

public int **read**(byte[] input, int offset, int length)

- Reading a byte at a time is inefficient, and hence these two methods.
  - The first method **attempts** to fill the specified array input.
  - The second method **attempts** to fill the specified sub-array of input, starting at offset and continuing for length bytes.
  - If such an attempt fails, an IOException would be raised
  - Sometimes, an attempt won’t completely fail but won’t completely succeed either.
    - Some of the requested bytes may be read, but not all of them.
    - To account for this, the multibyte read methods return the number of bytes actually read.

The InputStream Class

- Don’t do this:

```
byte[] input = new byte[1024];
int bytesRead = in.read(input); // in is an InputStream object
```

- Do this instead:

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
        bytesRead += in.read(input, bytesRead, bytesToRead - bytesRead);
}
```

The InputStream Class

- Or do this:

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
    int result = in.read(input, bytesRead, bytesToRead - bytesRead);
    if (result == -1) break; // end of stream
    bytesRead += result;
}
```

The InputStream Class

public int available()

- ❖ If you do not want to wait until all the bytes you need are immediately available, you can use the **available()** method to determine how many bytes can be read without blocking.
  - ❖ This returns the minimum number of bytes you can read.
  - ❖ You may in fact be able to read more, but you will be able to read at least as many bytes as **available()** suggests:

```
int bytesAvailable = in.available(); // in is an InputStream
byte[] input = new byte[bytesAvailable];
int bytesRead = in.read(input, 0, bytesAvailable);
// continue with rest of program immediately...
```

The InputStream Class

public long skip(long n)

- ❖ On rare occasions, you may want to skip over data without reading it, and this method does that.
  - ❖ This is less useful on network connections than when reading from files:
    - ❖ Network connections are <u>sequential</u> and normally quite slow, so it’s not significantly more time consuming to read data than to skip over it.
    - ❖ Files are <u>random access</u> so that skipping can be implemented simply by repositioning a file pointer rather than processing each byte to be skipped.

The InputStream Class

public void close()

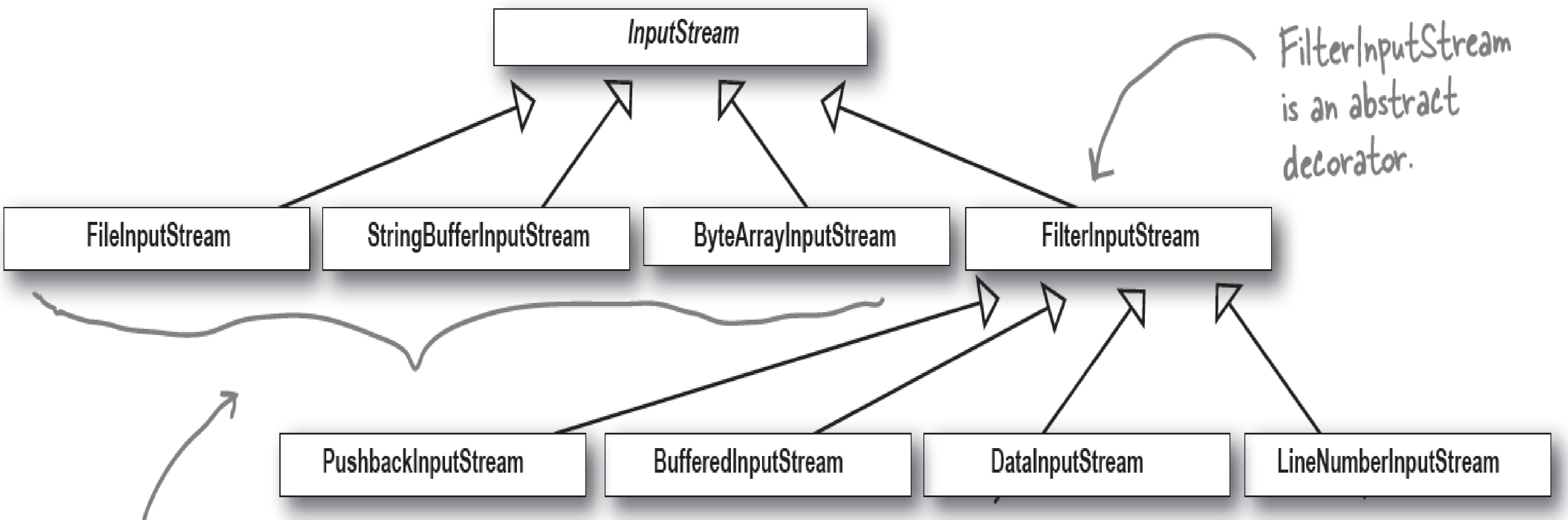
- ❖ Once your program has finished with an input stream, it should close it by invoking its close() method.
  - ❖ This releases any resources associated with the stream, such as file handles or ports.
  - ❖ Once an input stream has been closed, further reads from it throw IOExceptions.

The InputStream Class

- Subclasses of InputStream use these methods to read data from particular media.
  - A **FileInputStream** reads data from a file.
  - A **ByteArrayInputStream** reads data from an array of bytes.
  - But whichever source you're reading, you mostly use only those same six methods.
  - Sometimes you may not even know exactly what kind of stream you're reading from:
    - It's returned by various methods in various classes in java.net, such as the **getInputStream()** method of java.net.Socket.
      - These methods are declared to return only InputStream, not the more specific subclass.
      - That's the power of polymorphism.
        - If you know how to use the superclass, you know how to use all the subclasses, too.

Here's our abstract component.

FilterInputStream is an abstract decorator.



These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.

And finally, here are all our concrete decorators.

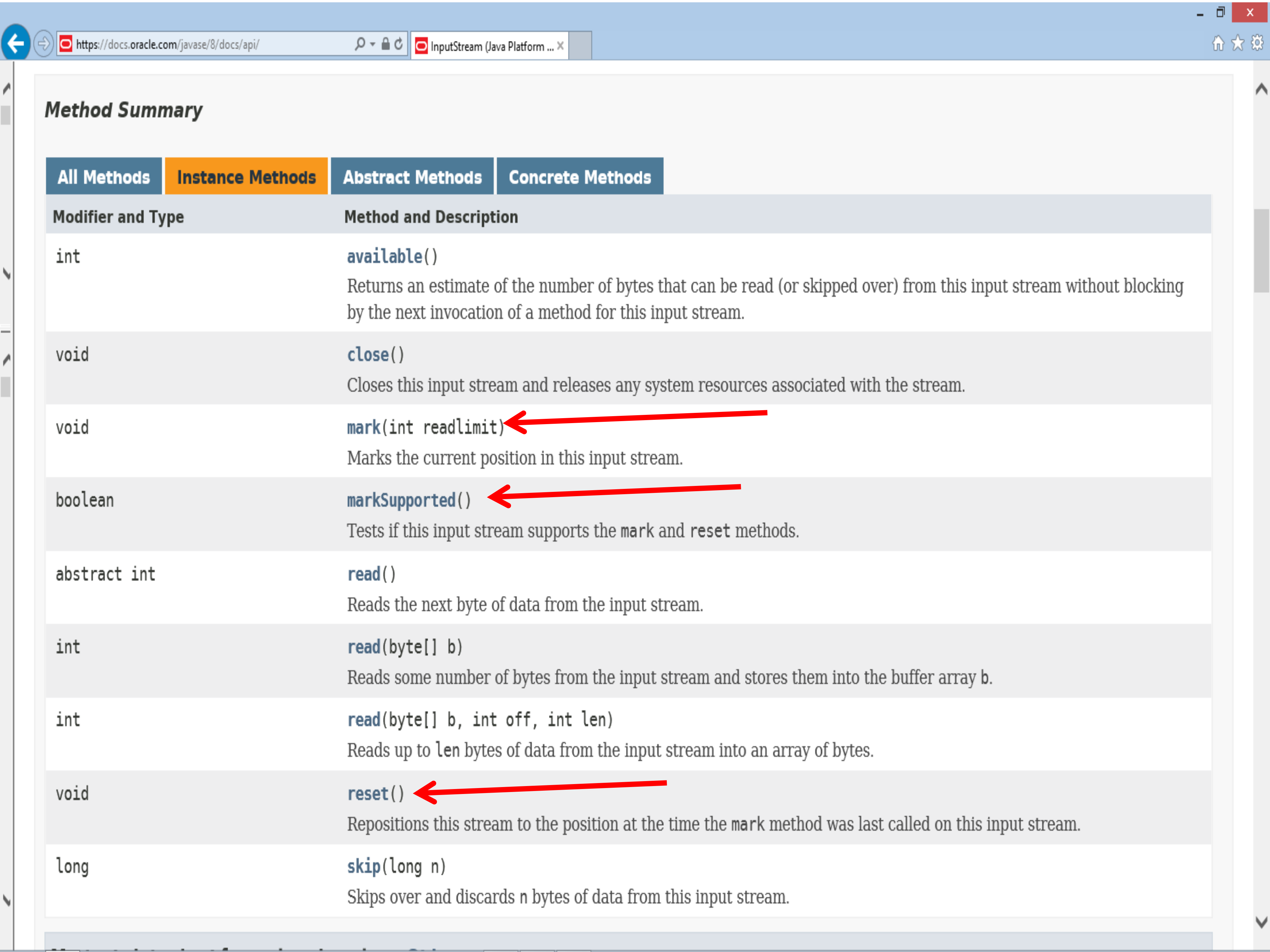
Marking and Resetting

- The InputStream class also has three less commonly used methods that allow programs to back up and **reread** data they've already read.
  - They are:

    public void **mark**(int readAheadLimit)

    public void **reset**() throws IOException

    public boolean **markSupported**()


https://docs.oracle.com/javase/8/docs/api/
InputStream (Java Platform ...
All Methods
Instance Methods
Abstract Methods
Concrete Methods

Method Summary

Modifier and Type	Method and Description
`int`	`available()` Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
`void`	`close()` Closes this input stream and releases any system resources associated with the stream.
`void`	`mark(int readlimit)` Marks the current position in this input stream.
`boolean`	`markSupported()` Tests if this input stream supports the `mark` and `reset` methods.
`abstract int`	`read()` Reads the next byte of data from the input stream.
`int`	`read(byte[] b)` Reads some number of bytes from the input stream and stores them into the buffer array `b`.
`int`	`read(byte[] b, int off, int len)` Reads up to `len` bytes of data from the input stream into an array of bytes.
`void`	`reset()` Repositions this stream to the position at the time the `mark` method was last called on this input stream.
`long`	`skip(long n)` Skips over and discards `n` bytes of data from this input stream.

Marking and Resetting

public void **mark**(int readAheadLimit)

- In order to reread data, mark the current position in the stream with the **mark()** method.
- The **readAheadLimit** argument to mark() specifies how far back you are allowed to reset the data.
- There can be only one mark in a stream at any given time - marking a second location erases the first mark.

public void **reset**() throws IOException

- At a later point, you can reset the stream to the marked position using the **reset()** method.
- Subsequent reads then return data starting from the marked position.
- If you try to reset too far back, an IOException is thrown.

Marking and Resetting

public boolean markSupported()

- Not all input streams (subclasses of InputStream) support this.
    - BufferedInputStream and ByteArrayInputStream do.
- Before trying to use marking and resetting, check whether the **markSupported()** method returns true.
    - If it does, the stream supports marking and resetting.
    - Otherwise, mark() will do nothing and reset() will throw an IOException.

The OutputStream Class

- Java’s basic output class is java.io.OutputStream:

  public abstract class OutputStream

- It provides the fundamental methods needed to write data.
  - The write methods are:

    public abstract void **write**(int b) throws IOException

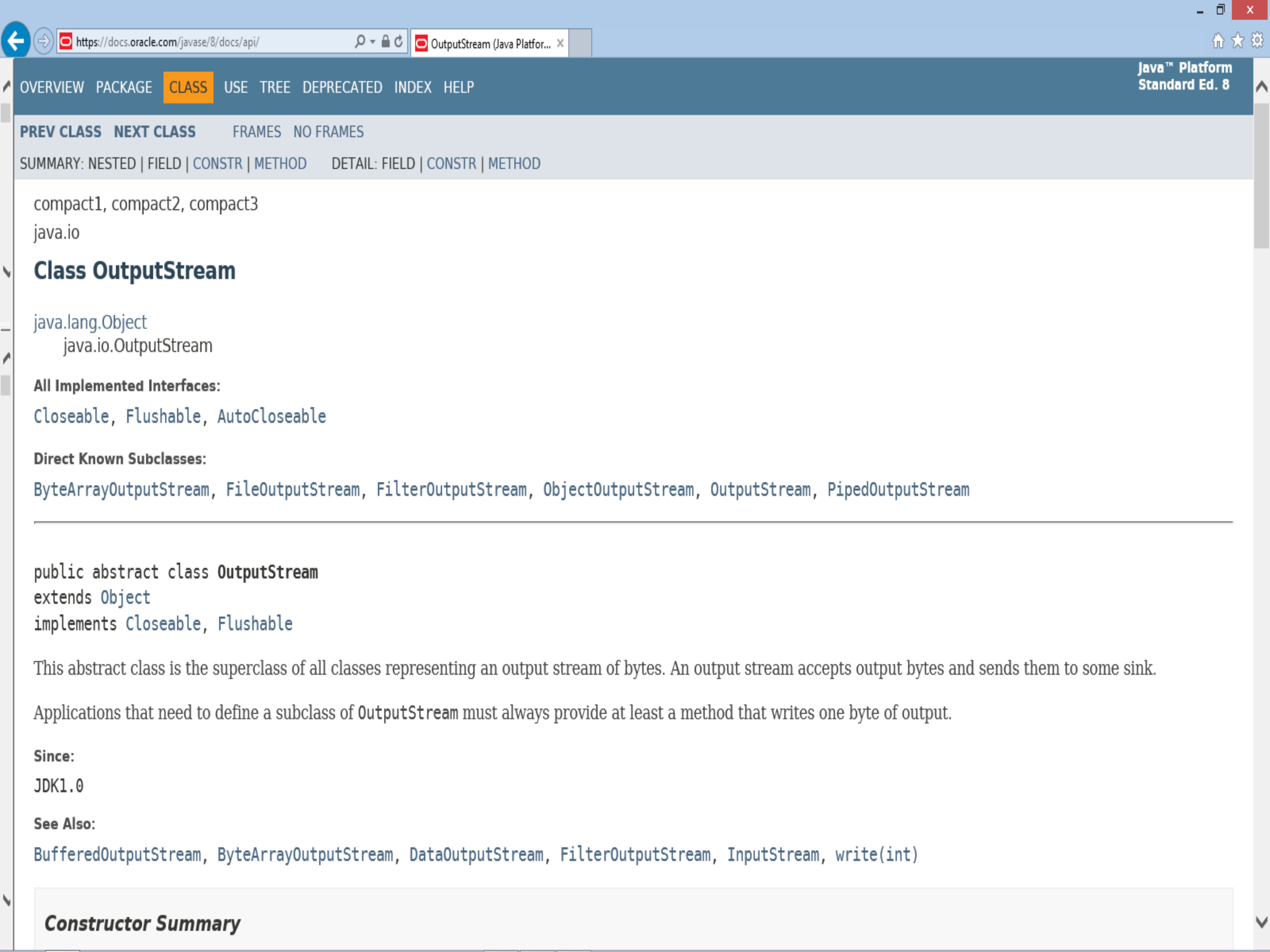
    public void **write**(byte[] data) throws IOException

    public void **write**(byte[] data, int offset, int length)
    throws IOException

  - Other supporting methods are:

    public void **flush**() throws IOException

    public void **close**() throws IOException


https://docs.oracle.com/javase/8/docs/api/
OutputStream (Java Platfor...
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java™ Platform Standard Ed. 8
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.io

Class OutputStream

java.lang.Object
    java.io.OutputStream

All Implemented Interfaces:

`Closeable`, `Flushable`, `AutoCloseable`

Direct Known Subclasses:

`ByteArrayOutputStream`, `FileOutputStream`, `FilterOutputStream`, `ObjectOutputStream`, `OutputStream`, `PipedOutputStream`

```
public abstract class OutputStream
extends Object
implements Closeable, Flushable
```

This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Applications that need to define a subclass of `OutputStream` must always provide at least a method that writes one byte of output.

Since:

JDK1.0

See Also:

`BufferedOutputStream`, `ByteArrayOutputStream`, `DataOutputStream`, `FilterOutputStream`, `InputStream`, `write(int)`

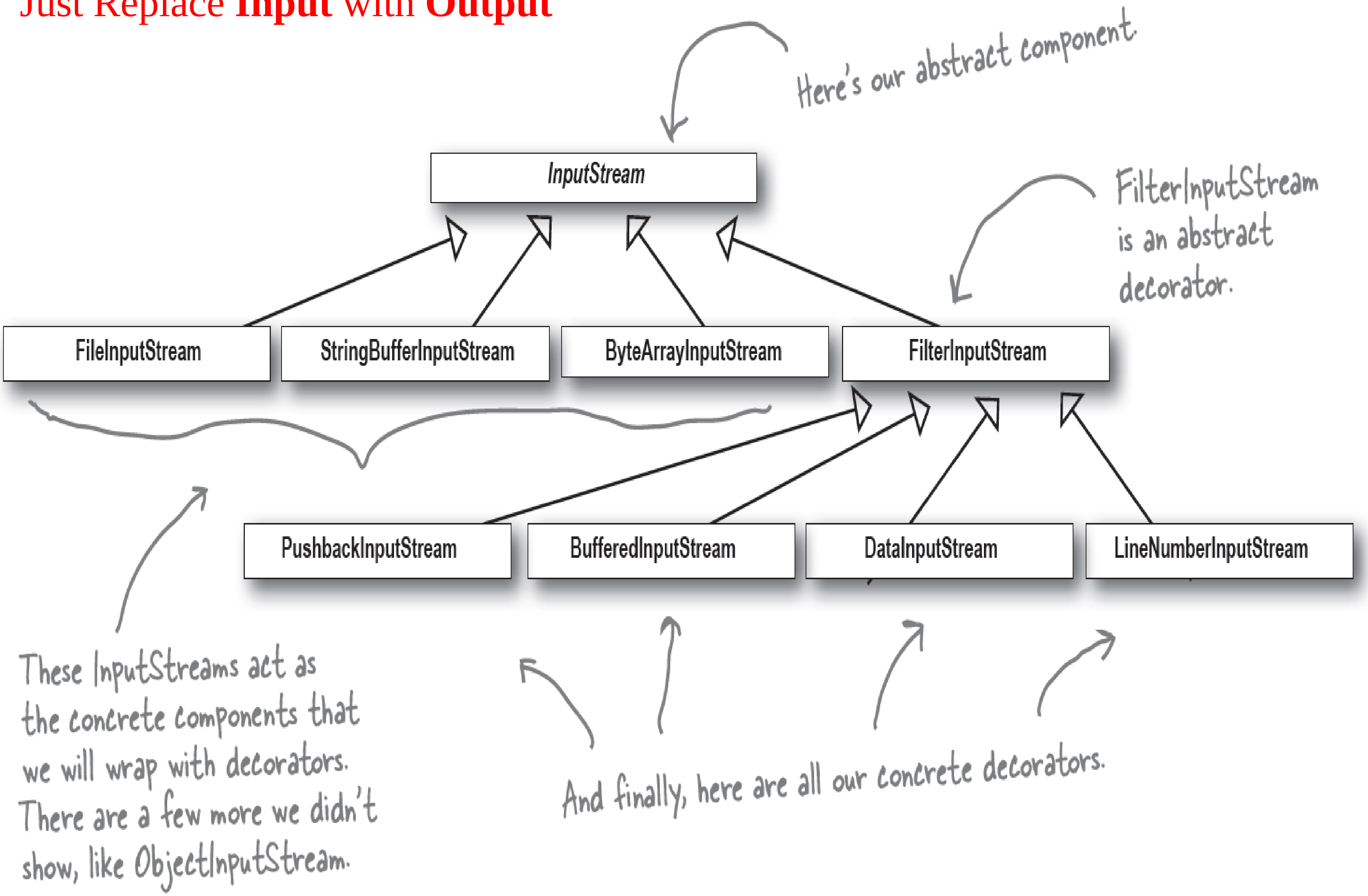
Constructor Summary

The OutputStream Class

- Subclasses of OutputStream use these methods to write data onto particular media.
  - **FileOutputStream** uses these methods to write data into a file.
  - **ByteArrayOutputStream** uses these methods to write data into an expandable byte array.

- Sometimes you may not even know exactly what kind of stream you're writing onto.
  - It's returned by various methods in various classes in java.net, like the **getOutputStream()** method of java.net.Socket.
  - However, these methods are declared to return only OutputStream, not the more specific subclass.
    - That's <u>polymorphism at work</u> once again.

Just Replace **Input** with **Output**



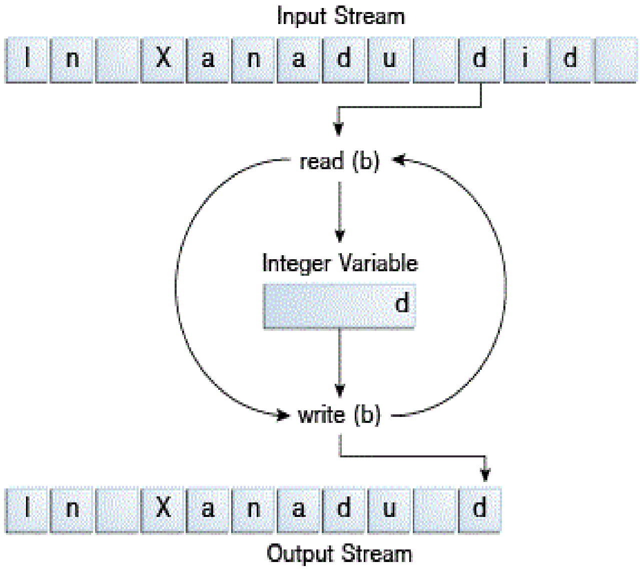
Here's our abstract component.
InputStream
FilterInputStream is an abstract decorator.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.
And finally, here are all our concrete decorators.

The OutputStream Class

- OutputStream’s fundamental method is

  public abstract void **write**(int b)

  - It takes an integer from 0 to 255 as an argument and writes the corresponding byte to the output stream.
  - This method is declared abstract because subclasses need to change it to handle their particular medium.
    - For example, a **FileOutputStream** will need to use native code that understands how to write data in files on the host platform.



The OutputStream Class

public void **write**(byte[] data)

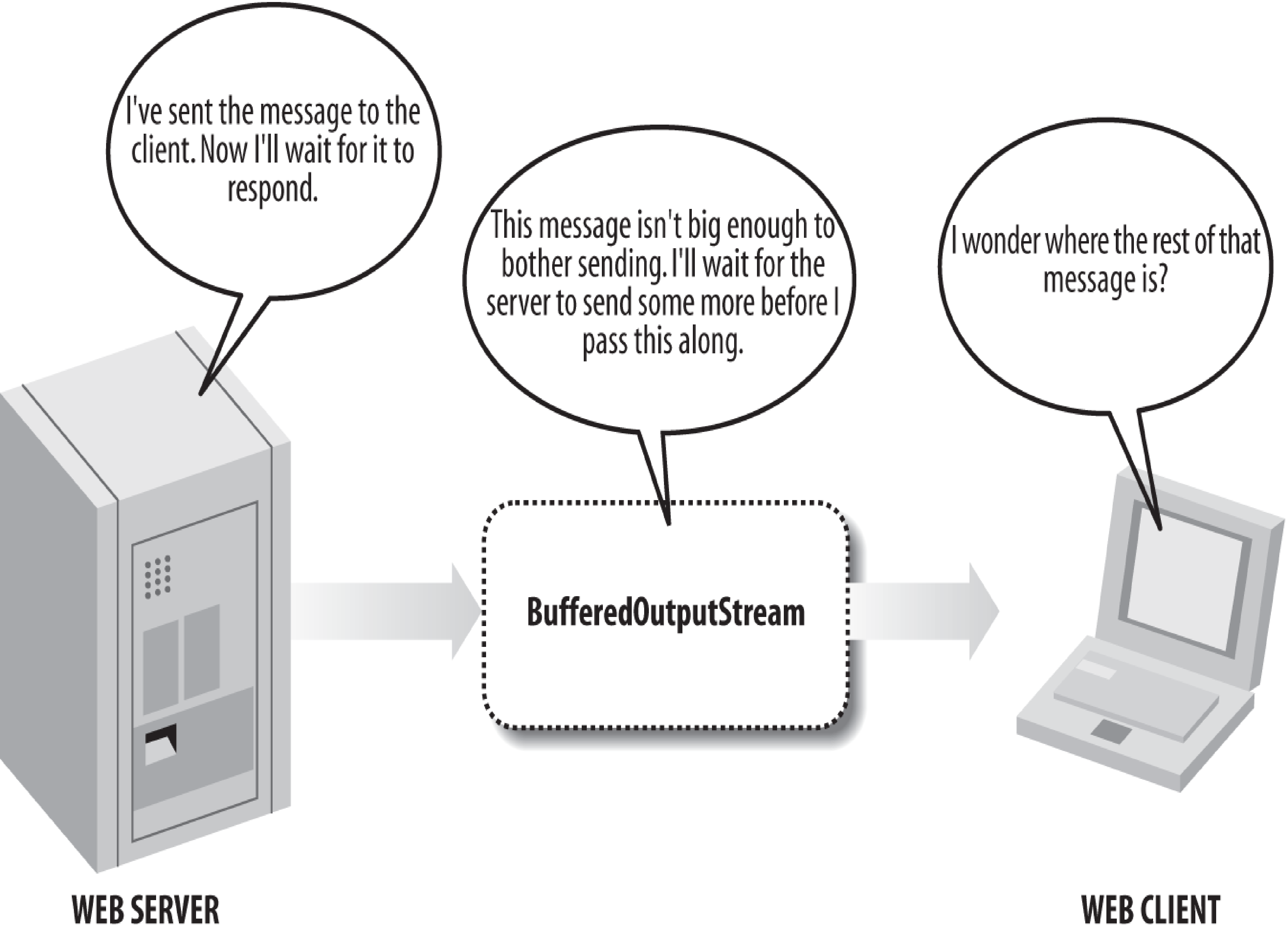
public void **write**(byte[] data, int offset, int length)

- Writing a single byte at a time is inefficient. Consequently, most TCP/IP implementations buffer data to some extent.
    - They accumulate bytes in memory and send them to their eventual destination only when a certain number have accumulated or a certain amount of time has passed.
    - If you have more than one byte ready to go, it’s not a bad idea to send them all at once, using one of these two methods.

The OutputStream Class

public void **flush**()

- As previously mentioned, Streams can also be buffered:
  - In software, directly in the Java code, as well as in the network hardware.
  - In Java, this is usually accomplished by **chaining** a **BufferedOutputStream** to the underlying stream.
- If you are done writing data, it’s important to **flush** the output stream.
  - If the output stream has a 1,024-byte buffer, the stream may be waiting for more data to arrive before it sends the data out of its buffer.
  - No more data will be written onto the stream until the server response arrives, but the response is never going to arrive because the request hasn’t been sent yet.


I've sent the message to the client. Now I'll wait for it to respond.
This message isn't big enough to bother sending. I'll wait for the server to send some more before I pass this along.
I wonder where the rest of that message is?
BufferedOutputStream
WEB SERVER
WEB CLIENT

The OutputStream Class

public void **close**()

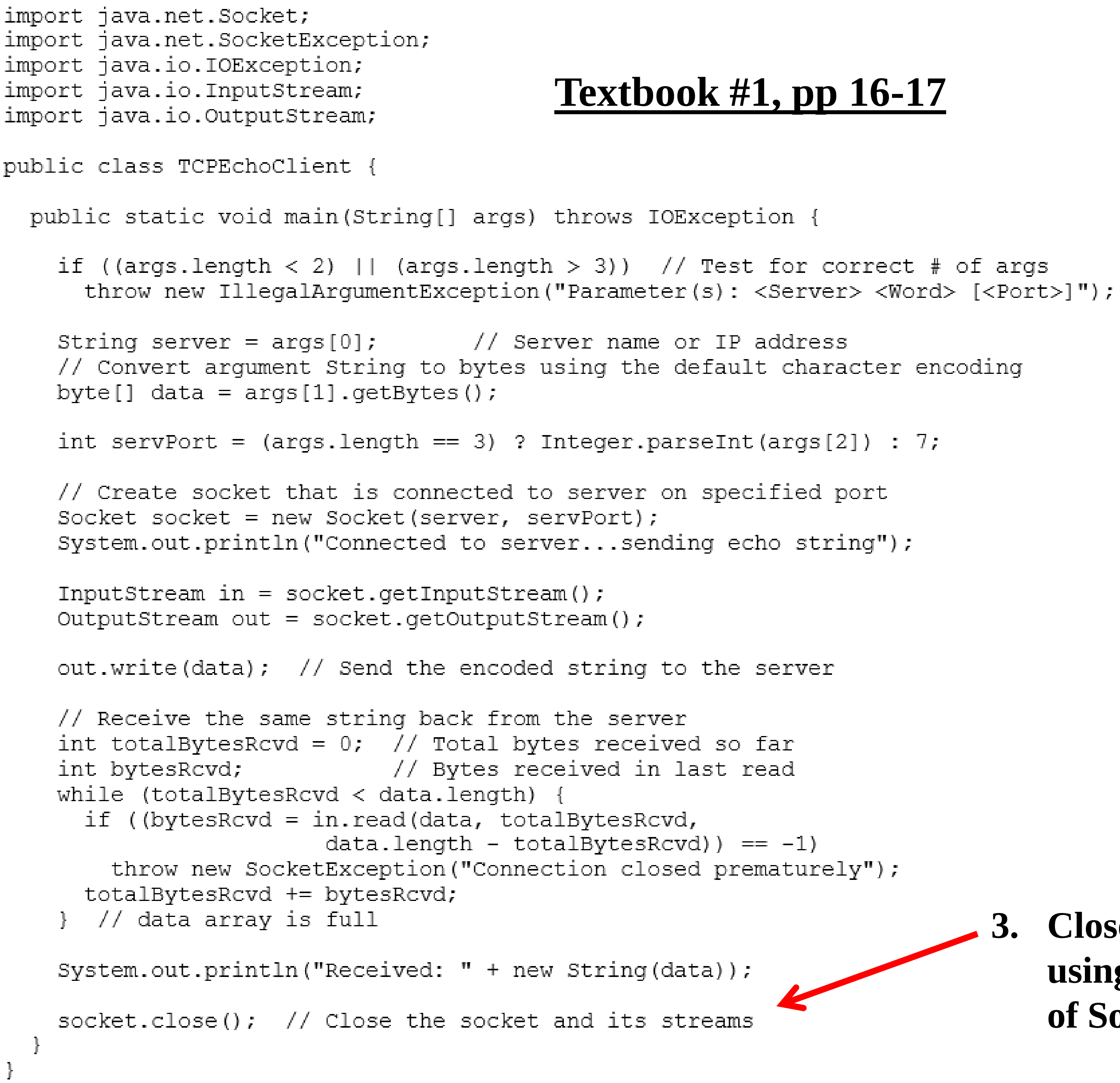
- When you’re done with a stream, close it by invoking its close() method.
  - This releases any resources associated with the stream, such as file handles or ports.
  - If the stream derives from a network connection, then closing the stream terminates the connection.
  - Once an output stream has been closed, further writes to it throw IOExceptions.
- Failure to close a stream in a long-running program can leak file handles, network ports, and other resources. (**Memory leaks**)
- You should **flush**() before you **close**().

The OutputStream Class: close()

- However, in Java 7 and above, you could do the following:

```
try (OutputStream out = new FileOutputStream("data.txt")) {
        // work with the output stream...
} catch (IOException ex) {
        System.err.println(ex.getMessage());
}
```

- Java automatically invokes close() on any **AutoCloseable** objects declared inside the argument list of the try block, in this case, **out**.



```java
import java.net.Socket;
import java.net.SocketException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

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");

    InputStream in = socket.getInputStream();
    OutputStream out = socket.getOutputStream();

    out.write(data);  // Send the encoded string to the server

    // Receive the same string back from the server
    int totalBytesRcvd = 0;  // Total bytes received so far
    int bytesRcvd;           // Bytes received in last read
    while (totalBytesRcvd < data.length) {
      if ((bytesRcvd = in.read(data, totalBytesRcvd,
                        data.length - totalBytesRcvd)) == -1)
        throw new SocketException("Connection closed prematurely");
      totalBytesRcvd += bytesRcvd;
    }  // data array is full

    System.out.println("Received: " + new String(data));

    socket.close();  // Close the socket and its streams
  }
}
```

Textbook #1, pp 16-17

3. Close the connection using the close() method of Socket:

TCP Client/Server Interaction

```
ServerSocket servSock = new ServerSocket(servPort);
```

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

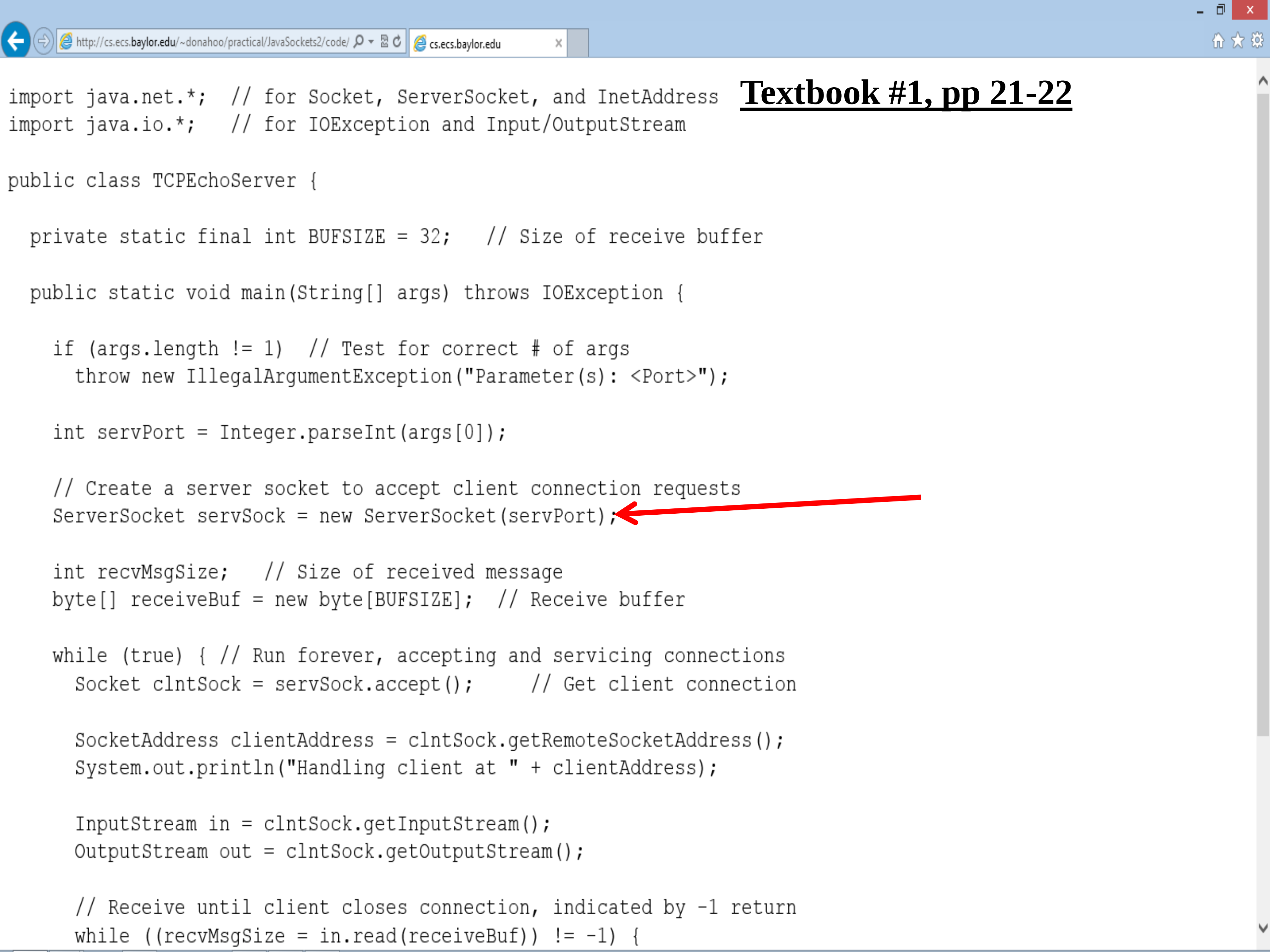
TCPEchoServer

- The server’s job is to set up a communication endpoint and passively wait for connections from clients. The typical TCP server goes through two steps:
  1. Construct a **ServerSocket** instance, specifying the **local port**. This socket listens for incoming connections to the specified port.
  2. Repeatedly:
     a. Call the **accept()** method of ServerSocket to get the next incoming client connection. Upon establishment of a new client connection, an instance of **Socket** for the new connection is created and returned by accept().

     b. Communicate with the client using the returned Socket’s **InputStream** and **OutputStream**.

     c. When finished, close the new client socket connection using the **close()** method of Socket.

Textbook #1, pp 21-22


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

```
import java.net.*;  // for Socket, ServerSocket, and InetAddress
import java.io.*;   // for IOException and Input/OutputStream

public class TCPEchoServer {

  private static final int BUFSIZE = 32;   // Size of receive buffer

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

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

    int servPort = Integer.parseInt(args[0]);

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(servPort);

    int recvMsgSize;   // Size of received message
    byte[] receiveBuf = new byte[BUFSIZE];  // Receive buffer

    while (true) { // Run forever, accepting and servicing connections
      Socket clntSock = servSock.accept();     // Get client connection

      SocketAddress clientAddress = clntSock.getRemoteSocketAddress();
      System.out.println("Handling client at " + clientAddress);

      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      // Receive until client closes connection, indicated by -1 return
      while ((recvMsgSize = in.read(receiveBuf)) != -1) {
```

The ServerSocket Class: Creation

ServerSocket(int **localPort**) throws IOException

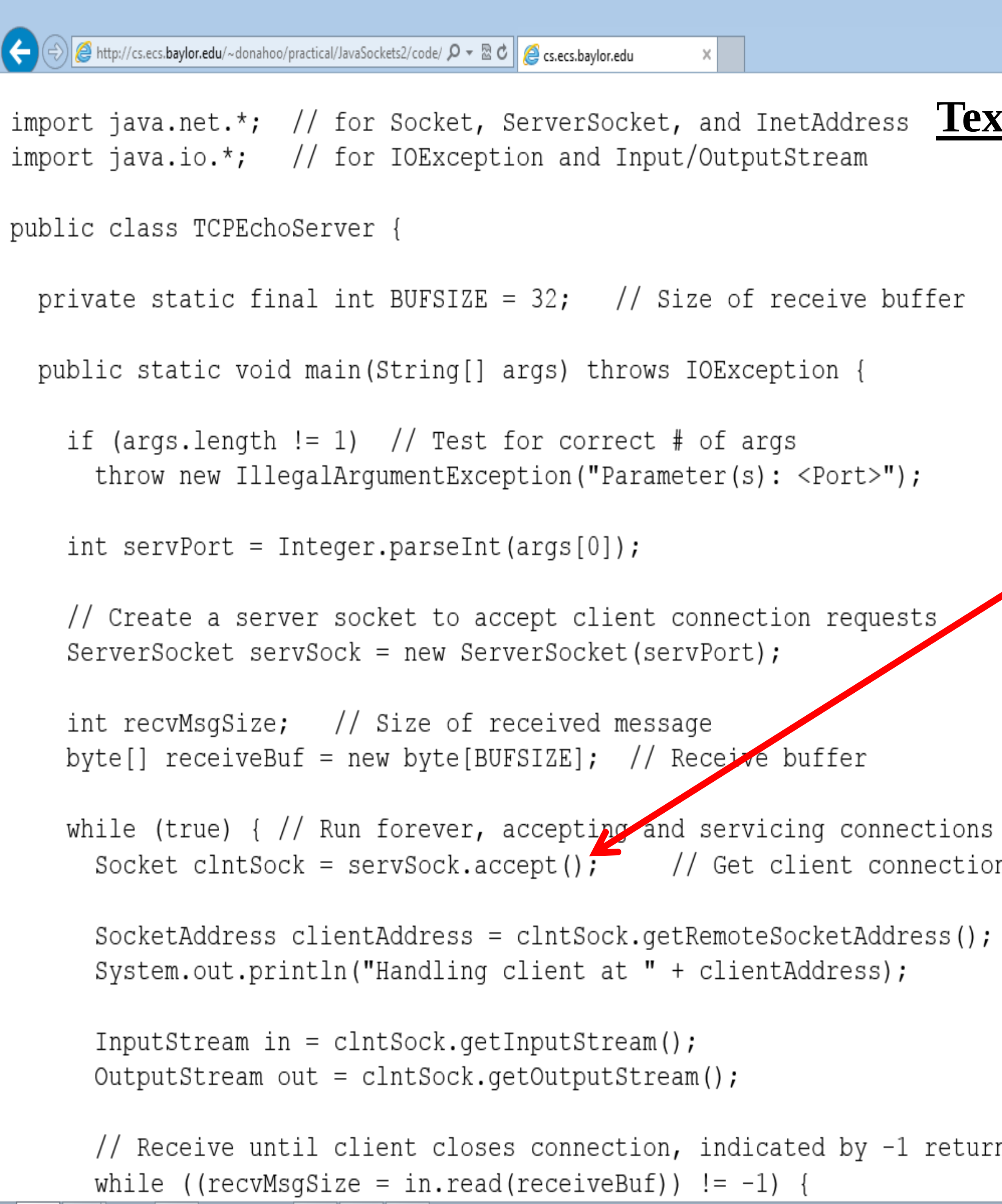
ServerSocket(int **localPort**, int **queueLimit**) throws IOException

ServerSocket(int **localPor**t, int **queueLimit**, InetAddress **localAddr**) …

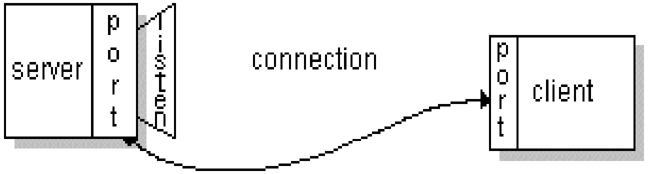
ServerSocket**()** throws IOException // unbound

- The first three constructors create a TCP endpoint that is associated with the specified local port and ready to **accept** incoming connections.
  - Valid port numbers are in the range 0–65,535. (If the port specified is zero, an arbitrary unused port will be picked. See next slide)
  - The local address, if specified, must be an address of one of this host’s **network interfaces**. If the address is not specified, the socket will accept connections to any of the host’s IP addresses. This may be useful for hosts with multiple network interfaces

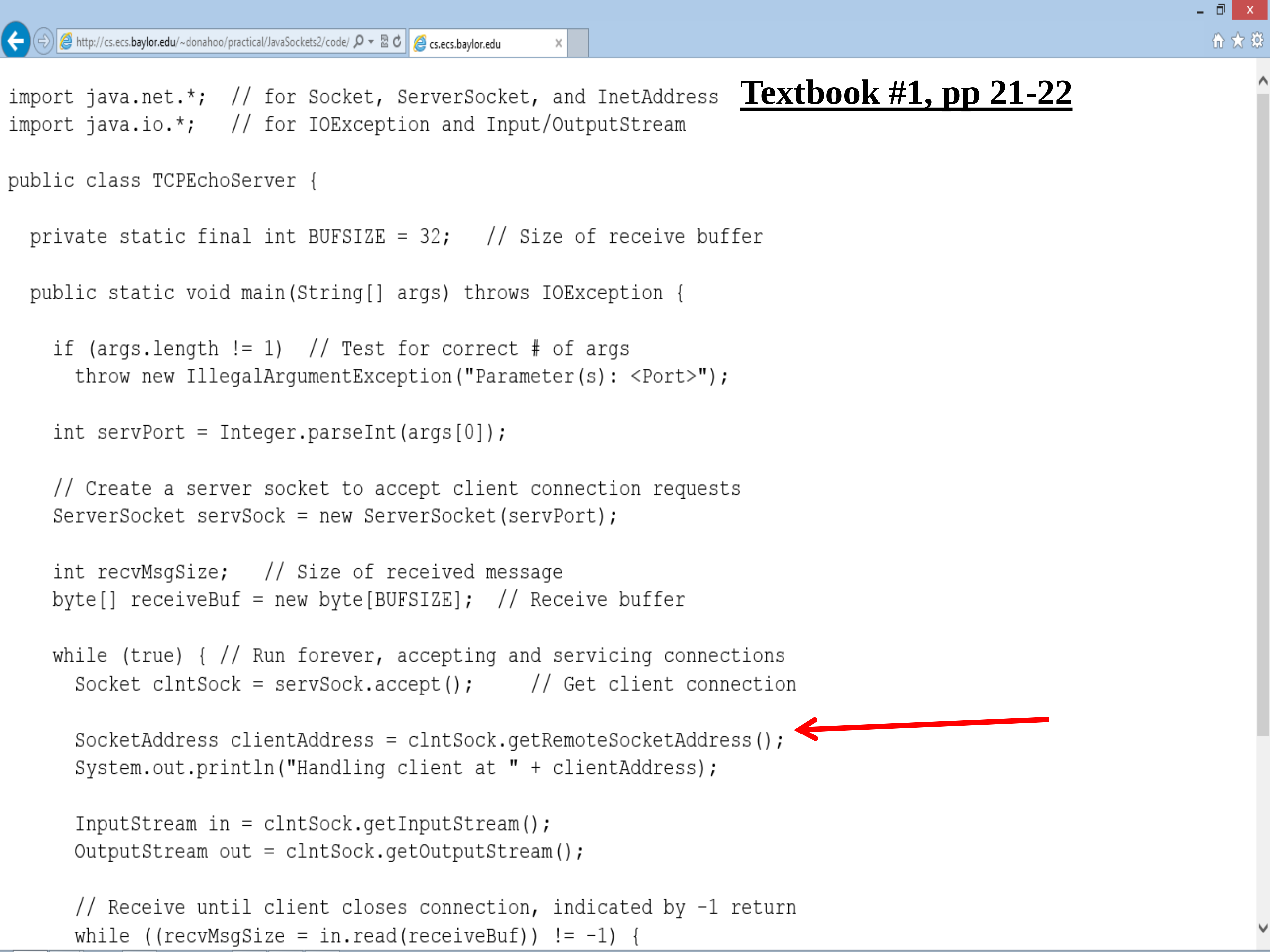
Textbook #1, pp 21-22


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

```
import java.net.*;  // for Socket, ServerSocket, and InetAddress
import java.io.*;   // for IOException and Input/OutputStream

public class TCPEchoServer {

  private static final int BUFSIZE = 32;   // Size of receive buffer

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

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

    int servPort = Integer.parseInt(args[0]);

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(servPort);

    int recvMsgSize;   // Size of received message
    byte[] receiveBuf = new byte[BUFSIZE];  // Receive buffer

    while (true) { // Run forever, accepting and servicing connections
      Socket clntSock = servSock.accept();     // Get client connection

      SocketAddress clientAddress = clntSock.getRemoteSocketAddress();
      System.out.println("Handling client at " + clientAddress);

      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      // Receive until client closes connection, indicated by -1 return
      while ((recvMsgSize = in.read(receiveBuf)) != -1) {
```

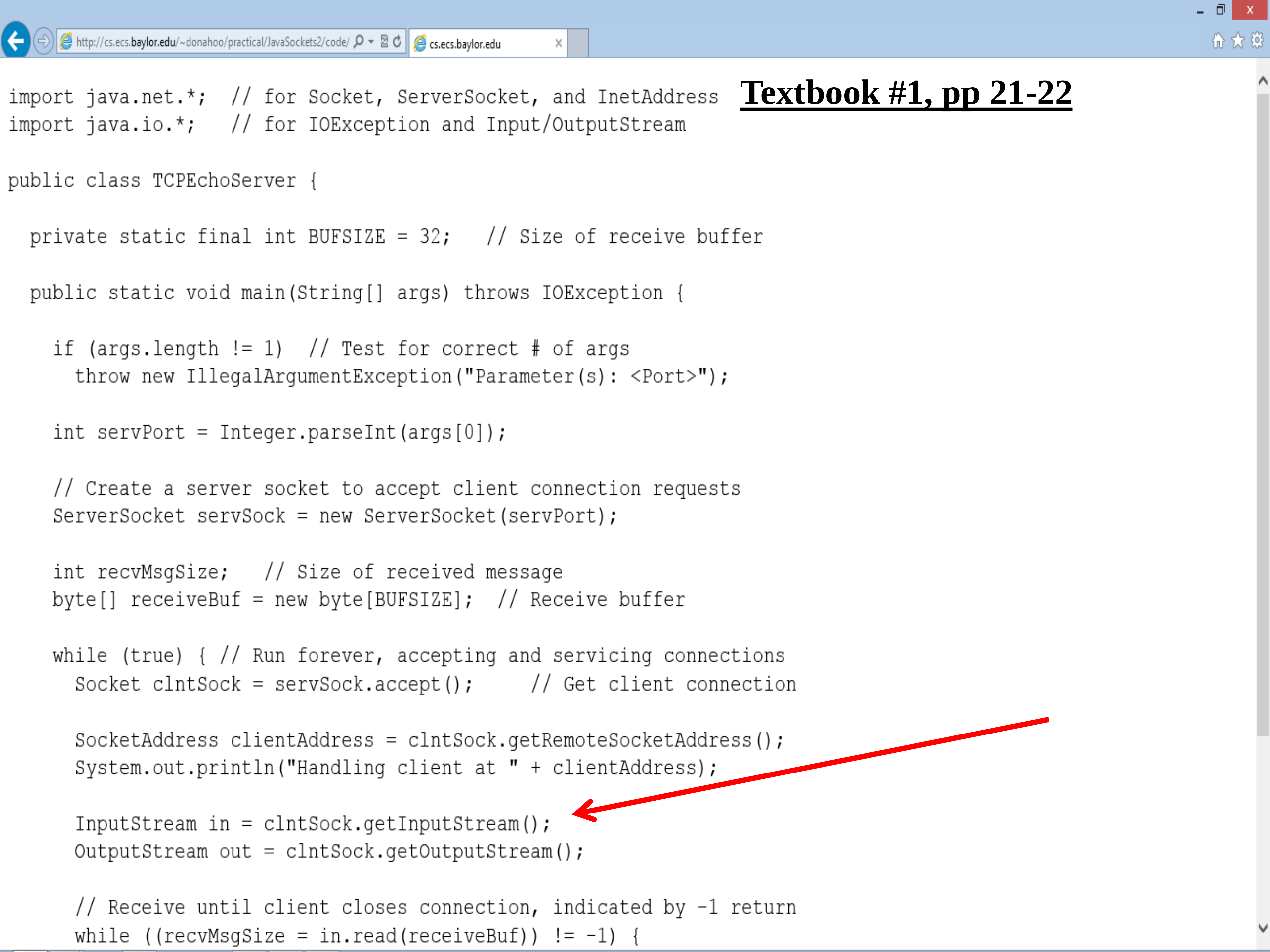


Textbook #1, pp 21-22


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

```
import java.net.*;  // for Socket, ServerSocket, and InetAddress
import java.io.*;   // for IOException and Input/OutputStream

public class TCPEchoServer {

  private static final int BUFSIZE = 32;   // Size of receive buffer

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

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

    int servPort = Integer.parseInt(args[0]);

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(servPort);

    int recvMsgSize;   // Size of received message
    byte[] receiveBuf = new byte[BUFSIZE];  // Receive buffer

    while (true) { // Run forever, accepting and servicing connections
      Socket clntSock = servSock.accept();     // Get client connection

      SocketAddress clientAddress = clntSock.getRemoteSocketAddress();
      System.out.println("Handling client at " + clientAddress);

      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      // Receive until client closes connection, indicated by -1 return
      while ((recvMsgSize = in.read(receiveBuf)) != -1) {
```

Textbook #1, pp 21-22


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

```
import java.net.*;  // for Socket, ServerSocket, and InetAddress
import java.io.*;   // for IOException and Input/OutputStream

public class TCPEchoServer {

  private static final int BUFSIZE = 32;   // Size of receive buffer

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

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

    int servPort = Integer.parseInt(args[0]);

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(servPort);

    int recvMsgSize;   // Size of received message
    byte[] receiveBuf = new byte[BUFSIZE];  // Receive buffer

    while (true) { // Run forever, accepting and servicing connections
      Socket clntSock = servSock.accept();     // Get client connection

      SocketAddress clientAddress = clntSock.getRemoteSocketAddress();
      System.out.println("Handling client at " + clientAddress);

      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      // Receive until client closes connection, indicated by -1 return
      while ((recvMsgSize = in.read(receiveBuf)) != -1) {
```

The ServerSocket Class: Creation

ServerSocket(int **localPort**) throws IOException

ServerSocket(int **localPort**, int **queueLimit**) throws IOException

ServerSocket(int **localPor**t, int **queueLimit**, InetAddress **localAddr**) …

ServerSocket**()** throws IOException // unbound

- ❖ The first three constructors create a TCP endpoint that is associated with the specified local port and ready to **accept** incoming connections.
  - ❖ Valid port numbers are in the range 0–65,535. (If the port specified is zero, an arbitrary unused port will be picked. See next slide)
  - ❖ The local address, if specified, must be an address of one of this host's **network interfaces**. If the address is not specified, the socket will accept connections to any of the host's IP addresses. This may be useful for hosts with multiple network interfaces

A Getter for the ServerSocket Class

public int **getLocalPort**() // return -1 if not bound

- The ServerSocket constructors allow you to listen on an unspecified port by passing 0 for the port number. This method lets you find out what port you’re listening on:

```
import java.io.*;
import java.net.*;
public class RandomPort { // From Textbook #2
        public static void main(String[] args) {
            try {
                    ServerSocket server = new ServerSocket(0);
                    System.out.println("This server runs on port "+ server.getLocalPort());
            } catch (IOException ex) {
                    System.err.println(ex);
            }
        }
}
```

Example 9-8. Look for local ports

```
import java.io.*;
import java.net.*;
public class LocalPortScanner { // From Textbook #2
        public static void main(String[] args) {
          for (int port = 1; port <= 65535; port++) {
            try {
                // the next line will fail and drop into the catch block if
                // there is already a server running on the port
                ServerSocket server = new ServerSocket(port);
            } catch (IOException ex) {
                System.out.println("There is a server on port " + port + ".");
            }
          }
        }
}
```

The ServerSocket Class: Operations

public void **bind**(SocketAddress endpoint) throws IOException

public void **bind**(SocketAddress endpoint, int queueLimit)
throws IOException

- The **noargs** constructor ServerSocket() you previously saw creates a ServerSocket object but does not actually bind it to a port, so it cannot initially accept any connections. It can be bound later using the **bind()** methods.
- If this ServerSocket instance is already associated with another port, or if the specified port is already in use, an IOException is thrown.

```
ServerSocket ss = new ServerSocket();
// set socket options...
SocketAddress http = new InetSocketAddress(80);
ss.bind(http);
```

The InetSocketAddress Class (Revisited)

- The **InetSocketAddress** class (which is the one and only known subclass of **SocketAddress**) is usually created with a host and a port (for clients) or just a port (for servers):

```
public InetSocketAddress(InetAddress address, int port)
public InetSocketAddress(String host, int port)
public InetSocketAddress(int port)
```

The SocketAddress Class (Revisited)

- The **SocketAddress** class represents a connection endpoint.

- It is an empty abstract class with no methods besides a default constructor.

- At least theoretically, the SocketAddress class can be used for both TCP and non-TCP sockets.

- In practice, only TCP/IP sockets are currently supported and the socket addresses you actually use are all instances of **InetSocketAddress**.

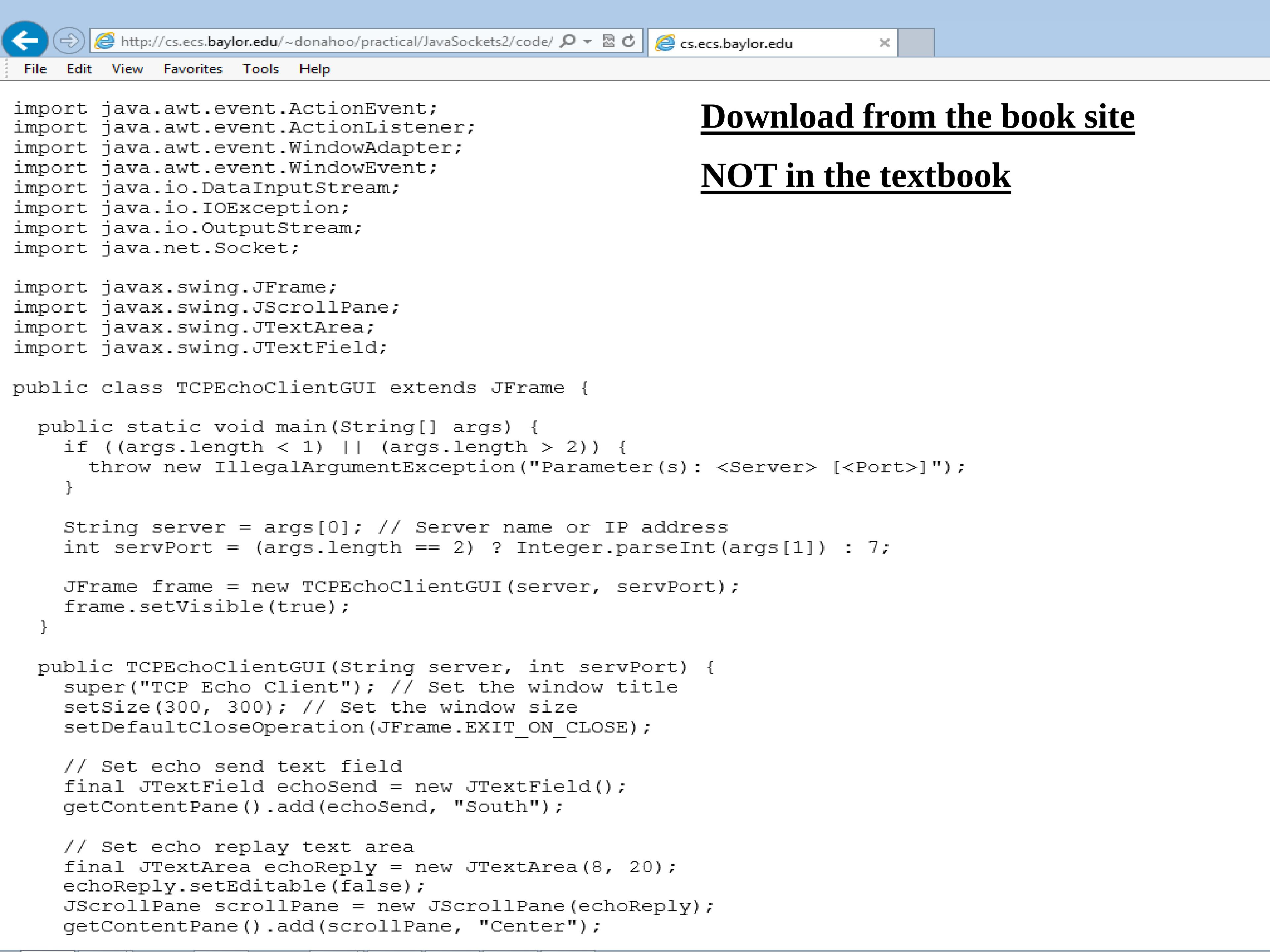
The ServerSocket Class: Operation

public Socket **accept()** throws IOException

- It returns a connected Socket instance for the next new incoming connection to the server socket.
- If no established connection is waiting, accept() blocks until one is established or a timeout occurs.

public void **close()** throws IOException

- The close() method closes the socket.
- After invoking this method, incoming client connection requests for this socket are rejected.


http://cs.ecs.baylor.edu/~donahoo/practical/JavaSockets2/code/
cs.ecs.baylor.edu
File
Edit
View
Favorites
Tools
Help

Download from the book site

NOT in the textbook

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.DataInputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class TCPEchoClientGUI extends JFrame {

  public static void main(String[] args) {
    if ((args.length < 1) || (args.length > 2)) {
      throw new IllegalArgumentException("Parameter(s): <Server> [<Port>]");
    }

    String server = args[0]; // Server name or IP address
    int servPort = (args.length == 2) ? Integer.parseInt(args[1]) : 7;

    JFrame frame = new TCPEchoClientGUI(server, servPort);
    frame.setVisible(true);
  }

  public TCPEchoClientGUI(String server, int servPort) {
    super("TCP Echo Client"); // Set the window title
    setSize(300, 300); // Set the window size
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Set echo send text field
    final JTextField echoSend = new JTextField();
    getContentPane().add(echoSend, "South");

    // Set echo replay text area
    final JTextArea echoReply = new JTextArea(8, 20);
    echoReply.setEditable(false);
    JScrollPane scrollPane = new JScrollPane(echoReply);
    getContentPane().add(scrollPane, "Center");
```