Key Points – I/O

- writing and reading must be compatible
  - match type of streams
    - character stream output → character stream input
      - any character stream writer (such as `PrintWriter`) can be matched with any character stream reader (such as `BufferedReader` or `Scanner`)
    - byte stream output → byte stream input
  - for byte streams, also match representation
    - e.g. `ObjectOutputStream` → `ObjectInputStream`
    - e.g. `DataOutputStream` → `DataInputStream`
  - match order
    - e.g. `readInt` then `readDouble` only works if it was written that way

Sockets

- a *socket* is one endpoint of a two-way connection between programs running on a network
  - it is bound to a *port* which identifies which running program gets the incoming data
- a program is reached via its *host* and *port*
  - host is a name like `localhost` or `chat.example.com` or an IP address like `192.0.2.17`
  - port is a number like 5000
    - *well-known ports* (0-1023) are established for system or core services
    - *registered ports* (1024-49151) are used for vendor-specific applications
      - avoid 300, 5173, 8080, 3306, 5432, 8000, 5000, 7000
      - most other ports 5000-9999 are likely to safe – if you get "address already in use", pick something else
    - *dynamic* or *private ports* (49152-65535) are used for temporary client-side sessions

Network Communications

Steps in network communications:

- establish a connection
  - the server creates a `ServerSocket` on a particular port, then waits for a client to connect
  - the client creates a `Socket` connected to the server's host and port
    - the server must already be waiting or this will fail
- read/write over the connection
  - both sides must match protocols and kinds of streams
- close connection

Establish a Connection

- server

```
ServerSocket socket = new ServerSocket(0);
```

port number to listen on
0 means to use any available port
(can use `socket.getLocalPort()` to find out which port that is)

or 5000-9999 is generally safe

```
Socket connection = socket.accept();
```

wait for a client (blocks until a client connects)

- client

```
Socket connection = new Socket(server,port);
```

server host name or IP address (String)

port number to connect to

Reading and Writing

- get stream objects for reading and/or writing over the socket
  - `connection.getInputStream()`, `connection.getOutputStream()`

'connection' is the socket

These streams can be used directly, or more commonly wrapped with another (more useful) stream or Reader/Writer. e.g.

'connection' is the socket

```
BufferedReader reader =
    new BufferedReader(new InputStreamReader(connection.getInputStream()));
PrintWriter writer =
    new PrintWriter(new OutputStreamWriter(connection.getOutputStream()));
```

  - be sure to `flush()` when done writing!

Close Connection

- close the connection when done

```
connection.close();
```

'connection' is the socket

echo server

```
public static void main ( String[] args ) throws IOException {
  ServerSocket server = new ServerSocket(6000);
  for ( ; ; ) {
    Socket socket = server.accept();          // waits until a client connects
    System.out.println("connected: " + socket.getRemoteSocketAddress());
    handle(socket);
  }
}

private static void handle ( Socket socket ) {
  try {
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream(),
                                                                 StandardCharsets.UTF_8));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),
                                                             StandardCharsets.UTF_8));
    for ( ; ; ) {
      String line = in.readLine();            // waits until a line arrives
      if ( line == null ) {
        break;                                // the client closed the connection
      }
      if ( line.equals("bye") ) {
        out.println("bye");
        out.flush();
        break;
      }
      out.println("echo: " + line);
      out.flush();                            // send it now
    }
  } catch ( IOException e ) {
    System.out.println("problem talking to the client: " + e.getMessage());
  } finally {
    try { socket.close(); } catch ( IOException e ) {}
  }
}
```

echo client

```
public static void main ( String[] args ) {
  String host = "localhost";
  int port = 6000;

  try {
    Socket socket = new Socket(host,port);             // connects, or throws
    BufferedReader in = new BufferedReader(new InputStreamReader(socket
        .getInputStream(),StandardCharsets.UTF_8));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(socket
        .getOutputStream(),StandardCharsets.UTF_8));
    Scanner keyboard = new Scanner(System.in);

    System.out.println("connected to " + host + ":" + port + " from local port "
        + socket.getLocalPort());
    for ( ; keyboard.hasNextLine() ; ) {
      String line = keyboard.nextLine();
      out.println(line);
      out.flush();                                     // send it now

      String reply = in.readLine();                    // wait for the answer
      if ( reply == null ) {
        System.out.println("the server closed the connection");
        break;
      }
      System.out.println(reply);
      if ( reply.equals("bye") ) {
        break;
      }
    }
    socket.close();
  } catch ( IOException e ) {
    System.out.println("can't talk to " + host + ":" + port + ": " + e.getMessage());
  }
 }
}
```

Simple Protocols

- client and server must agree on a *protocol*
  - what each side sends and how
- key element – what counts as one message?
  - the connection just carries a stream of bytes
- a simple case – one line is one message
  - `println()` sends a message, `readLine()` receives it
    - but: a message can't contain a newline
- the streams also have to match
  - same kind of stream on both sides
    - don't mix text (characters) and objects (byte)
  - same character set on both sides –
    - e.g. `StandardCharsets.UTF_8`
      - UTF-8 is the dominant standard on the Internet, default charset for Java since Java 18

Streams Over the Network

- other streams details
  - flush after each message
    - must `flush()` after each message in order to ensure that it sends
      - `PrintWriter` buffers output rather than writing immediately
      - even closing the socket doesn't flush



  - wrap each stream (only) once and keep using the same reader and writer
    - `BufferedReader` reads ahead, so a second reader on the same stream misses what the first one took
    - in general, a nozzle may
  - `null` from `readLine()` means the other side has closed the connection
    - check every time!

Common Failures

what happens	what you see
client can't determine IP of host it is connecting to	`UnknownHostException`
client connects, no server on that port	`Connection refused`
start a server twice	`BindException`: Address already in use (the first run still has the port)
the other side closes normally	`readLine()` returns `null`
the other side crashes or the network fails	`SocketException: Connection reset` or a read that never returns
forgotten flush	other end hangs, waiting for input it never gets

- all of the exceptions listed are kinds of `IOException`
  - catch `IOException` if error handling is the same for all
  - catch individual exceptions if specialized handling is needed
    - put catch clauses for the more specific types first

One Server, Several Clients

- to manage multiple clients at once, the server needs threads
  - one thread to listen for new clients
  - one thread per client to handle the client's needs

Echo Server

echo server

```
public class EchoHandler implements Runnable {
  private Socket socket_;

  public EchoHandler ( Socket socket ) {
    socket_ = socket;
  }

  @Override
  public void run () {
    // ...what's done to handle a single client...
  }
}
```

move the single client handling code into the body of a Runnable

---

```
ServerSocket server = new ServerSocket(6000);
int count = 0;
for ( ; true ; ) {
  Socket socket = server.accept();
  count++;
  Thread thread = new Thread(new EchoHandler(socket),"client-" + count);
  thread.start();
}
```

start a thread per client

Chat Server

- echo each message sent to all connected clients
- client must read from server as well as from the keyboard

- new ingredients
  - server – a chat hub to manage and message the connected clients
  - client – separate server and keyboard threads
    - reading blocks, and both user and server can type/send a message at any time

Chat Hub – Server

chat server

```
public class ChatHub {

  private List<ChatHandler> clients_;

  public ChatHub () {
    client_ = new ArrayList<ChatHandler>();
  }

  public void addClient ( ChatHandler client ) {
    clients_.add(client);
  }

  public void removeClient ( ChatHandler client ) {
    clients_.remove(client);
  }

  public void broadcast ( String message ) {
    for ( ChatHandler client : clients_ ) {
      client.send(message);
    }
  }
}
```


Chat Server – Server

chat server

```
public class ChatServer {

  public static final int PORT = 6000;

  public static void main ( String[] args ) throws IOException {
    ChatHub chat = new ChatHub();
    ServerSocket server = new ServerSocket(PORT);
    System.out.println("chat server listening on port " + PORT);

    int count = 0;
    for ( ; true ; ) {
      Socket socket = server.accept();
      count++;
      try {
        Thread thread =
            new Thread(new ChatHandler(socket,chat),"client-" + count);
        thread.start();
      } catch ( IOException e ) {
        System.out
            .println("couldn't set up client-" + count + ": " + e.getMessage());
        socket.close();
      }
    }
  }
}
```

Chat Handler – Server

chat server

```
public class ChatHandler implements Runnable {
  …
  private BufferedReader in_;
  private PrintWriter out_;

  public ChatHandler ( Socket socket, ChatHub chat ) throws IOException {
    …
    in_ = new BufferedReader(new InputStreamReader(socket.getInputStream(),…));
    out_ = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),…));
  }

  public void run () {
    try {
      …
      chat_.addClient(this);
      for ( ; true ; ) {
        String line = in_.readLine();
        if ( line == null || line.equals("bye") ) { break; }
        chat_.broadcast(name + ": " + line);
      }
    } catch ( IOException e ) {
      …
    } finally {
      chat_.removeClient(this);
      …
      try { socket_.close(); } catch ( IOException e ) {}
    }
  }
  …
}
```

Chat Handler – Server

chat server

```
public class ChatHandler implements Runnable {

  …

  public void send ( String message ) {
    out_.println(message);
    out_.flush();
  }
}
```