## Java I/O – Quick Review

- the `java.io` package is the main place for Java I/O classes.
- starting with 1.4, the `java.nio` package contains additional I/O functionality.

- Java uses ***streams*** to pass bytes and characters.
  - streams are easily connected together to implement in-line data processing (like buffering, encryption, compression, etc.)
  - streams allow for an abstraction of the source of or destination for the data—easily allowing other stream implementations to be slipped in.

- Java has two main kinds of data streams:
  - ***byte streams***
    - data passed is 8-bit byte values
    - bytes are read from an `InputStream`
    - bytes are written to an `OutputStream`

  - ***character streams***
    - data passed is 16-bit character values
    - characters are read from a `Reader`.
    - characters are written to a `Writer`.

  - Almost every constructor and method potentially throws an instance of `IOException` (or one of its subclasses). This is because so much can go wrong during I/O.

- Using `InputStream`
  - no explicit 'open'—automatically opened at the time of construction.
  - bytes are read with:

```
public int read() throws IOException
```

the byte read is returned, or `-1` is returned if it is the End-Of-File (EOF). Code like this is used to read one byte and check for EOF:

```
InputStream in = //...

int val = in.read();

if ( val == -1 ) {
    // end of file
} else {
    byte b = (byte) val; // strip off the high bits
}
```

  - the thread that calls `read()` ***blocks*** inside that method until a byte can be read. This is significant in network (client/server) based I/O. This needs to be taken into consideration when writing code.

  - when finished, the `InputStream` should be closed with:

```
public void close() throws IOException
```

- Using `OutputStream`
    - no explicit 'open'—automatically opened at the time of construction.
    - bytes are written with:

```
public void write(int b) throws IOException
```

the value written is the lower 8-bits of the `int`—an implicit `(byte) b` is done behind the scenes.

```
OutputStream out = //...

byte b = 23;
out.write(b); // works
```

- the thread that calls `write()` ***blocks*** inside that method until a byte can be written. This is significant in network (client/server) based I/O. This needs to be taken into consideration when writing code—although this is of less concern than the blocked `read()`.

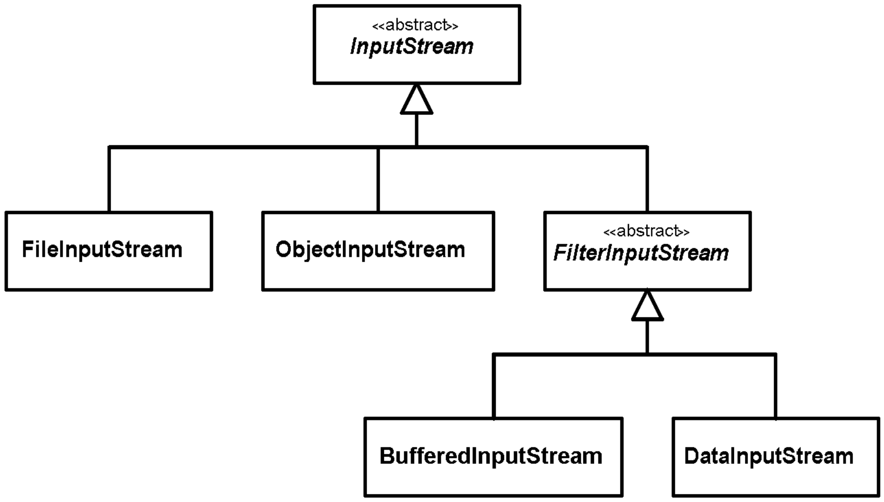
- most times, there is some kind of buffering present between the closest `OutputStream` and the ultimate destination for the data. To force the data to be pushed down the line, the stream should be *flushed*:

```
public void flush() throws IOException
```

- when finished, the `OutputStream` should be closed with:

```
public void close() throws IOException
```

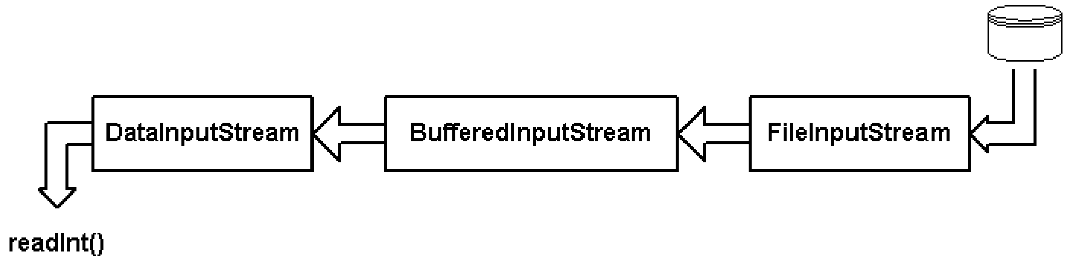
- `InputStream` is actually an `abstract` class. The actual implementations used are subclasses of `InputStream`:




  - `FileInputStream` is used to read bytes from a disk file.
  - `ObjectInputStream` is used to read serialized objects.
  - `FilterInputStream` is a subclass of `InputStream`, but is still an abstract class. `FilterInputStream`'s are used as inline data filters and take an `InputStream` as a parameter to the constructor:


  - Examples of `FilterInputStream` implementations are:
    - `BufferedInputStream` –used to read data in chunks at a time for more efficient I/O. Individual bytes can be read from the buffer without necessarily causing a real I/O call (like reading a byte from a disk). When the buffer is empty, the next read results in another large chuck being read into the buffer.
    - `DataInputStream` –used to read primitive types like `int`, `double`, `long`, `boolean`, etc. which in some cases result in more than one byte being read from the underlying stream.

- An example of filtering:




  - In this example, we want to read `int`'s from a file named "`integers.dat`". To be more efficient, we want to use buffering. The bytes are read from the file using a `FileInputStream` and are then buffered using a `BufferedInputStream`. From the buffer, 4 bytes at a time are converted into an `int` to be returned from the `readInt()` method of `DataInputStream`.

  - This can be implemented in code like this:

```
FileInputStream fis = new FileInputStream("integers.dat");
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream in = new DataInputStream(bis);

int i = in.readInt();
// ...
```

  - Or an even easier implementation is this:

```
DataInputStream in = new DataInputStream(
        new BufferedInputStream(
                new FileInputStream("integers.dat")));

int i = in.readInt();
// ...
```

- There are many more subclasses of `InputStream`—these are just a few examples.

- `OutputStream` is also an abstract class and there are many subclasses that are used in similar ways.

## Object Serialization

- Java's serialization mechanism allows us to take an object, serialize it into a stream of bytes, and later take that stream of bytes and turn it back into an object.

- The stream of bytes can be saved to a file to ***persist*** an object. Weeks later, this file can be read back in and the object will be restored.
- The stream of bytes can be sent "live" over a network connection (TCP/IP socket most likely) to another Java VM. This remote VM reads the stream of bytes from the network and creates an object.

- Object serialization-related classes are in the `java.io` package.

- To ***serialize*** objects into bytes, we use `ObjectOutputStream`:

```
import java.io.*;
// ...

try {
    ObjectOutputStream oos = //...
    Object obj = //...

    oos.writeObject(obj);
    oos.flush(); // if this is the last "for a while"
} catch ( NotSerializableException x ) {
    x.printStackTrace();
} catch ( IOException iox ) {
    iox.printStackTrace();
}
```

- To ***deserialize*** objects from bytes, we use `ObjectInputStream`:

```
import java.io.*;
// ...

try {
    ObjectInputStream ois = //...
    Object obj = ois.readObject();
} catch ( ClassNotFoundException x ) {
    x.printStackTrace();
} catch ( IOException iox ) {
    iox.printStackTrace();
}
```

- Only classes that implement the `Serializable` interface (either directly or indirectly by subclassing a class that does) can be serialized.
- The `Serializable` interface doesn't have any methods, but simply serves as a *marker* interface to indicate that serialization is permitted:

```
import java.io.*;

public class MyObject extends Object implements Serializable {
    private int x;
    private String name;
    private transient String secret; // not part of the serialization

    //...
}
```

- Member variables that are marked as `transient` are not part of the serialized object. When the object is deserialized, `transient` members take on their default value (like `0` for numeric types, `false` for `boolean`, `null` for references, etc).

- Example: serializing objects to a file:

```
MyObject mo1 = new MyObject();
MyObject mo2 = new MyObject();
//...

ObjectOutputStream oos = new ObjectOutputStream(
        new BufferedOutputStream(
                new FileOutputStream("objects.ser")));

oos.writeObject(mo1);
oos.writeObject(mo2);
oos.flush();
oos.close();
```

--and reading them back at some later time:

```
ObjectInputStream ois = new ObjectInputStream(
        new BufferedInputStream(
                new FileInputStream("objects.ser")));

MyObject moA = (MyObject) ois.readObject();
MyObject moB = (MyObject) ois.readObject();
ois.close();
```

- In a client/server setting where objects are serialized in both directions (from client to server and from server to client), care must be taken in setting up the object streams.
  - When an `ObjectOutputStream` is constructed, it writes a 4-byte header—and usually needs to be flushed right away like this:

    ```
    ObjectOutputStream oos = new ObjectOutputStream( //...
    oos.flush();  // push the header over right away
    //...
    ```

  - When an `ObjectInputStream` is constructed, it blocks inside the constructor until it reads that 4-byte header.

    ```
    ObjectInputStream ois = new ObjectInputStream( //...
    ```

  - If both the client and the server try to create their `ObjectInputStream`'s first, an inter-VM deadlock will result as both sides are blocking waiting to read the header from the other side (that never gets written).
  - If both sides try to create their `ObjectOutputStream`'s first, there's still some potential for trouble (if there's not enough buffering in the network to allow both headers to be simultaneously floating in opposite directions).
  - The solution is to create the `ObjectInputStream` first on one side while creating the `ObjectOutputStream` first on the other. The recommended pattern is (don't forget the flushes!):

| Client | Server |
|---|---|
| `ObjectInputStream ois =`<br>`    new ObjectInputStream( //...` | |
| | `ObjectOutputStream oos =`<br>`    new ObjectOutputStream( //...`<br>`oos.flush();  // push the header` |
| | `ObjectInputStream ois =`<br>`    new ObjectInputStream( //...` |
| ...`ObjectInputStream` receives the header and returns from the constructor... | |
| `ObjectOutputStream oos =`<br>`    new ObjectOutputStream( //...`<br>`oos.flush();  // push the header` | |
| | ...`ObjectInputStream` receives the header and returns from the constructor... |

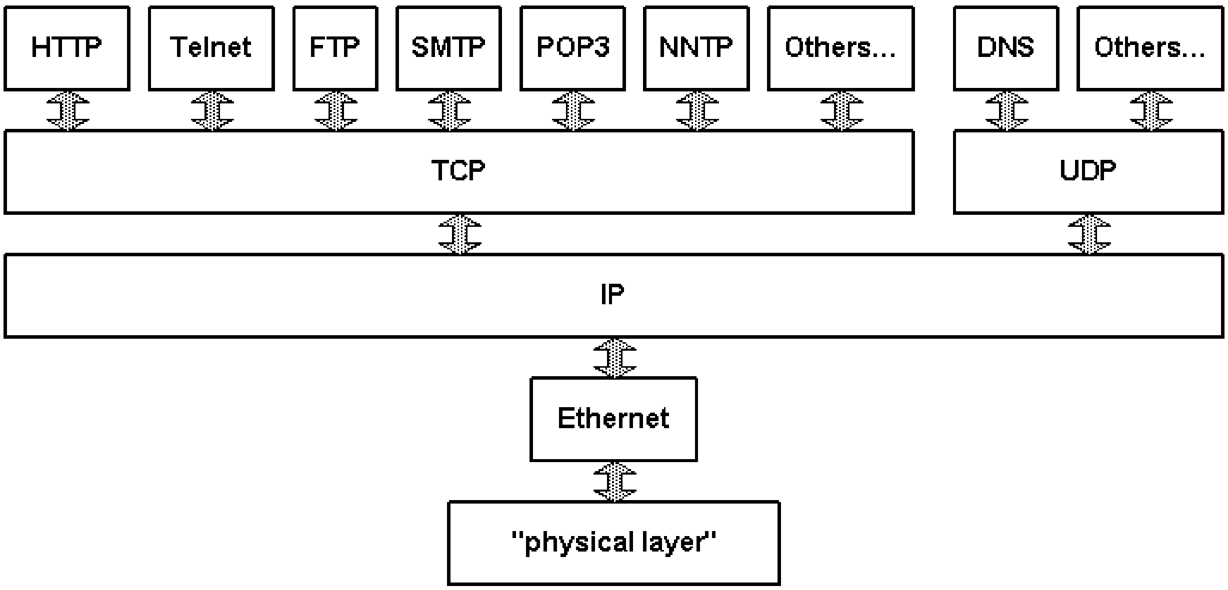
## TCP/IP Networking with Sockets

- TCP/IP Overview
  - Transfer Control Protocol / Internet Protocol –the networking protocol of the Internet and most intranets.

  - Packet-based
    - the data is broken up into small chunks and wrapped in packets. This would be similar to sending a printed document in the US Mail one page at a time.
    - each packet has a "To:" and "From:" address.
    - each packet has a sequence number "written" on the outside to help in reordering the packets on the receiving end if they arrive out of order.
    - packets are sent as quickly as they can given the available bandwidth. Under low traffic conditions one connection might enjoy most of the bandwidth.

  - Error Correcting –checksums are sent with each packet and are used to validate the data that is received. If a packet appears to be corrupted, a re-send is requested. Data is assumed to be received error-free through the TCP layer.

- Layered Protocol –each layer knows how to talk to the neighboring layers. This gives flexibility in implementation. Protocols on top of the TCP layer only need to know how to talk to TCP and don't need to be particularly concerned with the underlying layers (like the physical transport layer).




- the "physical layer" may be many things like twisted pair wired, phone lines, fiber optics, microwaves, or just about anything.
- in this diagram, "Ethernet" was shown, but other protocols like "PPP—Point to Point Protocol", and others can be used.
- the Internet Protocol—IP layer is the foundation.
- the User Datagram Protocol—UDP layer is for "connectionless" protocols.
    - a popular UPD protocol is Domain Name Service—DNS. This is used to convert hostnames to IP address and IP addresses back into hostnames. Like:

```
java.sun.com <-> 192.18.97.137
```

        - An IP address is a 32-bit address that uniquely identifies a machine or "host" on the Internet and is used to physically route the packets to the destination.
        - A hostname is used to more easily identify a machine—by a more convenient, more human friendly mechanism.
        - Usually one hostname translates into one IP address, but sometimes multiple IP addresses are returned for load balancing situations.
        - Usually one IP address translates back into one hostname, but sometimes, multiple small hosts are placed on one machine to keep hardware costs down.

- the Transfer Control Protocol—TCP layer is for "connection" protocols that persist a sense of a continuous connection between endpoints.

- HTTP – Hyper-Text Transfer Protocol is the protocol used to request web pages.
- Telnet – the protocol for logging into servers with terminal like functionality over the Internet.
- FTP – File Transfer Protocol is the protocol used to transfer ASCII and binary files between Internet hosts.
- SMTP – Simple Mail Transfer Protocol is used to send un-validated email (loosing popularity fast due to exploitation by spammers).
- POP3 – Post Office Protocol is used to sent and retrieve email.
- NNTP – Net News Transfer Protocol is used to post and retrieve Usenet (newsgroup) messages.
- Many custom protocols are implemented on top of the TCP layer.

**IP address and port numbers**

- IP addresses uniquely identify the source and destination hosts (machines).
- Port numbers are used to identify services or applications running various machines. For example, port 80 is the default port for the HTTP protocol. Port numbers are also used to direct return packets to the right application.

**Sockets**

- a socket is an IP address/port number combination on both ends. For example, if a host `209.112.34.121` wants to make a socket with `192.18.97.137` (`java.sun.com`) on port `80` (http) to retrieve a page, an unused local port (let's say `25401`) is needed to complete the socket:

```
209.112.34.121:25401 --- 192.18.97.137:80
```

- the general format is:

*<local ip>***:***<local port>* --- *<remote ip>***:***<remote port>*

- if two web browser windows are simultaneously retrieving two different pages from the same web server, then the two sockets only differ by the local port number:

```
209.112.34.121:25401 --- 192.18.97.137:80
209.112.34.121:25402 --- 192.18.97.137:80
```

--this is how the right data gets to the right window!

**Java and Sockets**

- use the `java.net` package

- to initiate a socket connection from a *client* to a *server*, use this constructor on `Socket`:

```
public Socket(String hostname, int port) throws IOException
```

  like this:

```
Socket sock = new Socket("java.sun.com", 80);
```

  which may throw many different kinds of exceptions including `UnknownHostException`, `NoRouteToHostException`, and many other subclasses of `IOException`.

- DNS services are automatically used behind the scene to resolve the hostname into an IP address.
- All sorts of exceptions—that are subclasses of `IOException`—can be thrown just about anywhere related to TCP/IP communications.

- to have a server *listen* for client socket connections, use this constructor on `ServerSocket`:

```
public ServerSocket(int portToListenTo) throws IOException
```

  like this to listen to port `3001` for incoming socket requests from clients:

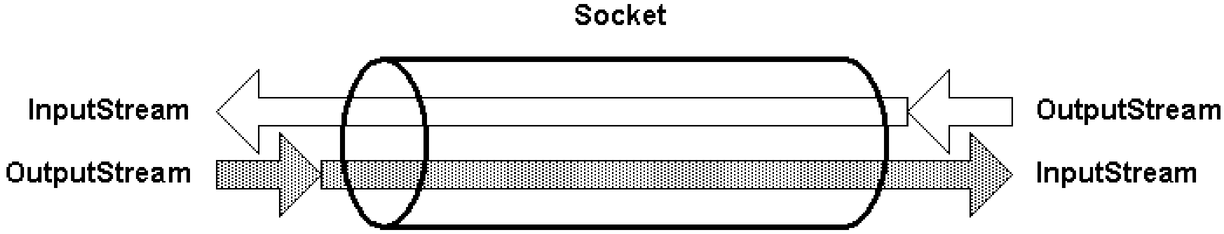
```
ServerSocket ss = new ServerSocket(3001);
```

- to wait for socket connections, use the `accept()` method on `ServerSocket`. The `accept()` method ***blocks*** the calling thread indefinitely until a socket request comes in. In addition, usually a loop is used to continually accept sockets:

```
while ( boolExpr ) {
    Socket sock = ss.accept();  // blocks until a connect occurs
    //... process ...
}
```

**Streams and Sockets**

- a socket has full-duplex communication and bytes are streamed in both directions simultaneously using streams:




- the `InputStream` on one end of the socket is connected to the `OutputStream` on the other end. The data might be delivered in "bursts" due to the packet nature of TCP/IP, but the data virtually moves continuously in both directions.

- to get a handle on the streams on one end of a socket, use these methods on `Socket`:

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

  like this:

```
Socket sock = //...
InputStream rawIn = sock.getInputStream();
OutputStream rawOut = sock.getOutputStream();
```

- the raw streams generally benefit from being wrapped in buffered streams to reduce the underlying network I/O activity.
- both the server and client ends of the socket "look the same" in after the instance of `Socket` is created.