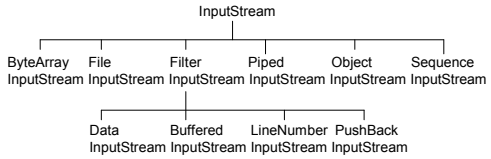
C14: Input/Output Streams

Streams vs. Readers/Writerse

- Confusing number of alternatives:
  - Streams: for binary 8bit (byte) quantities
  - Readers/Writers: for String and 16bit Unicode characters
- Streams: always 8bit quantities at the lowest level, but may provide higher-level abstractions as well, e.g.
  - readInt() [in class DataInputStream] will process four 8bit values to read a single 32bit int value

InputStream hierarchy



Physical: ByteArrayInputStream, FileInputStream, PipedInputStream

Virtual: all others

Class InputStream

```
abstract class InputStream {
    int read() throws IOException;
    int read(byte[] buffer) throws IOException;
    long skip(long n) throws IOException;
    int available() throws IOException;
    void close() throws IOException;
    }
```

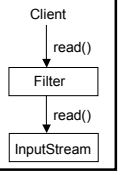
- IOException: e.g. file cannot be opened, read operation was interrupted, …

Physical input streams

- Some constructors:
  - ByteArrayInputStream(byte[] buffer);
  - ByteArrayInputStream(byte[] buffer, int offset, int count)
  - FileInputStream(File f);
  - FileInputStream(String fileName);
  - PipedInputStream(PipedOutputStream p);
- Why use File instead of String as argument to the FileInputStream constructor?

Virtual input streams

- Build on top of physical ones using composition, add additional behaviour, e.g.:
  - SequenceInputStream: appends a couple of InputStreams giving the impression/illusion of just a single stream
  - FilterInputStream: an instance of the "Wrapper design pattern"



DataInputStream

```
class DataInputStream extends FilterInputStream implements DataInput {
    …
    public boolean readBoolean() throws IOException
    public byte readByte() throws IOException
    public char readChar() throws IOException
    public double readDouble() throws IOException
    public int readInt() throws IOException
    public long readLong() throws IOException
    public short readShort() throws IOException
    public int readUnsignedByte() throws IOException
    public int readUnsignedShort() throws IOException
    }
```

- **Read bytes (binary representation, not textual), return them as primitive data types**
- **Possibly generated by DataOutputStream**
- **Same interface also used by RandomAccessFile**

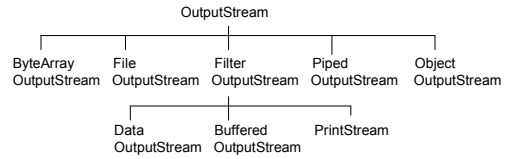
Stream Tokenizer

- Useful to break up text into tokens (words, numbers, others):

```
public class TokenTest {
    public static void main(String[] args) {
        StreamTokenizer tok = new StreamTokenizer(System.in);
        try {
        while (tok.nextToken() != tok.TT_EOF) {
            switch(tok.ttype) {
                case tok.TT_NUMBER:
                        System.out.println("number " + tok.val);
                        break;
                …
                case default: …
            }}
        } catch (IOException e) {}
}}
```

- "23-skidoo, kid!" yields: number 23.0 token – word skidoo …

Output stream hierachy



Similar to InputStream hierarchy:

physical: ByteArrayOutputStream, FileOutputStream, PipedOutputStream

virtual: all others

OutputStream

```
public abstract class OutputStream {
    public abstract void write(int b) throws IOException
    public void write(byte[] buffer) throws IOException
    public void flush() throws IOException
    public void close() throws IOException
    }
```

- Quite a few output streams use buffering (for efficiency reasons), flush() explicitly forces completion of pending output operations

PrintStream

```
public class PrintStream extends FilterOutputStream {
    …
    public void print(boolean bool)
    public void print(int inum)
    public void print(float fnum)
    public void print(double dnum)
    public void print(String str)
    public void print(Object o) { print(o.toString()); }
    …
}
```

- Generates textual representations
- print(Object o) is purely polymorphic

Object Serialization

- Transform instances into 8bit units can be send over networks, stored in filesystems
- Allows for object persistence, i.e. save state across program invocations
- Interface Serializable, classes ObjectInputStream and ObjectOutputStream
- Would be tricky: need to get shared structure right!

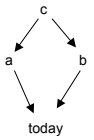
Object Serialization example

```
import java.io.Serializable;
class Pair implements Serializable {
    public Object first;
    public Object second;
}
class Holder implements Serializable {
    public Object value;
}
```

- Actually empty interface, so “implements Serializable” suffices

Object Serialization example contd.

```
import java.util.Date;
…
Date today = new Date();
Holder a = new Holder();
a.value = today;
Holder b = new Holder();
b.value = today;
Pair c = new Pair();
c.first = a;
c.second = b;
```



Object Serialization: write to file

```
try {
    FileOutputStream f = new FileOutputStream(“saveState”);
    ObjectOutputStream s = new ObjectOutputStream(f);
    s.writeObject(“The value of c is”);
    s.writeObject(c);
    s.flush();
} catch (IOException e)
    System.out.println(“received error “ + e);
}
```

Object Serialization: read back

```
try {
    FileInputStream f = new FileInputStream(“saveState”);
    ObjectInputStream s = new ObjectInputStream(f);
    String tag = (String) s.readObject();
    Pair c = (Pair) s.readObject();
} catch (IOException e) {
    System.out.println(“received IO exception “ + e);
} catch (ClassNotFoundException e) {
    System.out.println(“received class exception “ + e);
}
```

Object Serialization: check ==

```
…
Holder a = (Holder) c.first;
Holder b = (Holder) c.second;
if( a.value == b.value )
    System.out.println(“it’s the same object”);
…
```

Piped IO

- Producer/consumer paradigm
- Multiple threads and pipes
- A pipe: buffered data area used for both reading and writing
- May suspend on IO operations:
  - Read, because empty
  - Write, because full
- In Java: matched pair of streams:

```
PipedInputStream in = new PipedInputStream();
PipedOutputStream out = new PipedOutputStream(in);
```

Piped IO Example: Fibonacci + Prime

```
class FibMaker extends Thread {
    private DataOutputStream out;
    public FibMaker(DataOutputStream o) {out = o;}
    public void run() {
        int n = 0;
        int m = 1;
        try {
            out.writeInt(m);
            while(m<100000) {
                int newValue = n+m;
                n=m;
                m=newValue;
                out.writeInt(newValue);
                }
            out.close();
            } catch (IOException e) {return; }}}
```

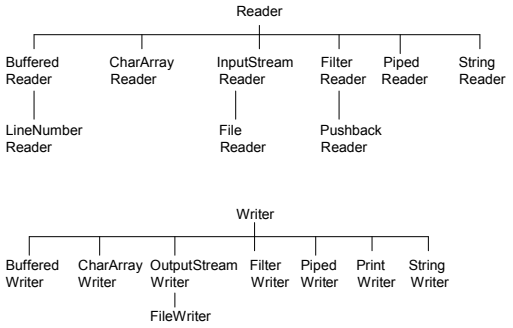
Piped IO Example contd.

```
class PipeTest {
    static public void main(String[] args)
        {PipeTest w = new PipeTest(); w.show(System.out);}
    private PipeTest(PrintStream out) {
        DataInputStream fibs = makeFibs();
        DataInputStream primes = makePrimes();
        try {
            int x = fibs.readInt();
            int y = primes.readInt();
            while (x < 100000) {
                if (x == y) {
                  out.println(x + " is both prime and fib");
                  x = fibs.readInt();
                  y = primes.readInt(); }
                else if (x<y) x = fibs.readInt();
                else y = primes.readInt();
                }
            } catch (IOException e) { System.exit(0);}}
```

Piped IO Example contd.

```
private DataInputStream makeFibs() {
    try {
        PipedInputStream in = new PipedInputStream();
        PipedOutputStream out =
                new PipedOutputStream(in);
        Thread fibThread = new FibMaker
                (new DataOutputStream(out));
        fibThread.start();
        return new DataInputStream(in);
        } catch (IOException e) { return null;}
    }
```

Readers and Writers hierarchy



Readers/Writers

- Purely textual, but similar to IO streams

```
try {
    FileReader f = new FileReader("filename");
    BufferedReader b = new BufferedReader(b);
    … } catch (…){ … } …
```

- Also wrap around IO streams:

```
FileInputStream f = new FileInputStream("f");
InputStreamReader r = new InputStreamReader(f,
    "MacCyrillic");
BufferedReader b = new BufferedReader(r);
String text = r.readLine();
```