# CPS122 Lecture: Input-Output

**Last Revised April 9, 2019**

*Objectives:*

1. To discuss IO to System.in/out/err
2. To introduce the abstract notion of a "stream"
3. To introduce the java File, Input/OutputStream, and Reader/Writer abstractions
4. To show how to read and write primitive types and strings as either binary or text data
5. To show how to serialize/deserialize objects
6. To discuss exceptions in the context of input-output

*Materials:*

1. Magnetic tape to show
2. Projectable showing binary and text representations of data, and different kinds of text representations
3. Projectable of various representations for the line "Hello"
4. Handout of diagram showing layered structure of major java.io classes, plus version to project.
5. Demo programs: BinaryFileMaker.java, BinaryFileAccessor.java, TextFileMaker.java, TextFileAccessor.java, RandomFileAccessor.java
6. Projectable of read and write code from LibraryDatabase.java

## I. Introduction to Input-Output

A. Computer programs generally don't exist in isolation, but rather as part of complete software systems (e.g. a payroll system; a web-based order taking system, etc). Moreover, it is quite common for a complete system to involve programs running on multiple computers communicating with one another over a network.

Such systems must work with three basic types of data:

1. Permanent data - data that the system must maintain over a long period of time - e.g.

   a) Payroll system: data on employees (names, SSN's, pay rates, year-to-date pay and withholding, etc.)

   b) Web-based order system: data on customers (name, address, credit card number, etc.) and available goods (description, price, etc.)

   This data typically resides on disk

2. Transient data - data that is exists outside of any particular program, but which is needed only for a brief period of time (hours, or days)

   a) Payroll system: weekly "timecard" information for employees who worked during a given pay period (typically keyed into the computer by a payroll clerk, perhaps from written time sheets)

   b) Web-based order system: individual customer orders (typically sent over the Internet from the customer's browser).

   This data may reside on a variety of media: currently often on disk or as packets transmitted over the Internet.

3. Temporary data - data that is only in existence when a given program is running. The variables that we have been using to store and manipulate data fall into this category. They reside in the computer's RAM only during the time the program of which they are a part is running.

4. In addition, many programs are responsible for producing different kinds of printed reports.

B. In the case of disk files used for permanent or transient data, and also in the case of communication streams between systems, some further distinctions are important:

1. The distinction between *pure binary data* and *text data*.

a) In pure binary data, information is stored as a sequence of bits identical to the internal binary representation of the entity being stored.

*EXAMPLE:* ints are stored as 32 bits, grouped into four 8-bit bytes. The int 42 has binary representation

00000000 00000000 00000000 00101010 **PROJECT**

b) Text data is represented in binary as well, but with two important distinctions:

(1)In text data, information is stored as the binary representation of a series of codes for characters that constitute the printed representation of the value. If the file represents characters in ASCII (as most files do - even those accessed by Java) - each character is represented by one byte. For example, the int 42 has ASCII representation `00110100 00110010`

**PROJECT**

(In this case, the text representation is more compact. That is not the case in general - binary encoding can store values up to 2 billion in four bytes; this would require 10 characters.)

(2)Of course, textual data can also be represented using Unicode, in which case 42 would have the representation
`00000000 00110100 00000000 00110010`

PROJECT

(3)The data is typically stored as a series of lines, with the lines marked in one of several ways (depending on the platform being used).

(a)Some platforms have used a “count + data” format for lines, where each line is represented by a pure binary integer encoding the number of characters in the line, followed by the individual characters, each encoded in a form such as ASCII.

(b)Most platforms represent a line by a sequence of characters followed by a special “line-end” marker - which varies from platform to platform.

i) On Unix-based platforms (including Mac OS X): \n

ii) On Windows platforms: \r \n

Example: if a line consisted of the word "Hello", its binary representation would be one of the following:

PROJECT

Unix/OSX: `01001000 01100101 01101100 01101100 01101111 00001010`
Windows: `01001000 01100101 01101100 01101100 01101111 00001101 00001010`

c) Note that for either binary data or text data the data on disk is a bit pattern; but you have to know which way it is represented to interpret it properly - e.g. if the binary representation for 42 were interpreted as if it were text, it would be interpreted as ‘*’; and if the text representation for 42 were interpreted as if it were binary, it would be interpreted as 13362.

d) Frequently, permanent data is stored in binary files. Transient data is often stored in text files, so that it can be easily manipulated by utility programs that work with text files such as text editors. (Transient data may be sent over a network in either form.) Internally, temporary data is generally represented in binary, unless it is textual - e.g. the Java primitive types (int, boolean, double, etc) are represented in binary, but Strings are text data.

2. The distinction between *sequential* and *direct-access* files on disk.

a) Historically, the earliest media used for long-term storage of data only supported reading and writing data sequentially, beginning at the start of the medium.

*EXAMPLE:* Magnetic tape [ SHOW ]

Also, some sorts of data come in an inherently sequential form - e.g. data typed on a keyboard, or packets arriving over the Internet.

b) Disks allow direct access to data anywhere on the disk in fairly short time.

*A technical note on terminology:* Disk files are *direct access* but not *random access*. The difference is that random access implies that the time to access a given item of data is independent of where it is located. On a disk, it is possible to access any location directly, but a location that is physically closer to the last one accessed can be accessed more quickly.

c) Frequently, transient data is stored in sequential files. Often, permanent data is stored in direct access files to permit direct access to the desired information on demand. (E.g. a web order system would exhibit unacceptable performance if it had to read through all the customer information from the beginning to find the desired customer when a new order arrived!)

C. In the overall structure of a computer system, a program is directly responsible for managing its own variables, but all other kinds of data (disk files, network packets, printed documents) are managed by the operating system.

1. The portion of the operating system that deals with these issues is called the *input-output system - or IO system for short*.

   (Note the use of the terms BIOS = basic input output system and DOS = disk operating system in the PC world)

2. Indeed, this was one of the original motivations for developing operating systems in the first place.

   a) To avoid burdening each program with detailed knowledge of the particular physical IO devices.

   b) To make it easy to change the specific device used for input-output without requiring massive overhaul of the program.

   c) To provide a measure of security for data, so that an erroneous (or malicious) program cannot easily destroy important data.

3. Although the file system portion of the operating system was designed to work with disk files, it generally allows access to other kinds of input output devices by treating them as if they were disk files. This includes devices such as the keyboard, as well as the network.

   Example: On Unix systems when terminal input-output is used, the "file" `/dev/tty` is actually the user's terminal

D. When a program needs to access a file, it typically does this through an abstraction called a stream - which represents a connection to a file (or some other medium).

## II. **Input-Output in Java**

A. The designers of Java faced an important challenge. The input-output system is part of the operating system, and there are significant differences between the input output systems between various operating systems.

Example: file specification syntax

1. On Unix systems, the names of disks and directories appearing in the full path specification for a file are separated by slashes (e.g. your Library project on our server is actually stored in the directory `/home/group-login-name/Library`).
2. On Wintel systems, a backslash (\) is used instead.
3. On older (OS 9 and before - which is what was in use at the time Java was first developed) Macintoshes, a colon (:) was used.

   Since Java is meant to be platform-independent, it was necessary to develop a Java IO system with its own set of conventions that can be implemented on top of each operating systems own mechanisms.

B. Reflecting Java's Unix heritage, Java programs run from the command line have easy access to three standard streams: System.in (standard input), System.out (standard output) and System.err (standard error). This can be used for simple IO, but is not sufficient for more sophisticated applications, which typically use graphical user interfaces (GUIs) for interacting with the user, and files for storage of transient and permanent data. We focus on the latter today.

C. The Java IO system is housed in the package java.io. This package implements a hierarchy of fundamental abstractions, implemented by objects that use other, lower-level objects. There are quite a few classes in this package. We will look at only a few.

D. The structure of this package is based on a an architectural pattern we looked at earlier - the notion of a *layered system* in which a complex task

is broken down into a series of layers, with each layer using the services of the layer immediately below it. Note: We will not discuss all the methods at each layer - only the most critical ones in terms of establishing the functionality of each layer.

*HANDOUT DIAGRAM AND PROJECT IT*

1. At the lowest level, the class File represents the full *name* of a physical file or directory on a disk, which may or may not actually exist. This class deals with platform-specific issues in terms of path-specification syntax.

   a) A File object supports basic operations such as inquiries about the file's existence and, if it exists, inquires about its size and last modification date and operations like renaming or deleting the file.

   b) A File object does *not* support transferring information to or from the file. (I.e. it represents the name of a container for information, but provides no way to actually access the information inside it.)

   c) It would, perhaps, have been clearer if the designers of java had called the class something other than File - e.g. FileName or FilePath.

2. At the next level up, there are three options, depending on whether the file being accessed is a sequential binary file, a sequential text file, or a direct access file. In each case, the object at this layer can be built on top of a File object, or in some cases on top of some other facility (such as a network connection).

   a) For work with sequential binary files, the classes InputStream and OutputStream are provided. <u>These presume that the underlying information is stored/transmitted in binary.</u>

(1)An InputStream permits a program to read an individual byte, or a series of bytes (the read method has two variants).

(2)An OutputStream permits a program to write an individual byte, or a series of bytes.

(3)The classes at this level only provide for manipulation of individual bytes (in a binary file).

b) For work with text files, the classes Reader and Writer are provided. <u>These presume that the underlying information is stored/transmitted in textual form.</u>

(1)A Reader provides methods to read the code(s) for a single character or a series of characters

(2)A Writer provides methods to write a single character or series of characters

(3)The classes at this level only provide for manipulation of individual characters.

(4)One of the issues these classes must deal with is the fact that Java represents characters internally using 16-bit Unicode, but many disk files are 8-bit ASCII, and data transmitted over the network may use an encoding known as UTF-8. (The latter is a scheme under which characters that can be represented using 7 bit ASCII are represented by a single byte, while all other characters are represented using 2 or 3 bytes.)

(a)Conversion between these schemes is handled by Reader / Writer objects. A Reader or Writer object "knows" what representation is actually used for characters in a particular file or network connection.

(b)Thus, Reader and Writer should always be used with text data - the various forms of binary streams should never be used, since they do not deal with this issue.

c) For work with direct access files, the class RandomAccessFile is provided. This presumes that the underlying information is stored in binary (there is no analogous structure for text or for information transmitted over a network, because direct access is not really practical in these cases due to varying-size data representation in text or transmission-time issues over a network.)

A RandomAccessFile can support both reading and writing of data from/to a direct access disk file. (Note that the name of the class is really not well chosen - since disk files support direct access but not true random access!) This class actually corresponds to both this layer and the next layer up in the design, so for now we just note that it has methods to read individual bytes - or a series of bytes - and to write individual bytes - or a series of bytes. We will discuss it again in conjunction with the next level up.

The `seek()` method is used to position the file to read or write at a specific point.

A RandomAccessFile object supports both reading and writing data with the same file This is because it is common with permanent data to read, modify, and update some value "in place" (e.g. the total outstanding amount owed by a customer). A given file may be opened in read-only mode, which would make the write operations fail on that file.

3. The next level up again provides two options, depending on whether the program wants to read/write a binary file or a text file. In each case, the object at this layer is built on top of an object of the appropriate kind at the next layer down. The class RandomAccess file also provides services at this layer.

a) For work with sequential binary files, the classes DataInputStream and DataOutputStream are provided.

(1)A DataInputStream is always constructed on top of an InputStream. It allows the program to read any of the primitive data types from a binary file, by getting the appropriate number of bytes from the underlying stream and converting them into the appropriate internal type. It has methods (see handout)

```
boolean readBoolean()
char readChar()
byte readByte()
short readShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
```

There is no method to read a String. To read a String, the program must know how long of a String to expect, and then must do repeated readChar() operations. This is because there is no standard way of marking the end of a string in a binary file.

(2)A DataOutputStream is always constructed on top of an OutputStream. It allows the program to write any of the primitive types to a binary file, by converting the internal type into a sequence of bytes and then writing them to the underlying stream. It has methods (see handout)

```
void writeBoolean(boolean)
void writeChar(char)
void writeByte(byte)
void writeShort(short)
void writeInt(int)
void writeLong(long)
void writeFloat(float)
void writeDouble(double)
```

It also has a method void writeChars(String) - which simply does a writeChar for each character in the string. (Not doing anything special to specify the end of the string.)

(3)In the case of character data (char and String), data is read/written as 16-bit Unicode values - which is not the standard representation for character data on most platforms, of course. However, this sort of stream is usually processed only by other programs written in Java.

b) For work with text files, the classes BufferedReader and PrintWriter are defined.

(1)A BufferedReader is always constructed on top of a Reader. It adds one new method

```
String readLine() // Reads a line - returns null (not "") if
                  // at end of file
```

(2)A PrintWriter is always constructed on top of a Writer. It allows the program to write the textual representation of any primitive type, with optional line termination after it, by converting the internal type to a sequence of characters and then writing each character in turn to the underlying stream.

```
void print/println(boolean)
void print/println(char)
void print/println(byte)
void print/println(short)
void print/println(int)
void print/println(long)
void print/println(float)
void print/println(double)
void println()              // Just output newline - no data
```

(3)There is an asymmetry in the structure of the java library: there are provisions for writing any primitive type in its textual form, but no direct provision for reading the textual form of a primitive type.

(a)The principal reason for this is that reading requires some sort of *lookahead read* - e.g. when reading the number 42

from a file, you don't know you've read all the digits until you see some non-digit character such as a space. (If you've seen 4 and 2 thus far, the next character could be a 3 in which case the number is 423.)

(b) To read numeric data from a text file, one typically reads a line of characters from a BufferedReader, and then converts it to binary form using various "parse" methods.

c) The class RandomAccessFile provides methods at this layer as well for use with direct-access binary files: (see handout)

```
boolean readBoolean()
char readChar()
byte readByte()
short readShort()
int readInt()
long readLong()
float readFloat()
double readDouble()

void writeBoolean(boolean)
void writeChar(char)
void writeByte(byte)
void writeShort(short)
void writeInt(int)
void writeLong(long)
void writeFloat(float)
void writeDouble(double)
void writeBytes(String)    // Writes each character as 8 bits
void writeChars(String)    // Writes each character as 16 bits
```

4. In addition to what we have discussed here, there are many other options - e.g. a stream can be built on top of the console or a network connection or a pipe or even an array of bytes; and a reader/writer can be built on top of a stream (and hence anything a stream can be built on top of) or a character string.

5. In addition to the operations we have discussed thus far, there are two other very important operations on a file.

a) *opening or creating the file* - which initially establishes access to it. In Java, this is taken care of by the constructor for the various classes that provide access to a file. (But not by the class File per se, because it doesn't actually provide access to the content of a file.)

(1)The constructors for sequential input files (FileInputStream, FileReader) require that the user specify an existing file to be opened.

(2)The constructors for sequential output files (FileOutputStream, FileWriter) normally create a new file. However, there is one form of the constructor that allows specification of appending to an existing file instead.

(3)The constructor for a random access file opens an existing file if one of the specified name exists; otherwise it creates a new, empty file. It allows the user to specify one of two modes: "r" - read only, or "rw", read and write/update.

b) *closing the file* - which terminates access to the file. All of the file access classes provide a method

```
void close()
```

In the case of a file that you are writing data to, it is *extremely important to remember to close the file!* The reason for this is that data is transferred to/from a disk in units called blocks (typically some multiple of 512 bytes.) When you write data to a file, it is accumulated in memory until there is enough to make a complete block, and then the whole block is physically written to the disk. This means that, typically, when a program finishes writing to a file there will be a partial block that has not yet been physically written to disk. One thing the close() operation does is to flush this last partial block to the disk. Failing to close a file when you are through writing to it can result in loss of data.

6. As a final note: printers are considered input-output devices on most computer systems, and most operating systems provide access to printers through their file system. However, the Java support for access to a printer is through the abstract windowing tool kit (awt) - in essence, one can paint on a printer the way one paints in a window.

E. The java.io package makes extensive use of the Decorator (Wrapper) pattern, which we introduced when we talked about design patterns.

1. the java.io package provides many kinds of InputStreams - objects that can read streams of bytes from a variety of different sources - e.g.

   a) FileInputStream - reads bytes from a disk file

   b) ByteArrayInputStream - "reads" bytes from an array

   c) PipedInputStream - reads bytes from a pipe (a connection between two processes running on the same computer such that one process can "feed" data to another through the pipe).

   d) A network socket - reads bytes sent over a network from another computer.

2. Recall that, in java, characters of text are not represented by single bytes, but by using Unicode, which represents characters as pairs of bytes - but most platforms represent characters in text files either using plain ASCII or an encoding called UTF-8 which stores "standard" characters using one byte, but may use up to 3 bytes for some less common characters.

   a) Therefore, the java.io package defines a Reader class (and various subclasses) that supports a read() method that reads a single character from some source To this, it may actually need to read just one byte and pad it out with 0's - or it may need to read two or even three bytes.

b) Rather than having to define a Reader subclass for every possible kind of input source, the java.io package includes a wrapper called InputStreamReader that allows any input stream to function as a Reader - i.e. each call to its read() method reads the appropriate number of bytes from the underlying input stream and returns the result as a Unicode character.

(The package also includes several "convenience classes" such as FileReader that work with an InputStreamReader under the hood.)

3. Moreover, when processing text programs often need to work with complete lines of text - not just individual characters. A line of text is a sequence of characters terminated by a line terminator.

a) However, as you recall, the line terminator is platform-specific and environment-specific

b) Therefore, the java.io package includes a wrapper called BufferedReader that allows any reader to be used as a source of complete lines. A BufferedReader has - in addition to the standard individual character read() method - a readLine() method that keeps reading characters until it has read a line terminator, and then returns a String composed of all the characters read except the terminator (i.e. a platform-independent representation of the contents of the line.)

4. A major advantage of the “wrapper” approach used by the java.io package is that it avoids a proliferation of classes. - e.g. we don’t need a special kind of BufferedReader for files, and a different one for pipes, and a different one for network connections, etc. - instead, a single BufferedReader wrapper can build the necessary functionality on top of any kind of Reader; and a single InputStreamReader wrapper can build the necessary functionality on top of any input stream.

We will see examples of this in the demos which follow.

F.*SHOW, DEMO*

1. BinaryFileMaker.java

   a) Use of JFileChooser

   b) Use of constructors wrapped around constructors

   c) Writing of various kinds of data to file

   d) Show contents of file in TextWrangler - note how most of it is unreadable

   e) Show raw binary using `od -t u1`

   f) Get file size - show how this follows from data written to it

      1 byte for boolean, byte
      2 bytes for char, short
      4 bytes for int, float
      8 bytes for long, double
      4 bytes for length of string, plus 2 bytes per character

2. BinaryFileAccessor.java

   a) Use of JFileChooser

   b) Use of constructors wrapped around constructors

   c) Reading of various kinds of data from file

3. TextFileMaker.java

   a) Use of JFileChooser

   b) Use of constructors wrapped around constructors

   c) Writing of various kinds of data to file

   d) Show contents of file in TextWrangler- note how all of it is readable

   e) Also show binary using `od -t u1, od -t c`

4. TextFileAccessor.java

   a) Use of JFileChooser

   b) Use of constructors wrapped around constructors

   c) Reading of various kinds of data from file

5. RandomFileAccessor.java

   a) Use of JFileChooser

   b) Use of constructors wrapped around constructors

   c) Positioning using seek

   d) Reading of various kinds of data from various positions in the file:

      Position 0 - boolean is read correctly
      Position 1 - char is read correctly
      Position 3 - byte is read correctly
      Position 4 - short is read correctly
      Position 6 - int is read correctly
      Position 10 - long is read correctly
      Position 18 - float is read correctly
      Position 22 - double is read correctly

## III. **Serialization**

A. The various types of files we have discussed provide support for reading and writing the various primitive types. What about reading and writing objects?

B. A solution used in many OO languages, and in the earliest version of Java, was to require that the programmer create methods to read or write an object that needs to be saved to disk by reading and writing its individual fields.

C. Early in its history, Java added support for *object serialization*, which allows objects to be written and read in binary form just like primitive types.

1. This is the approach used in your Library project.

   PROJECT Excerpts from LibraryDatabase.java that handle reading and writing

   a) The class ObjectInputStream builds on an InputStream, and supports a method readObject() (as well as methods for reading primitive types)

   b) The class ObjectOutput builds on an OutputStream, and supports a method writeObject() (as well as methods for writing primitive types)

2. To be eligible to be read/written from/to a stream, an object's class must be declared as implementing Serializable. The Java language takes care of creating the code necessary to actually read and write the object.

D. There is a lot more involved here than might first appear, because objects can contain references to other objects.

   Example: Suppose we stored registration information about courses using a structure like this (simplified form what we used earlier in a lab)


   That is, each Student object holds references to the Course objects for the courses the student is enrolled in (perhaps an array or some other sort of collection; and each Course object similarly holds references to the Student objects for the students registered in it.

   Now consider what happens if we try to save one Student object to disk:

1. That Student object needs to be saved

2. The Course objects for each course the student is in needs to be saved.

3. The Student objects for each student in each course needs to be saved - which can result in saving more Course objects ...

   It is not inconceivable that saving one Student object might result in saving the entire database!

4. Of course, we only want to save each object once - e.g. if ten students in this course are both enrolled in Calculus II, we only want to save the Calculus II object once; and when we do, we don't want to save the Student objects for these individuals again.

E. This is all taken care of by object serialization.

## IV.Exceptions

A. IO operations are inherently prone to possible failure for a variety of reasons

1. Attempting to perform an operation that is inherently impossible:

   a) Attempting to open a non-existent file

   b) Attempting to create a file or write on a read-only device (e.g. a CDROM) or a device that has no room available.

   c) Attempting to read past the end of a file.

2. Many operating systems have a file permission mechanism that allows the owner of a file to control what users may access it. An attempt to access a file in violation of these permissions is not allowed.

3. IO systems can have hardware problems such as

   a) Disk errors due to defects in the disk

   b) Network errors due to another system that one is communicating with going down or communications problems

4. Data format problems when reading data from a text file - e.g. encountering a letter when trying to read a number. - etc.

B. We have already met the general mechanism Java uses for dealing with issues like this - the exception mechanism. Exception-handling is particularly prominent when dealing with IO operations.

   1. The package `java.io` defines a general exception category known as `IOException` with many subclasses - e.g. (briefly explain each)

      a) `EOFException`

      b) `FileNotFoundException`

      c) `InterruptedIOException`

      d) etc., etc.

      e) Specific implementations may also create specific subclasses of `IOException` to deal with various specialized problems.

      Most of the io methods are declared as throwers (or propagators) for `IOException`.

   2. In addition, if you are reading input as character strings to be parsed as numbers, you also need to be prepared to deal with exceptions such as `NumberFormatException`.

## V. Communicating over a Network

A. The examples we have considered thus far have focussed on performing IO operations on disk files. Often, though, programs communicate with other programs over a network connection.

B. For network communication, the fundamental abstraction is called a sockets, which serves as an endpoint for communication to another computer.

1. When two computers are communicating, there exists a socket on each which is connected over the network to the corresponding socket on the other machine. Thus, sockets normally come in pairs - one on each machine.

2. Associated with each socket is an input stream and an output stream. Whatever is written to the output stream on one computer becomes available to be read from the input stream on the other computer, and vice versa.

   That is, a pair of connected sockets behaves like a "tin-can" telephone.

3. Of course, this creates the question of how a socket pair gets created in the first place. To make this possible, there is a special kind of socket called a server socket which isn't actually connected to any other socket. Rather, it waits for a connection attempt from some other computer and then creates a socket connected to the socket that attempted the connection.

4. We will look at this in more detail in CPS221 next year.