13. Java Input and Output

CMPT 101: For an introductory programming course, it is only necessary to study sections 13.1, 13.5, 13.9, and 13.10.

Java's input and output (I/O) programming features are not simple. There are a *bewildering* number of useful classes for input/output! In addition, to get these classes to do your input and output, you usually have to have several of them working together for you.

Java I/O is based on the concept of 'streams'. A stream is a hose for data coming in from input devices like a keyboard, file, or a network connection. The keyboard stream System.in is already available to you. For files and network connections, you will have to give the particular stream constructor that you need the name of the file or Uniform Resource Locator (URL) of the network point from which you wish to read. Similarly, there are output streams.

In addition, there are filtering streams that are like a hose that you can connect to another hose. Filtering streams convert the data passing through them from one format (e.g. two's complement integer) to another (Unicode or ASCII) characters. The latter format is required for North American keyboards, printers, and DOS/NT console windows. e.g. For output of int variables to a human readable file, you want to connect a filtering hose to a destination hose which terminates in an actual file.

Java has a wonderful array of filtering, and source and destination streams. The can provide data compression, compute checksums, connect to web sites, etc.

REQUIRED READINGS: none.

OPTIONAL READINGS: Ch. 9 of [Savitch2001]

Section Table Of Contents

13.1 Data Formats and Conversions

In most computer languages, there are only two general formats: textual and binary.

In conventional text format, the data is in ASCII (American Standard Code for Information Interchange) or, for IBM mainframe computers, in EBCDIC. These are just mapping tables indicating what byte value is used inside a computer, and for keyboard and printer, to store say an upper case A, or a comma. If you want to print a number out so that a human can read it, it must be converted from whatever internal numerical format is has to a sequence of bytes containing the ASCII digit characters. Unfortunately, different ethnic languages use the same ASCII codes above 128 for different accented characters.

In conventional binary format, numbers are stored in non-ASCII form. These are more compact and easier to compute with. Unfortunately, different computers store (particularly floating point) numbers in binary form differently from each other. In addition, there is a terrible incompatibility between computer processors as to whether the most significant byte of an integer or floating point number should be lower or higher in RAM memory. When sending such numbers to files or networks, the byte with the lowest address is sent first. However, if this data goes to a computer that stores things the other way around, chaos results.

So, before Java things were sometimes not very compatible, but were relatively simple:

1) characters were written directly in binary as no conversion was needed (at least if you were using the correct type of printer).

2) numbers written to printers and character windows had to be converted (called formatted I/O) from internal format (e.g. twos complement) to ASCII characters (e.g. '1' '2' '8' '6' '4' '3' '.' '9'). Most programming languages had functions like System.out.println(int) to perform these functions, and reverse ones for reading.

3) numbers were written directly to files and network connections in binary with no conversion (and hopefully you didn't have a 'which end first' problem reading them on another computer).

Java attempts to alleviate the representation, ethnic, and byte-ordering difficulties mentioned above. Java adopts international formats for storing numbers, and international conventions for whether the least significant or most significant byte of a number is sent to a file or network connection first. (Java uses 'big end' first like Sun computers and most network standards. The Intel microprocessor used in IBM PCs uses 'little end' first.)

In addition, Java adopted the so-called Unicode character set. This set uses two bytes to store each character. Most Java GUI I/O is based on Unicode characters, and there are many input and output stream classes to help convert characters coming in and out of Java from non-Java files.

Unfortunately, now that there are two bytes for each character, Java had to adopt an character byte ordering convention: most significant byte first. And, it is no longer

straight forward to write characters to a printer or DOS/NT console window as the underlying hardware requires ASCII bytes, not Unicode.

Now with Java:

1) Java characters (2 bytes) cannot be read directly to or from ASCII I/O (1 byte) devices, but must go through conversion functions or filtering streams.

2) However, you can also write Java Unicode characters in binary (i.e. directly), storing the 16 bits directly for later use when read back into Java.

3) Java number variables can be read and written to files and network connections in binary without conversion, and are compatible with network transmission standards and some computers. They are always compatibly read by another Java program.

4) To print Java numbers for humans to read, they have to be converted. However, you now have two choices. You can convert them to ASCII with a PrintStream like System.out. Or you can convert them to Unicode characters using either a PrintWriter, or the number's toString( ) method. You might use the latter for writing to a Java GUI text box (which is designed to display/edit/enter Unicode characters).

13.2 Intro to Streams

There are 5 subclasses that form the first part of the inheritance tree for Java I/O.

Object subclasses:

1) /*abstract*/ InputStream
2) /*abstract*/ OutputStream
3) RandomAccessFile
4) /*abstract*/ Reader
5) /*abstract*/ Writer

The subclasses of InputStream are designed simply to read bytes or arrays of bytes. They do not know how to convert the bytes to Unicode characters, nor how to read some ASCII digits and convert them into twos complement representation for storage in an 'int' variable. Nonetheless, the constructors of its further subclasses provide your program with a hose that has data coming out of it a byte at a time. The constructors normally take some parameter that indicates where the data should be sourced from (e.g. a file name). **There is a default input stream provided by Java called System.in that is pre-connected for you to the standard input device (usually the keyboard).**

The subclasses of OutputStream do the opposite.

RandomAccessFile is a special kind of subclass for streams on which you can both read and write. In addition, you can

seek( ) to any position in the file before beginning to read and write. Thus, you can write into the middle of a file. Random access is also good for writing primitives to the file in a machine-independent binary format. That means that an integer is written to file is its compact twos complement form. This form is unreadable to humans, but requires little format conversion, and is very compact (both good for reading back in later).

Random access files are not used very frequently. To get this same binary I/O functionality requires putting special filters streams on existing input or output stream subclasses.

Reader streams allow you to read into Unicode char variables or char[ ] arrays. Various concrete subclasses will allow you to read characters from a file (with any conversion you want (say ASCII to Unicode), read from a String, read from a pipe (some output stream), etc.

Writer streams allow you to write Unicode characters. You can write a char or char[ ] to a file, to a string, to a pipe. Or, to a byte stream with appropriate conversion.

InputStreams are very similar to Readers. The main difference is the former has read byte member functions while the latter has read char member functions.

OutputStreams are very similar to Writers. The former has write byte member functions while the latter has write char member functions.

The subclasses of these 4 have enhanced functionality as indicated by their name (though you will have to look up

the details in the Java documentation). Here are some of the subclasses of each of the 4 classes we have just discussed.

13.3 Subclasses of Input Stream

1) FileInputStream - includes reading bytes from pseudo file devices like keyboard, network sockets, etc.)

2) StringBufferInputStream – read bytes from a StringBuffer in RAM memory.

3) ObjectInputStream - allows you to read whole objects at once from a machine-independent binary source.

4) PipedInputStream – can read from a PipedOutputStream.

5) ByteArrayInputStream – read portion at a time from a byte array in memory.

6) /*abstract*/ FilterInputStream with subclasses:

   a. DataInputStream – allows you to read all the primitives individually from binary, machine-independent format. Note: the readLine( ) function is deprecated and you should use the one in BufferedReader instead.

   b. BufferedInputStream – for efficiency, data is acquired from the underlying source in big hunks. Application reads out bytes from the big buffer as needed, so each read does not requiring going to the underlying source device.

   c. PushPackInputStream

   d. CheckedInputStream – for streams with checksum.

e. various InflatorInputStreams for GZIP, Zip, and Jar compressed file formats.

f. DigestInputStream

g. LineNumberInputStream – counts lines.

13.4 Subclasses of Reader

1) InputStreamReader – a filter that converts a byte input stream to a character stream using a locale and machine-dependent conversion mapping.

    a. FileReader – can be opened by supplying constructor with a file name and a locale and machine-dependent mapping.

2) BufferedReader – reads buffer of input at a time, for efficiency. Also, great for reading a line of input characters (up to line terminator) from another reader.

    a. LineNumberReader

3) PipedReader

4) StringReader

5) CharArrayReader

6) /*abstract*/ FilterReader

    a. PushBackReader

13.5 Connecting Streams and Hydrants

To read, you have to get an object like a fire hose for data, and connect it to a data source like a fire hydrant. The default fire hydrant that is provided by Java is named System.in and is of some subclass of InputStream. The hose you connect to it is an InputStreamReader instance, which is good for converting ASCII keyboard input into Unicode characters needed by your Java program.

To do output, you get a hose of type PrintStream and connect it to whatever output device you want. System.out is a PrintStream instance that is pre-connected for you to the standard output device, usually your command line (i.e. DOS or Unix shell) window. Note that PrintStream has some characteristics of the newer, better PrintWriter, in that both can convert Unicode to ASCII for printers and command line windows. PrintStream has not been abandoned because that would break to many existing Java programs.

To get what you want, you often have to connect a number of hoses together. In fact, Java has the worst input this author has ever seen for reading in a simple integer from the keyboard. Java I/O is very flexible, but consider how many lines it takes to read an integer!


```
InputStreamReader isr;
isr = new InputStreamReader(System.in);
BufferedReader br;
br = new BufferedReader (isr);
String s = br.readLine();
s = s.trim();
int i = Integer.parseInt(s);
```

Isn't that horrible?

Fortunately, the first 4 statements above can be compressed down to 1 statement (which usually spans two lines) as follows:

```
BufferedReader br = new BufferedReader(
    new InputStreamReader(System.in));
```

This takes System.in and cloaks it behind a InputStreamReader converter hose that converts from ASCII to Unicode. This is fed to a BufferedReader that is a kind of hose that reads a whole line at a time. A buffered reader provides a whole line (up until you press the Enter/CarriageReturn key) so your program can use all the digits of a number. BufferedReader is one of the few streams with a very useful readLine( ) member function which will fill a string variable with more than one character.

In addition, the last 3 lines of the above input code can be compressed down to one, as shown here:

```
i = Integer.parseInt(br.readLine().trim());
```

This calls the readLine( ) function of the BufferedReader, which returns a String. The string may have leading and trailing white space (e.g. tabs, or space characters) that will mess up the parsing. So the String instance function named trim( ) is called to remove leading and trailing white space. Then the static Integer function called parseInt( ) is called. It takes a string of Unicode characters like "12345.67" and converts it into an int (stored in just 4 bytes in twos complement form).

Note that once you have a BufferedReader constructed in your program, you don't need another each time you are going to read some data. You construct the reader just once, and then use its readLine( ) member function over and over again.

So perhaps Java input is not as bad as I indicated. But wait! In addition, the readLine( ) and parseInt( ) functions can throw exceptions.

readLine( ) can potentially throw an IOException or subclass of IOException (e.g. EndOfFile, floppy drive empty, etc.). And the parseInt( ) static function can throw a NumberFormatException (e.g. try entering "Mary" when the program is trying to read an integer).

So, either the function this code is within must advertise that it throws these exceptions, or the above code must be within a try block with an appropriate catch clause(s)!

Here is a complete test program for reading integers.

```
//file ReadKeyboard.java

import java.io.*;   //Needed for Readers.

class ReadKeyboard{

  public static void main(String[] args){

    System.out.println("Enter an integer:");

    BufferedReader br = new BufferedReader(
      new InputStreamReader(System.in), 1);

      //the 1 above helps a Win95 bug.

    try {

      String s = (br.readLine()).trim();
      int i = Integer.parseInt(s);

      System.out.println(
        "The integer read was " + i);
    }
    catch(Exception e){
      System.out.println(e.toString());
      System.exit(1);
    }
  }
}
```

Note that the catch(Exception e) clause above will catch either IOException or NumericFormatException. Not only that, but when you print e.toString( ), each different kind of exception will by default print a slightly different message indicating what kind of exception it was. (Unfortunately, the programmer of the parse functions did not put much useful in the contained string of the exception, so e.toString( ) which prints the exception type is more useful than e.getMessage( )). If you wanted to deal with NumericFormatExceptions differently (than IOExceptions), and perhaps ask the user to re-enter their data, then you need separate catch clauses for each exception type.

If you have been using the [Savitch2001] textbook, you might be curious how his SavitchIn class provides introductory students with simplified input:

- SavitchIn.readLineInt( ) trims the string, calls parseInt( ) and catches NumberFormatException. To read its data it uses:
- SavitchIn.readLine( ) that provides much of the functionality of a BufferedReader readLine( ), in that it reads a whole line and discards the line terminator. To read its data it uses:
- SavitchIn.readChar( ) that catches IOException. To read its data it uses:
- System.in.read( ) that reads a byte from an ASCII input device of type InputStream. This particular function expands the byte into a 2 byte Unicode character.

13.5.1 More on Keyboard Reading

The previous section gives a good example of how (awkward it is) to read from the keyboard in a console application. Further details are available in the String sub-section of the course. Recall for example, the functions:

```
Byte.parseByte(String)
Short.parseShort(String)
Integer.parseInt(String)
Long.parseLong(String)
Float.parseFloat(String)
Double.parseDouble(String)
new Boolean(String).booleanValue( )
```

Note that in Java 1.1 and earlier, there was no parseFloat( ) and parseDouble( ). Instead you had to use a round-about method similar to that shown above for booleans.

These conversions from strings to primitives are very important, as Java has no facilities for formatted input such as C's scanf( ) function, or C++'s overloaded input extraction operator.

Unfortunately, the information just mentioned does not explain how to read in primitives from the keyboard when there are multiple primitives on a line. The secret to doing this is to use the StringTokenizer class or a StreamTokenizer reader. They scan their input and break the incoming characters up into 'tokens'. These are available one at a time through the nextToken( ) member function call. The type of the next token (either a number

or non-numeric string is also available. You can specify what characters are to be considered delimiters: spaces, commas, tabs, newlines, etc. Be careful though, as these classes don't seem to read floating-point numbers in scientific notation very well (e.g. 6.02e26).

13.6 Subclasses of OutputStream

1) FileOutputStream - includes writing to pseudo file devices like keyboard, network sockets, etc.)

2) ObjectOutputStream - allows you to write whole objects at once in machine-independent binary.

3) PipedOutputStream – can sink to a PipedInputStream.

4) ByteArrayOutputStream – to write a bit at a time to a byte array in RAM memory.

5) /*abstract*/ FilterOutputStream with subclasses:

   a. DataOutputStream – allows you to write all the primitives individually in a binary, machine-independent format.

   b. BufferedOutputStream – for efficiency writes to a buffer, and when buffer is full only then outputs the whole thing to sink. If you want the data out to the sink immediately, use flush( ).

   c. CheckedInputStream – for streams terminated by checksums.

   d. various DeflatorOutputStreams for GZIP, Zip, and Jar compressed file formats.

   e. DigestOutputStream

   f. PrintStream – this is the underlying type for System.out and System.err. But like the readLine() function in DataInputStream, it does not handle Uncode characters very well. Like that

function, the whole Printstream class has been deprecated. For reasons of backward compatibility, Printstream will still be used for System.out and System.err, but all new code should be written to use PrintWriter.

13.7 Subclasses of Writer

1) OutputStreamWriter – a filter that provides character output functions which are converted to a byte stream using a locale and machine-dependent conversion mapping.

    a. FileWriter – can be opened by supplying constructor with a file name and a locale and machine-dependent mapping.

2) BufferedWriter – allows character writes to be buffered for efficiency. Only if the buffer is full does the data actually get send to the sink device. You can flush( ) it if you need the data output immediately.

3) PrintWriter – a great class for writing any of the primitives to a character stream in human readable form.

4) PipedWriter – for writing characters to a PipedReader.

5) StringWriter – for writing characters a few at a time to compose a String in RAM memory.

6) CharArrayWriter – for writing characters a few at a time to a char[ ] in RAM memory.

7) /*abstract*/ FilterWriter – a base class if you want to design your own writer filter class.

13.8 Which Ones Are Really Sources and Filters

Unfortunately, several Java class which do filtering are not descendents of the abstract filtering classes. In all the many Java I/O classes, it is therefore hard to tell which ones are really hoses connected to some actual sink or source, like a file or network or array. Or, to tell which ones are really just filters. This section contains 4 subsections. The first lists actual sources, the second input filters, the third actual sinks, and the fourth output filters.

13.8.1 Actual Sources

These classes are input hoses that actually connect to some data source (vs. a filter that just modifies already incoming data). To get one, you have to tell the constructor what actual source of data the far end of the hose is connected to (e.g. the name of the file).

```
FileInputStream
RandomAccessFile
URL.getInputStream or
URLconnection.getInputStream.
ByteArrayInputStream
PipedInputStream
StringBufferInputStream

FileReader
CharArrayReader
PipedReader
StringReader
```

13.8.2 Input Filters

This section lists filters; hoses which connect to other hoses rather than to an actual data source.

```
DataInputStream
BufferedInputStream
ObjectInputStream
SequencedInputStream
CheckedInputStream
DigestInputStream
various InflatorImputSteam subclasses
LineNumberInputStream
PushbackInputStream

InputStreamReader
BufferedReader
CharArrayReader
LineNumberReader
PushBackReader
/*abstract*/ FilterReader
StreamTokenizer
```


13.8.3 Actual Sinks

These classes provide actual sinks for data. During their construction, you have to specify the actual data sink that the downstream end of the hose is connected to (e.g. file name).

```
FileOutputStream
RandomAccessFile
URLconnection.getOutputStream()
PipedOutputStream
ByteArrayOutputStream

FileWriter
StringWriter
CharArrayWriter
PipedWriter
```

13.8.4 Output Filters

Not all of the filters are descendents of the filtering classes. This sub-section lists those output classes that provide a hose that is ***not*** connected to an actual destination sink, but instead just to another hose which goes eventually to a sink.

```
DataOutputStream
BufferedOutputStream
ObjectOutputStream
PrintStream (deprecated)
CheckedOutputStream
DigestOutputStream
various DeflatorOutputStream subclasses

OutputStreamWriter
BufferedWriter
PrintWriter
/*abstract*/ FilterWriter
```


13.9 File I/O

It would be nice to have a whole section here on file I/O. However there is not time and there are other complications.

There are two kinds of file I/O:

- Text I/O
- Binary and Random Access I/O

13.9.1 Text I/O

When the compiler is reading your programs, it is doing input text I/O. Text I/O means two things:

1) You want basically to read characters. Perhaps you want to read numbers too, but they are written in either ASCII or Unicode representation. If you are reading numbers written in ASCII or Unicode, you might want to call certain functions that will read the next string of characters, and hoping they are digits, convert them into an internal form like int or float.
2) You are dealing with lines of text, each terminated with a line termination character. The line termination you get when pressing the Enter/CarriageReturn key is slightly different on an IBM PC vs. MacIntosh vs. Unix machine. However, Java mostly hides the difference. The point here though is that in addition to reading single characters, you may also use readLine( ) functions to read as many characters as there are in the next line.

Many other kinds of applications use text I/O. Notice that there is usually conversion. Either:

- ASCII to Unicode.
- Internal to external representation of numbers
- Line terminators removed, or modified to be compatible with the brand of OS (Mac, PC, Unix).

13.9.2 Binary I/O

Binary I/O is basically writing or reading the bytes exactly as they are, without conversion. The concept of a line terminator disappears as it is just another byte(s) going in or out. So there is no concept of reading a 'line'. To specify how many bytes to read or write, you basically tell the function the byte count, or tell it to read an float (which is knows is 4 bytes long in internal format.

On reason not to do format conversion in binary I/O is that it is silly to convert an int from twos complement internal format, to human readable digit characters on disk or network, only to later read them back in and have to convert they back into the internal form. First, this slows the computer down, and second it takes extra space on disk to store the characters '1' '2' '8' '6' '4' '3' '.' '9' instead of just 4 bytes.

13.9.3 Random-Access I/O

Many times when you are using binary I/O you might also be using Random-Access I/O. Random access I/O permits you to seek( ) to various places in a file without having to read half the file to get to the middle of the file. This would be ridiculous in this modern day of disks (c.f. tape storage).

To move to the beginning of 1000th byte, you just call seek(999). The bytes are numbered starting at zero. You can also consider this to mean skip the first 999 bytes.

What is even more interesting is that you are allowed to write into the middle of a binary file. This is unlike text files which, for historical reasons related to tape drive hardware, do not allow you to write into the middle of a file without messing up the rest of the file beyond the write point.

Most databases are implemented using binary, random-access files containing fixed length records. To instantly read the 57th record you just do:

```
seek(56 * recordLength);
```


Java has a class called RandomAccessFile for doing binary random-access I/O. The RandomAccessFile class supports these functions from the so-called DataOutput interface.

Method	Explanation
`writeBoolean( boolean v )`	Writes the boolean 'v' as a byte in binary format.
`writeByte( int v )`	Writes 'v' as a one-byte value in binary format.
`writeChar( int v )`	Writes 'v' as a two-byte binary unicode character.
`writeDouble( double v )`	Writes 'v' as an 8-byte binary floating-point number.
`writeFloat( float v )`	Writes 'v' as a 4-byte binary floating-point number.
`writeInt( int v )`	Writes 'v' as a 4-byte binary signed integer.
`writeLong( long v )`	Writes 'v' as an 8-byte binary signed integer.
`writeShort( int v )`	Writes 'v' as a 2-byte binary signed integer.
`writeBytes( String s )`	Writes 's' as a sequence of bytes (high 8 bits of each character in 's' are discarded) in binary format.
`writeChars( String s )`	Writes 's' as a sequence of Java unicode characters in binary format.
`writeUTF( String s )`	Write 's' as a string using binary UTF-8 coding.

RandomAccessFile class also provide the DataInput interface functions:

Method	Explanation
`boolean readBoolean()`	Reads a boolean from the file.
`byte readByte()`	Reads a signed byte from the file.
`int readUnsignedByte()`	Reads an unsigned byte from the file.
`short readShort()`	Reads a signed 16-bit number from the file.
`int readUnsignedShort()`	Reads an unsigned 16-bit number from the file.
`char readChar()`	Reads a 16-bit Unicode character from the file.
`int readInt()`	Reads a signed 32-bit integer from the file.
`long readLong()`	Reads a signed 64-bit integer from the file.
`float readFloat()`	Reads a float (32-bit) from the file.
`double readDouble()`	Reads a double (64-bit) from the file.
`String readLine()`	Reads a line of text terminated by a newline character ('\n'), or end-of-file.

Note that Java has terrible support for writing arrays of characters or arrays of bytes, even though these are common in other programming languages.

However, Java has great support for writing to existing databases using a protocol called Structured Query Language (SQL). It also has interesting support for writing whole objects (including variable length strings referred to by the instance attributes) to a file. However, since these records are not all the same length, because the strings might be different lengths from object to object of the same class, you cannot use the instance random-access abilities of Java with such Object files.

13.10 Summary

If you are writing primitives out into human readable form, use the print( ) and println( ) functions in PrintWriter. PrintWriter (and the deprecated PrintStream) provide different print functions that convert from binary to String for each different primitive type. Also, as you have seen throughout the course, they are great for concatenating strings and primitives together for output (as the primitives will all be converted to strings using the primitive's toString( ) method). You can also use most class's toString( ) member functions (e.g. Integer.toString( )) to convert internal binary numbers to Strings or char[ ] for display on graphical screens.

If you are doing binary output (i.e. no conversion) of primitives, use either RandomAccessFile or a DataOutputStream.

If you are reading primitives without conversion in from a binary input, use DataInputStream or RandomAccessFile.

Finally, if you are reading in primitives from the keyboard, read strings using readLine( ) from a BufferReader sourcing from an InputStreamReader. Alternatively, get the basic string from GUI text field objects. You then have to parse them (e.g. Integer.parseInt( )), or use a tokenizer (e.g. StringTokenizer or StreamTokenizer). Parsing or tokenizing converts them from human readable strings into the appropriate binary format for each primitive.