

---

## CSC 143 Java

### Streams

Reading: 16

---

## Overview

- Topics
  - Data representation – bits and bytes
  - Streams – communicating with the outside world
  - Basic Java files
  - Other stream classes

---

## GREAT IDEAS IN COMPUTER SCIENCE

### REPRESENTATION VS. RENDERING

---

## Data Representation

- Underneath it's all bits (binary digits – 0/1)
- Byte – group of 8 binary digits
  - Smallest addressable unit of memory
- Meaning depends on interpretation
  - Non-negative base-10 integers represented as base-2 integers
  - Characters formats include ASCII (1 byte) or Unicode (2 byte) encodings
    - 01000001 = integer 65 = ASCII 'A'
    - Unicode 'A' is 0000000001000001
    - 00111111 = integer 63 = ASCII '?'
    - 00110110 = integer 54 = ASCII '6'
- But it's still just bits

---

## Representation of Primitive Java Types

- Boolean – 1 byte (0 = false; 1 = true)
- Integer types
  - byte – 1 byte (-128 to 127)
  - short – 2 bytes (-32768 to 32767)
  - int – 4 bytes (-2147483648 to 2147483647)
  - long – 8 bytes (-9223372036854775808 to 9223372036854775807)
- Floating-point (real number) types
  - float – 4 bytes; approx. 6 decimal digits precision
  - double – 8 bytes; approx. 15 decimal digits precision
- Character type
  - char – 2 bytes; **Unicode** characters w/decimal values 0 to 65535

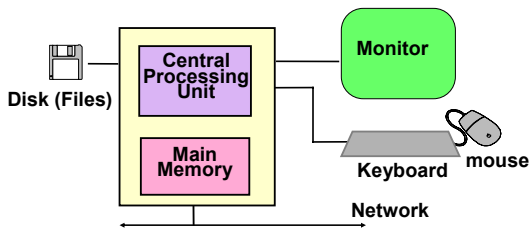
---

## Unicode

- International standard
  - Java was first major language to adopt
- Intended to include all the world's writing systems
- Characters are 2 bytes (16 bits)
  - Given by two Hex digits, e.g. 4EB9
- Specifications: [www.unicode.org](http://www.unicode.org)
- Unicode 3.1 (2001) introduced characters outside the original 16 bit range

## Input and Output

- Communicating with the outside world

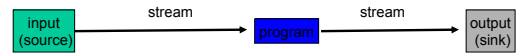


(c) University of Washington

12-7

## Streams

- Java model of communication: streams
  - Sequence of data flowing from a source to a program, or from a program to a destination (sink)
  - Files are common sources and sinks

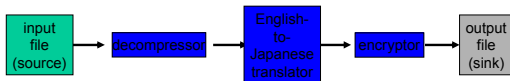


(c) University of Washington

12-8

## Stream after Stream...

- Stream are a useful model for processing data along the way, in a pipeline



(c) University of Washington

12-9

## Streams vs. Files

- Many languages don't make clear distinction
  - Programmers, too!
- In Java:
  - "file" is the collection of data, managed by the operating system
  - "stream" is a flow of data from one place to another
- It's possible for a stream to flow from or to from... URL, remote computer, hardware device, etc.

(c) University of Washington

12-10

## Java Stream Library

- Huge variety of stream classes in java.io.\*
  - Some are data sources or sinks
  - Others are converters that take data from a stream and transform it somehow to produce a stream with different characteristics
- Highly modular
  - Lots of different implementations all sharing a common interface; can be mixed and matched and chained easily
  - Great OO design example, in principle
  - In practice, it can be very confusing

(c) University of Washington

12-11

## Common Stream Processing Pattern

- Basic idea the same for input & output

```
// input
open a stream
while more data {
    read & process next data
}
close stream

// output
open a stream
while more data {
    write data to stream
}
close stream
```

(c) University of Washington

12-12

## Opening & Closing Streams

- Before a stream can be used it must be *opened*
  - Create a stream object and connect it to source or destination of the stream data
  - Often done implicitly as part of creating stream objects
- When we're done with a stream, it should be *closed*
  - Cleans up any unfinished operations, then breaks the connection between the program and the data source/destination

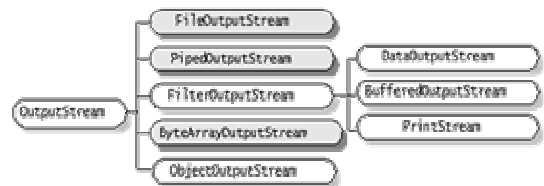
## Java Streams

- 2 major families of stream classes, based on the type of data
- **Byte streams** – read/write `byte` values
  - Corresponds to physical data – network and disk I/O streams
  - Abstract classes: `InputStream` and `OutputStream`
- **Character streams** – read/write `char` values
  - Added in Java 1.1
  - Primary (Unicode) text input/output stream classes
  - Abstract classes: `Reader` and `Writer`
- `System.out` should be a character stream... is it??

## Character Input Streams



## Byte Output Streams



## Streams and Exceptions

- All operations can throw `IOException`
- Normally throws a specific subclass of `IOException`
  - depending on the actual error
- `IOException` is "checked" – what does this imply?

## Basic Reader/Writer Operations

- Reader

```
int read(); // return Unicode value of next character; -1 if end-of-stream
int read(char[] cbuf); // read several characters into array; return -1 if end-of-stream
void close(); // close the stream
```
- Writer

```
void write(int c); // write character whose Unicode value is c
void write(char[] cbuf); // write array contents
void write(String s); // write string
void close(); // close the stream
```
- To convert Unicode `int` to `char`, or vice versa: use cast syntax

## File Readers and Writers

- To read a (Unicode) text file (not a binary data file), instantiate `FileReader`
  - A subclass of `Reader`: implements read and close operations
  - Constructor takes the name of the file to open and read from
- To write to a text file, instantiate `FileWriter`
  - A subclass of `Writer`: implements write and close operations
  - Constructor takes the name of the file to open/create and overwrite (can also append to an existing file using a different constructor)

## Text Files vs Char Data

- Most of the world's text files use 8  $\text{bit}$  characters
  - ASCII and variations of ASCII
  - Internal to Java, char data is *always* 2-byte Unicode
  - Java Reader deals only with Unicode
- Big problem: how to read and write normal (ASCII) text files in Java?
- Solution: stream classes which adapt 8  $\text{bit}$  chars to Unicode

## Copy a Text File, One Character at a Time

```
public void copyFile(String sourceFilename, String destFilename)
    throws IOException {
    FileReader inFile = new FileReader(sourceFilename);
    FileWriter outFile = new FileWriter(destFilename);
    int ch;
    while ( (ch = inFile.read()) != -1 ) {
        outFile.write(ch);
        System.out.println("The next char is \"" + (char)ch + "\""); // why ' ' ?
    }
    inFile.close();
    outFile.close();
}
```

## More Efficient I/O – BufferedReader/Writer

- Can improve efficiency by reading/writing many characters at a time
- `BufferedReader`: a converter stream that performs this chunking
  - `BufferedReader` constructor takes any kind of `Reader` as an argument -- can make any read stream buffered
  - `BufferedReader` supports standard `Reader` operations -- clients don't have to change to benefit from buffering
  - Also supports `readLine()`
    - `String readLine();` // read an entire line of input; or null if end-of-stream reached
    - [handles the complexities of how end-of-line is represented on different systems]
- `BufferedWriter`: a converter stream that performs chunking on writes
  - `BufferedWriter` constructor takes any kind of `Writer` as an argument
  - `BufferedWriter` supports standard `Writer` operations
  - Also supports `newLine()`
    - `void newLine();` // write an end-of-line character

## Copy a Text File, One Line at a Time

```
public void copyFile(String sourceFilename, String destFilename)
    throws IOException {
    BufferedReader inFile = new BufferedReader( new FileReader(sourceFilename));
    BufferedWriter outFile = new BufferedWriter( new FileWriter(destFilename));
    String line;
    while ( (line = inFile.readLine()) != null ) {
        outFile.write(line);
        outFile.newLine();
        System.out.println("The next line is \"" + line + "\"");
    }
    inFile.close();
    outFile.close();
}
```

## PrintWriter

- `PrintWriter` is another converter for a write stream
  - Adds `print` & `println` methods for primitive types, strings, objects, etc., just as we've used for `System.out`
  - Does not throw exceptions (to make it more convenient to use)
  - Optional 2<sup>nd</sup> boolean parameter in constructor to request output be flushed (force all output to actually appear) after each `println`
    - Useful for interactive consoles where messages need to appear right away

## Copy a Text File, Using PrintWriter

```
public void copyFile(String srcFilename, String destFilename)
    throws IOException {
    BufferedReader inFile = new BufferedReader(new FileReader(srcFilename));
    PrintWriter outFile =
        new PrintWriter( new BufferedWriter( new FileWriter(destFilename)));
    String line;
    while ( (line = inFile.readLine()) != null) {
        outFile.println(line);
        System.out.println("The next line is '" + line + "'");
    }
    inFile.close();
    outFile.close();
}
```

(c) University of Washington

12-25

## StringReader and StringWriter

- StringReader: convert from a String to a character stream  
`StringReader inStream = new StringReader("the source");`  
*// could now write inStream to a file, or somewhere else*
- StringWriter: convert from a stream to a String  
`StringWriter outStream = new StringWriter();`  
*// now write onto outStream, using outStream.write(...), outStream.print(...), etc.*  
`String theResult = outStream.toString();`

(c) University of Washington

12-26

## Binary Streams

- For processing binary data (encoded characters, executable programs, other low level data), use InputStreams and OutputStreams
- Operations are similar to Reader and Writer operations
  - Replace char with byte in read; no write(String)
- Many analogous classes to Readers and Writers:
  - FileInputStream, FileOutputStream
  - BufferedInputStream, BufferedOutputStream
  - ByteArrayInputStream, ByteArrayOutputStream
  - ObjectInputStream, ObjectOutputStream -- read & write whole objects!

(c) University of Washington

12-27

## Conversion from Binary to Text Streams

- InputStreamReader: creates a Reader from an InputStream  
*// System.in is of type InputStream*  
`Reader inStream = new InputStreamReader(System.in);`  
*// now can treat it nicely as a character stream*
- OutputStreamWriter: creates a Writer from an OutputStream  
*// System.out is of type OutputStream*  
`Writer outStream = new OutputStreamWriter(System.out);`  
*// now can treat it nicely as a character stream*

(c) University of Washington

12-28

## Network Streams

- Import java.net.\*
- Use URL to create a name of something on the web
- Use `openStream()` method to get a InputStream on the contents of the URL  
`URL url = new URL("http://www.cs.washington.edu/index.html");`  
`InputStream inStream = url.openStream();`  
*// now read from inStream*
- Use `openConnection()` and `URLConnection` methods to get more control  
`URLConnection connection = url.openConnection();`  
`OutputStream outStream = connection.getOutputStream();`  
*// now write to outStream (assuming target url allows writing!)*
- `Socket` class for even more flexible network reading & writing

(c) University of Washington

12-29

## Other Possible Kinds of Stream Converters

- Compression
- Encryption
- Filtering
- Translation
- Statistics gathering
- Security monitoring
- Routing/Merging
- Reducing Bandwidth (Size & Detail), e.g. of graphics or sound
  - "lossy compression"
- Noise reduction, image sharpening, ...
- Many, many more...

(c) University of Washington

12-30