To Ponder

Consider:
String s = “Hello World”;

☐ How much space does s occupy in memory?
☐ How much space does s (probably) occupy when written to disk?

File IO

Lecture 21
I/O Package Overview

- Package java.io
- Core concept: streams
  - Ordered sequences of data that have a source (for input) or a destination (for output)
  - Can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays
  - Support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects
- See Java API documentation for details

Input Streams

- A program uses an input stream to read data from a source, one item at a time
Output Streams

- A program uses an *output* stream to write data to a destination, one item at a time

Types Of Streams

- Two major flavors:
  - Byte streams
    - 8 bits at a time, data-based (binary) information
    - Input streams and output streams
  - Character streams
    - 16 bits at a time, text-based information
    - Readers and writers
Byte Streams

- Two abstract base classes: InputStream and OutputStream
  - InputStream (for reading bytes) defines:
    - An abstract method for reading 1 byte at a time
      ```java
      public abstract int read()
      ```
    - Returns next byte value (0-255) or -1 if end-of-stream encountered
    - Concrete input stream overrides this method to provide useful functionality
  - Methods to read an array of bytes or skip a number of bytes

- OutputStream (for writing bytes) defines:
  - An abstract method for writing 1 byte at a time
    ```java
    public abstract void write(int b)
    ```
    - Upper 24 bits are ignored
  - Methods to write bytes from a specified byte array
  - Close the stream after reading/writing
    ```java
    public void close()
    ```
    - Frees up limited operating system resources
  - All of these methods can throw IOException

(Partial) Byte Stream Hierarchy
Example 1: Measuring File Size

```java
import java.io.*;
class CountBytes {
    public static void main(String[] args)
        throws IOException {
        InputStream in = new FileInputStream(args[0]);
        int total = 0;
        while (in.read() != -1) {
            total++;
        }
        in.close();
        System.out.println(total + " bytes");
    }
}
```

Standard Streams

- Three standard streams for console IO
  - System.in
    - Input from keyboard
  - System.out
    - Output to console
  - System.err
    - Output to error (console by default)

- These streams are byte streams!
  - System.in is an InputStream, the others are PrintStreams (extend OutputStream)
  - *Should* be character streams not byte streams, but they predate the inclusion of character streams in Java
Example 2: Console Streams

```java
import java.io.*;

class TranslateBytes {
    public static void main(String[] args) throws IOException {
        byte from = (byte)args[0].charAt(0);
        byte to = (byte)args[1].charAt(0);
        int x;
        while((x = System.in.read()) != -1) {
            System.out.write(x == from ? to : x);
        }
    }
}

If you run "java TranslateBytes b B" and enter text bigboy via the keyboard the output will be: BigBoy
```

Character Streams

- Two abstract base classes: Reader and Writer
- Similar methods to byte stream counterparts
- Reader abstract class defines:
  - public int read()
  - Returns value in range 0..65535 (or -1)
  - public int read(char[] cbuf)
  - Returns number of characters read
  - public void skip(int n)
- Writer abstract class defines:
  - public void write(int c)
  - public void write(char[] cbuf)
  - public abstract void flush()
  - Ensures previous writes have been sent to destination
  - Useful for buffered streams
- Both classes define:
  - public void close()
Example 3: File Streams

```java
import java.io.*;
public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inStream = null;
        FileWriter outStream = null;
        try {
            inStream = new FileReader("input.txt");
            outStream = new FileWriter("output.txt");
            int c;
            while ((c = inStream.read()) != -1) {
                outStream.write(c);
            }
        } finally {
            if (inStream != null) { inStream.close(); }
            if (outStream != null) { outStream.close(); }
        }
    }
}
```
Converting Byte/Character Streams

- Conversion streams: InputStreamReader and OutputStreamWriter
  - Subclasses of Reader and Writer respectively
- InputStreamReader
  - public InputStreamReader(InputStream in)
  - public InputStreamReader(InputStream in, String encoding)
  - An encoding is a standard map of characters to bits (e.g., UTF-16)
  - public int read()
  - Reads bytes from associated InputStream and converts them to characters using the appropriate encoding for that stream
- OutputStreamWriter
  - public OutputStreamWriter(OutputStream out)
  - public OutputStreamWriter(OutputStream out, String enc)
  - public void write(int c)
  - Converts argument to bytes using the appropriate encoding and writes these bytes to its associated OutputStream
- Closing the conversion stream also closes the associated byte stream—may not always be desirable

Efficient IO

- Buffering greatly improves IO performance
- Example: BufferedReader for character input streams
  - public BufferedReader(Reader in)
  - The buffered stream “wraps” the unbuffered stream
- Example declarations of BufferedReaders
  - An InputStreamReader inside a BufferedReader
    Reader r = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(r);
  - A FileReader inside a BufferedReader
    Reader fr = new FileReader("fileName");
    BufferedReader in = new BufferedReader(fr);
  - Then you can invoke in.readLine() to read from the stream line by line
Example 4: Buffered File Streams

import java.io.*;
public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inStream = null;
        PrintWriter outStream = null;
        try {
            inStream = new BufferedReader(new FileReader("input.txt"));
            outStream = new PrintWriter(
                    new BufferedWriter(new FileWriter("output.txt")));
            String line;
            while ((line = inStream.readLine()) != null) {
                outStream.println(line);
            }
        } finally {
            if (inStream != null) { inStream.close(); }
            if (outStream != null) { outStream.close(); }
        }
    }
}

The File Class

- Useful for retrieving information about a file or a directory
  - Represents a path, not necessarily an underlying file
  - Does not open/close files or provide file-processing capabilities
- Three constructors
  - public File(String name)
  - public File(String pathToName, String name)
  - public File(File directory, String name)
- Main methods
  - boolean canRead() / boolean canWrite()
  - boolean exists()
  - boolean isFile() / boolean isDirectory()
  - String getAbsolutePath() / String getPath()
  - String getParent()
  - String getName()
  - long length()
  - long lastModified()
Working with Files

- A file can be identified in one of three ways
  - A String object (file name)
  - A File object
  - A FileDescriptor object
- Sequential-Access file: read/write at end of stream only
  - FileInputStream, FileOutputStream, FileReader, FileWriter
  - Each file stream type has three constructors
- Random-Access file: read/write at a specified location
  - RandomAccessFile
  - A file pointer is used to guide the starting position
    - seek(pos), getFilePointer()
  - Not a subclass of any of the four basic IO classes
    (InputStream, OutputStream, Reader, or Writer)
  - Supports both input and output
  - Supports both bytes and characters

Example 5: A Random Access File

```java
public static void main(String args[]) {
    RandomAccessFile fh1 = null;
    RandomAccessFile fh2 = null;
    try {
        fh1 = new RandomAccessFile(args[0], "r");
        fh2 = new RandomAccessFile(args[1], "rw");
    } catch (FileNotFoundException e) { . . . }
    try {
        int bufsize = (int) (fh1.length())/2;
        byte[] buffer = new byte[bufsize];
        fh1.seek(bufsize); // set file pointer to middle of file
        fh1.readFully(buffer, 0, bufsize); //read half of file
        fh2.write(buffer, 0, bufsize); //write all of array
    } catch (IOException e) {
        . . .
    }
}
```
java.util.Scanner

- A simple text scanner which can parse primitive types and strings using regular expressions
- A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace
- The resulting tokens may then be converted into values of different types using the various `next` methods:
  - `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, etc.
  - `nextLine()`
  - `nextBigInteger()`, `nextBigDecimal()`
- Can check for token existence with various `hasNext` methods:
  - `hasNextInt()`, `hasNextLong()`, `hasNextDouble()`, etc.
  - `hasNext()`
  - `hasNextBigInteger()`, `hasNextBigDecimal()`

Example 6: Text Console Input

```java
import java.util.Scanner;

public class AddNumbers {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int sum = 0;
        while (input.hasNextInt()) {
            int x = input.nextInt();
            sum += x;
        }
        System.out.println(sum);
    }
}
```
Example 7: Text File Input

```java
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class DumpFile {
    public static void main(String[] args) throws IOException {
        File file = new File("input.txt");
        Scanner input = new Scanner(file);
        String line = input.nextLine();
        while (line != null) {
            System.out.println(line);
            line = input.nextLine();
        }
        input.close();
    }
}
```

Supplemental Reading

- Java Tutorial "Basic I/O" trail
  - download.oracle.com/javase/tutorial/essential/io/
- Java APIs
  - java.io package
  - java.util.Scanner
Summary

- Metaphor: Streams
  - Use of checked exceptions
  - Remember to close a stream when done
- Two flavors
  - Data (byte): InputStream & OutputStream
  - Text (character): Reader & Writer
- Wrapping streams
  - For converting byte/character
  - For efficiency with buffering
- java.util.Scanner