

---

# Java Input / Output

CSE 413, Autumn 2002  
Programming Languages

<http://www.cs.washington.edu/education/courses/413/02au/>

# Readings and References

---

- Reading
  - » Chapter 12, Sections: Streams, The Complete Stream Zoo, Putting Streams to Use, File Management, *CoreJava, Volume 1, Fundamentals*, by Horstmann and Cornell
- Other References
  - » Section "I/O" of the Java tutorial
  - » <http://java.sun.com/docs/books/tutorial/essential/io/index.html>

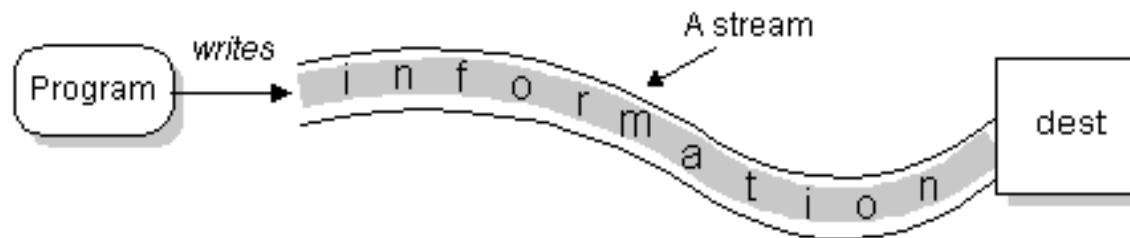
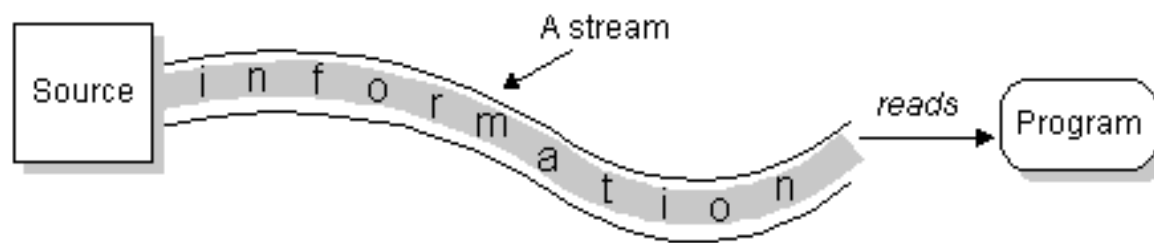
# Input & Output

---

- Program input can come from a variety of places:
  - » the mouse, keyboard, disk, network...
- Program output can go to a variety of places:
  - » the screen, speakers, disk, network, printer...

# "Streams" are the basic I/O objects

keyboard,  
disk file,  
network,  
etc



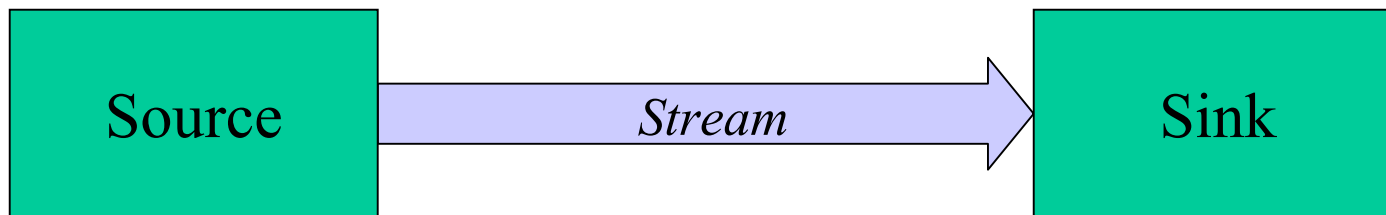
display,  
disk file,  
network,  
etc

from Sun tutorial on I/O

# The stream model

---

- The stream model views all data as coming from a source and going to a sink



- Sources and sinks can be files, memory, the console, network ports, serial ports, etc

# Streams

---

- Getting data from source to sink is the job of an object of a *stream* class
- Use different streams for doing different jobs
- Streams appear in many packages
  - » java.io - basic stream functionality, files
  - » java.net - network sockets
  - » javax.comm - serial ports
  - » java.util.zip - zip files

# Streams are *layered* classes

---

- Inheritance and composition both play key roles in defining the various types of streams
- Each layer adds a little bit of functionality
- The nice thing about this design is that many programs don't need to know exactly what kind of stream they are working with
  - » one `OutputStream` is as good as another in many situations, as long as it knows how to move bytes

# OutputStream

---

- An OutputStream sends bytes to a sink
  - » OutputStream is an abstract class
  - » the actual "write" method depends on the device being written to
- Key methods:

```
abstract void write(int b) throws IOException
void write(byte[] b) throws IOException
void close() throws IOException
```



# OutputStream subclasses

---

- Subclasses differ in how they implement write() and in what kind of sink they deal with:
  - » **FileOutputStream**: sink is a file on disk
  - » **ByteArrayOutputStream**: sink is an array of bytes
  - » **PipedOutputStream**: sink is a pipe to another thread
- Other subclasses process output streams
  - » **FilterOutputStream**: process the stream in transit
  - » **ObjectOutputStream**: primitives and objects to a sink

# FilterOutputStream

---

- Constructor takes an instance of OutputStream
- Resulting object is also instance of OutputStream
- These classes *decorate* the basic OutputStream implementations with extra functionality
- Subclasses of FilterOutputStream in java.io:
  - » BufferedOutputStream: adds buffering for efficiency
  - » PrintStream: supports display of data in text form (using the default encoding only)
  - » DataOutputStream: write primitive data types and Strings (in binary form)

# InputStream

---

- An InputStream gets bytes from a source
  - » InputStream is an abstract class
  - » The actual "read" method depends on the source being read from
  - » Key methods:

```
abstract int read() throws IOException
int read(byte[] b) throws IOException
void close() throws IOException
```

# InputStream subclasses

---

- Subclasses differ in how they implement read() and in what kind of source they deal with:
  - » **FileInputStream**: source is a file on disk
  - » **ByteArrayInputStream**: source is an array of byte
  - » **PipedInputStream**: source is pipe from another thread
- Other subclasses process input streams
  - » **FilterInputStream**: process the stream in transit
  - » **ObjectInputStream**: primitives and objects from a source

# FilterInputStream

---

- Constructor takes an instance of `InputStream`
- Resulting object is also instance of `InputStream`
- These classes “decorate” the basic `InputStream` implementations with extra functionality
- Some useful subclasses
  - » `BufferedInputStream`: adds buffering for efficiency
  - » `ZipInputStream`: read zip files
  - » `DataInputStream`: read primitive data types and Strings (in binary form)

# Reader and Writer

---

- Reader and Writer are abstract classes that are **Unicode** aware and can use a specified encoding to translate Unicode to/from bytes
- Subclasses implement most of the functionality
  - » InputStreamReader, OutputStreamWriter  
rely on the underlying streams to actually move bytes
  - » BufferedReader, BufferedWriter  
add buffering for efficiency
  - » StringReader, StringWriter
  - » PipedReader, PipedWriter

# Reader and Writer guidelines

---

- In general:
  - » If you're working with text (Strings and chars), use Readers and Writers
  - » If you're working with primitive data types, use InputStreams and OutputStreams
  - » If you get an InputStream or OutputStream from somewhere else, you can convert it to a Reader or a Writer as needed by wrapping it with an InputStreamReader or OutputStreamWriter

# System.in, System.out

---

- System.in is a predefined InputStream
- You can convert to a BufferedReader like this:

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

- System.out is a predefined OutputStream
  - » actually, it's a PrintStream
- You can convert to a PrintWriter like this:

```
PrintWriter w =  
    new PrintWriter(new OutputStreamWriter(System.out), true);
```



# Read a String from the console

---

```
/* ask for the names we were not given */

BufferedReader console =
    new BufferedReader(new InputStreamReader(System.in));

for (int i=count; i<3; i++) {
    System.out.print("name "+i+"? ");
    String petName = console.readLine();
    if (petName == null) {
        petName = "<blank>";
    }
    names.add(petName);
}
```

# Sources and Sinks - Console

---

- When reading from the console
  - » the keyboard is the source
  - » a data structure in your application is the sink
  
- When writing to the console
  - » a data structure in your application is the source
  - » the monitor (terminal window) is the sink

# Sources and Sinks - Files

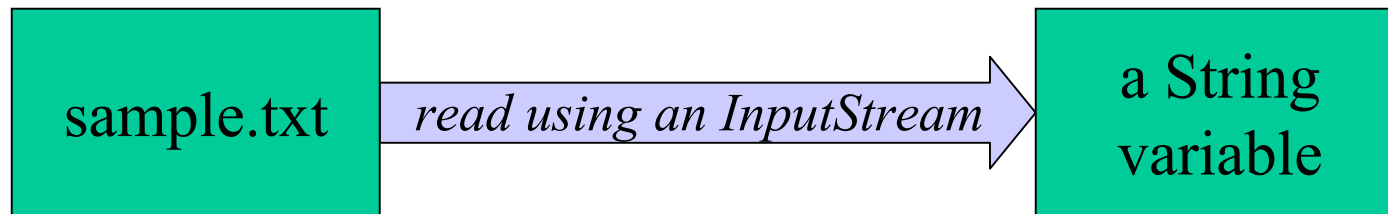
---

- When reading from a file
  - » the file is the source
  - » a data structure in your application is the sink
- When writing to a file
  - » a data structure in your application is the source
  - » the file is the sink

# The stream model applied to files

---

- The source can be a file on disk
  - » in this case, the sink is some variable in your program



# FileInputStream and FileOutputStream

---

- The file streams read or write from a file on the native file system
  - » FileInputStream
    - retrieve bytes from a file and provide them to the program
  - » FileOutputStream
    - send bytes to a file from your program
- If used by themselves, FileInputStream and FileOutputStream are for binary I/O
  - » just plain bytes in and out with no interpretation as characters or anything else

# FileInputStream methods

---

## **int available()**

Returns the number of bytes that can be read from this file input stream without blocking.

## **void close()**

Closes this file input stream and releases any system resources associated with the stream.

## **protected void finalize()**

Ensures that the close method of this file input stream is called when there are no more references to it.

## **FileDescriptor getFD()**

Returns the FileDescriptor object that represents the connection to the actual file in the file system being used by this FileInputStream.

## **int read()**

Reads a byte of data from this input stream.

## **int read(byte[] b)**

Reads up to b.length bytes of data from this input stream into an array of bytes.

## **int read(byte[] b, int off, int len)**

Reads up to len bytes of data from this input stream into an array of bytes.

## **long skip(long n)**

Skips over and discards n bytes of data from the input stream.

## **void mark(int readlimit)**

Marks the current position in this input stream.

## **boolean markSupported()**

Tests if this input stream supports the mark and reset methods.

## **void reset()**

Repositions this stream to the position at the time the mark method was last called on this input stream.

# "bytes from a file" and "bytes as text"

---

- Create new `FileInputStream` and connect it to a specific file
- "decorate" the stream with an `InputStreamReader` that will do Unicode translation for you

## `FileInputStream(String name)`

Create a `FileInputStream` by opening a connection to an actual file, the file named by the path name in the file system.

## `InputStreamReader(InputStream in)`

Create an `InputStreamReader` that uses the default character encoding.

## `InputStreamReader(InputStream in, String enc)`

Create an `InputStreamReader` that uses the named character encoding.

# "bytes from a file as text"

---

- Create new `FileReader` and connect it to a file
  - » `FileReader` is a convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an `InputStreamReader` on a `FileInputStream`.

`FileReader(File file)`

Creates a new `FileReader`, given the `File` to read from.

`FileReader(FileDescriptor fd)`

Creates a new `FileReader`, given the `FileDescriptor` to read from.

`FileReader(String fileName)`

Creates a new `FileReader`, given the name of the file to read from.



# prepare to read a file

---

<sup>1</sup>  
open an InputStream connected to the filename provided

```
public TextRead(String fn) throws IOException {  
    InputStream in;  
    in = new FileInputStream(fn);  
    textReader = new BufferedReader(new InputStreamReader(in));  
}
```


<sup>3</sup>  
add buffering capability so that we  
can read an entire line at once

<sup>2</sup>  
make it a Reader so that we  
get valid Unicode characters

# BufferedReader constructor from Sun

---

```
/**
 * Create a buffering character-input stream that uses an input buffer of
 * the specified size.
 *
 * @param in    A Reader
 * @param sz    Input-buffer size
 * @exception IllegalArgumentException  If sz is <= 0
 */
public BufferedReader(Reader in, int sz) {
    super(in);
    if (sz <= 0)
        throw new IllegalArgumentException("Buffer size <= 0");
    this.in = in;
    cb = new char[sz];
    nextChar = nChars = 0;
}
```



the buffer is allocated here as an array of characters

# readline()

---

- Read one line from a `BufferedReader`
  - » returns a `String` containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

```
/**
 * Read one line from the text file and return it as a String to the caller.
 * Note that the line might be null (at end of file), empty (0 characters) or
 * blank (all whitespace). Of course, it might also be a non-blank String with
 * some useful characters in it.
 * @return a String containing the next line or null if
 * we are at the end of the file
 */
private String getNextLine() throws IOException {
    return textReader.readLine();
}
```

# Detecting end of file

---

- End of file is expected when using `readline()`
  - » you will eventually read all the characters in a file
- So the method returns **null** if we are end of file
  - » you must check for null after every `readline()` call

```
String myLine = tr.getNextLine();
while (myLine != null) {
    System.out.println(">> "+myLine);
    myLine = tr.getNextLine();
}
```

# close when done

---

- After reading through the file, you should close the stream, since an open file takes up system resources and prevents other programs from using the file

```
/**
 * Close the stream.
 */
public void close() throws IOException {
    textReader.close();
}
```

# "bytes to a file as text"

---

- Create new `PrintWriter` and connect it to a file using a `FileWriter`
  - » `PrintWriter` provides the text formatting capabilities
  - » `FileWriter` provides the connection between the `PrintWriter` and the actual file
  - » `FileWriter` is a convenience class like `FileReader`
    - could use `OutputStreamWriter` with a `FileOutputStream`

`PrintWriter(Writer out)`

Create a new `PrintWriter`, without automatic line flushing.

`FileWriter(String fileName)`

Constructs a `FileWriter` object given a file name.

# prepare to write a file

1  
create a new file with the name given to us for writing

```
public TextRW(String fn) throws IOException {  
    File sink = new File(fn);  
    sink.createNewFile();  
    System.out.println("Created "+sink.getAbsolutePath());  
    textWriter = new PrintWriter(new BufferedWriter(new FileWriter(sink)));  
}
```

4  
add formatting so that  
Java can convert values  
to character strings for us

3  
add buffering for  
efficiency

2  
open the file as a  
Writer so Unicode  
works correctly

# println(...)

---

- Print formatted representations of objects and primitive type to a text-output stream
  - » does not contain methods for writing raw bytes, for which a program should use unencoded byte streams

```
/**
 * Write one line on the output file.
 * @param line the line of text to write out
 */
public void writeOneLine(String s) {
    textWriter.println(s);
}
```



# close when done

---

- After writing the file, you should close the stream
  - » the last data that you have written may not actually have gotten all the way out to the disk - closing makes sure that the data is **flushed to disk**
  - » an open file takes up system resources and prevents other programs from using the file

```
/**
 * Close the stream.
 */
public void close() throws IOException {
    textWriter.close();
}
```

# The File class

---

- Manages an entry in a directory (a pathname)
- Several constructors are available
  - » `File(String pathname)`  
pathname string
  - » `File(String parent, String child)`  
parent pathname string and a child pathname string.
  - » `File(File parent, String child)`  
parent abstract pathname and a child pathname string.
- The `File()` constructors create a pathname object in memory, NOT a new file on disk

# File class examples

---

```
File f = new File("c:\autoexec.bat");
```

```
File app = new File("c:\apps\JPadPro", "JPadPro.exe");
```

```
File jppDir = new File("c:\apps\JPadPro");
```

```
File jppApp = new File(jppDir, "JPadPro.exe");
```

- Creating a new File object just creates a tool for managing files, it does not create a new file on disk!
  - » Creating a new Dog object did not create a new dog running around the room ...

# File class methods

---

- Create, rename, delete a file
  - » `createNewFile()`, `createTempFile()`, `renameTo()`, `delete()`
- Determine whether a file exists and access limitations
  - » `exists()`, `canRead()`, `canWrite()`
- Get file info
  - » `getParent()`, `getCanonicalPath()`, `length()`, `lastModified()`
- Create and get directory info
  - » `mkdirs()`, `list()`, `listFiles()`, `getParent()`
- Etc, etc

# Appendix

---

# Writing output to the console

---

- Java provides standard PrintStream System.out
  - » has methods to print text to the console window
- Some operations:
  - System.out.println( <expression> );
  - System.out.print( <expression> );
- expression can be
  - » primitive type: an int, double, char, boolean
  - » or an object of any class type

# Printing primitives on System.out

---

- System.out is a PrintStream object
- PrintStream defines a whole bunch of print(...) methods, one for each type

```
void print(boolean b)
void print(char c)
void print(char[] s)
void print(double d)
void print(float f)
void print(int i)
void print(long l)
```

```
void print(Object obj)
void print(String s)
```

# Printing objects on System.out

---

- Any object can be printed on System.out

```
Rectangle rect = new
    Rectangle(30,50,100,150,Color.blue,true);
System.out.println(rect);
```
- Can be very useful for debugging
  - » Put System.out.print or println method calls in your code to display a message when that place is reached during execution
  - » Particularly useful if the string version of the object has useful information in a readable format



# Object Representation on System.out

---

- What actually happens when an object is printed?
  - » The toString() method belonging to the object provides the string to be printed
  - » All classes have a default toString( ), the one defined by the Object class (not very descriptive)

```
public String toString() {  
    return getClass().getName()+"@"+Integer.toHexString(hashCode());  
}
```

- » But you can provide a custom version of toString() in any of your classes very easily