CISC 370: I/O Streams

Sara Sprenkle

June 20, 2006

Review

- Static Methods (Assignment 1)
- Inheritance
- Exceptions
- Scanner

Quiz

Streams

- Java handles input/output using streams

input stream: an object from which we can read a sequence of bytes

Abstract class: InputStream

Streams

- Java handles input/output using streams

output stream: an object to which we can write a sequence of bytes

Abstract class: OutputStream

Streams Basics

- Java handles input/output using streams
  - MANY (80+) types of Java streams
  - In java.io package
- Why streams?
  - information stored in different source (e.g., in a file, on a web server across the network, a string) is accessed in essentially the same way
  - allows the same methods to read or write data, regardless of its source
    - create an InputStream or OutputStream of the appropriate type

Stream Basics

- Streams are automatically opened when created
- Close a stream by calling its close() method
  - close a stream as soon as program is done with it
  - free up system resources

Reading & Writing Bytes

- The InputStream class has an abstract method read() that reads one byte from the stream and returns it.
- Concrete input stream classes override read() to provide the appropriate functionality
  - e.g., **FileInputStream** class: read() reads one byte from a *file*
- Similarly, OutputStream class has an abstract method write() to write a byte to the stream

Reading & Writing Bytes

- read() and write() are blocking operations
  - If a byte cannot be read from the stream, the method waits (does not return) until a byte is read
- isAvailable() allows you to check the number of bytes that are available for reading before you call read()

```
int bytesAvailable = System.in.isAvailable();
if (bytesAvailable > 0)
      System.in.read(byteBuffer);
```


More Powerful Stream Objects

- Reading and writing bytes is very time-consuming and code-intensive
- **DataInputStream** class
  - directly reads Java primitive types through method calls such as readDouble(), readChar(), readBoolean()
- **DataOutputStream** class
  - directly writes all of the Java primitive types with writeDouble(), writeChar(), …

File Input and Output Streams

- **FileInputStream**: provides an input stream that can read from a disk file
  - Constructor takes the name of the file:

```
FileInputStream fin = new
          FileInputStream("chicken.data");
```

  - Or, uses a **File** object …

```
File inputFile = new File("chicken.data");
FileInputStream fin = new FileInputStream(inputFile);
```

Copy.java

Filtered Streams

- **FileInputStream** has no methods to read numeric types
- **DataInputStream** has no methods to read from a file
- Java allows you to combine these two types of streams into a connected stream

Filtered Streams

- Subclasses of FilterInputStream or FilterOutputStream
- Communicate with another stream
- Add functionality
  - Automatically buffered IO
  - Automatic compression
  - Automatic encryption
  - Automatic conversion between objects and bytes
- As opposed to Data source streams
  - communicate with a data source
    - file, byte array, network socket, or URL

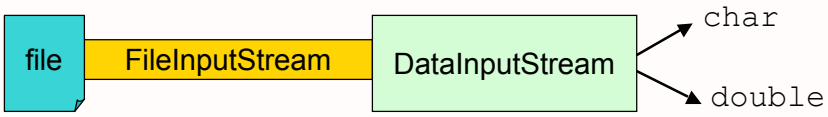
Filtered Streams: Reading from a file

- If we wanted to read numbers from a file
  - Create a FileInputStream to read the bytes from the file
  - Create a DataInputStream to handle numeric type reading
- Connect the DataInputStream to the FileInputStream
  - FileInputStream gets the bytes from the file and the DataInputStream reads them as assembled types

```
FileInputStream fin = new
          FileInputStream("chicken.data");
DataInputStream din = new
          DataInputStream(fin);  "wrap" fin in din
double num1 = din.readDouble();
```

Connected Streams

- Think of a stream as a “pipe”
- FileInputStream knows how to read from a file
- DataInputStream knows how to read an InputStream into useful types
- Connect the **out** end of the FileInputStream to the **in** end of the DataInputStream…



DataIODemo.java

Aside: String Buffers vs Strings

- Strings are “read-only” or immutable
- Use StringBuffer to manipulate a String
- Instead of
  - String str = prevStr + “ more!”;
  - Which creates a new string
- Use
  - StringBuffer str = new StringBuffer(prevStr);
  - str.append(“ more!”);
- Many StringBuffer methods, including toString() to get the resultant string back

Buffered Streams

- Use a **BufferedInputStream** class object
  - to buffer your input streams
- A pipe in the chain that adds buffering

```
DataInputStream din = new DataInputStream (
   new BufferedInputStream (
      new FileInputStream("chicken.data")));
```


A More Connected Stream

- FileInputStream reads bytes from the file
- BufferedInputStream buffers bytes
  - speeds up access to the file.
- DataInputStream reads buffered bytes as types

Connected Streams

- Combine the many different types of streams to can get the functionality you want
- Similar for output
  - For buffered output to the file and to write types
    - create a FileOutputStream
    - attach a BufferedOutputStream
    - attach a DataOutputStream
    - perform the typed writing using the methods of the DataOutputStream object

Text Streams

- We have seen streams that operate on binary data, not text.
- Text streams are somewhat complicated because Java uses Unicode to represent characters/strings and some operating systems do not
  - we need something that will convert the characters from Unicode to whatever encoding the underlying operating system uses, as they are written out.

Text Streams

- Writing text can be accomplished using classes derived from **Reader** and **Writer**
  - The terms Reader and Writer generally refer to text I/O in Java).
- make an input reader that will read from the keyboard
  - Create an object of type **InputStreamReader**

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

  - in will read characters from the keyboard and convert them into Unicode for Java

Text Streams and Encodings

- You can also attach an InputStreamReader to a FileInputStream to read from a text file…

```
InputStreamReader in = new InputStreamReader(
      new FileInputStream("employee.data"));
```

  - assumes the file has been encoded in the default encoding of the underlying operating system
  - You can specify a different encoding by listing it in the constructor of the InputStreamReader…

```
InputStreamReader in = new InputStreamReader(
      new FileInputStream("employee.data"), "ASCII");
```

Convenience Classes

- Reading and writing to text files is common
- Two convenience classes that combine a InputStreamReader with a FileInputStream and similarly for output of text file.
- For example,

```
FileWriter out = new FileWriter("output.txt");
```

is equivalent to

```
OutputStreamWriter out = new OutputStreamWriter(
     new FileOutputStream("output.txt"));
```


PrintWriters

- When writing text output, use **PrintWriter**
  - Easiest writer to use
  - Very similar to a DataOutputStream
  - No destination
- Combine a PrintWriter with a destination writer, such as a FileWriter

```
PrintWriter out = new PrintWriter(
     new FileWriter("output.txt"));
```

- To write strings using a PrintWriter, use the same print() and println() methods you use with System.out to display strings

PrintWriters

- Write data to the destination writer
  - in this case the file…

```
PrintWriter out = new PrintWriter(new
      FileWriter("output.txt"));

String myName = "Sara 'InMyDreams' Sprenkle";
double mySalary = 325000;

out.print(myName);
out.print(" makes ");
out.print(salary);
out.println(" per year.");
      or
out.println(myName + " makes " + salary +
            " per year.");
```


PrintWriters and Buffering

- PrintWriters are always buffered
- You can set the writer to autoflush mode
  - causes any writes to be executed directly on the target destination (in effect defeating the purpose of the buffering).
  - constructor with second parameter set to true

```
// create an autoflushing PrintWriter
PrintWriter out = new PrintWriter(
      new FileWriter("output.txt"), true);
```

Formatted Output

- **PrintStream** has new functionality in Java 1.5
  - printf()

```
double f1=3.14159, f2=1.45, total=9.43;
// simple formatting...
System.out.printf("%6.5f and %5.2f", f1, f2);
// getting fancy (%n = \n or \r\n)...
System.out.printf("%-6s%5.2f%n", "Tax:", total);
```

- Can make formatted output easy
  - before 1.5, required java.util.Formatter objects to generate the string to be passed to System.out.println()

Reading Text from a Stream

- There is no PrintReader class
- Use a BufferedReader
  - call readLine()
    - reads in a line of text and return it as a String
    - returns null when no more input is available

Reading Text from a Stream

- Make a BufferedReader…

```
BufferedReader in = new BufferedReader(
      new FileReader("inputfile.txt"));
```

- Read the file, line-by-line…

```
String line;
while ((line = in.readLine()) != null)
{
      // process the line
}
```


Reading Text from a Stream

- You can also attach a BufferedReader directly to an InputStreamReader…

```
BufferedReader in2 = new BufferedReader(
      new InputStreamReader(System.in));
BufferedReader in3 = new BufferedReader(
      new InputStreamReader(url.openStream());
```

Scanners

- The best way to read from the console used to be this combination of a BufferedReader and InputStreamReader wrapped around System.in
- Scanner: new to 1.5
  - We used these last time

```
Scanner sc = new Scanner(System.in);
```


Using Scanners

- Use *nextXXX()* to read from it…

```
long tempLong;

// create the scanner for the console
Scanner sc = new Scanner(System.in);

// read in an integer and a string
int i = sc.nextInt();
String restOfLine = sc.nextLine();

read in a bunch of long integers
while (sc.hasNextLong())
{ tempLong = sc.nextLong(); }
```

Scanner Details I

- Scanners do not throw IOExceptions!
- For a simple console program, main() does not have to deal with or throw IOExceptions
  - required with the BufferedReader/InputStreamReader combination
- A Scanner breaks its input into tokens using a delimiter pattern, which matches whitespace
  - What is “delimiter pattern”?
  - What is “whitespace”?
- Resulting tokens are converted into values of different types using *nextXXX()*

Scanner Details II

- A Scanner throws an InputMismatchException when the token does not match the pattern for the expected type
  - e.g., nextLong() called with the next token equal to “AAA”
  - RuntimeException (no catching required)
- You can change the token delimiter from the default of whitespace
- Scanners assume numbers are input as decimal
  - Can specify a different radix
- Scanners are for Java 1.5 and up only!

Writing and Reading Objects

- So now we know how to write and read primitive types and text to and from places
- What about objects?
- To save object data, you need to create an **ObjectOutputStream** object

```
ObjectOutputStream out = new ObjectOutputStream(
      FileOutputStream("chicken.data"));
```


Writing Objects

- To save objects to the file, call writeObject() on ObjectOutputStream…

```
Chicken baby = new Chicken("Baby", 10, 2.8);
Rooster foghorn = new Rooster("Foghorn", 38, 5.8);

out.writeObject(baby);
out.writeObject(foghorn);
```

Reading Objects

- To read objects, create an **ObjectInputStream** object and call readObject()

```
ObjectInputStream in = new ObjectInputStream(
     new FileInputStream("Chickens.data"));

Chicken c1 = (Chicken)in.readObject();
Chicken c2 = (Chicken)in.readObject();
```


Reading Objects

- readObject() reads an object and returns it as an Object
  - Make sure you cast it to the appropriate class
- Use getClass() to dynamically determine the class of the object you just read in
  - Where is getClass() defined?
- When reading objects back in, you must carefully keep track of how many you saved, their order, and their type

Object Serialization

- Serialization: process of converting an object to ordered data, to operate with streams
- To allow a class of objects to be written and read with Object[Output/Input]Stream
  - the class must implement the **Serializable** interface
- Serializable interface contains no methods
  - "Marker interface"
  - used to tag a class as able to be serialized
  - refers to the class's ability to be converted into a single byte stream, which is then saved
- All classes are inherently serializable
  - But you have to mark them as Serializable

Object Serialization

- When an object is written to a stream, it is written as a sequence of bytes.
- Stores, in order
  - fingerprinting information that describes the class
  - instance fields and their values
  - a more complete class description.
- The ability to convert an object from its "object" state to a byte stream state is what makes the direct writing and reading of objects possible

Using Serialization

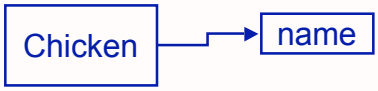
- Light-weight persistence
  - Archive an object for use in later invocation of same program
- Remote Method Invocation (RMI)
  - Communication between objects via sockets

Object References and Serialization

- When an object is serialized and saved to disk, any instance fields that are references to other objects need to be handled.
  - For example, a Chicken object has a String name
- If the serialization process stored the object variable (the pointer), this would not work correctly when the object was reloaded at a later point
  - the object is not guaranteed to exist in memory at that point any longer (or at all!)

Object References and Serialization

- Java will serialize all referred objects in an object being serialized
  - allows the complete reconstruction of the state of that object when it is read back in



- Recursive process
  - any objects referred to inside an object being serialized will also be serialized
    - any objects referred to inside those objects will be serialized
      - and so forth.

Object References and Serialization

- What happens if one object is referred to by two different objects that are both serialized?
  - for the Chicken class, add an instance field that is an object variable to a Farmer class object, representing that owner of the farm
- Since more than one chicken can share a farmer, both chicken1 and chicken2 could refer to the same farmer object

- Does farmer get serialized and saved twice?

Object References and Serialization

- No, it does not get serialized more than once
- When an object is serialized and written to the disk, Java assigns it a serial number.
- If that same object is referred to again
  - Java simply stores the serial number of the first time it serialized it
  - Does not serialize again because it is the same object!

Object Serialization

- Serialize Chicken1
  - Points to Farmer, so serialize Farmer
- When Chicken2 is serialized
  - Serialize chicken 2 but …
  - Points to object with serial number XX001

Object Serialization

- Serialization is completely automatic
- Implement the Serializable interface and store your objects to disk using an ObjectOutputStream and read them using an ObjectInputStream

Using Serialization for Cloning

- The complete state of an object can be converted into a byte stream
  - allows serialization to be used as an alternate approach to object cloning
- For a class to use serialization to clone itself
  - implement the Cloneable interface
  - Inside clone()
    - serialize the object to a byte stream
    - read that same byte stream back in and store it as the clone

Using Serialization for Cloning

- You do not have to write the serialized version of the object to a file
  - use a **ByteArrayOutputStream** to save it in memory as a byte array
  - read it using a **ByteArrayInputStream** object
- You can get the generated byte array by calling output stream object's toByteArray() method

Using Serialization for Cloning

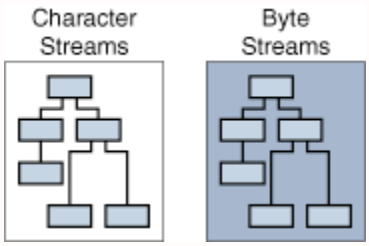
```
public Object clone()
{
   try {
      ByteArrayOutputStream bout =
         new ByteArrayOutputStream();
      ObjectOutputStream out = new ObjectOutputStream(bout);
      out.writeObject(this);
      out.close();
      ByteArrayInputStream bin = new ByteArrayInputStream(
         bout.toByteArray());
      ObjectInputStream in = new ObjectInputStream(bin);
      Object cloned = in.readObject();
      in.close();
      return cloned;
   } catch (Exception e) {
      return null;
   }
}
```

Externalizable Interface (FYI)

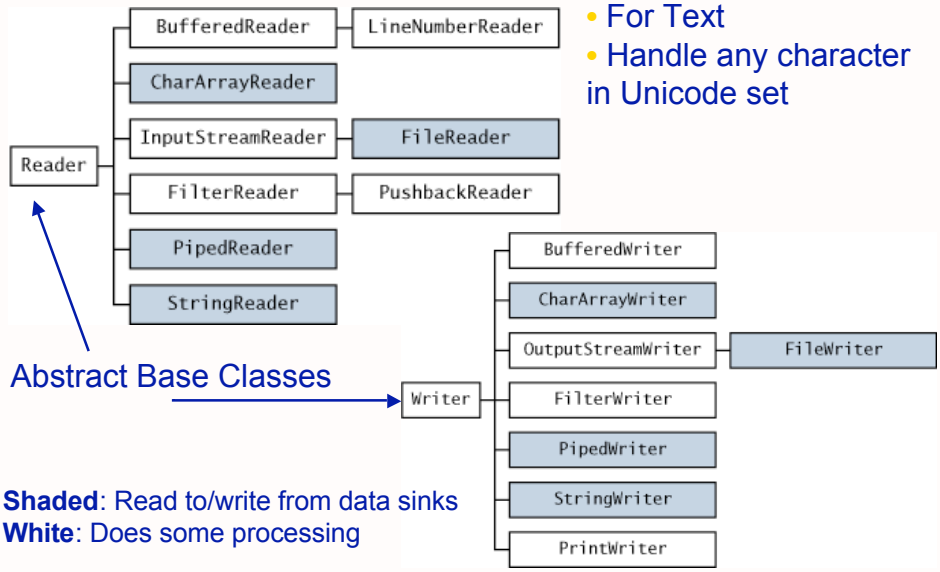
- To control data serialization, implement the Externalizable interface
- void readExternal(ObjectInput in)
- void writeExternal(ObjectOutput out)

java.io Classes Overview

- Two types of stream classes
  - Based on datatype: Character, Byte



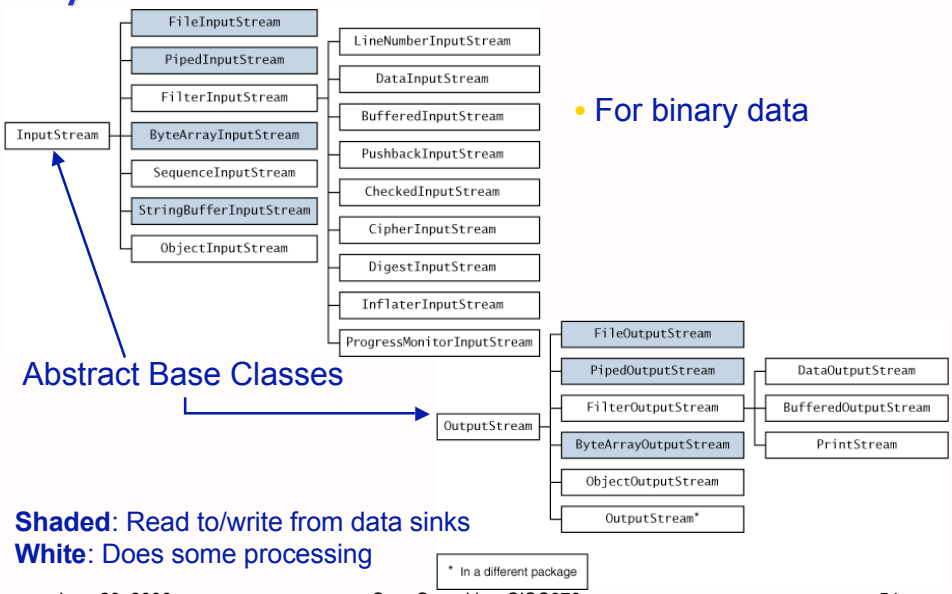
Character Streams



- For Text
- Handle any character in Unicode set

Shaded: Read to/write from data sinks
White: Does some processing

Byte Streams



- For binary data

Shaded: Read to/write from data sinks
White: Does some processing

Readers and Input Streams

Similar APIs for different data types

characters

- Reader
  - int read()
  - int read(char cbuf[])
  - int read(char cbuf[], int offset, int length)

bytes

- InputStream
  - int read()
  - int read(byte cbuf[])
  - int read(byte cbuf[], int offset, int length)

Writers, Output Streams are similarly parallel

java.nio.*

- Additional classes for I/O
  - scalable I/O
  - fast buffered byte and character I/O
  - character set conversion
- Designed for performance tuning

A little bit about Files

- More to file management than input and output
- Stream classes deal with with contents of a file
- **File** class deals with file functionality
  - the actual storage of the file on the disk
  - determine if a particular file exists
  - when file was last modified
  - Rename file
  - Remove/delete file

Making a File Object

- The simplest constructor for a File object simply takes the full file name (including the path)
  - If you do not supply a path, Java assumes the current directory

```
File f1 = new File("chicken.data");
```

- Creates a File object representing a file named "chicken.data" in the current directory…

Making a File Object

- Does **not** create a file with this name on disk
  - creates a File object that represents a file with that pathname on the disk, even if file does not exist
- File's exists() method
  - Determines if a file exists on the disk
  - Create a File object that represents file and call exists() method.

Other File Constructors

- a String for the path and a String for the filename…

```
File f2 = new File(
   "/home/sprenks/datafiles","chicken.data");
```

- a File object representing the directory

```
File f3 = new File("/home/sprenks/datafiles");
```

- Plus a String representing the filename

```
File f4 = new File(f3, "chicken.data");
```

"Break" any of Java's Principles?

Not Portable

- Accessing the file system is inherently not portable
  - In Windows, paths are "c:\\dir"
  - In Unix, paths are "/home/sprenks/dir"
- Relies on underlying file system/operating system to perform actions

Handling Portability Issues

- Fields in File class
  - static separator
    - Unix: "/"
    - Windows: "\\"
  - static pathSeparator
    - For separating a list of paths
    - Unix: ":"
    - Windows: ";"
- Use relative paths, with separators

Why two \\?

Files and Directories

- A File object can represent a file **or** a directory
  - directories are special files in most modern operating systems
- Use isDirectory() and/or isFile() to see what type of file is abstracted in the File object

The File Class

- 25+ methods of the File class
  - manipulate files and directories
  - creating and removing directories
  - making, renaming, and deleting files
  - Information about file (size, last modified)
  - Creating temporary files
  - …
- see online API documentation