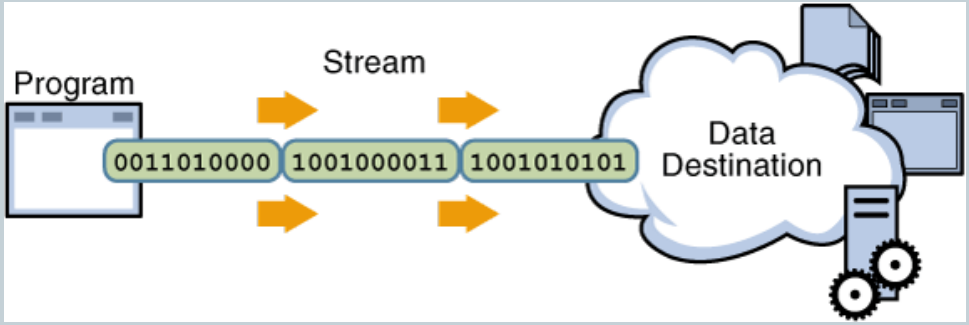
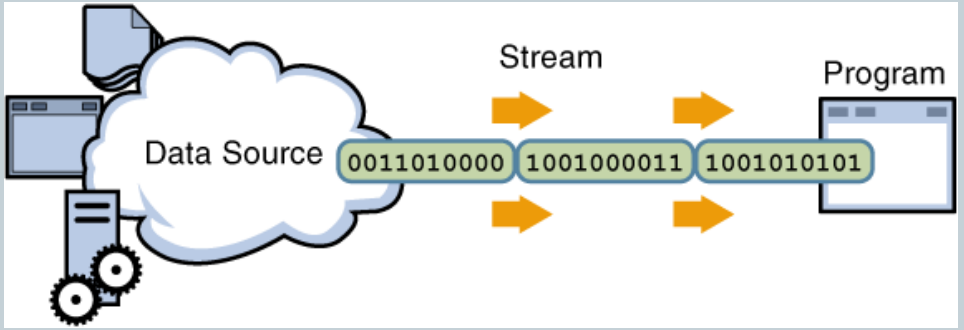
Streams and File I/O



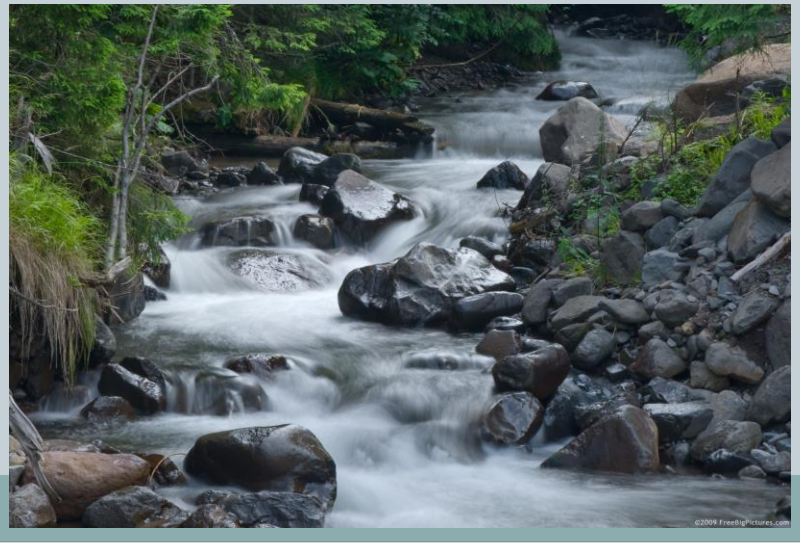


Outline

- Overview of Streams and File I/O
  - Buffering
- Text File I/O
- Binary File I/O

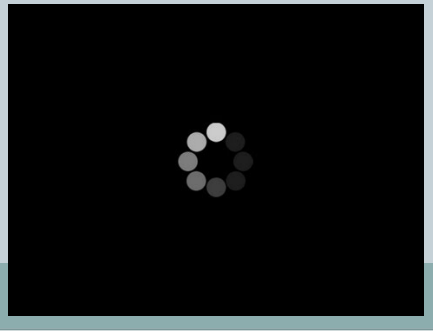
Streams

- ***Stream***: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
  - A stream connects a program to an I/O object
- ***Input stream***: a stream that provides input to a program
  - `System.in` is an input stream
- ***Output stream***: a stream that accepts output from a program
  - `System.out` is an output stream

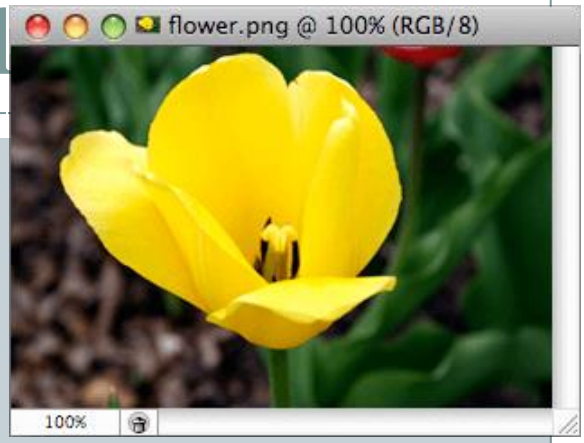


Buffering

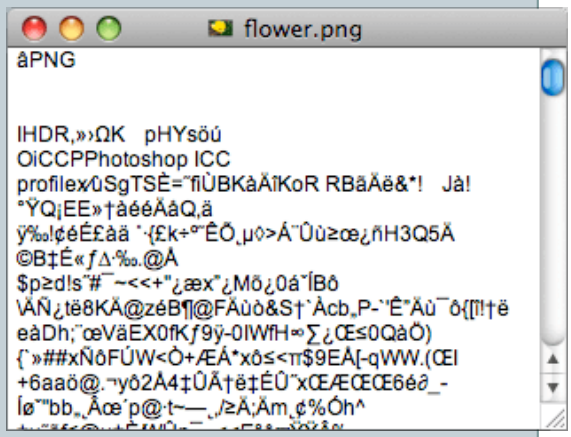
- Not buffered: each byte is read/written from/to disk as soon as possible
  - "little" delay for each byte
  - A disk operation per byte - higher overhead
- Buffered: reading/writing in "chunks"
  - Some delay for some bytes
    - Assume 16-byte buffers
    - Reading: access the first 4 bytes, need to wait for all 16 bytes are read from disk to memory
    - Writing: save the first 4 bytes, need to wait for all 16 bytes before writing from memory to disk
  - One disk operation per buffer of bytes---lower overhead



Binary Versus Text Fil



- *All* data and programs are ultimately just zeros and ones
  - each digit can have one of two values, hence *binary*
  - *bit* is one binary digit
  - *byte* is a group of eight bits
- *Text files*: the bits represent printable characters
  - one byte per character for ASCII, the most common code
  - for example, Java source files are text files
  - so is any file created with a "text editor"
- *Binary files*: the bits represent other types of encoded information, such as executable instructions or numeric data
  - these files are easily read by the computer but not humans
  - they are *not* "printable" files
    - actually, you *can* print them, but they will be unintelligible
    - "printable" means "easily readable by humans when printed"



Java: Text Versus Binary Files

- Text files are more readable by humans
- Binary files are more efficient
  - computers read and write binary files more easily than text
- Java binary files are portable
  - they can be used by Java on different machines
  - reading and writing binary files is normally done by a program
  - text files are used only to communicate with humans

Java Text Files

- Source files
- Occasionally input files
- Occasionally output files

Java Binary Files

- Executable files (created by compiling source files)
- Usually input files
- Usually output files

Text Files vs. Binary Files

- Number: 127 (decimal)
  - Text file
    - Three bytes: “1”, “2”, “7”
    - ASCII (decimal): 49, 50, 55
    - ASCII (octal): 61, 62, 67
    - ASCII (binary): 00110001, 00110010, 00110111
  - Binary file:
    - One byte (`byte`): 01111111
    - Two bytes (`short`): 00000000 01111111
    - Four bytes (`int`): 00000000 00000000 00000000 01111111

Text File I/O

- Important classes for text file **output** (to the file)
  - `PrintWriter`
  - `FileOutputStream` [or `FileWriter`]
- Important classes for text file **input** (from the file):
  - `BufferedReader`
  - `FileReader`
- **`FileOutputStream`** and **`FileReader`** take file names as arguments.
- **`PrintWriter`** and **`BufferedReader`** provide useful methods for easier writing and reading.
- Usually need a combination of two classes
- To use these classes your program needs a line like the following:

```
import java.io.*;
```

Text File Output

- To open a text file for output: connect a text file to a stream for writing

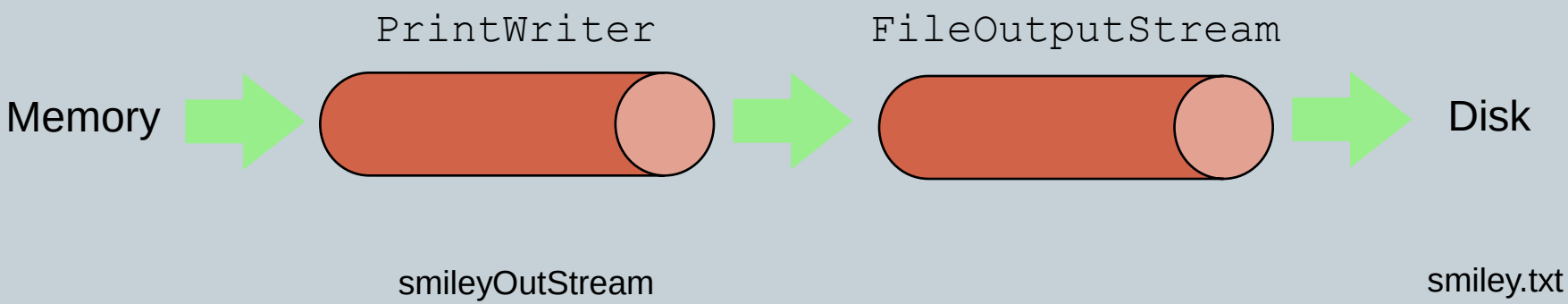
```
PrintWriter outputStream =
  new PrintWriter(new FileOutputStream("out.txt"));
```

- Similar to the long way:

```
FileOutputStream s = new FileOutputStream("out.txt");
PrintWriter outputStream = new PrintWriter(s);
```

- Goal: create a `PrintWriter` object
  - which uses `FileOutputStream` to open a text file
- `FileOutputStream` "connects" `PrintWriter` to a text file.

Output File Streams




```
PrintWriter smileyOutStream = new PrintWriter( new FileOutputStream("smiley.txt") );
```

Methods for `PrintWriter`

- Similar to methods for `System.out`
- `println`

```
outputStream.println(count + " " + line);
```

- `print`
- `format`
- `flush`: write buffered output to disk
- `close`: close the `PrintWriter` stream (and file)

Text File Output Demo

```
public static void main(String[] args)
{
   PrintWriter outputStream = null;
   try
   {
       outputStream =
            new PrintWriter(new FileOutputStream("out.txt"));
   }
   catch(FileNotFoundException e)
   {
       System.out.println("Error opening the file out.txt. "
                          + e.getMessage());
       System.exit(0);
   }
System.out.println("Enter three lines of text:");
String line = null;
int count;
   for (count = 1; count <= 3; count++)
   {
       line = keyboard.nextLine();
       outputStream.println(count + " " + line);
   }
   outputStream.close();
   System.out.println("... written to out.txt.");
}
```

Gotcha: Overwriting a File

- Opening an output file creates an empty file
  - creates a new file if it does not already exist
  - opening an output file that already exists eliminates the old file and creates a new, empty one
    - data in the original file is lost
  - can also append to a file (next slide)

Appending to a Text File

- To add/append to a file instead of replacing it, use a different constructor for **`FileOutputStream`**:

```
outputStream =
   new PrintWriter(new FileOutputStream("out.txt", true));
```

- Second parameter: append to the end of the file if it exists?
- Sample code for letting user tell whether to replace or append:

```
System.out.println("A for append or N for new file:");
char ans = keyboard.next().charAt(0);
boolean append = (ans == 'A' || ans == 'a');
outputStream = new PrintWriter(
      new FileOutputStream("out.txt", append));
```

true if user enters 'A'

Closing a File

- An output file should be closed when you are done writing to it (and an input file should be closed when you are done reading from it).
- Use the `close` method of the class `PrintWriter` (`BufferedReader` also has a `close` method).
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open.

Basic Binary File I/O

- Important classes for binary file **output** (to the file)
  - `ObjectOutputStream`
  - `FileOutputStream`
- Important classes for binary file **input** (from the file):
  - `ObjectInputStream`
  - `FileInputStream`
- Note that `FileOutputStream` and `FileInputStream` are used only for their constructors, which can take file names as arguments.
  - `ObjectOutputStream` and `ObjectInputStream` cannot take file names as arguments for their constructors.
- To use these classes your program needs a line like the following:

```
import java.io.*;
```

Java File I/O: Stream Classes

- `ObjectInputStream` and `ObjectOutputStream`:
  - have methods to either read or write data one byte at a time
  - automatically convert numbers and characters into binary
    - binary-encoded numeric files (files with numbers) are not readable by a text editor, but store data more efficiently
- Remember:
  - *input* means data into a program, not the file
  - similarly, *output* means data out of a program, not the file

Using `ObjectOutputStream` to Output Data to Files:

- The output files are binary and can store any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type
  - You can store reference types – we'll talk about that later in the semester
- The files created can be read by other Java programs but are not printable
- The Java I/O library must be imported by including the line: `import java.io.*;`
  - it contains `ObjectOutputStream` and other useful class definitions
- An `IOException` might be thrown

Example: Opening an Output File

To open a file named `numbers.dat`:

```
ObjectOutputStream outputStream =
  new ObjectOutputStream(
  new FileOutputStream("numbers.dat"));
```

- The constructor for `ObjectOutputStream` requires a `FileOutputStream` argument
- The constructor for `FileOutputStream` requires a `String` argument
  - the `String` argument is the output file name
- The following two statements are equivalent to the single statement above:

```
FileOutputStream middleman =
  new FileOutputStream("numbers.dat");
ObjectOutputStream outputStream =
  new ObjectOutputSteam(middleman);
```

Some `ObjectOutputStream` Methods

- You can write data to an output file after it is connected to a stream class
  - Use methods defined in `ObjectOutputStream`
    - `writeInt(int n)`
    - `writeDouble(double x)`
    - `writeBoolean(boolean b)`
    - etc.
- Note that each write method throws `IOException`
  - eventually we will have to write a catch block for it
- Also note that each write method includes the modifier `final`
  - `final` methods cannot be redefined in derived classes

Closing a File

- An Output file should be closed when you are done writing to it
- Use the `close` method of the class `ObjectOutputStream`
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open

Writing a Character to a File: an Unexpected Little Complexity

- The method `writeChar` has an annoying property:
  - it takes an `int`, not a `char`, argument
- But it is easy to fix:
  - just cast the character to an int
- For example, to write the character 'A' to the file opened previously:

```
outputStream.writeChar((int) 'A');
```

- Or, just use the automatic conversion from `char` to `int`

Writing Strings to a File: Another Little Unexpected Complexity

- Use the `writeUTF` method to output a value of type `String`
  - there is no `writeString` method
- UTF stands for Unicode Text Format
  - a special version of Unicode
- Unicode: a text (printable) code that uses 2 bytes per character
  - designed to accommodate languages with a different alphabet or no alphabet (such as Chinese and Japanese)
- ASCII: also a text (printable) code, but it uses just 1 byte per character
  - the most common code for English and languages with a similar alphabet
- UTF is a modification of Unicode that uses just one byte for ASCII characters
  - allows other languages without sacrificing efficiency for ASCII files

Using `ObjectInputStream` to Read Data from Files:

- Input files are binary and contain any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type
- The files can be read by Java programs but are not printable
- The Java I/O library must be imported including the line:
  `import java.io.*;`
  - it contains `ObjectInputStream` and other useful class definitions
- An `IOException` might be thrown

Opening a New Input File

- Similar to opening an output file, but replace "output" with "input"
- The file name is given as a `String`
  - file name rules are determined by your operating system
- Opening a file takes two steps
  1. Creating a `FileInputStream` object associated with the file name `String`
  2. Connecting the `FileInputStream` to an `ObjectInputStream` object
- This can be done in one line of code

Example: Opening an Input File

To open a file named `numbers.dat`:

```
ObjectInputStream inStream =
  new ObjectInputStream (new
  FileInputStream("numbers.dat"));
```

- The constructor for `ObjectInputStream` requires a `FileInputStream` argument
- The constructor for `FileInputStream` requires a `String` argument
  - the `String` argument is the input file name
- The following two statements are equivalent to the statement at the top of this slide:

```
FileInputStream middleman =
  new FileInputStream("numbers.dat");
ObjectInputStream inputStream =
  new ObjectInputStream (middleman);
```

Some `ObjectInputStream` Methods

- For every output file method there is a corresponding input file method
- You can read data from an input file after it is connected to a stream class
  - Use methods defined in `ObjectInputStream`
    - `readInt()`
    - `readDouble()`
    - `readBoolean()`
    - etc.
- Note that each write method throws `IOException`
- Also note that each write method includes the modifier `final`

Input File Exceptions

- A `FileNotFoundException` is thrown if the file is not found when an attempt is made to open a file
- Each read method throws `IOException`
  - we still have to write a catch block for it
- If a read goes beyond the end of the file an `EOFException` is thrown

Avoiding Common `ObjectInputStream` File Errors

There is no error message (or exception) if you read the wrong data type!

- Input files can contain a mix of data types
  - it is up to the programmer to know their order and use the correct read method
- `ObjectInputStream` works with binary, not text files
- As with an output file, close the input file when you are done with it

Common Methods to Test for the End of an Input File

- A common programming situation is to read data from an input file but not know how much data the file contains
- In these situations you need to check for the end of the file
- There are three common ways to test for the end of a file:
  1. Put a sentinel value at the end of the file and test for it.
  2. Throw and catch an end-of-file exception.
  3. Test for a special character that signals the end of the file (text files often have such a character).

The `EOFException` Class

- Many (but not all) methods that read from a file throw an end-of-file exception (`EOFException`) when they try to read beyond the file
  - all the `ObjectInputStream` methods do throw it
- The end-of-file exception can be used in an "infinite" (`while(true)`) loop that reads and processes data from the file
  - the loop terminates when an `EOFException` is thrown
- The program is written to continue normally after the `EOFException` has been caught

Using `EOFException`

`main` method from `EOFExceptionDemo`

Intentional "infinite" loop to process data from input file

Loop exits when end-of-file exception is thrown

Processing continues after `EOFException`: the input file is closed

Note order of catch blocks: the most specific is first and the most general last

```
try
{
   ObjectInputStream inputStream =
     new ObjectInputStream(new FileInputStream("numbers.dat"));
   int n;

   System.out.println("Reading ALL the integers");
   System.out.println("in the file numbers.dat.");
   try
   {
      while (true)
      {
         n = inputStream.readInt();
         System.out.println(n);
      }
   }
   catch(EOFException e)
   {
      System.out.println("End of reading from file.");
   }
   inputStream.close();
}
catch(FileNotFoundException e)
{
   System.out.println("Cannot find file numbers.dat.");
}
catch(IOException e)
{
   System.out.println("Problem with input from file numbers.dat.");
}
```

Summary

- Overview of Streams and File I/O
  - Buffering
- Text-File I/O
- Basic Binary-File I/O