Chapter 13

Input/Output

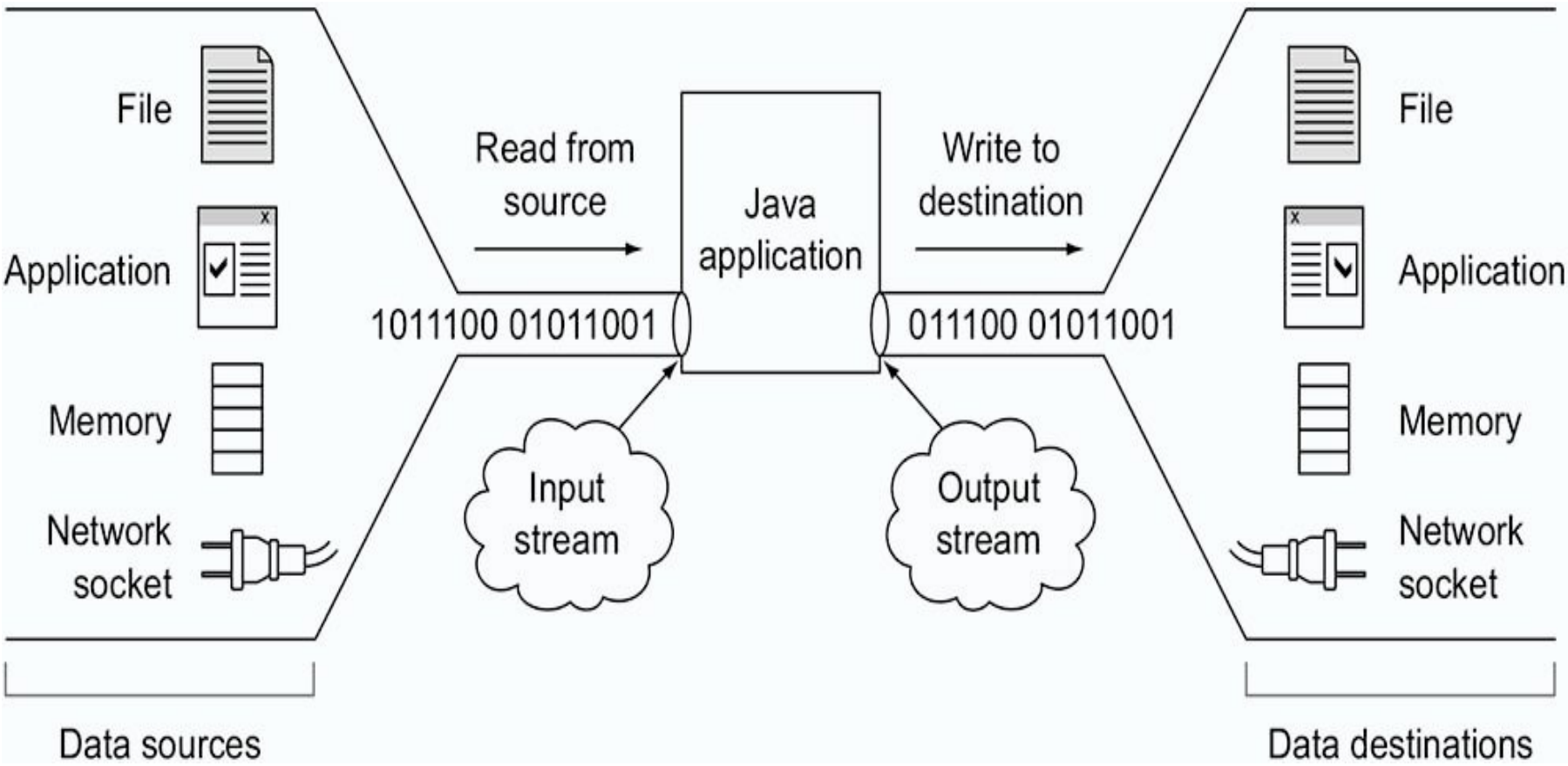
I/O Overview

- *I/O* = Input/Output
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
  - permanent copy
  - output from one program can be input to another
  - input can be automated (rather than entered manually)

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
- ***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
- A stream connects a program to an I/O object
  - System.out connects a program to the screen
  - System.in connects a program to the keyboard

Streams


File
Application
Memory
Network socket
Data sources
Read from source
Java application
Write to destination
1011100 01011001
011100 01011001
Input stream
Output stream
File
Application
Memory
Network socket
Data destinations

Binary Versus Text Files

- *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)**:** 01111110
    - Two bytes (short): 00000000 01111110
    - Four bytes (int): 00000000 00000000 00000000 01111110

Text file: an example

127 smiley

faces

0000000 061 062 067 011 163 155 151 154

First line 1 2 7 \t s m i l

0000010 145 171 012 146 141 143 145 163

Second line e y \n f a c e s

0000020 012

Third line \n

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.*;
```

Every File Has Two Names

1. the stream name used by Java
   - `outputStream` in the example
2. the name used by the operating system
   - `out.txt` in the example

How to do I/O: Steps

- `import java.io.*;`

- *Open* the stream
- *Use* the stream (read, write, or both)
- *Close* the stream

Why Java I/O is hard?

- Java I/O is very powerful, with an overwhelming number of options
- Any given kind of I/O is not particularly difficult
- The trick is to find your way through the maze of possibilities

Types of Streams

Streams can be broadly categorized into:

- **Byte Streams**: Used to handle binary data (e.g., images, audio files). Examples include FileInputStream and FileOutputStream.
- **Character Streams**: Used to handle character data (e.g., text files). Examples include FileReader and FileWriter.

Opening a stream

- There is data external to your program that you want to get, or you want to put data somewhere outside your program
- When you open a stream, you are making a connection to that external place
- Once the connection is made, you forget about the external place and just use the stream

Example of opening a stream

- FileReader is a class or object provided by many programming languages and libraries for reading data from files. It is commonly used for file input operations and is particularly useful for reading text and binary data from files.
- A FileReader is a used to connect to a file that will be used for input:

```
FileReader fileReader =
    new FileReader(fileName);
```

- The fileName specifies where the (external) file is to be found
- You never use fileName again; instead, you use fileReader
- FileReader is a subclass of InputStreamReader, which itself is a type of Reader.

Using a stream

- Some streams can be used only for input, others only for output, still others for both
- *Using* a stream means doing input from it or output to it
- But it’s not usually that simple - you need to manipulate the data in some way as it comes in or goes out

Example of using a stream

```
int ch;
ch = fileReader.read();
```

- The fileReader.read() method reads one character and returns it as an integer, or -1 if there are no more characters to read
- The meaning of the integer depends on the file encoding (ASCII, Unicode, other)

Manipulating the input data

- Reading characters as integers isn't usually what you want to do
- A BufferedReader will convert integers to characters; it can also read whole lines
- The constructor for BufferedReader takes a FileReader parameter:

```
BufferedReader bufferedReader =
    new BufferedReader(fileReader);
```

Reading lines

```
String s;
s = bufferedReader.readLine();
```

- A BufferedReader will return null if there is nothing more to read

Closing

- A stream is an expensive resource
- There is a limit on the number of streams that you can have open at one time
- You should not have more than one stream open on the same file
- You must close a stream before you can open it again
- *Always close your streams!*

The Predefined Stream

- All java programs automatically import the java.lang package.
- This package defines a class called System.
- System contains three predefined steam variables in, out, err.
- System.out refers to standard output stream (Console).
- System.in refers to standard input stream (Keyboard).
- System.err refers to standard error stream.

Reading Characters

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead {
  public static void main(String args[])
    throws IOException
  {
    char c;
    BufferedReader br = new
          BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter characters, 'q' to quit.");
    // read characters
    do {
      c = (char) br.read();
      System.out.println(c);
    } while(c != 'q');
  }
}
```

Reading String

```
// Read a string from console using a BufferedReader.
import java.io.*;
class BRReadLines {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
    String str;
    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    do {
      str = br.readLine();
      System.out.println(str);
    } while(!str.equals("stop"));
  }
}
```

Writing Console Output

```
// Demonstrate System.out.write().
class WriteDemo {
  public static void main(String args[]) {
    int b;

    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

The PrintWriter Class

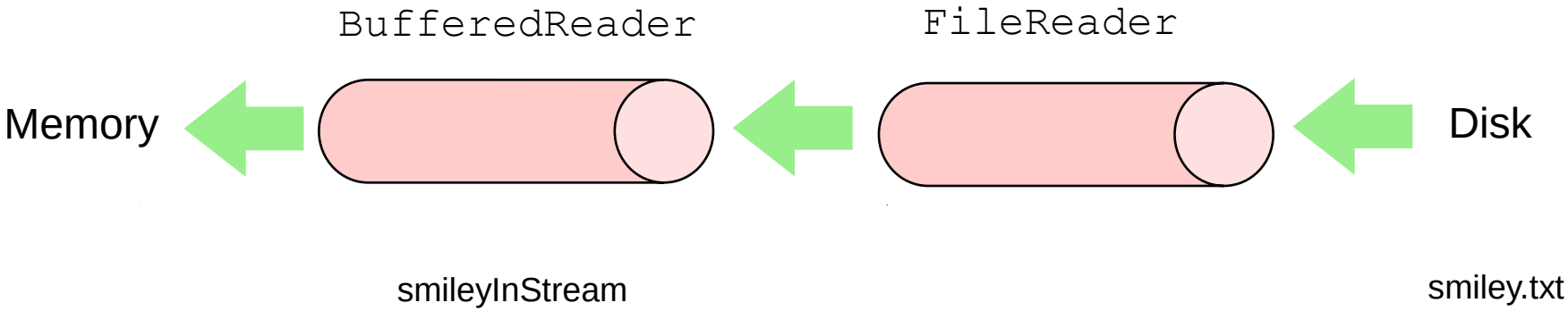
```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);
    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

Text files

- Text (.txt) files are the simplest kind of files
  - text files can be used by many different programs
- Formatted text files (such as .doc files) also contain binary formatting information

Input File Streams




```
BufferedReader smileyInStream =
new BufferedReader( new FileReader("smiley.txt") );
```

Basics of the LineReader constructor

- Create a FileReader for the named file:

```
FileReader fileReader =
    new FileReader(fileName);
```

- Use it as input to a BufferedReader:

```
BufferedReader bufferedReader =
    new BufferedReader(fileReader);
```

- Use the BufferedReader; but first, we need to catch possible Exceptions

My LineReader class

```
class LineReader {
  BufferedReader bufferedReader;

  LineReader(String fileName) {...}

  String readLine( ) {...}

  void close( ) {...}
}
```

The full LineReader constructor

```
LineReader(String fileName)
{
    FileReader fileReader = null;
    try
    {
       fileReader = new FileReader(fileName);
     }
    catch (FileNotFoundException e)
    {
      System.err.println
         ("LineReader can't find input file: " + fileName);
      e.printStackTrace( );
    }
    bufferedReader = new BufferedReader(fileReader);
  }
```

readLine

```
String readLine( ) {
    try {
       return bufferedReader.readLine( );
    }
    catch(IOException e) {
        e.printStackTrace( );
    }
    return null;
}
```

close

```
void close() {
    try {
        bufferedReader.close( );
    }
    catch(IOException e) { }
}
```

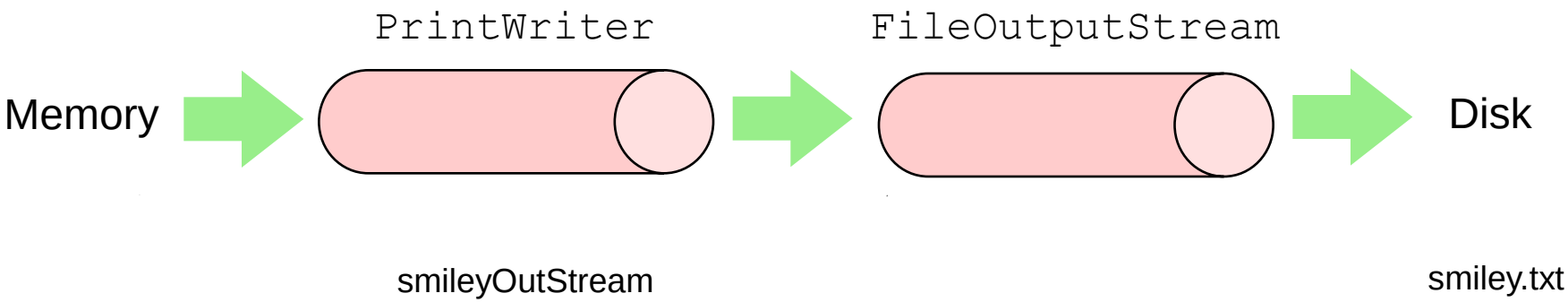
Reading from a file

```
/* Display a text file.
   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.
   java ShowFile TEST.TXT
*/
import java.io.*;
class ShowFile {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream fin;
```

Reading from a file

```
try {
    fin = new FileInputStream(args[0]);
  } catch(FileNotFoundException e) {
    System.out.println("File Not Found");
    return;
  } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Usage: ShowFile File");
    return;
  }
  // read characters until EOF is encountered
  do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);
  fin.close();
 }
}
```

Output File Streams




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

The LineWriter class

```
class LineWriter {
    PrintWriter printWriter;

    LineWriter(String fileName) {...}

   void writeLine(String line) {...}

   void close( ) {...}
}
```

The constructor for LineWriter

```
LineWriter(String fileName) {
    try {
       printWriter =
             new PrintWriter(
                new FileOutputStream(fileName, true);
        }
    catch(Exception e) {
       System.err.println("LineWriter can't " +
          "use output file: " + fileName);
    }
 }
```

Flushing the buffer

- When you put information into a buffered output stream, it goes into a buffer
- The buffer may not be written out right away
- If your program crashes, you may not know how far it got before it crashed
- Flushing the buffer is forcing the information to be written out

PrintWriter

- Buffers are automatically flushed when the program ends normally
- Usually it is your responsibility to flush buffers if the program does not end normally
- PrintWriter can do the flushing for you

```
public PrintWriter(OutputStream out,
                   boolean autoFlush)
```

writeLine

```
void writeLine(String line) {
      printWriter.println(line);
  }
```

close

```
void close( )
{
    printWriter.flush( );
    try
    {
                    printWriter.close( );
    }
    catch(Exception e)
    {
    }
 }
```

Writing Files

```
/* Copy a text file.
   To use this program, specify the name
   of the source file and the destination file.
   For example, to copy a file called FIRST.TXT
   to a file called SECOND.TXT, use the following
   command line.
   java CopyFile FIRST.TXT SECOND.TXT
*/
import java.io.*;
class CopyFile {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream fin;
    FileOutputStream fout;
```

Writing Files

```
try {
    // open input file
    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException e) {
      System.out.println("Input File Not Found");
      return;
    }
    // open output file
    try {
      fout = new FileOutputStream(args[1]);
    } catch(FileNotFoundException e) {
      System.out.println("Error Opening Output File");
      return;
    }
  } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Usage: CopyFile From To");
    return;
  }
```

Writing Files

```
// Copy File
   try {
     do {
       i = fin.read();
       if(i != -1) fout.write(i);
     } while(i != -1);
   } catch(IOException e) {
     System.out.println("File Error");
   }

   fin.close();
   fout.close();
  }
}
```

End of Chapter 13

Questions?