Java Input/Output

Today's Topics

- An introduction to the Java I/O library.
- The **File** class
- Using command line arguments
- More on the Java I/O classes
- The **SimpleInput** class, in detail
- Java's compression classes (briefly)

Java Input/Output

- I/O libraries are hard to design, and everyone can find something to complain about.
- Java's I/O library is extensive, and it seems like you need to know 100 things before you can start using it.
- Today, let's learn just five things and still do something useful.

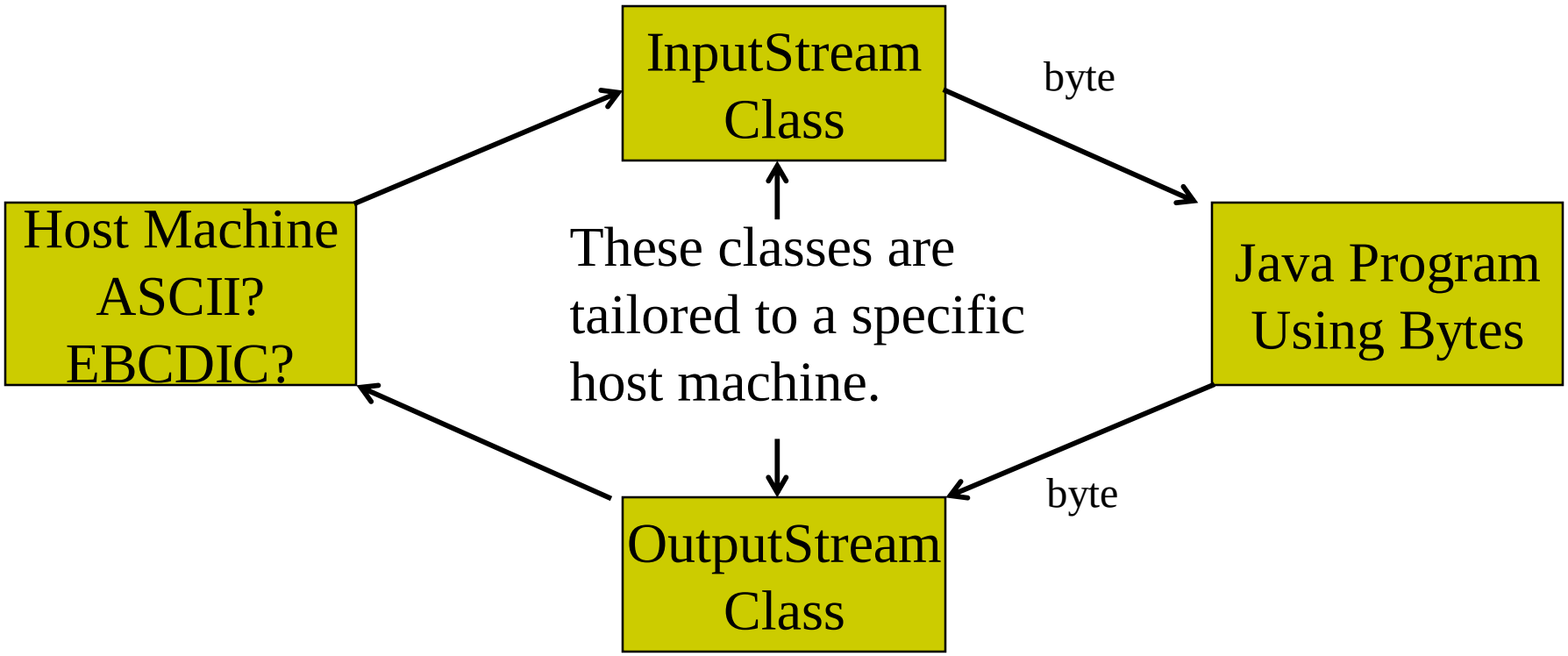
#1: Why Is Java I/O Hard?

- Java is intended to be used on many very different machines, having
  - different character encodings (ASCII, EBCDIC, 7- 8- or 16-bit…)
  - different internal numerical representations
  - different file systems, so different filename & pathname conventions
  - different arrangements for EOL, EOF, etc.
- The Java I/O classes have to “stand between” your code and all these different machines and conventions.

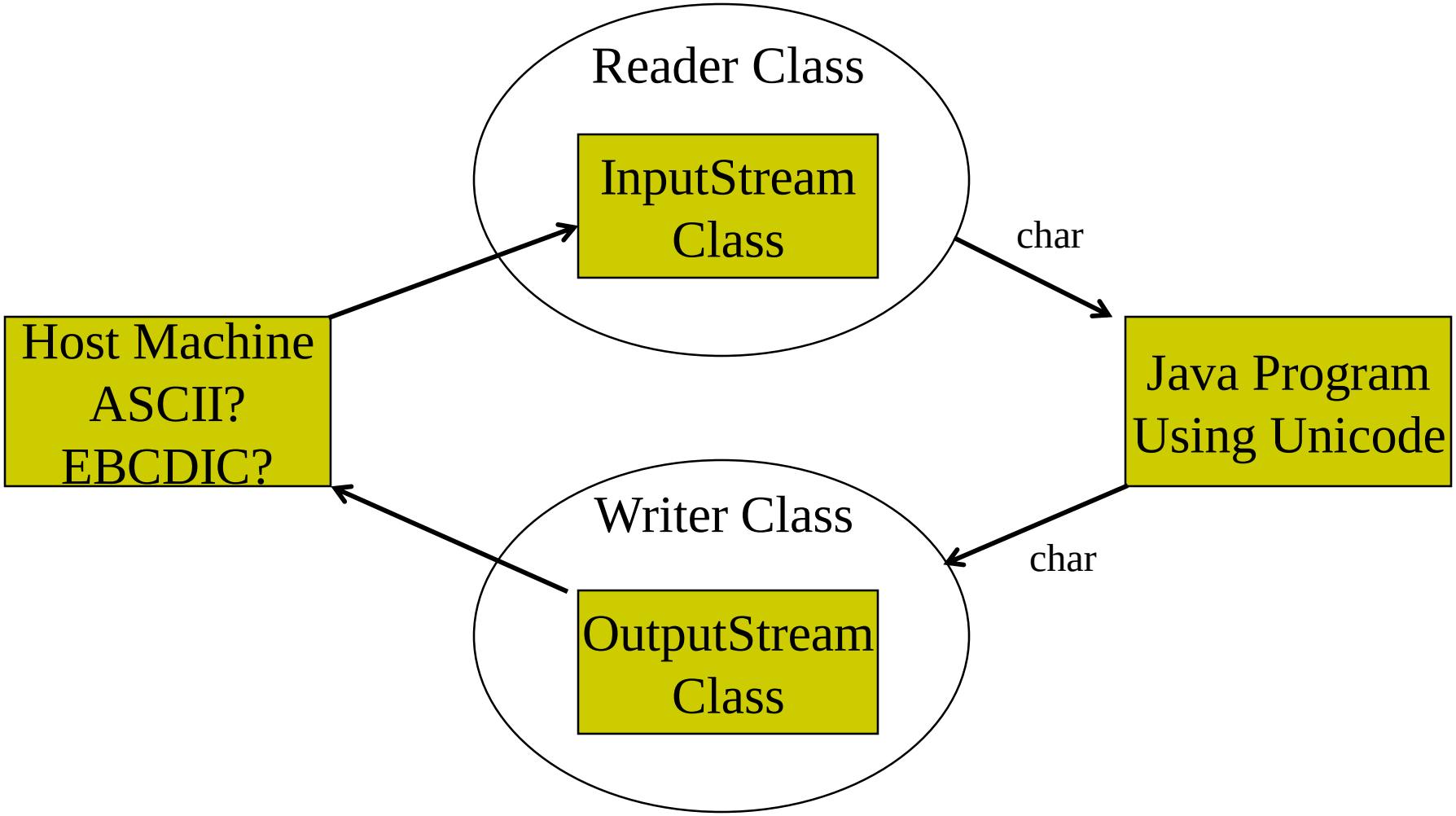
#2: Java's Internal Characters

- Unicode. 16-bit. Good idea.
- So, the primitive type **char** is 16-bit.
- Reading from a file using 8-bit ASCII characters (for example) requires conversion.
- Same for writing.
- But binary files (e.g., graphics) are "byte-sized", so there is a primitive type **byte**.
- So Java has two systems to handle the two different requirements.
- Both are in **java.io**, so import this *always*!
- I don't show imports in the examples below.

Streams


InputStream Class
byte
Host Machine ASCII? EBCDIC?
These classes are tailored to a specific host machine.
Java Program Using Bytes
byte
OutputStream Class

Readers and Writers



#3: Is Java “Platform Independent”?

- Yes, to the extent that you, the Java programmer, needn’t care about the platform your code will run on.
- No, to the extent that the Java I/O classes, the compiler, and any browser your clients use, must be programmed specifically for the host machine.
- This is *not* a new idea, just well-hyped by Sun (recall “p-code” from the 1970’s).

#4: What Are The Input Sources?

- **System.in**, which is an **InputStream** connected to your keyboard. (**System** is **public**, **static** and **final**, so it's always there).
- A file on your local machine. This is accessed through a **Reader** and/or an **InputStream**, usually using the **File** class.
- Resources on another machine through a **Socket**, which can be connected to an **InputStream**, and through it, a **Reader**.

#5: Why Can't We Read Directly From These?

- We can, but Java provides only "low-level" methods for these types. For example, **InputStream.read()** just reads a byte…
- It is assumed that in actual use, we will "wrap" a basic input source within another class that provides more capability.
- This "wrapper" class provides the methods that we actually use.

“Wrapping”

- Input comes in through a stream (bytes), but usually we want to read characters, so “wrap” the stream in a Reader to get characters.

```
public static void main(String[] args) {
    InputStreamReader isr = new InputStreamReader(System.in);
    int c;
    try {
        while ((c = isr.read()) != -1)
        System.out.println((char) c);
    }
    catch(IOException e) {
    }
}
```

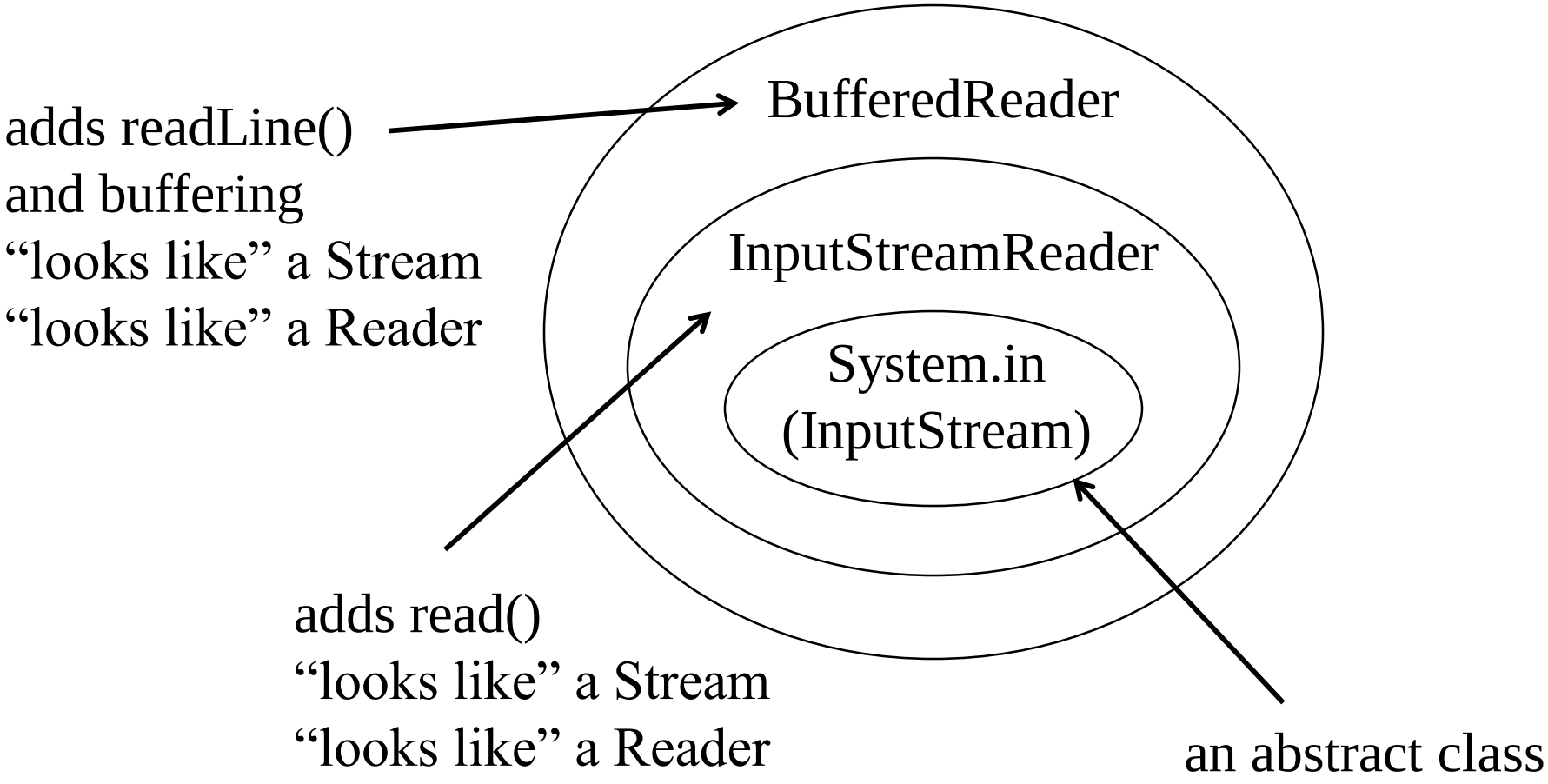
InputStreamReader

- This is a bridge between bytes and chars.
- The **read()** method returns an **int**, which must be cast to a **char**.
- **read()** returns -1 if the end of the stream has been reached.
- We need more methods to do a better job!

Use a **BufferedReader**

```
public static void main(String[] args) {
    BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
    String s;
    try {
        while ((s = br.readLine()).length() != 0)
        System.out.println(s);
    }
    catch(IOException e) {
    }
}
```

“Transparent Enclosure”



Reading From a File

- The same idea works, except we need to use a **FileInputStream**.
- Its constructor takes a string containing the file pathname.

```
public static void main(String[] args) throws IOException {
    InputStreamReader isr = new
        InputStreamReader(new FileInputStream("FileInput.java"));
    int c;
    while ((c = isr.read()) != -1)
        System.out.println((char) c);
    isr.close();
}
```

Reading From a File (cont.)

- Here we check for a -1, indicating we've reached the end of the file.
- This works just fine if the file to be read is in the same directory as the class file, but an absolute path name is safer.
- The **read()** method can throw an **IOException**, and the **FileInputStream** constructor can throw a **FileNotFoundException**
- Instead of using a try-catch construction, this example shows main() declaring that it throws **IOException**. This is a "dirty trick".

The **File** Class

- Think of this as holding a file *name*, or a list of file *names* (as in a directory).
- You create one by giving the constructor a pathname, as in

  File f = new File("d:/www/java/week10/DirList/.");
- This is a directory, so now the **File f** holds a list of (the names of) files in the directory.
- It’s straightforward to print them out.

Listing Files

```
import java.io.*;
import java.util.*;
public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        System.out.println(path.getAbsolutePath());
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new DirFilter(args[0]));
        for (int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
}
```

With No Command Line Args…

d:\www\java\week10\DirList\.
DirFilter.class
DirFilter.java
DirList.class
DirList.java
DirList.java~

With ".java" on the Command Line

d:\www\java\week10\DirList\.
DirFilter.java
DirList.java
DirList.java~

DirFilter is a **FilenameFilter**

- Its only method is **accept()**:

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

public class DirFilter implements FilenameFilter {
    String afn;
    DirFilter(String afn) { this.afn = afn; }
    public boolean accept(File dir, String name) {
        String f = new File(name).getName();
        return f.indexOf(afn) != -1;
    }
}
```

Using the “args” in **main()**

- All this time we’ve been dumbly typing
  public static void main(String[] args) {…
- **args** is an array of **String**s, but for us it’s usually been empty.
- It contains any *command line parameters* we choose to include.
- If we’re at a DOS or Unix command line, we might type **>java DirList .java**
- In Eclipse, we set the parameters via the Run/Run.

Other **File** Methods

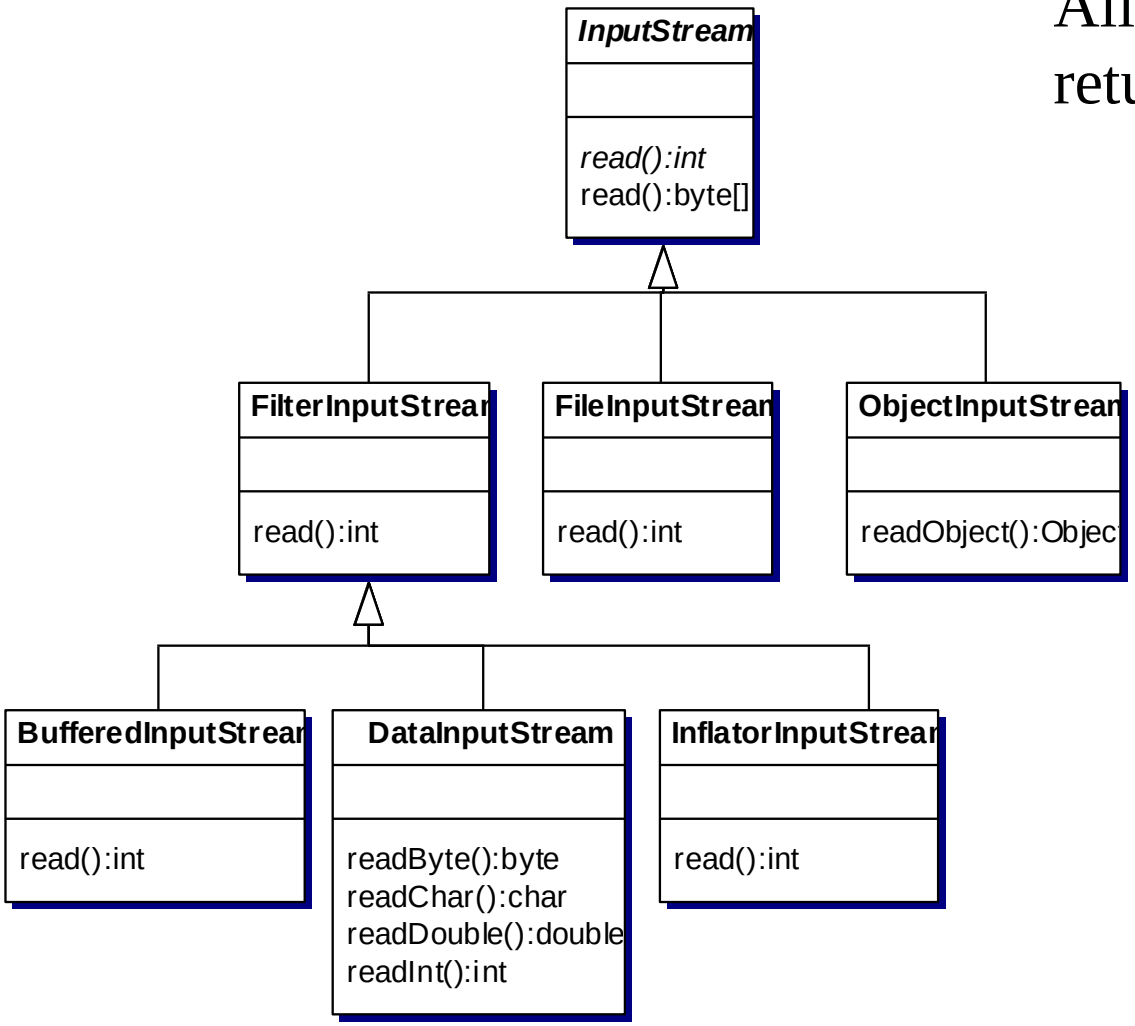
- canRead()
- canWrite()
- exists()
- getParent()
- isDirectory()
- isFile()
- lastModified()
- length()

File Methods for Modifying

- createNewFile()
- delete()
- makeDir()
- makeDirs()
- renameTo()
- setLastModified()
- setReadOnly()

More on Input

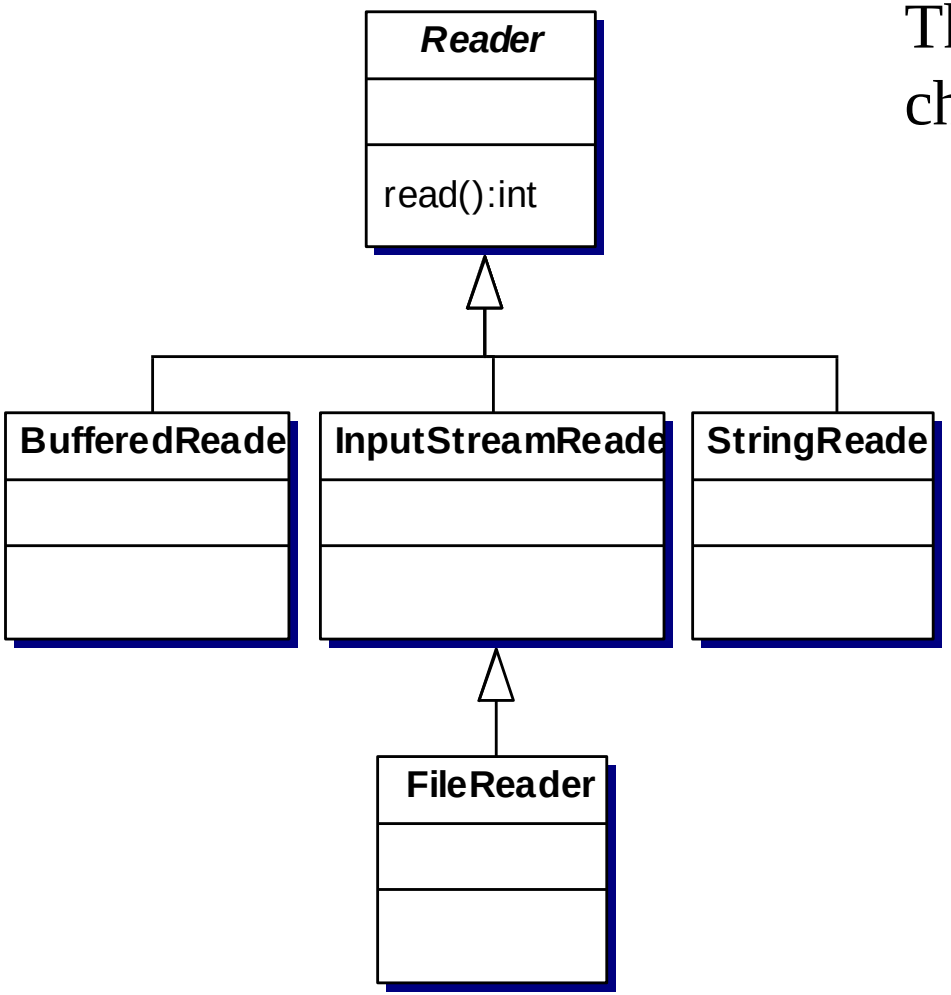
All of these return bytes!



FilterInputStream JavaDoc

- A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
- The class FilterInputStream itself simply overrides all methods of InputStream with versions that pass all requests to the contained input stream.
- Subclasses of FilterInputStream may further override some of these methods and may also provide additional methods and fields.

Readers



These return chars!

We Saw These Last Time

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

InputStreamReader isr = new
        InputStreamReader(new
    FileInputStream("FileInput.java")); //slow: unbuffered
```

This is easier (if we're happy with the default character encoding and buffer size:

```
InputStreamReader isr = new
        FileReader(" FileInput.java");
```

OutputStreams and **Writer**s

- Basically, a “mirror image” of **InputStream**s and **Reader**s.
- Wrapping is the same, e.g.,

```
BufferedWriter bw =
        new BufferedWriter(new OutputStreamWriter(System.out));
String s;
try {
      while ((s = br.readLine()).length() != 0) {
       bw.write(s, 0, s.length());
       bw.newLine();
       bw.flush();
      }
}
```

FileWriter

- Again, basically the same. The constructors are
  - **FileWriter(File file)**
  - **FileWriter(FileDescriptor fd)**
  - **FileWriter(String s)**
  - **FileWriter(String s, boolean append)**
- The last one allows appending, rather than writing to the beginning (and erasing an existing file!).
- These *will* create files!
- There is also **PrintWriter**

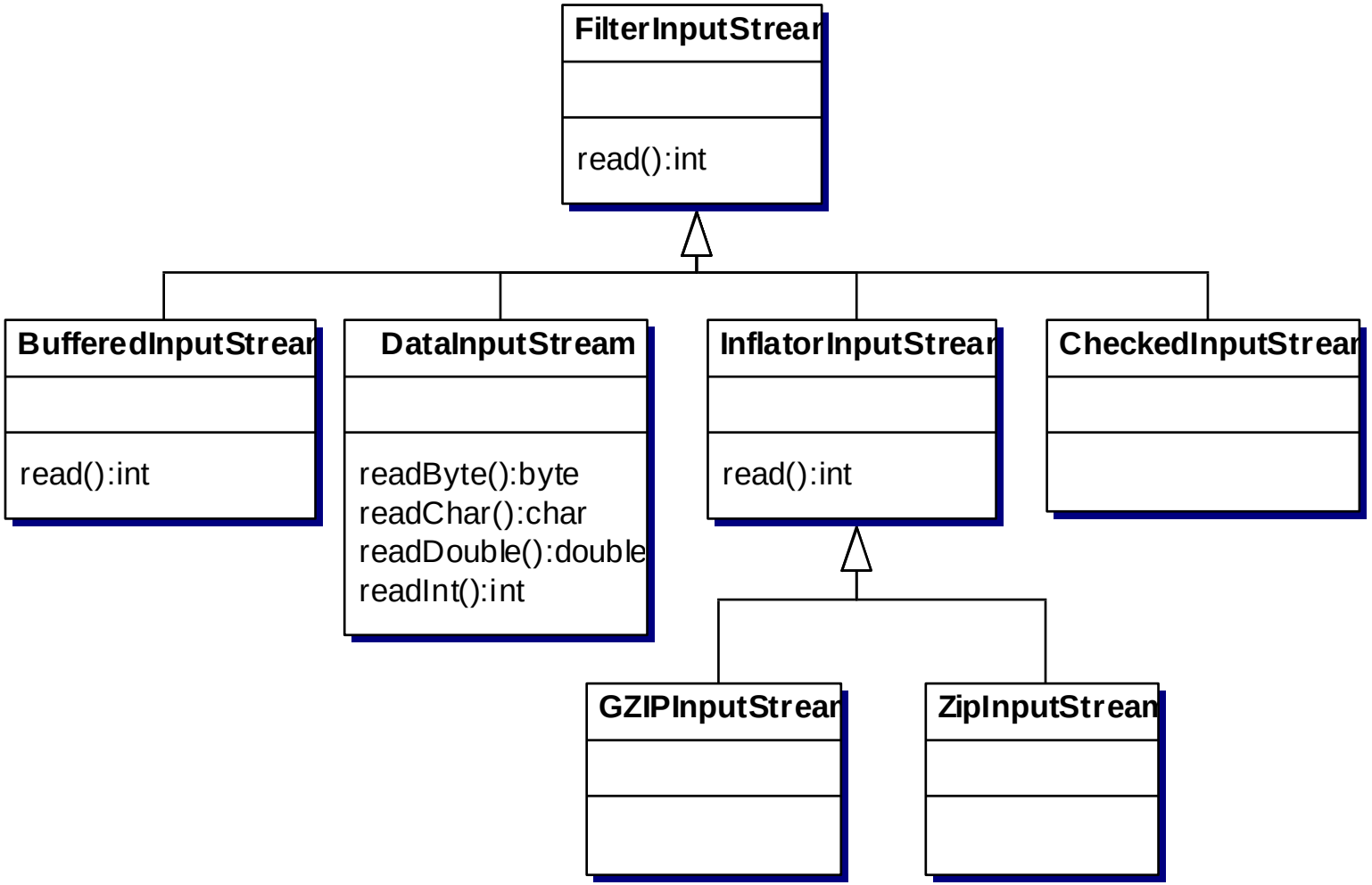
PrintWriter

```
PrintWriter out =
    new PrintWriter(new BufferedWriter(new FileWriter("Test.txt")));
String s;
try {
      while ((s = br.readLine()).length() != 0) {
       bw.write(s, 0, s.length());
       bw.newLine();
       bw.flush();
       out.println(s);
       //out.flush();
      }
}
catch(IOException e) {
}
out.close();   // also flushes
```

Java's Compression Classes

- These are used to write and read streams in Zip and GZIP formats.
- As always, these classes are wrappers around existing I/O classes, for "transparent" use.
- These classes are astonishingly easy to use!
- C++ should have this…
- Here is a picture of the input classes (the output classes are similar):

The Compression Input Classes



Eckel's GZIP Example (1st part)

```
import java.io.*;
import java.util.zip.*;
public class GZIPCompress {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(
            new FileReader(args[0]));
        BufferedOutputStream out = new BufferedOutputStream(
             new GZIPOutputStream(new FileOutputStream("test.gz")));
        System.out.println("Writing file");
        int c;
        while((c = in.read()) != -1)
            out.write(c);
        in.close();
        out.close();
```

GZIP Example (2nd part)

```
        System.out.println("Reading file");
        BufferedReader in2 =
            new BufferedReader(
                new InputStreamReader(
                    new GZIPInputStream(
                        new FileInputStream("test.gz"))));  // whew!
        String s;
        while((s = in2.readLine()) != null)
            System.out.println(s);
    }
}
```

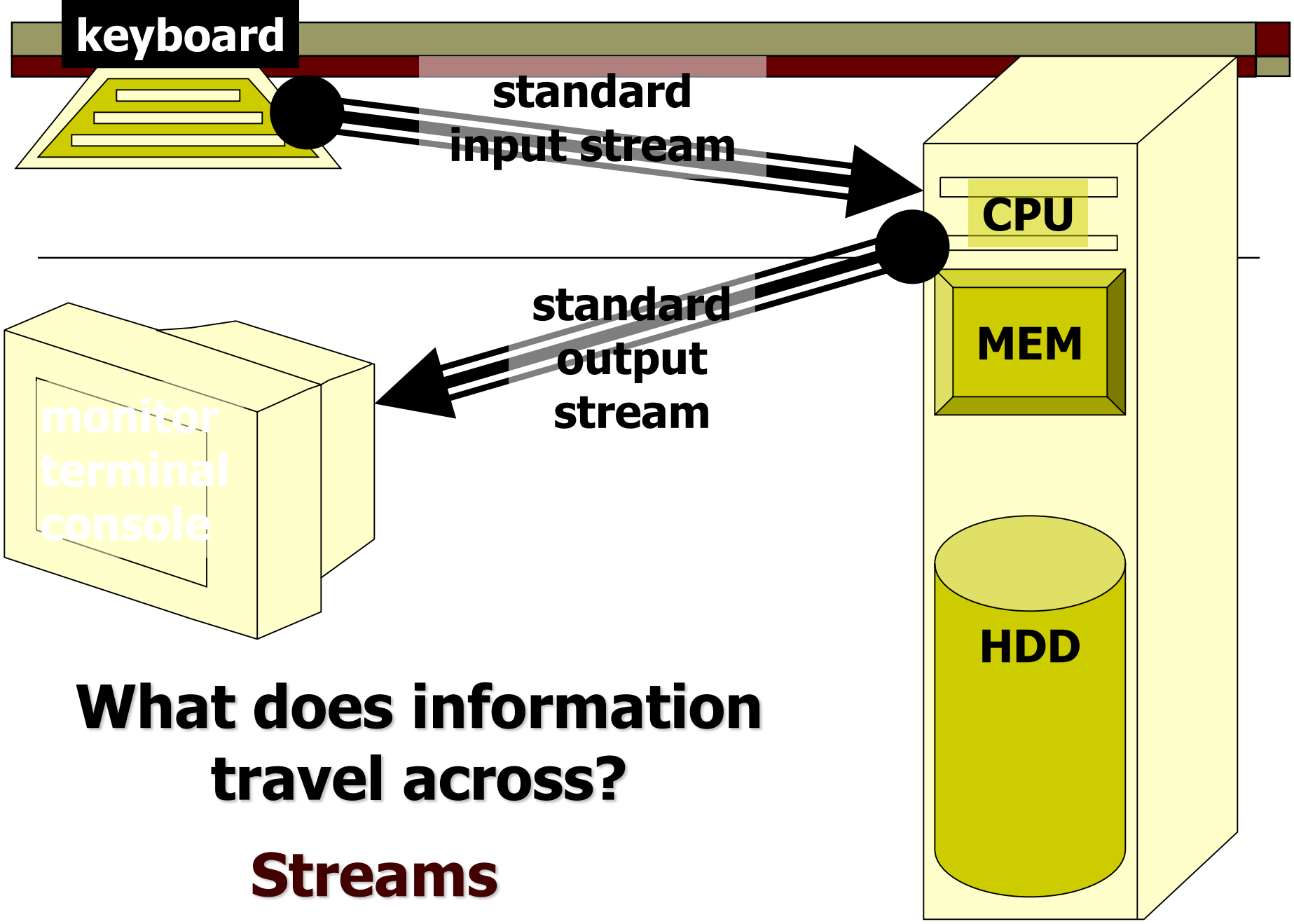
Comments

- GZIP and Zip are specific algorithms for compressing and uncompressing. You'll have to wait for details until Prof. McCarthy's course.
- This program works pretty well:
  - DancingMen.txt is 51KB
  - test.gz is 21KB

Chapter 13 – Files and Streams

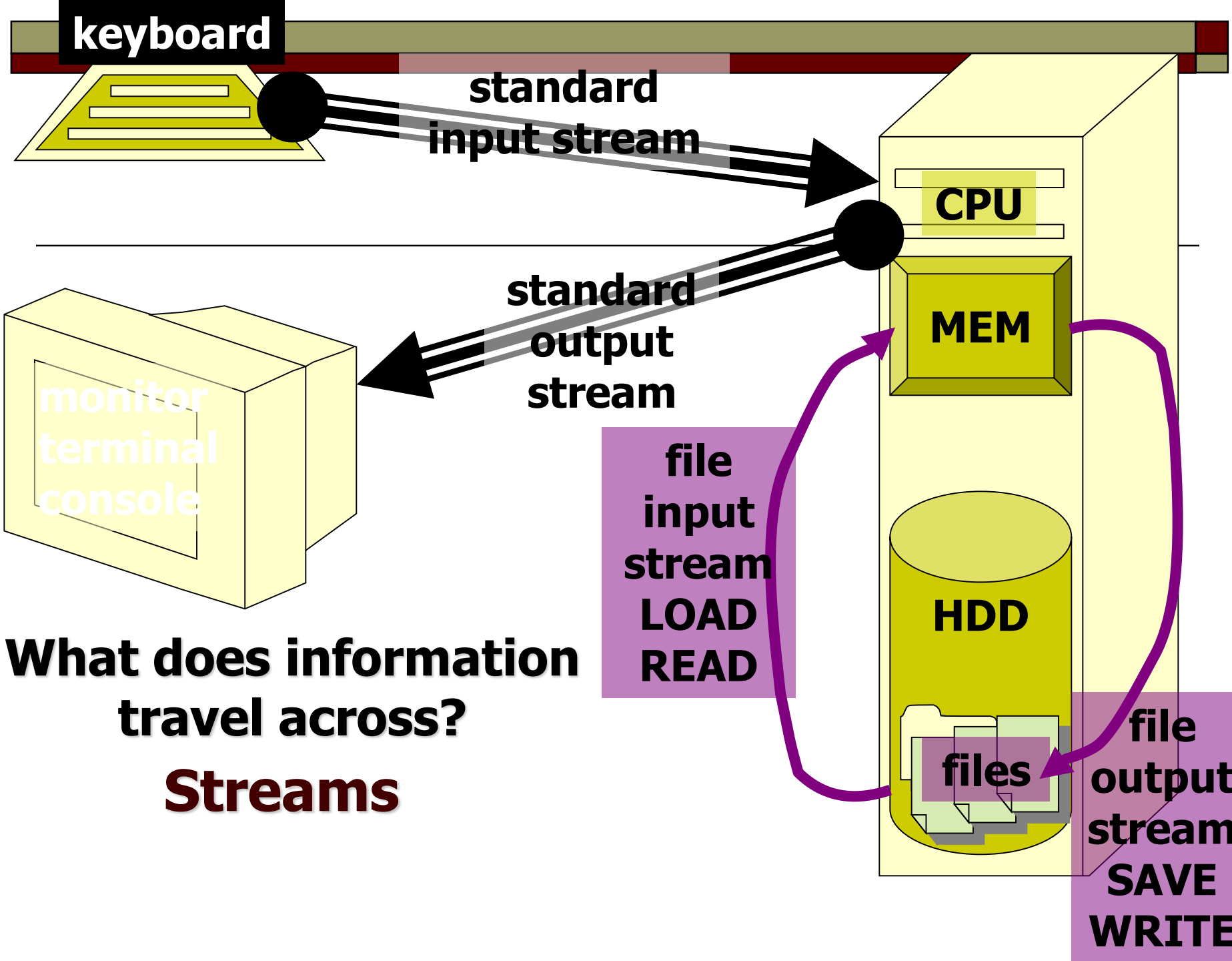
Goals

- To be able to read and write text files
- To become familiar with the concepts of text and binary formats
- To learn about encryption
- To understand when to use sequential and random file access
- To be able to read and write objects using serialization



What does information travel across?

Streams



What does information travel across?

Streams

Reading and Writing Text Files

- Text files – files containing simple text
  - Created with editors such as notepad, html, etc.
- Simplest way to learn it so extend our use of **`Scanner`**
  - Associate with files instead of **`System.in`**
- All input classes, except Scanner, are in java.io
  - **`import java.io.*;`**

Review: Scanner

- We've seen Scanner before
- The constructor takes an object of type **`java.io.InputStream`** – stores information about the connection between an input device and the computer or program
  - Example: **`System.in`**
- Recall – only associate ***one*** instance of **`Scanner`** with **`System.in`** in your program
  - Otherwise, get bugs

Numerical Input

- 2 ways (we've learned one, seen the other)
  - Use **`int`** as example, similar for **`double`**

- First way:
  - Use **`nextInt()`**

```
int number = scanner.nextInt();
```

- Second way:
  - Use **`nextLine()`, `Integer.parseInt()`**

```
String input = scanner.nextLine();
int number = Integer.parseInt(input);
```

Numerical Input

- Exceptions
  - **`nextInt()`** throws **`InputMismatchException`**
  - **`parseInt()`** throws **`NumberFormatException`**

- Optimal use
  - **`nextInt()`** when there is multiple information on one line
  - **`nextLine() + parseInt()`** when one number per line

Reading Files

- The same applies for both console input and file input
- We can use a different version of a Scanner that takes a ***File*** instead of **`System.in`**
- Everything works the same!

Reading Files

- To read from a disk file, construct a **`FileReader`**

- Then, use the **`FileReader`** to construct a **`Scanner`** object

```
FileReader rdr = newFileReader("input.txt");
Scanner fin = new Scanner(rdr);
```

Reading Files

- You can use **`File`** instead of **`FileReader`**
  - Has an **`exists()`** method we can call to avoid **`FileNotFoundException`**

```
File file = new File ("input.txt");
Scanner fin;
if(file.exists()){
   fin = new Scanner(file);
} else {
   //ask for another file
}
```

Reading Files

- Once we have a Scanner, we can use methods we already know:
  - **`next`**, **`nextLine`**, **`nextInt`**, etc.

- Reads the information from the file instead of console

File Class

- `java.io.File`
  - associated with an actual file on hard drive
  - used to check file's status
- Constructors
  - `File(<full path>)`
  - `File(<path>, <filename>)`
- Methods
  - `exists()`
  - `canRead()`, `canWrite()`
  - `isFile()`, `isDirectory()`

File Class

- **`java.io.FileReader`**
  - Associated with `File` object
  - Translates data bytes from File object into a stream of characters (much like InputStream vs. InputStreamReader)
- Constructors
  - `FileReader( <File object> );`
- Methods
  - `read(),readLine()`
  - `close()`

Writing to a File

- We will use a **`PrintWriter`** object to write to a file
  - What if file already exists? → Empty file
  - Doesn't exist? → Create empty file with that name

- How do we use a **`PrintWriter`** object?
  - Have we already seen one?

Writing to a File

- The out field of the System class is a **`PrintWriter`** object associated with the console
  - We will associate our **`PrintWriter`** with a file now

```
PrintWriter fout = new PrintWriter("output.txt");
fout.println(29.95);
fout.println(new Rectangle(5, 10, 15, 25));

fout.println("Hello, World!");
```

- This will print the exact same information as with **`System.out`** (except to a file "output.txt")!

Closing a File

- Only main difference is that we have to close the file stream when we are done writing
- If we do not, not all output will written
- At the end of output, call **`close()`**

```
fout.close();
```

Closing a File

- Why?
  - When you call **`print()`** and/or **`println()`**, the output is actually written to a buffer. When you close or flush the output, the buffer is written to the file
  - The slowest part of the computer is hard drive operations – much more efficient to write once instead of writing repeated times

File Locations

- When determining a file name, the default is to place in the same directory as your .class files
- If we want to define other place, use an absolute path (e.g. c:\My Documents)

```
in  = new
   FileReader("c:\\homework\\input.dat");
```

- Why `\\` ?

Sample Program

- Two things to notice:
  - Have to import from java.io
  - I/O requires us to catch checked exceptions
    - **`java.io.IOException`**

Java Input Review

CONSOLE:

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

FILE:

```
Scanner inFile = new Scanner( new
   FileReader(srcFileName ));
```

Java Output Review

- CONSOLE:

```
System.out.print("To the screen");
```

- FILE:

```
PrintWriter fout =
   new PrintWriter(new File("output.txt");
fout.print("To a file");
```

```
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class LineNumberer{
   public static void main(String[] args){
      Scanner console = new Scanner(System.in);
      System.out.print("Input file: ");
      String inFile = console.next();

      System.out.print("Output file: ");
      String outFile = console.next();

      try{
            FileReader reader = new FileReader(inFile);
            Scanner in = new Scanner(reader);
```

```
        PrintWriter out = new
           PrintWriter(outputFileName);
        int lineNumber = 1;

        while (in.hasNextLine()){
           String line = in.nextLine();
           out.println("/* " + lineNumber + " */ " +
           line);
           lineNumber++;
        }

        out.close();
     } catch (IOException exception){
          System.out.println("Error processing file: "
           + exception);
     }
   }
}
```

An Encryption Program

- Demonstration: Use encryption to show file techniques

- File encryption
  - To scramble a file so that it is readable only to those who know the encryption method and secret keyword
  - (Big area of CS in terms of commercial applications – biometrics, 128-bit encryption breaking, etc.)

Modifications of Output

- Two constraints so far:
  - Files are overwritten
  - Output is buffered and not written immediately
- We have options to get around this

File Class

- **`java.io.FileWriter`**
  - Associated with **`File`** object
  - Connects an output stream to write bytes of info
- Constructors
  - **`FileWriter( <filename>, <boolean> );`**
    - **true to append data, false to overwrite all of file**
- This will overwrite an existing file
  - To avoid, create File object and see if `exists()` is true

Java File Output

- `PrintWriter`
  - composed from several objects

```
PrintWriter out =
    new PrintWriter(
    new FileWriter( dstFileName, false ), true );
```

  - requires `throws FileNotFoundException`, which is a sub class of `IOException`

- Methods
  - `print(),println()`: **buffers** data to write
  - `flush()`: sends buffered output to destination
  - `close()`: flushes and closes stream

Java File Output

```
// With append to an existing file
PrintWriter outFile1 =
   new PrintWriter(
     new FileWriter(dstFileName,true),false);

// With autoflush on println
PrintWriter outFile2 =
   new PrintWriter(
     new FileWriter(dstFileName,false),true);

outFile1.println( "appended w/out flush" );
outFile2.println( "overwrite with flush" );
```

To flush or not to flush

- Advantage to flush:
  - Safer – guaranteed that all of our data will write to the file
- Disadvantage
  - Less efficient – writing to file takes up time, more efficient to flush once (on close)

Caeser Cipher

- Encryption key – the function to change the value

- Simple key – shift each letter over by 1 to 25 characters
  - If key = 3, A → D B → E etc.

- Decryption = reversing the encryption
  - Here we just subtract the key value

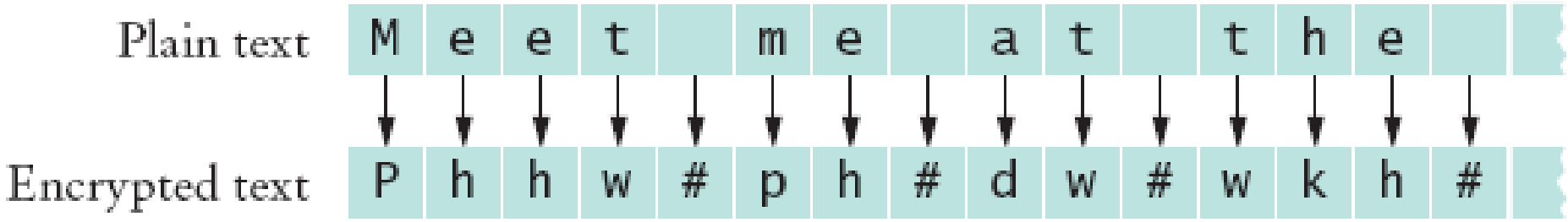


Figure 2 The Caesar Cipher

Binary File Encryption

```
int next = in.read();
if (next == -1)
   done = true;
else {
   byte b = (byte) next;
   //call the method to encrypt the byte
   byte c = encrypt(b);
   out.write(c);
}
```

```
01: import java.io.File;
02: import java.io.FileInputStream;
03: import java.io.FileOutputStream;
04: import java.io.InputStream;
05: import java.io.OutputStream;
06: import java.io.IOException;
07:
08: /**
09:    An encryptor encrypts files using the Caesar cipher.
10:    For decryption, use an encryptor whose key is the
11:    negative of the encryption key.
12: */
13: public class Encryptor
14: {
15:    /**
16:       Constructs an encryptor.
17:       @param aKey the encryption key
18:    */
19:    public Encryptor(int aKey)
20:    {
21:       key = aKey;
22:    }
23:
24:    /**
25:       Encrypts the contents of a file.
26:       @param inFile the input file
27:       @param outFile the output file
28:    */
29:    public void encryptFile(String inFile, String outFile)
30:          throws IOException
31:    {
32:       InputStream in = null;
33:       OutputStream out = null;
34:
35:       try
```

```
37:          in = new FileInputStream(inFile);
38:          out = new FileOutputStream(outFile);
39:          encryptStream(in, out);
40:       }
41:       finally
42:       {
43:          if (in != null) in.close();
44:          if (out != null) out.close();
45:       }
46:    }
47:
48:    /**
49:       Encrypts the contents of a stream.
50:       @param in the input stream
51:       @param out the output stream
52:    */
53:    public void encryptStream(InputStream in, OutputStream out)
54:          throws IOException
55:    {
56:       boolean done = false;
57:       while (!done)
58:       {
59:          int next = in.read();
60:          if (next == -1) done = true;
61:          else
62:          {
63:             byte b = (byte) next;
64:             byte c = encrypt(b);
65:             out.write(c);
66:          }
67:       }
68:    }
69:
70:    /**
71:       Encrypts a byte.
72:       @param b the byte to encrypt
73:       @return the encrypted byte
74:    */
75:    public byte encrypt(byte b)
76:    {
77:       return (byte) (b + key);
78:    }
79:
80:    private int key;
81: }
```

```
01: import java.io.IOException;
02: import java.util.Scanner;
03:
04: /**
05:    A program to test the Caesar cipher encryptor.
06: */
07: public class EncryptorTester
08: {
09:    public static void main(String[] args)
10:    {
11:       Scanner in = new Scanner(System.in);
12:       try
13:       {
14:          System.out.print("Input file: ");
15:          String inFile = in.next();
16:          System.out.print("Output file: ");
17:          String outFile = in.next();
18:          System.out.print("Encryption key: ");
19:          int key = in.nextInt();
20:          Encryptor crypt = new Encryptor(key);
21:          crypt.encryptFile(inFile, outFile);
22:       }
23:       catch (IOException exception)
24:       {
25:          System.out.println("Error processing file: " + exception);
26:       }
27:    }
28: }
29:
```

Object Streams

- Last example read BankAccount field individually
  - Easier way to deal with whole object
- **`ObjectOutputStream`** class can save a entire objects to disk
- **`ObjectOutputStream`** class can read objects back in from disk
- Objects are saved in binary format; hence, you use streams and not writers

Write out an object

- The object output stream saves all instance variables

```
BankAccount b = . . .;

ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream("bank.dat"));

out.writeObject(b);
```

Read in an object

- **`readObject`** returns an **`Object`** reference
- Need to remember the types of the objects that you saved and use a cast

```
ObjectInputStream in = new ObjectInputStream(
   new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```

Exceptions

- `readObject` method can throw a `ClassNotFoundException`
- It is a checked exception
- You must catch or declare it

Writing an Array

- Usually want to write out a collection of objects:

```
BankAccount[] arr = new BankAccount[size];

// Now add size BankAccount objects into arr
out.writeObject(arr);
```

Reading an Array

- To read a set of objects into an array

```
BankAccount[] ary = (BankAccount[])
   in.readObject();
```

Object Streams

- Very powerful features
  - Especially considering how little we have to do

- The **`BankAccount`** class as is actually will not work with the stream
  - Must implement **`Serializable`** interface in order for the formatting to work

Object Streams

```
class BankAccount implements Serializable
{
   . . .
}
```

- IMPORTANT: **`Serializable`** interface has no methods.
- No effort required

Serialization

- Serialization: process of saving objects to a stream
  - Each object is assigned a serial number on the stream
  - If the same object is saved twice, only serial number is written out the second time
  - When reading, duplicate serial numbers are restored as references to the same object

Serialization

- Why isn’t everything serializable?
  - Security reasons – may not want contents of objects printed out to disk, then anyone can print out internal structure and analyze it
  - Example: Don’t want SSN ever being accessed
  - Could also have temporary variables that are useless once the program is done running

Tokenizing

- Often several text values are in a single line in a file to be compact

  `"25 38 36 34 29 60 59"`

- The line must be broken into parts (i.e. ***tokens***)

  `"25"`

  `"38"`

  `"36"`

- tokens then can be parsed as needed

  `"25"` can be turned into the integer `25`

Tokenizing

- Inputting each value on a new line makes the file very long
- May want a file of customer info – name, age, phone number all on one line
- File usually separate each piece of info with a **delimiter** – any special character designating a new piece of data (space in previous example)

Tokenizing in Java

- use a `StringTokenizer` object
  - default delimiters are: space, tab, newline, return
  - requires: `import java.util.*`

- Constructors
  - `StringTokenizer(String line)//default dlms`
  - `StringTokenizer(String ln, String dlms)`

- Methods
  - `hasMoreTokens()`
  - `nextToken()`
  - `countTokens()`

StringTokenizing in Java

```
Scanner stdin = new…
System.out.print( "Enter a line with comma
   seperated integers(no space): " );
String input = stdin.nextLine();

StringTokenizer st;
String delims = ",";
st = new StringTokenizer( input, delims );

while ( st.hasMoreTokens() )
{
  int n = Integer.parseInt(st.nextToken());
  System.out.println(n);
}
```

```
File gradeFile = new File("scores.txt");
if(gradeFile.exists()){
   Scanner inFile = new Scanner(gradeFile);

   String line = inFile.nextLine();

   while(line != null){
      StringTokenizer st = new
            StringTokenizer(line, ":");
      System.out.print(" Name: " + st.nextToken());

      int num = 0;
      double sum = 0;

      while ( st.hasMoreTokens() )
      {
            num++;
            sum += Integer.parseInt(st.nextToken());
      }
      System.our.println(" average = "+ sum/num);
      line = inFile.nextLine();
```

```
    }

    inFile.close();
}
```

If you call `nextToken()` and there are no more tokens, `NoSuchElementException` is thrown

Tokenizing

- Scanner tokenizes already…

```
Scanner in = new Scanner(…);
while(in.hasNext()) {
   String str = in.next();
   …
}
```