Object-Oriented Programming

CSE-703029

Faculty of Computer Science

Phenikaa University

Lecture 10 : I/O

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.

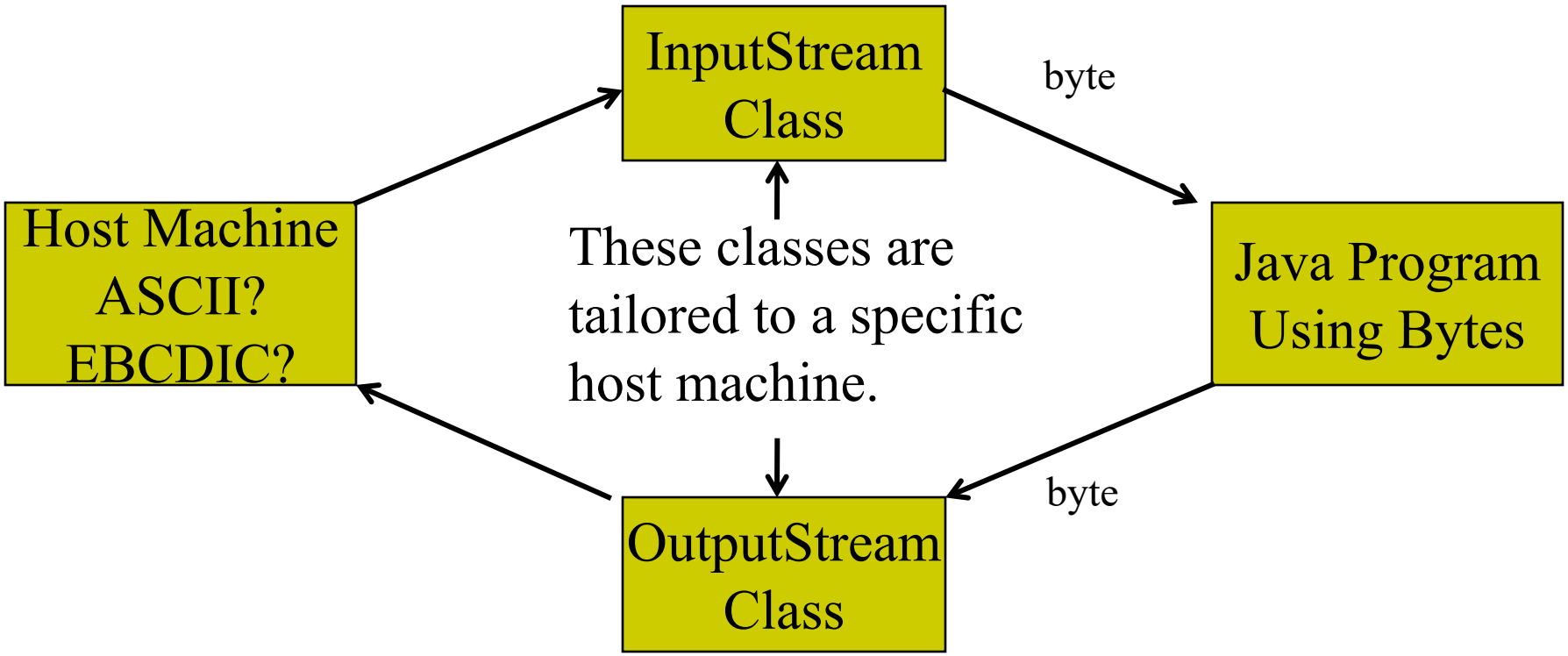
#1: Why Is Java I/O Hard?

- 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.

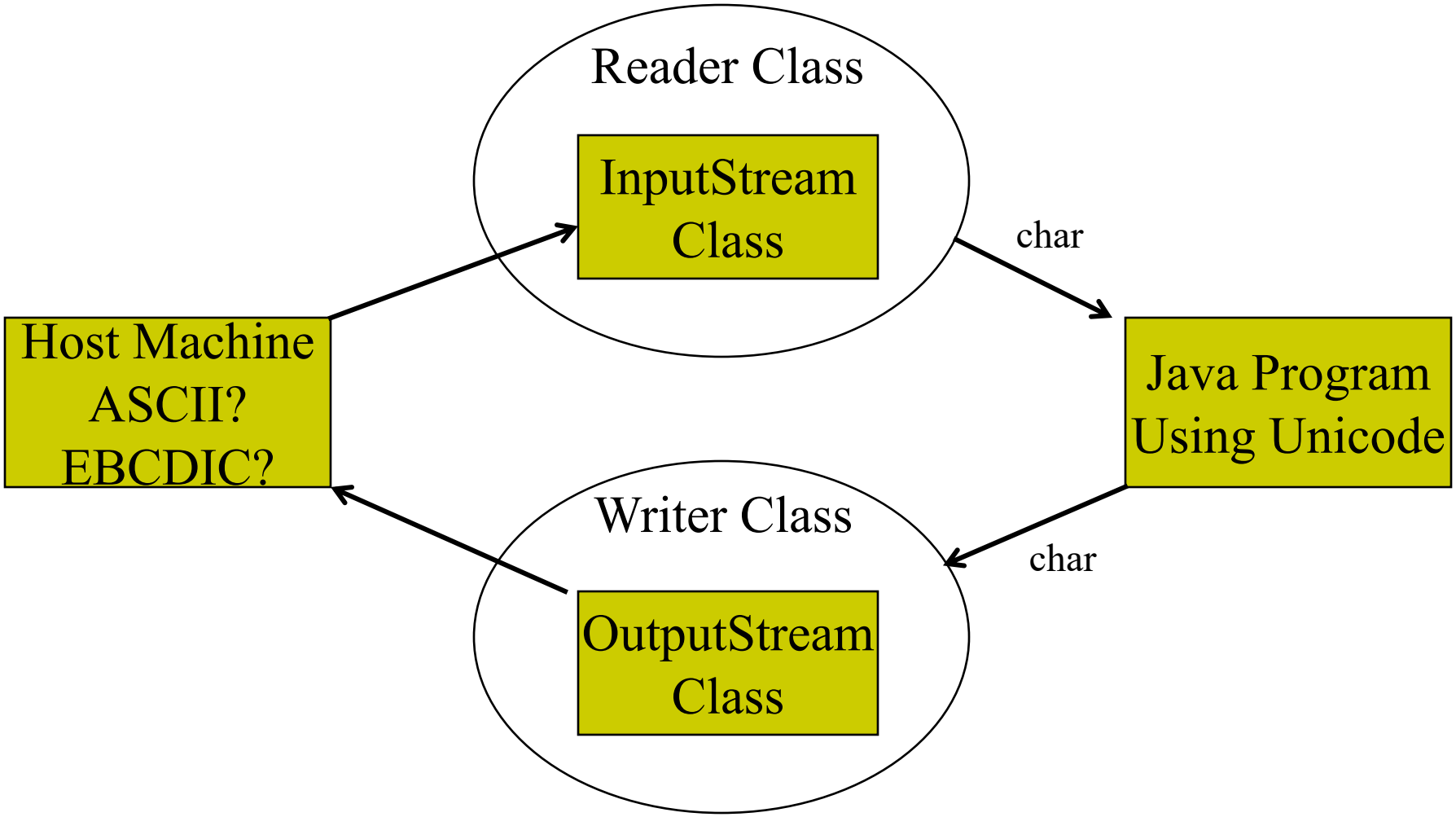
#2: Java's Internal Characters

- 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**.
- 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) {
    }
}
```

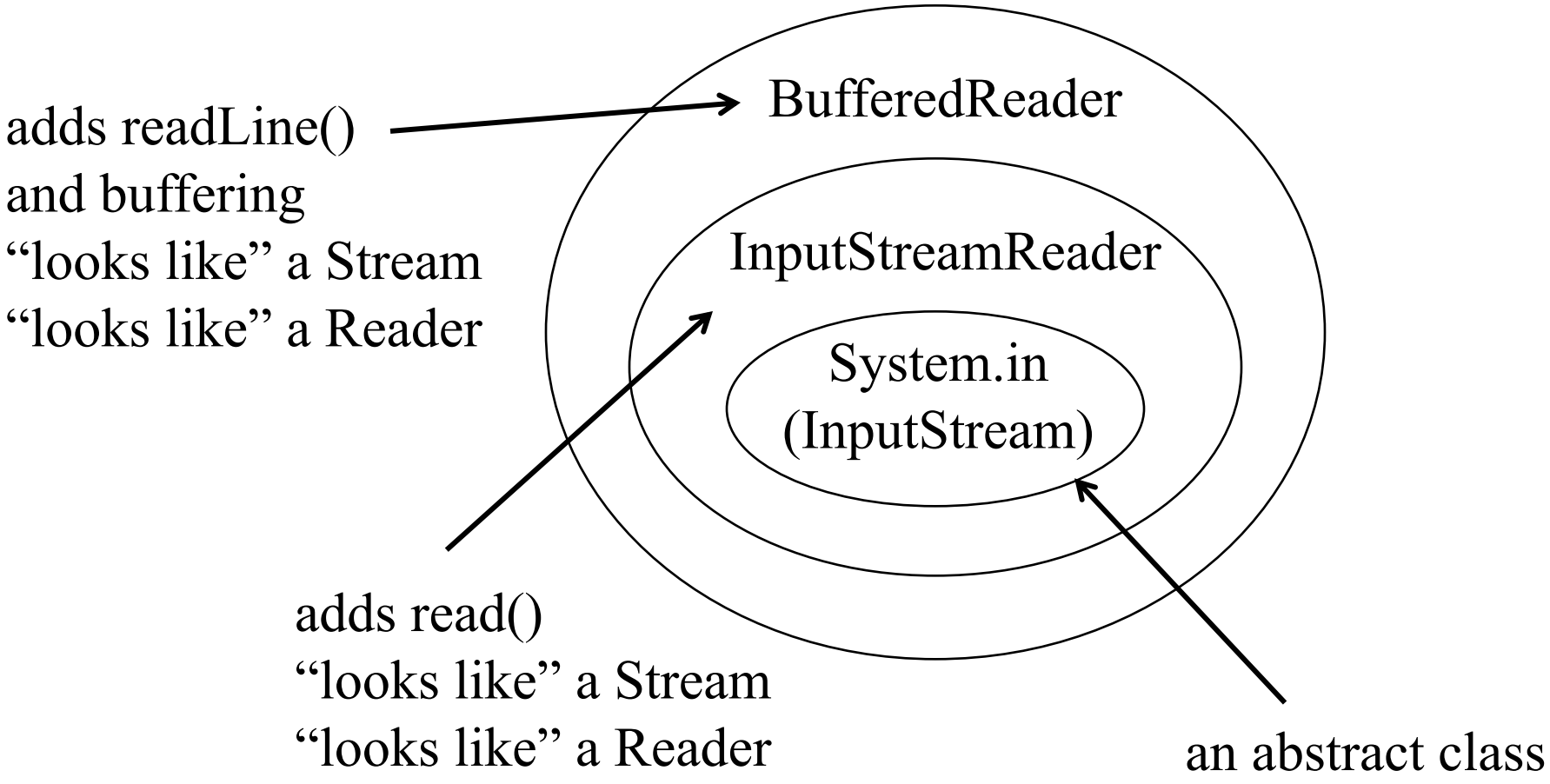
InputStreamReader

- 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 / File**
- Its constructor takes a string containing the file pathname.

Reading From a File

```
import java.io.File;
import java.io.FileInputStream;
public class fileReader {
public static void main(String[] args) {
   File f = new File("final/src/test.txt");
   try {
        FileInputStream fis = new FileInputStream(f);
        int c;
        while ((c = fis.read()) != -1) {
               System.out.println((char) c);
        }
        } catch (Exception e) {
            e.printStackTrace();
}} }
```

Reading From a File (cont.)

- = -1 Reached the end of the file.
- Path file
- The **read()** method can throw an **IOException**, and the **FileInputStream** constructor can throw a **FileNotFoundException**

  meant no file found, check path, check file reading

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

https://docs.oracle.com/javase/8/docs/api/java/io/File.html

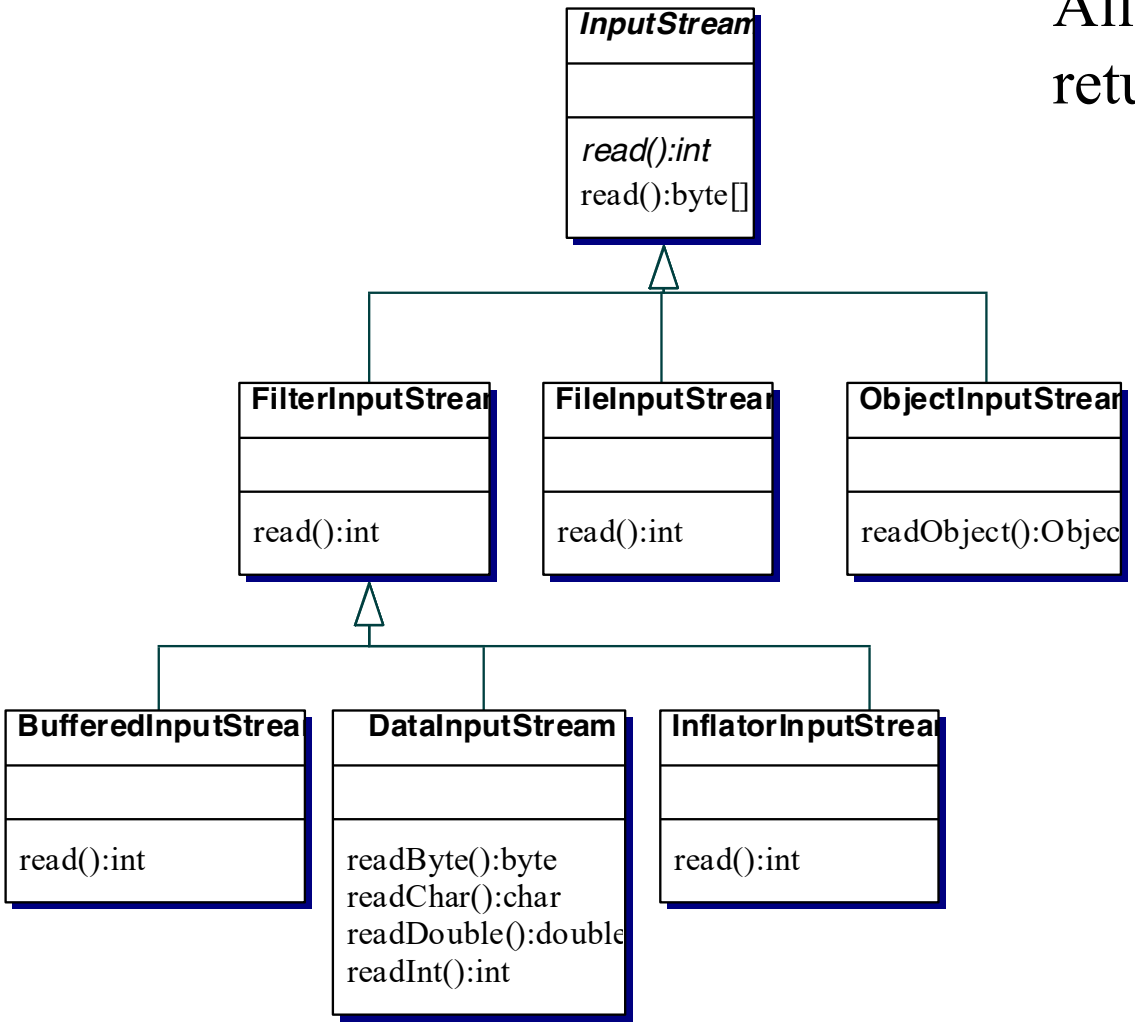
- canRead()
- canWrite()
- exists()
- getParent()
- isDirectory()
- isFile()
- lastModified()
- length()

File Methods for Modifying

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

More on Input

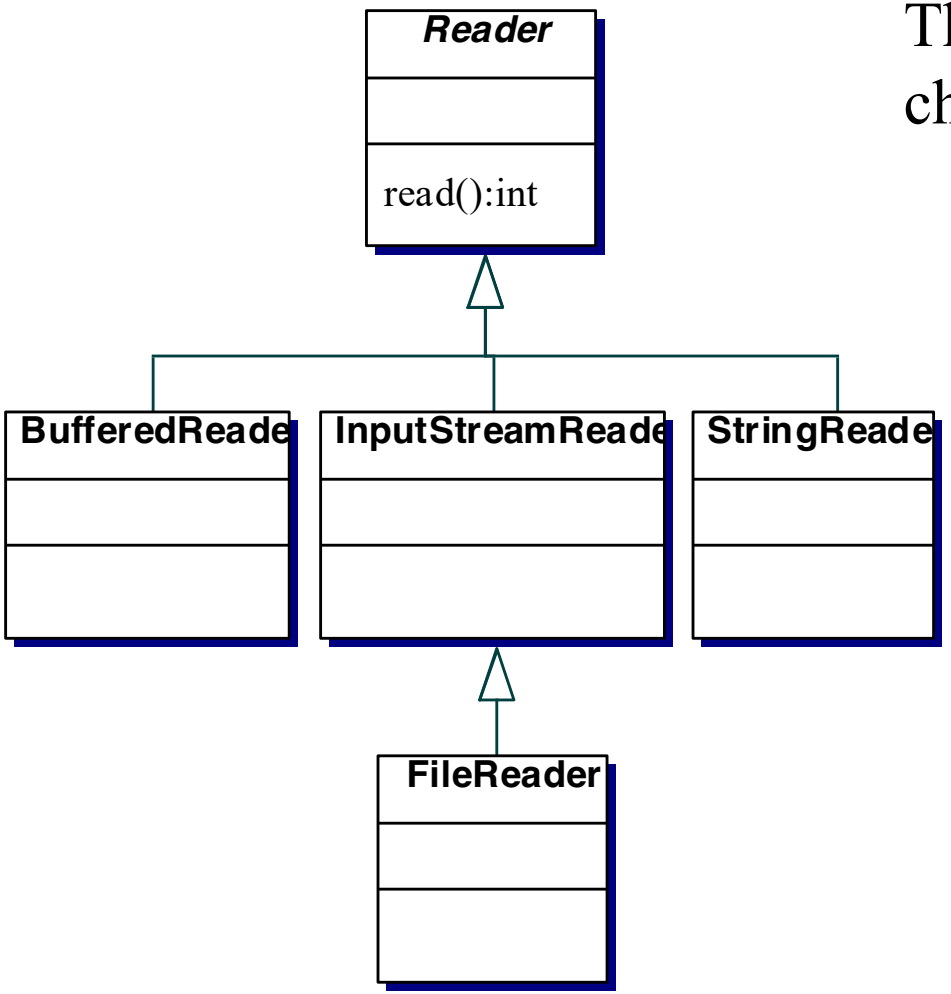
All of these return bytes!



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**

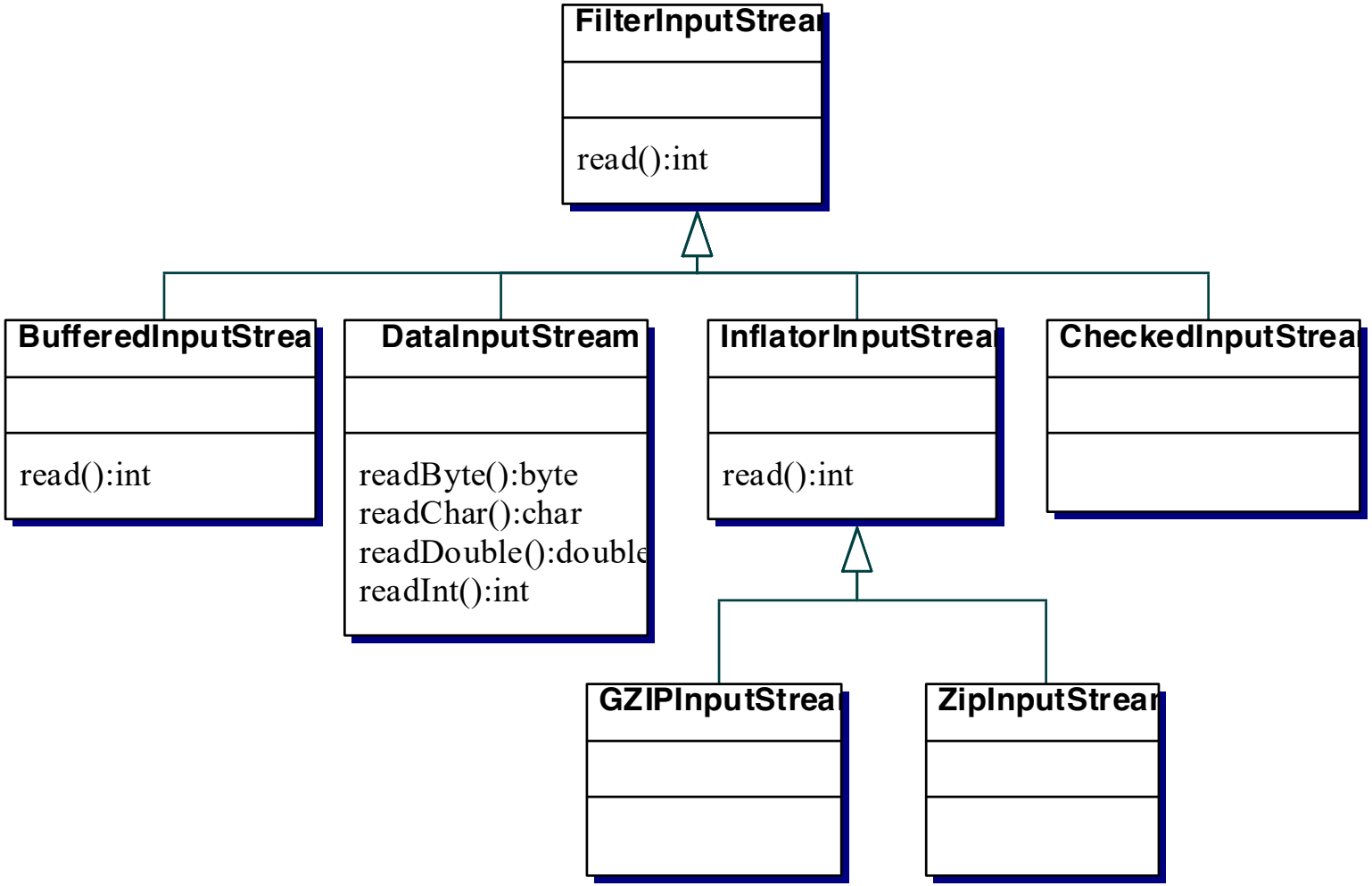
PrintWriter

```
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