CSE2101: Object Oriented Programming-II (Java)

Lecture 12

CSE2101-OOP in Java

Input / Output

I/O mechanisms

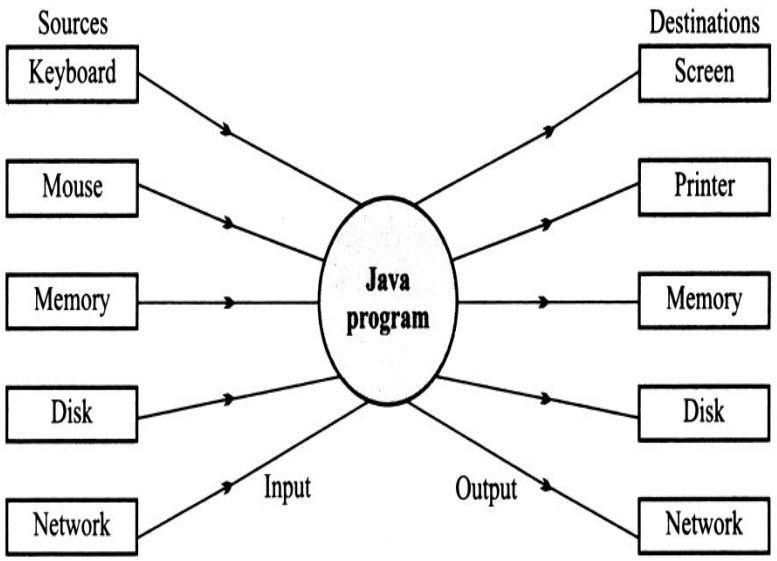
- All programs need I/O to communicate with the outside world
- I/O can be textual, graphical, through sensors, external devices, etc
- For desk computers: usually screen, keyboard, network, file system
- For embedded systems: all sorts of sensors and machinery

Output Mechanisms

- Text to terminal, printer, etc.
- Data to file.
- Data to other programs, network, etc.
- Windows, graphics (GUI).

I/O and Data Movement

- The flow of data into a program (input) may come from different devices such as keyboard, mouse, memory, disk, network, or another program.
- The flow of data out of a program (output) may go to the screen, printer, memory, disk, network, another program.
- Both input and output share a certain common property such as unidirectional movement of data – a sequence of bytes and characters and support to the sequential access to the data.



Relationship of Java program with I/O devices

The Java IO package

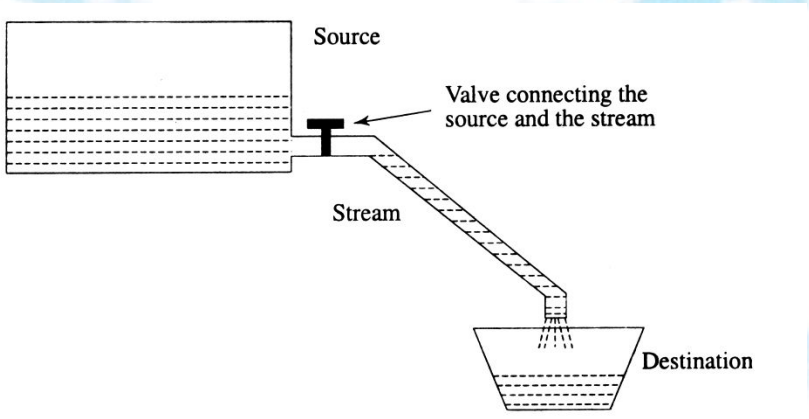
- Provided as part of the JDK
  - Contains classes and interfaces to use when dealing with different types of input and output
  - includes
    - 10 Interfaces
    - 50 Classes
    - 16 Runtime Exceptions

Java I/O mechanisms

- In Java, IO is done through a set of classes
- For I/O: no new language construct (just new classes)
- The classes provide several interfaces to ***streams*** and other IO concepts
- First, we look at streams.

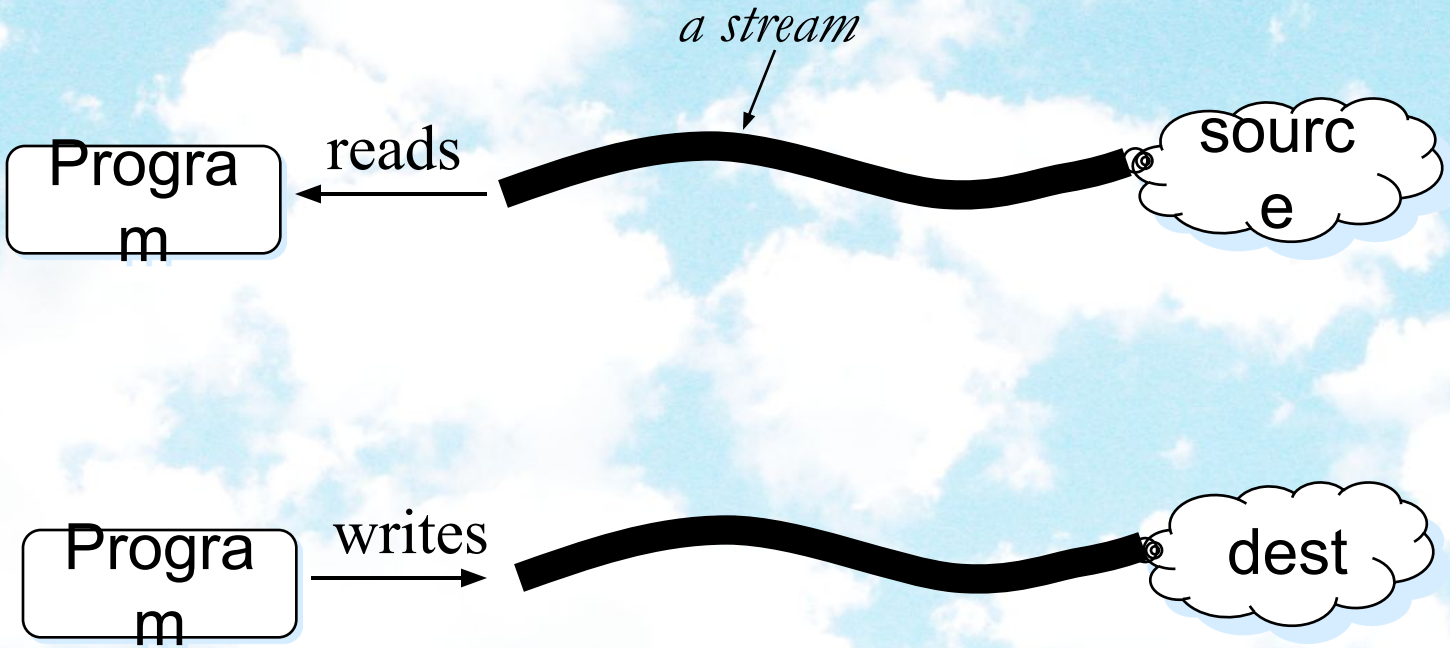
Streams

- Java Uses the concept of Streams to represent the ordered sequence of data, a common characteristic shared by all I/O devices.
- Streams presents a uniform, easy to use, object oriented interface between the program and I/O devices.
- A stream in Java is a path along which data flows (like a river or pipe along which water flows).



Conceptual view of a stream

Streams



Java idioms

Writing

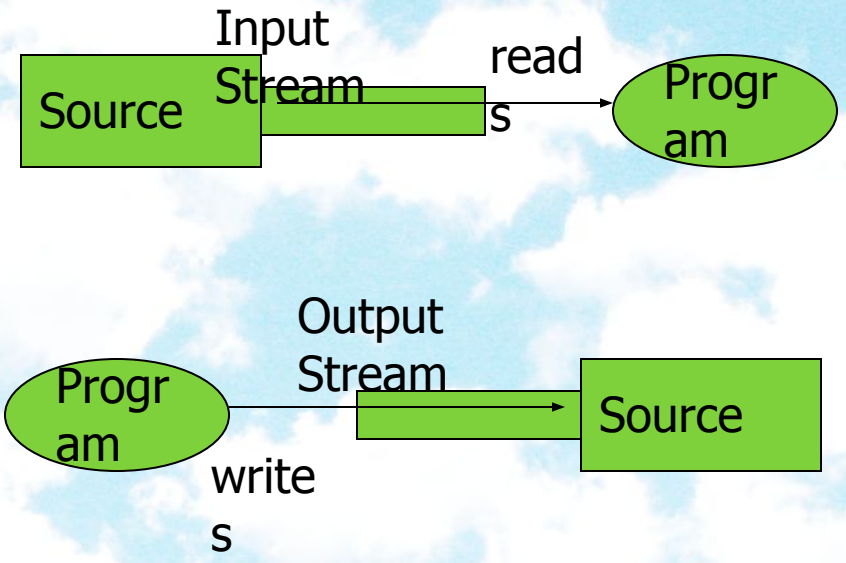
```
open a stream
while(more information)
   write information to stream
close the stream
```

Reading

```
open a stream
while(more information)
   read information from stream
close the stream
```

Stream Types

- The concepts of sending data from one stream to another (like a pipe feeding into another pipe) has made streams powerful tool for file processing.
- Connecting streams can also act as filters.
- Streams are classified into two basic types:
  - Input Steam
  - Output Stream



Java Stream Classes

- Input/Output related classes are defined in java.io package.
- Input/Output in Java is defined in terms of streams.
- A *stream* is a sequence of data, of no particular length.
- Java classes can be categorised into two groups based on the data type one which they operate:
  - *Byte streams*
  - *Character Streams*

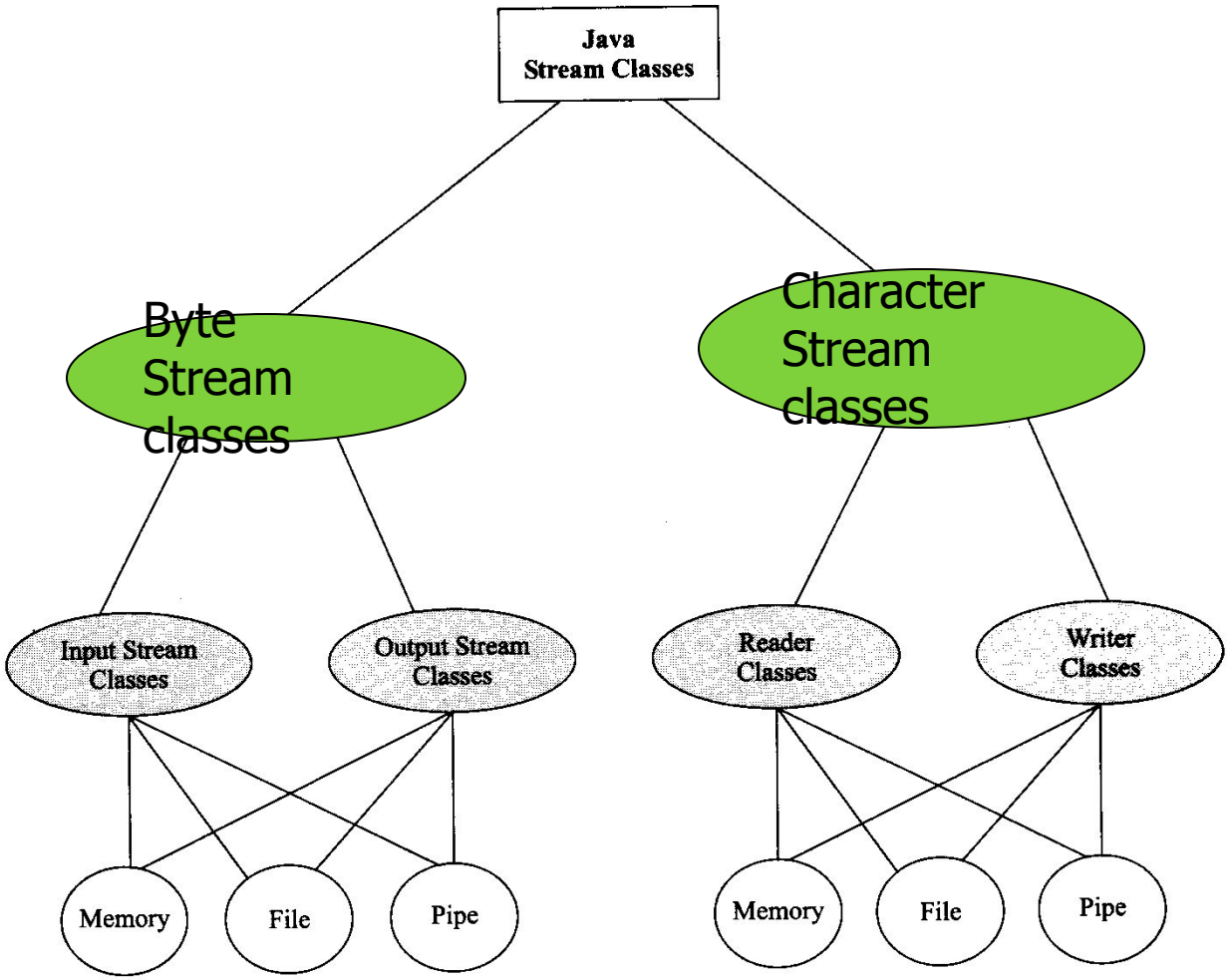
Text Data Vs Binary Data

- Text files are human and text editor readable
- Binary files are program readable

Streams

Byte Streams	Character streams
Operated on 8 bit (1 byte) data.	Operates on 16-bit (2 byte) unicode characters.
Input streams/Output streams	Readers/ Writers

Classification of Java Stream Classes

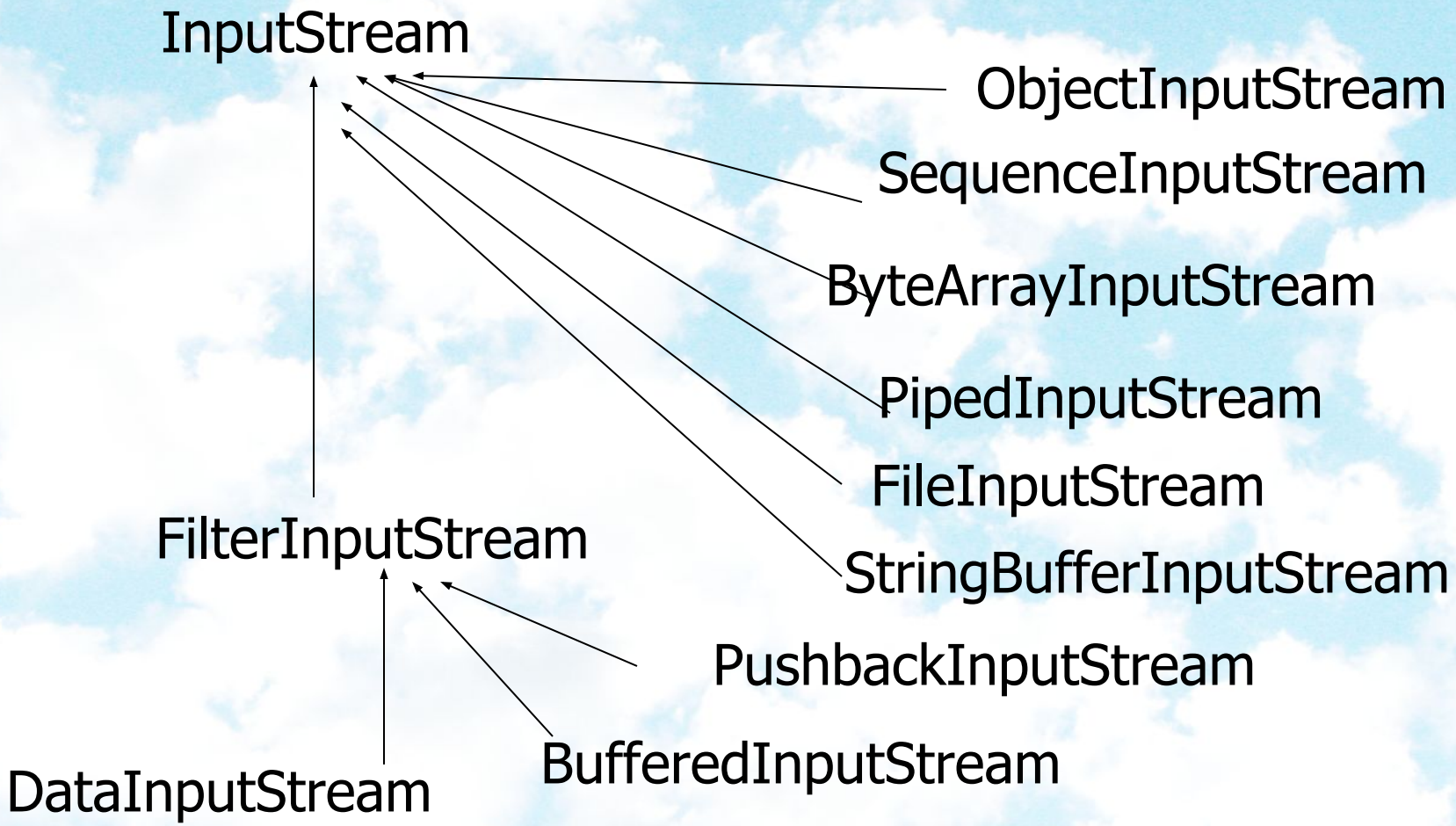


Classification of Java stream classes

Byte Streams Vs Reader/writer

- Reader/writer (characters)
  - Read,write characters
  - Are used for text input/output
  - Are preferred for text I/O
- Streams (bytes)
  - Read/write bytes
  - Can be used for any data type
  - Are used for "serialisation"

Byte Input Streams



Byte Input Streams - operations

public abstract int read()	Reads a byte and returns as a integer 0-255
public int read(byte[] buf, int offset, int count)	Reads and stores the bytes in buf starting at offset. Count is the maximum read.
public int read(byte[] buf)	Same as previous offset=0 and length=buf.length()
public long skip(long count)	Skips count bytes.
public int available()	Returns the number of bytes that can be read.
public void close()	Closes stream

Java I/O - Using InputStreams

- I/O in Java:

```
InputStream in = new
  FileInputStream("c:\\temp\\myfile.txt");
int b = in.read();
//EOF is signalled by read() returning -1
while (b != -1)
{
  //do something…
  b = in.read();
}
in.close();
```

Java I/O - Using InputStreams

- Basic pattern for output is as follows:

```
Open a stream
While there's data to write
  Write the data
Close the stream
```

Reading: an example

```
BufferedReader reader =
 new BufferedReader(new InputStreamReader(System.in));
try
{
   while(true)
   {
      String line = reader.readLine();
      System.out.println("the line was: " + line);
   }
}
catch(IOException exc)
{
   // an IO error occurred
}
```

Reading (1)

Example from KeyboardReader Class

```
public KeyboardReader()
{
reader = new BufferedReader(new
       InputStreamReader(System.in));
}
```

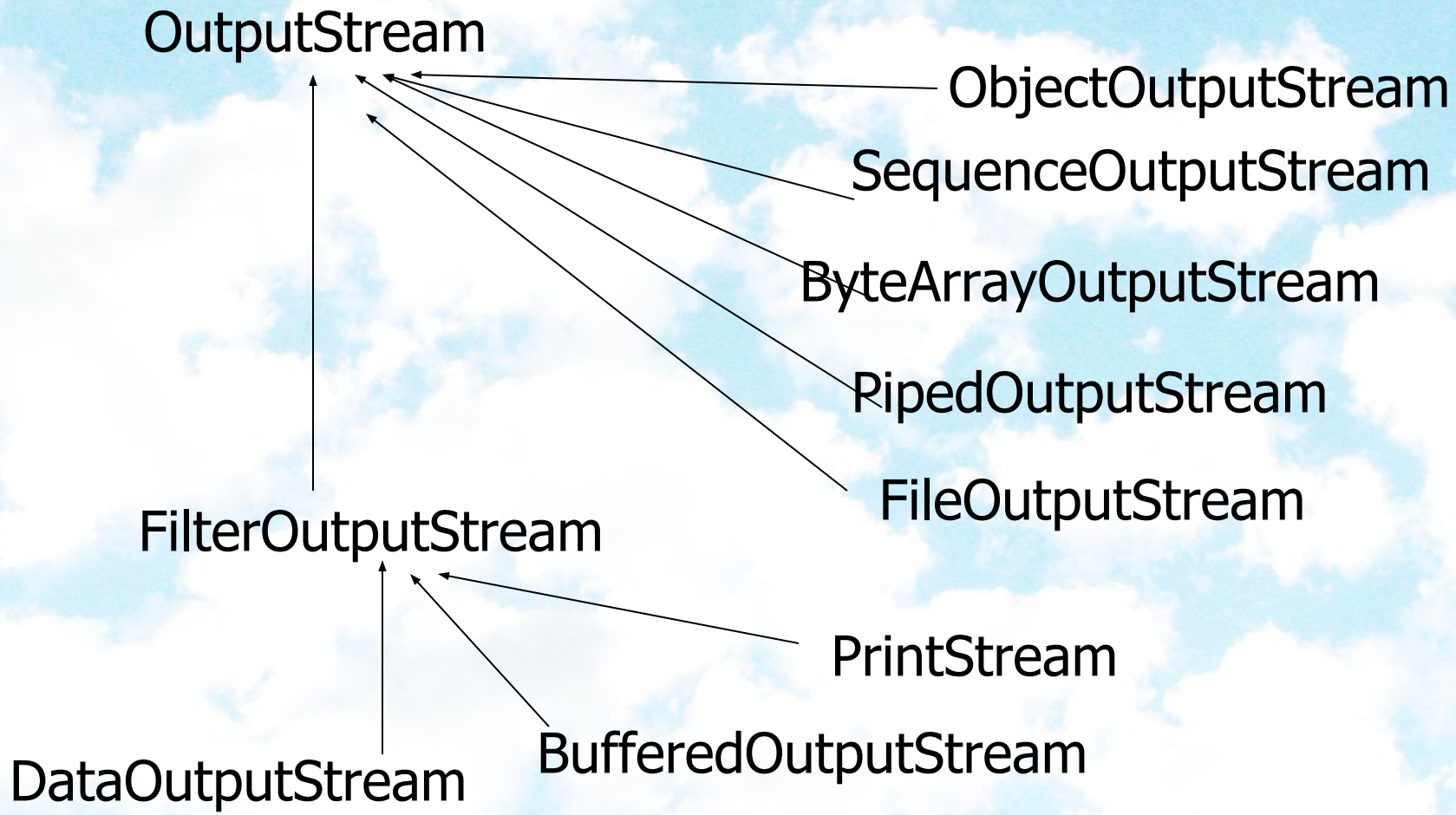
KeyboardReader Class Methods

```
public String readString()
{
 inputLine = "";
 try
   {
    inputLine = reader.readLine();
   }
   catch(java.io.IOException anException)
  {
  // code not shown to save space
   }
            return inputLine;
 }
```

KeyboardReader Class Methods


```
public int readInteger()
{
  inputInteger = 0;
  try {
    inputLine = reader.readLine();
    inputInteger = Integer.parseInt(inputLine);
    }
  catch(java.io.IOException anException) {}
  catch(NumberFormatException anException){}
  return inputInteger;
}
```


Byte Output Streams



Byte Output Streams - operations

public abstract void write(int b)	Write *b* as bytes.
public void write(byte[] buf, int offset, int count)	Write *count* bytes starting from *offset* in *buf.*
public void write(byte[] buf)	Same as previous *offset=0* and *count = buf.length()*
public void flush()	Flushes the stream.
public void close()	Closes stream

PrintStream methods

```
public void print(boolean b)
public void print(char c)
public void print(double d)
public void print(float f)
public void print(int i)
public void print(long l)
public void print(String s)
public void print(Object o)
```

```
public void println(boolean b)
public void println (char c)
public void println (double d)
public void println (float f)
public void println (int i)
public void println (long l)
public void println (String s)
public void println (Object o)
```

Byte Output Stream - example

- Read from standard in and write to standard out

```
import java.io.*;

class ReadWrite {
        public static void main(string[] args)
                throws IOException
        {
                int b;
                while (( b = System.in.read()) != -1)
                {
                        System.out.write(b);
                }

}
```

System.out

- You have already used an io class implicitly when you write information to the screen.

- System.out refers to a data attribute of the System class.
  - This attribute is an instance of the class **PrintStream**.
  - This class is used for output only.

- The PrintStream class includes a method, **println**, that prints a string to the specified output.
  - We don't know where this output comes from!
  - We don't need to know – we just need to know where it goes to (i.e. the screen).

Handling User Input

- To get input from the user, we use another data attribute of the System class:

```
System.in
```

- This attribute is an instance of the **InputStream** class.
  - The InputStream class is not easy to use (it works only with bytes).
  - To make things easier, we can wrap the input stream in other io classes.

- In particular, we use two classes:
  - **InputStreamReader**. This class converts the byte stream into a character stream (this is much more useful – honestly!).
  - **BufferedReader**. This class buffers data from the enwrapped stream, allowing smoother reading of the stream (it doesn't wait for the program to ask for the next character in the stream).

Handling User Input

- To wrap an io object, we create a new io object, passing the old io object into the constructor.
  - For example, to wrap System.in within an InputStreamReader, we write:

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

- To use the BufferedReader (which, remember, is an optimisation), we wrap the InputStreamReader object we created above.
  - For example:

```
new BufferedReader(new InputStreamReader(System.in))
```

- Once we have a BufferedReader, we can use a nice method it provides called readLine() which reads a line of text!
  - This makes life much easier!

Handling User Input

- Now that we have a way of reading a line of text from the System input stream (i.e. the keyboard), how do we use it?

```
public String readString() {
    BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));
    String line = null;
    try {
        line = in.readLine();
        while (line == null) {
            line = in.readLine();
        }
    } catch (IOException ie) {
        System.out.println("The following problem occurred " +
                           "when reading input: " + ie);
   }
   return line;
}
```

File Processing

- So far we have used variables and arrays for storing data inside the programs. This approach poses the following limitations:
  - The data is lost when variable goes out of scope or when the program terminates. That is data is stored in temporary/main memory is released when program terminates.
  - It is difficult to handle large volumes of data.
- We can overcome this problem by storing data on secondary storage devices such as floppy or hard disks.
- The data is stored in these devices using the concept of Files and such data is often called persistent data.

File Processing (cont.)

- Storing and manipulating data using files is known as file processing.
- Reading/Writing of data in a file can be performed at the level of bytes, characters, or fields depending on application requirements.
- Java also provides capabilities to read and write class objects directly. The process of reading and writing objects is called object serialisation.

The File Class

java.io package includes:

- **classes used to extract data from files**
- **classes used to place data within files**
- **class used to provide Java programs with mechanism for interacting with the file system: the File class**

Class File

java.lang.Object

|

+--**java.io.File**

All Implemented Interfaces:

Comparable, Serializable

The File Class

- Represents the pathname of a particular File, or directory
  - Ex: \Users\Smith\Java\read.me
  - Ex: read.me (in the current directory)
- Uses
  - Listing existing files and directories
  - Creating new files and directories
  - Check for the existence of files
  - …

Creating a new File: Example

File (String pathname)

Creates a new File instance by converting the given pathname string into an abstract pathname.

Example:

```
String fileName =
  "C:/fit2034.txt"
File file = new File(fileName);
```


Listing a Directory

```
public void listADirectory(String dirName)
    {
        File dirList = new File(dirName);
        String[] dList = dirList.list();
        for(int i = 0; i < dList.length; i++)
        System.out.println(dList[i]);
    }
```

String[] list() : Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

FileReader interface

A class for reading character files

class FileReader (extends InputStreamReader):

```
public FileReader(File file)
   throws FileNotFoundException
```

Creates a new FileReader, given the File to read from

```
public FileReader(FileDescriptor fd)
```

Creates a new FileReader, given the FileDescriptor to read from

```
public FileReader(String fileName)
   throws FileNotFoundException
```

Creates a new FileReader, given the name of the file to read from

Creating a file reader

The code used to create a FileReader:

```
inputFile = new File(fromFile);
try
{
   FileReader in = new FileReader(inputFile);
   . . .
}
catch (FileNotFoundException fNF)
{
   System.out.println("File Not Found");
}
```


FileReader methods (Inherited from Reader)

public int **read**() throws IOException

Read a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.

- *Reads each character as an integer*
- *Returns the character read, as an integer, or –1 when the end of the stream is reached.*

FileReader Methods (Inherited from Reader)

`abstract void close()`

Close the stream.

`void mark(int readAheadLimit)`

Mark the present position in the stream.

`boolean markSupported()`

Tell whether this stream supports the mark() operation. The default implementation always returns false.

`Int read(char[] cbuf)`

Read characters into an array. Cbuf is the destination buffer. Return the number of characters read, or –1 when the end of the stream is reached.

FileReader Methods (Inherited from Reader)

`abstract int read(char[] cbuf, int off, int len)`

Read characters into a portion of an array. Return the number of characters read…

`boolean ready()`

Tell whether this stream is ready to be read.

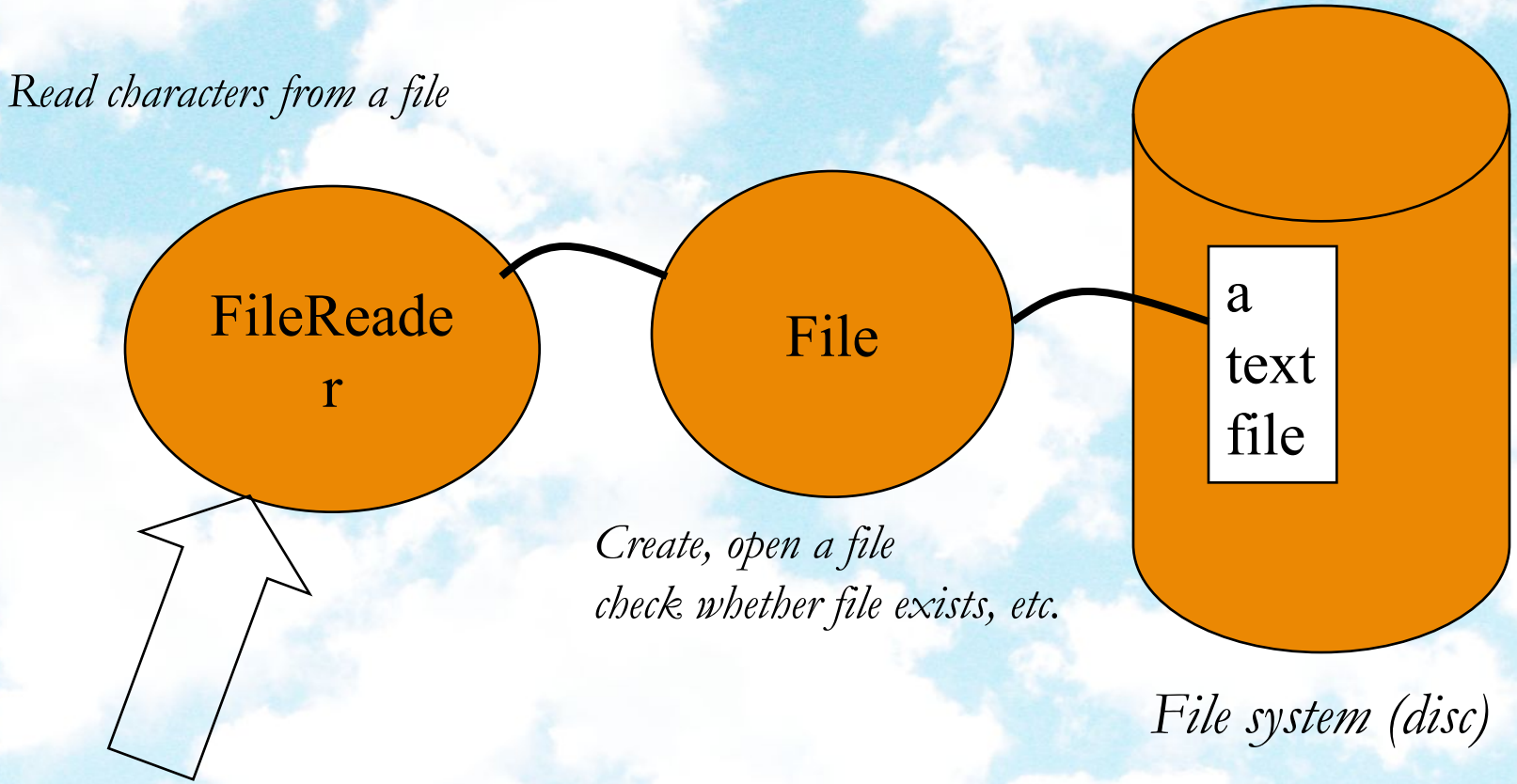
`Void reset()`

Reset the stream. If the stream has been marked, then attempt to reposition it at the mark.

`long skip(long n)`

Skip characters. N is the number of characters to skip

The file reader structure



Casting to char

```
public void processFile(String fileName)
    throws IOException
{
    File inputFile = new File(fileName);
    FileReader fr = new FileReader(inputFile);
    int ch;

    while ((ch = fr.read()) != -1)
       processCharacter((char)ch);

    fr.close();
}

public void processCharacter(char aChar)
{
    ...
}
```

cast needed!

FileWriter interface

Class for writing character files

class **FileWriter**
  extends OutputStreamWriter

FileWriter (File file)
  Constructs a FileWriter object given a File object.

FileWriter (FileDescriptor fd)
  Constructs a FileWriter object associated with a file descriptor.

FileWriter interface (2)

FileWriter(File file, boolean append)
Constructs a FileWriter object given a File object. If the boolean is true, then bytes will be written to the end of the file rather than the beginning.

FileWriter(String fileName)
Constructs a FileWriter object given a file name.

FileWriter(String fileName, boolean append)
Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written. If boolean is true, then the data will be written to the end of the file rather than the beginning

FileWriter Methods (Inherited from Writer)

- Reflect those found in `FileReader`

abstract void **close**()
Close the stream, flushing it first.

abstract void **flush**()
Flush the stream.

void **write**(char[] cbuf)
Write an array of characters.

abstract void **write**(char[] cbuf, int off, int len)
Write a portion of an array of characters.

void **write**(int c)
Write a single character.

void **write**(String str)
Write a string.

void **write**(String str, int off, int len)
Write a portion of a string.

An example: copyFile

```
public void copyFile(String fromFile, String toFile)
    throws IOException
{
    File inputFile = new File(fromFile);
    File outputFile = new File(toFile);

    FileReader in = new FileReader(inputFile);
    FileWriter out = new FileWriter(outputFile);
    int ch;

    while ((ch = in.read()) != -1)
       out.write(c);
    in.close();
    out.close();
}
```

Thank you