JAVA PROGRAMMING

Week 5: Java I/O

Lecturer:

- HO Tuan Thanh, M.Sc.

Plan

1. Java's I/O is built upon streams
2. Byte stream classes
3. Reading and writing binary data
4. Character stream classes

Overview

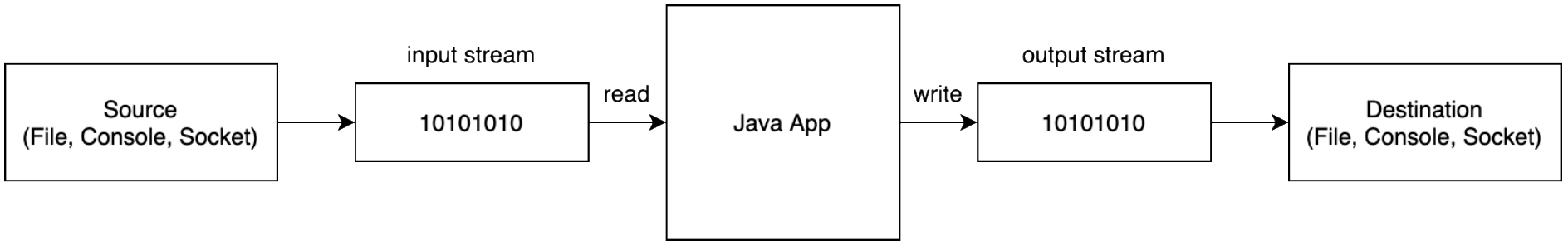
3

Java's I/O is built upon streams [1]

- Java programs perform I/O through streams.
- An I/O stream is an abstraction that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices they are linked to differ → the same I/O classes and methods can be applied to different types of devices.

Java's I/O is built upon streams [2]

- Java implements I/O streams within class hierarchies defined in the java.io package.



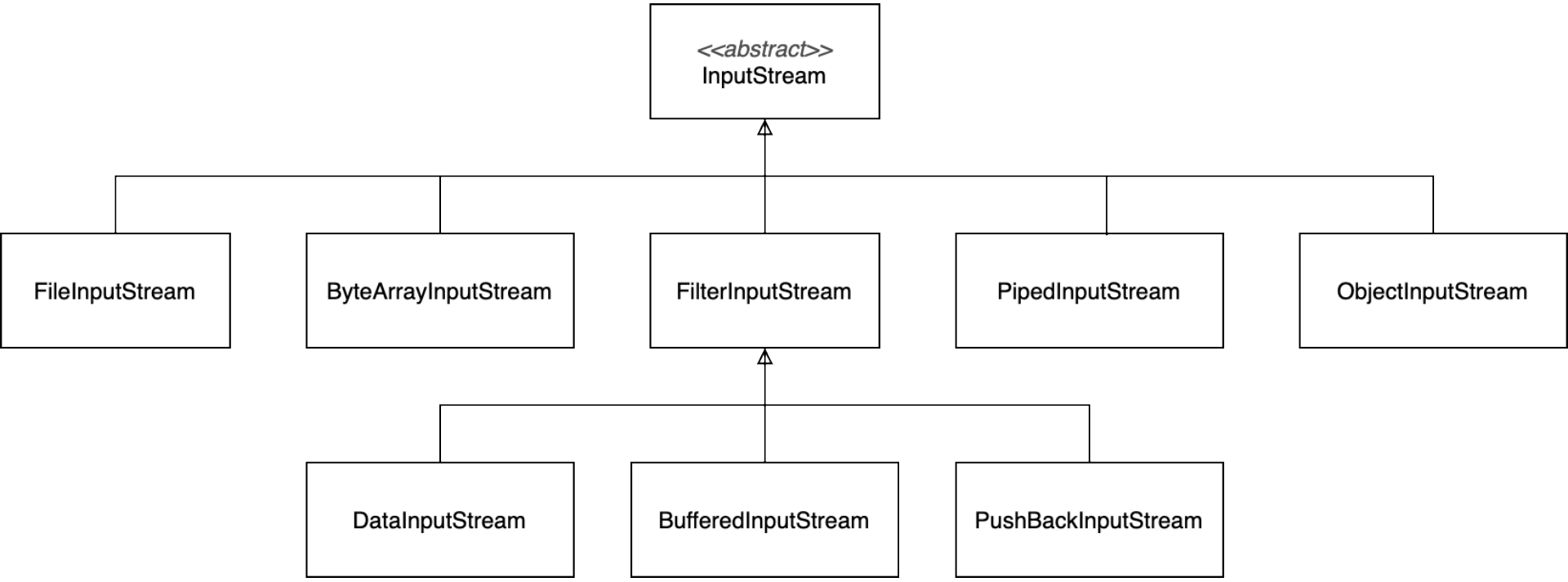
Byte streams and character streams

- Two types of I/O streams: **byte and character.**
- Byte streams provide a convenient means for handling input and output of bytes.
  - Can be used when reading or writing binary data.
  - Are especially helpful when working with files.
- Character streams are designed for handling the input and output of characters.
  - They use Unicode → can be internationalized.
  - In some cases, character streams are more efficient than byte streams.
- **Notice**: At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

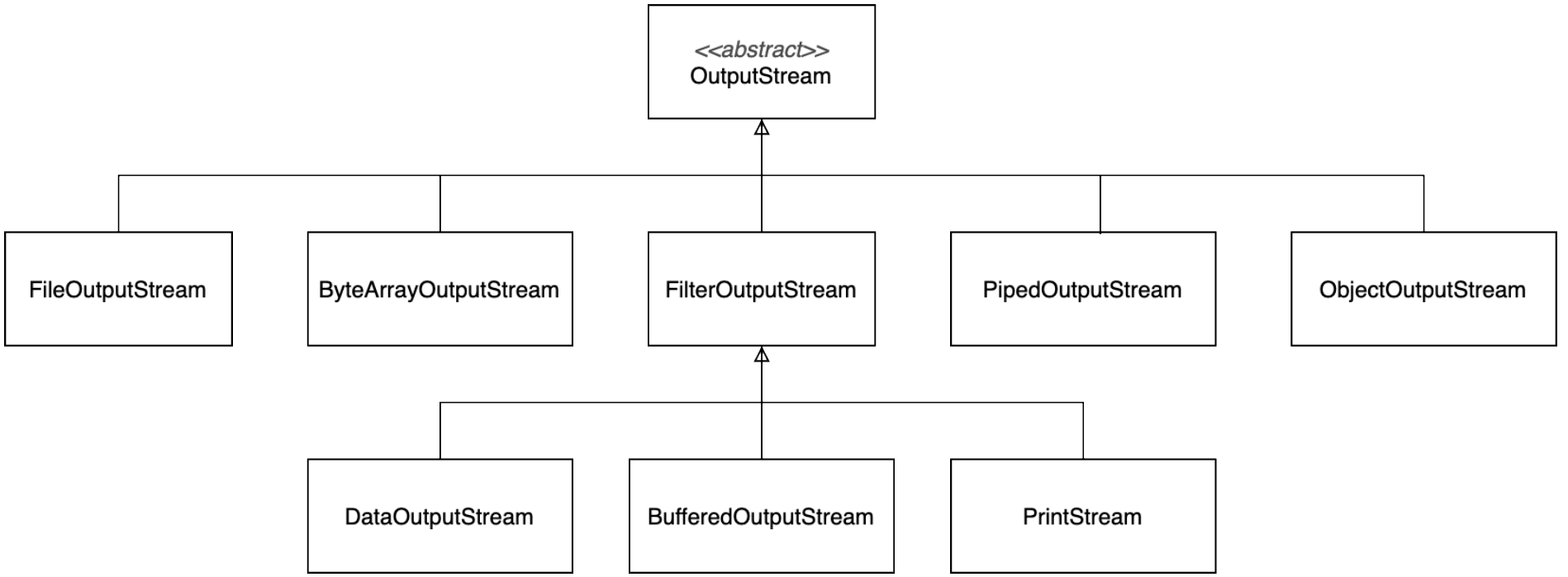
Byte stream classes

Byte Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that allows bytes to be returned to the stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Byte - InputStream



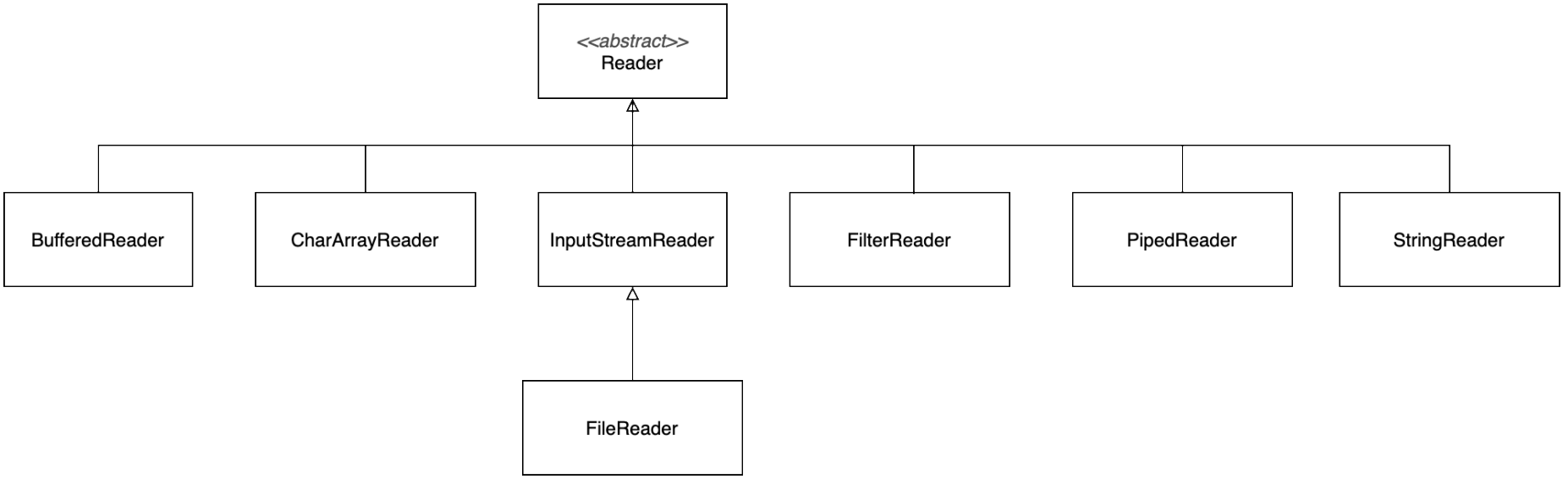
Byte - OutputStream



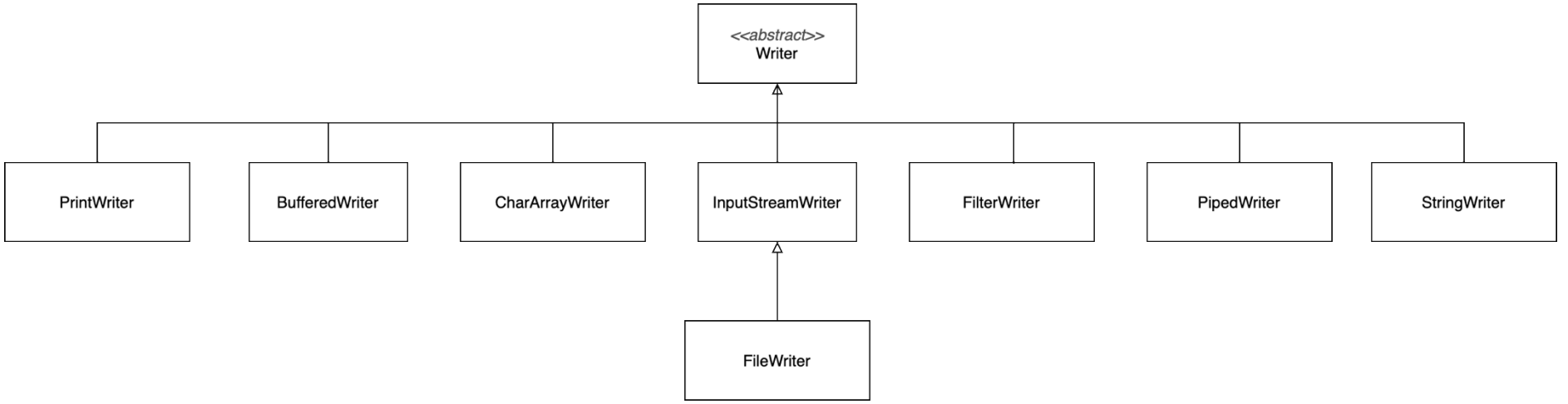
Character stream classes

Character Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Character - Reader



Character - Writer



Pre-defined streams

Pre-defined streams [1]

- Class System, defined in java.lang, encapsulates several aspects of the run time environment.
  - It contains three predefined stream variables: in, out, and err.
  - These fields are declared as public, final, and static → can be used by any other part of your program and without reference to a specific System object.
- System.out refers to the standard output stream.
  - Default: console.
- System.in refers to standard input stream.
  - Default: keyboard.
- System.err refers to the standard error stream.
  - Default: console.

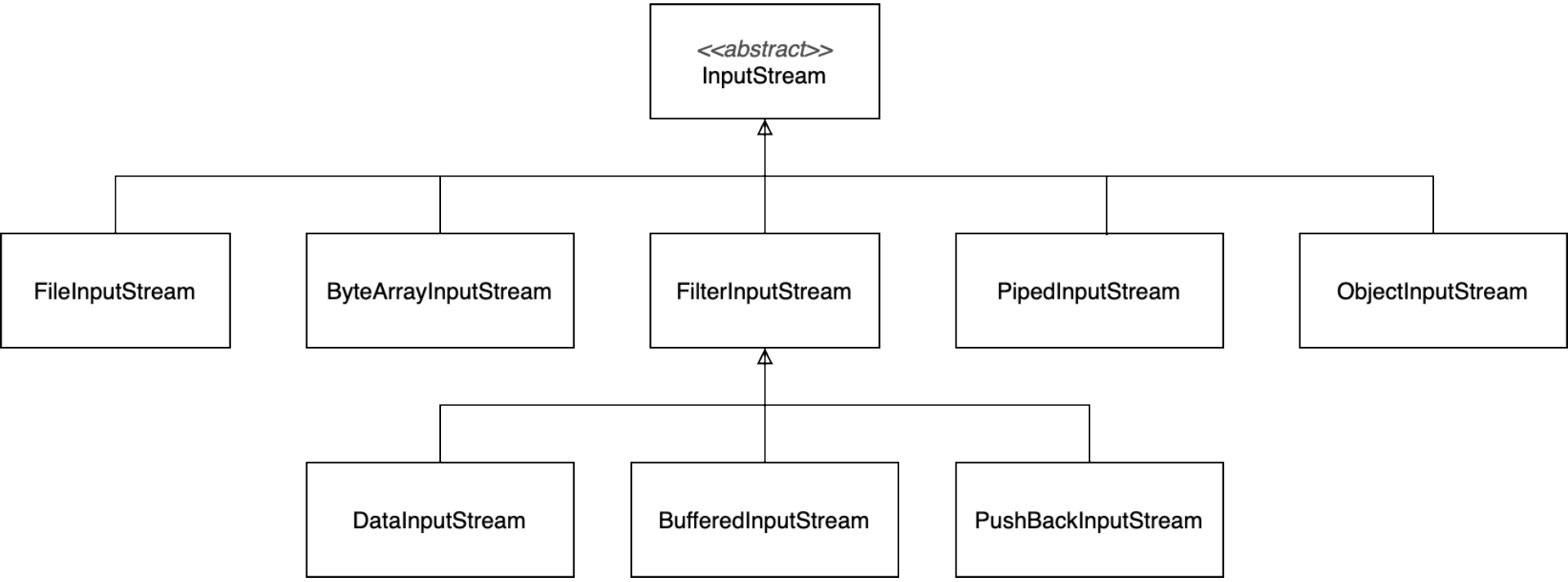
Pre-defined streams [2]

- These streams can be redirected to any compatible I/O device.
- System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream.
  - These are byte streams, even though they are typically used to read and write characters from and to the console.
  - Reason: the pre-defined streams were part of the original specification for Java, which did not include the character streams.

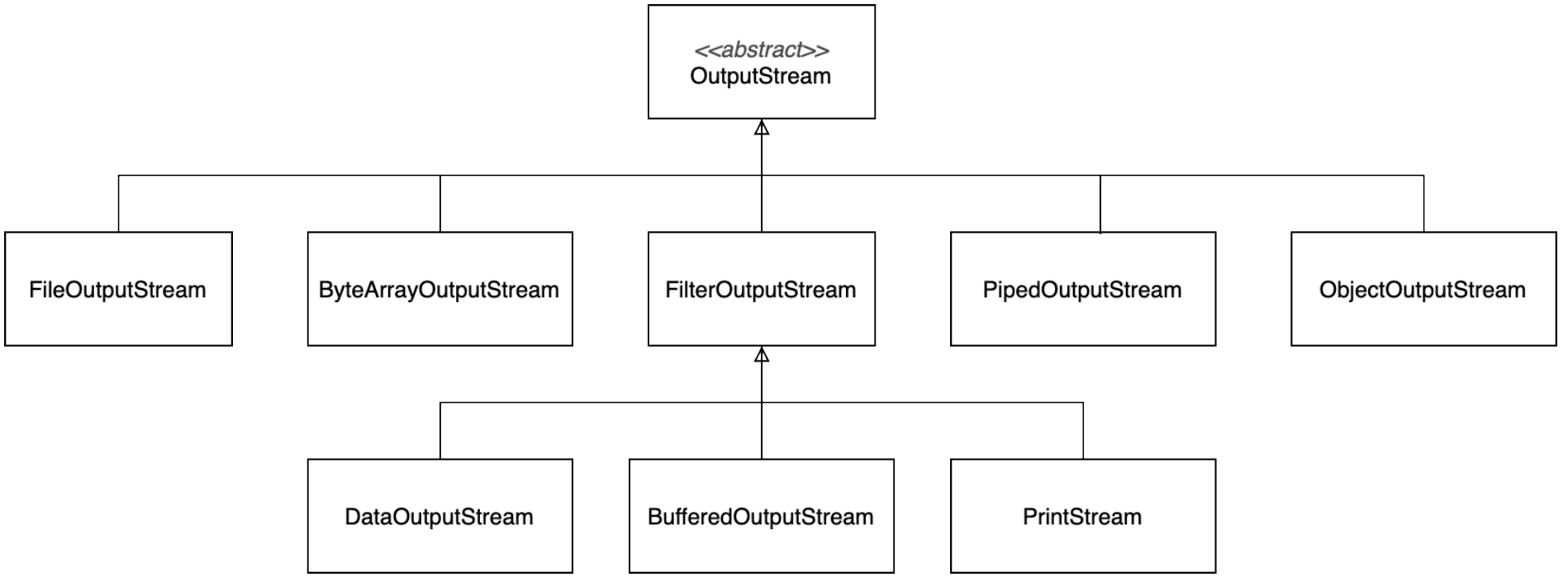
Byte streams

16

Byte - InputStream



Byte - OutputStream



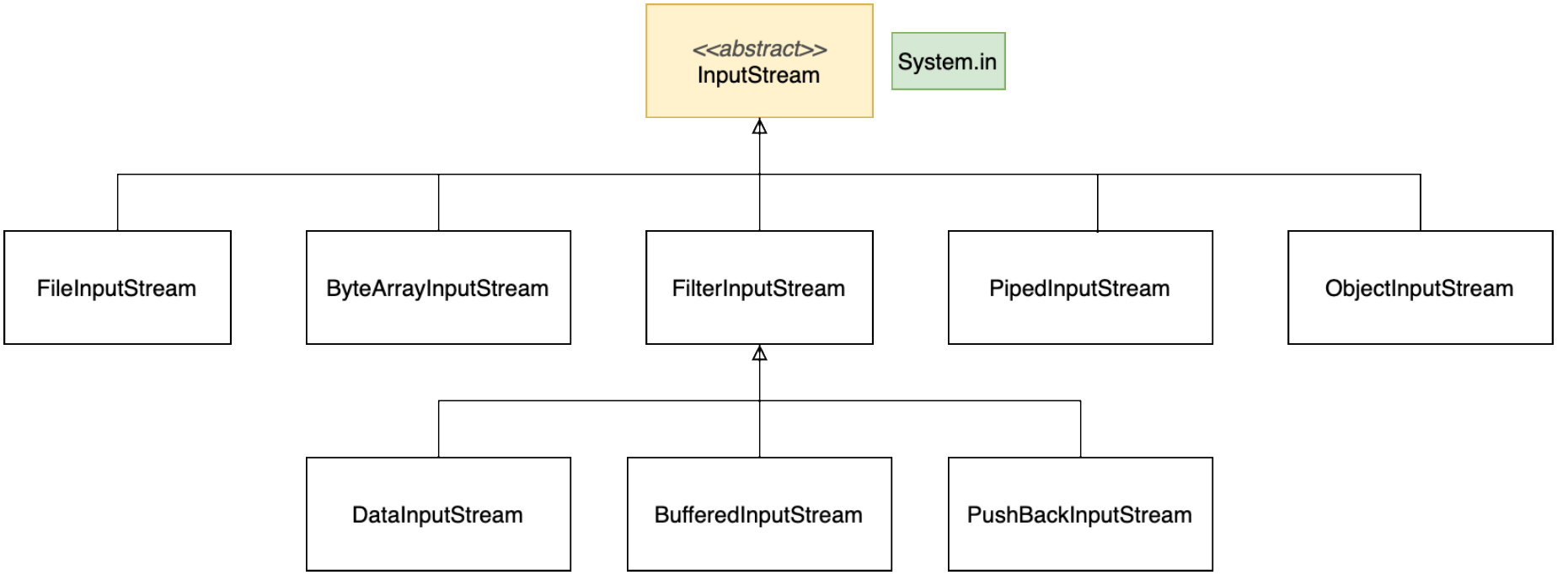
Methods defined by InputStream

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Subsequent read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns true if **mark( )**/**reset( )** are supported by the invoking stream.
static InputStream nullInputStream( )	Returns an open, but null stream, which is a stream that contains no data. Thus, the stream is always at the end of the stream and no input can be obtained. The stream can, however, be closed. (Added by JDK 11.)
int read( )	Returns an integer representation of the next available byte of input. −1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. −1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. −1 is returned when an attempt is made to read at the end of the stream.
byte[ ] readAllBytes( )	Reads and returns, in the form of an array of bytes, all bytes available in the stream. An attempt to read at the end of the stream results in an empty array.
byte[ ] readNBytes(int *numBytes*)	Attempts to read *numBytes* bytes, returning the result in a byte array. If the end of the stream is reached before *numBytes* bytes have been read, then the returned array will contain less than *numBytes* bytes. (Added by JDK 11.)
int readNBytes(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. An attempt to read at the end of the stream results in zero bytes being read.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.
long transferTo(OutputStream *outStrm*)	Copies the contents of the invoking stream to *outStrm*, returning the number of bytes copied.

Methods defined by OutputStream

Method	Description
void close( )	Closes the output stream. Subsequent write attempts will generate an **IOException**.
void flush( )	Causes any output that has been buffered to be sent to its destination. That is, it flushes the output buffer.
static OutputStream nullOutputStream( )	Returns an open, but null output stream, which is a stream to which no output is written. The stream can, however, be closed. (Added by JDK 11.)
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

Reading console input [1]



Reading console input [2]

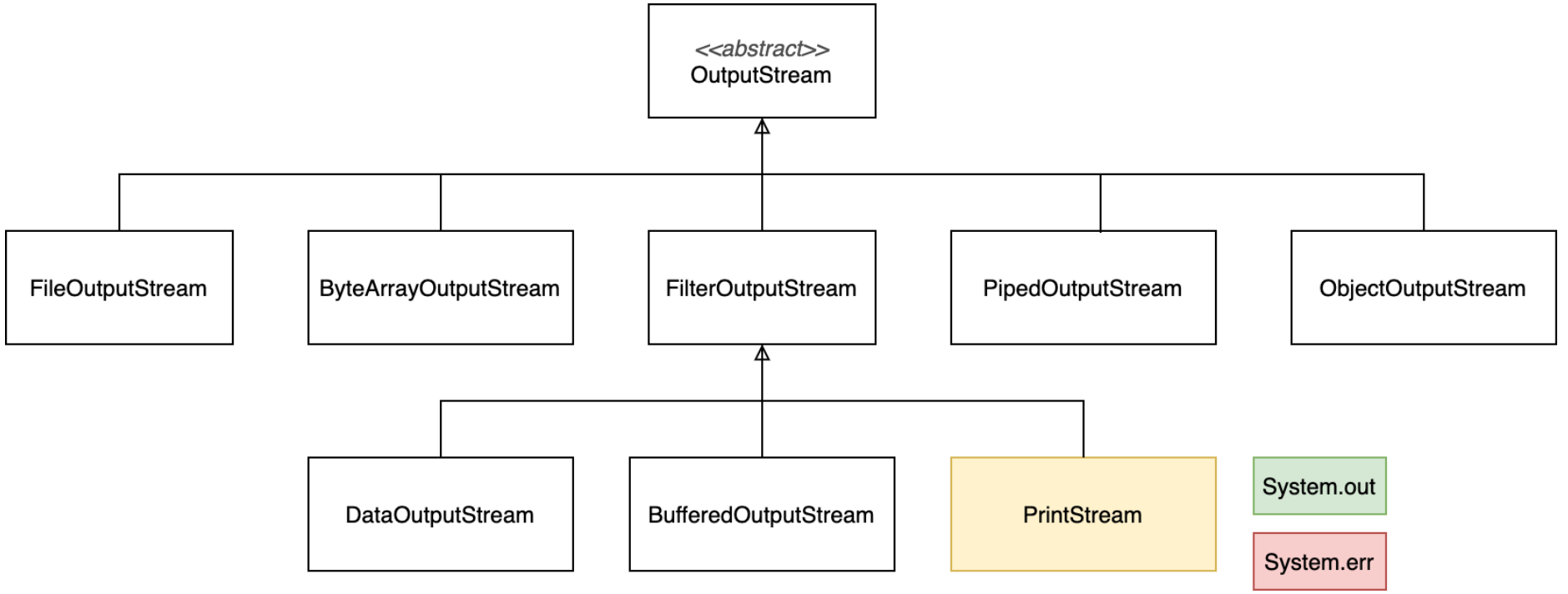
- Can use both byte or character streams.
- Preferred method: character-oriented stream.
  - Make your program easier to internationalize and easier to maintain.
  - Also more convenient to operate directly on characters rather than converting back and forth between characters and bytes.
- System.in is an instance of InputStream → automatically have access to the methods defined by InputStream → we can use the read() method to read bytes from System.in.
- Three versions of read():

```
int read() throws IOException
int read(byte data[]) throws IOException
int read(byte data[], int off, int len)
                                    throws IOException
```

Example


```
1.  import java.io.*;
2.  class ReadBytes {
3.      public static void main(String args[])
4.                                          throws IOException {
5.          byte data[] = new byte[10];
6.
7.          System.out.println("Enter some characters.");
8.          System.in.read(data);
9.          System.out.print("You entered: ");
10.         for(byte c:data)
11.             System.out.print((char)c);
12.         System.out.println();
13.     }
14. }
```

Writing console output [1]



Writing console output [2]

- For the most portable code, character streams are recommended.
- Because System.out is a byte stream→ byte-based console output is still widely used.
- Console output is most easily accomplished with print() and println().
  - These methods are defined by the class PrintStream
  - Even though System.out is a byte stream, it is still acceptable to use this stream for simple console output.
- PrintStream is an output stream derived from OutputStream → it also implements the low-level method write() → possible to write to the console by using write().

Writing console output [3]

- Simplest form of write() defined by PrintStream:

```
void write(int byteval)
```

- This method writes the byte specified by byteval to the file.
- Although byteval is declared as an integer, only the low-order 8 bits are written.

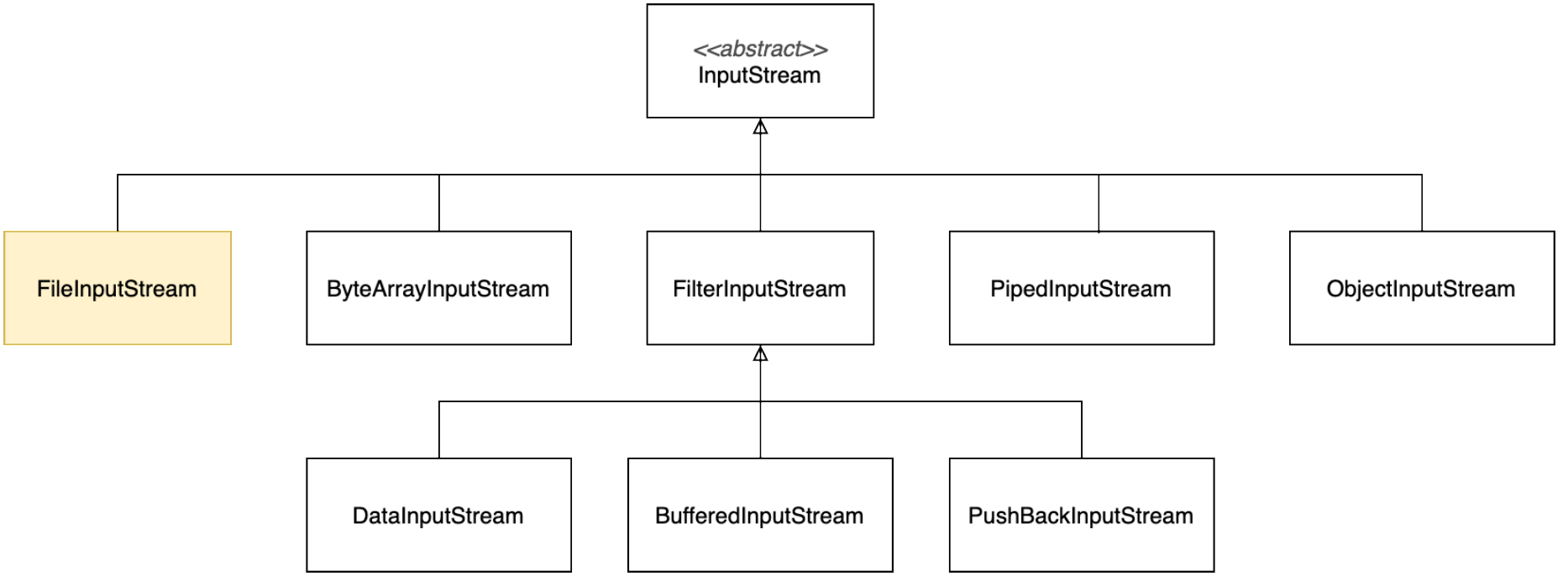
Example


```
1.  // Demonstrate System.out.write().
2.  class WriteDemo {
3.      public static void main(String args[]) {
4.          int b;
5.
6.          b = 'X';
7.          System.out.write(b);
8.          System.out.write('\n');
9.      }
10. }
```

Reading and writing files using byte streams

- All files are byte-oriented, and Java provides methods to read and write bytes from and to a file → reading and writing files using byte streams is very common.
- Java also allows you to wrap a byte-oriented file stream within a character-based object.
- To create a byte stream linked to a file, use FileInputStream or FileOutputStream.
- To open a file:
  - create an object of one of these classes, specifying the name of the file as an argument to the constructor.
  - Once the file is open, you can read from or write to it.

Inputting from a file [1]



Inputting from a file [2]

- A file is opened for input by creating a FileInputStream object.

  FileInputStream(String fileName)

  throws FileNotFoundException

  - fileName specifies the name of the file to open.
  - If the file does not exist, then FileNotFoundException is thrown.
  - FileNotFoundException is a subclass of IOException.
- To read from a file:

  int read() throws IOException
- When you are done with a file, you must close it by calling close():

  void close() throws IOException

Example [1]


```
1.  import java.io.*;
2.  class ShowFile {
3.      public static void main(String args[]) {
4.      int i; FileInputStream fin;
5.      // First make sure that a file has been specified.
6.      if (args.length != 1) {
7.          System.out.println("Usage: ShowFile File");
8.          return;
9.      }
10.     try {
11.         fin = new FileInputStream(args[0]);
12.     } catch (FileNotFoundException exc) {
13.         System.out.println("File Not Found"); return;
14.     }
15.     // ....
```

Example [2]

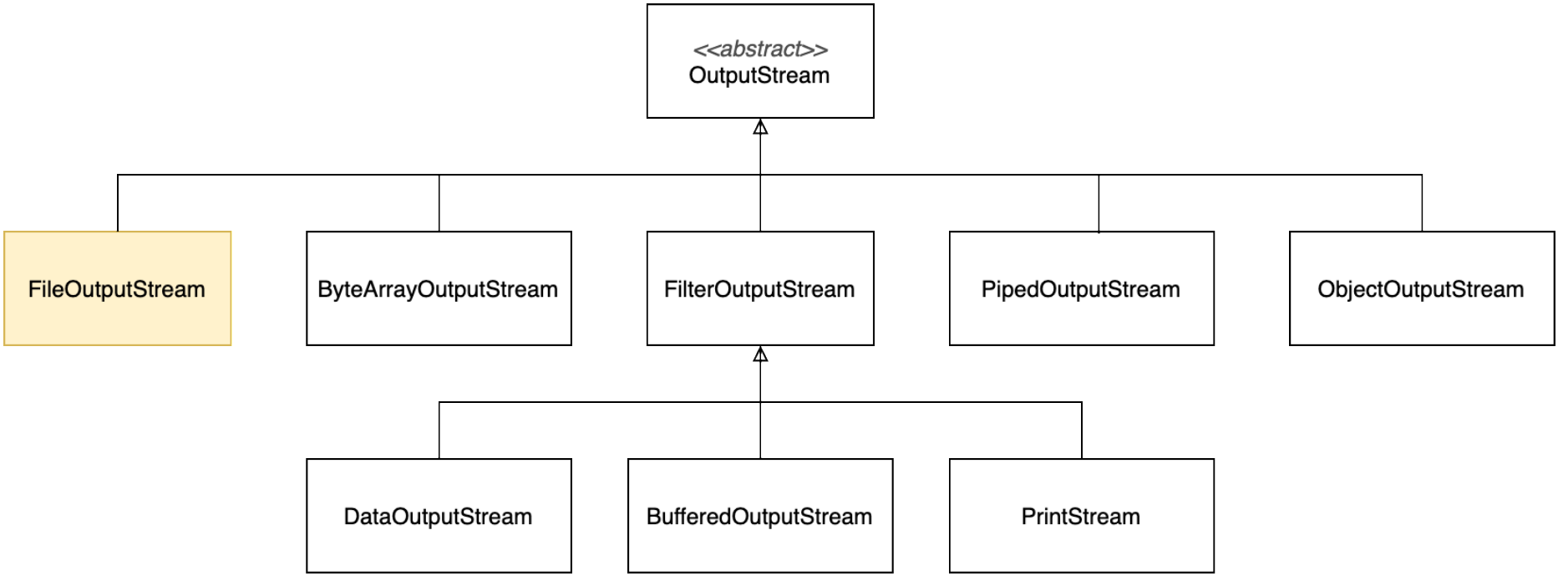

```
1.          // ...
2.          try { // read bytes until EOF is encountered
3.                  do {
4.                          i = fin.read();
5.                          if (i != -1)
6.                                  System.out.print((char) i);
7.                  } while (i != -1);
8.          } catch (IOException exc) {
9.                  System.out.println("Error reading file.");
10.         }
11.         try {
12.                 fin.close();
13.         } catch (IOException exc) {
14.                 System.out.println("Error closing file.");
15.         }
16. }
```

Version 2


```
1.  import java.io.*;
2.  class ShowFile1 {
3.      public static void main(String args[]) {
4.          int i;
5.          FileInputStream fin = null;
6.
7.          // First, confirm that a file name has been specified.
8.          if (args.length != 1) {
9.              System.out.println("Usage: ShowFile filename");
10.             return;
11.         }
12.         /* The following code opens a file, reads characters until EOF is encountered, and
13.         then closes the file via a finally block. */
14.
15.
```

```
1.              try {
2.                  fin = new FileInputStream(args[0]);
3.                  do {
4.                      i = fin.read();
5.                      if (i != -1) System.out.print((char) i);
6.                  } while (i != -1);
7.              } catch (FileNotFoundException exc) {
8.                      System.out.println("File Not Found.");
9.              } catch (IOException exc) {
10.                     System.out.println("An I/O Error Occurred");
11.             } finally { // Close file in all cases.
12.                 try {
13.                     if (fin != null) fin.close();
14.                 } catch (IOException exc) {
15.                     System.out.println("Error Closing File");
16.                 }
17.             }
18.     }
19. }
```

Writing to a file [1]



Writing to a file [2]

- To open a file for output, create a FileOutputStream object.
- If cannot create file: FileNotFoundException is thrown.

FileOutputStream(String fileName)

throws FileNotFoundException

- When an output file is opened, any pre-existing file by the same name is destroyed.

FileOutputStream(String fileName, boolean append)

throws FileNotFoundException

- If append is true, then output is appended to the end of the file.

Writing to a file [3]

- To write to a file:

  void write(int byteval) throws IOException

  - Writes the byte specified by byteval to the file.
  - Although byteval is declared as an integer, only the low-order 8 bits are written to the file.
  - If an error occurs during writing, an IOException is thrown.
- Once done with an output file: must close it

  void close() throws IOException

  - Closing a file releases the system resources allocated to the file, allowing them to be used by another file.
  - It also helps ensure that any output remaining in an output buffer is actually written to the physical device.

Example: Copy file


```
1.  import java.io.*;
2.  class CopyFile {
3.      public static void main(String args[]) throws IOException {
4.          int i;
5.          FileInputStream fin = null;FileOutputStream fout = null;
6.          // First, make sure that both files has been specified.
7.          if (args.length != 2) {
8.              System.out.println("Usage: CopyFile from to"); return;
9.          }
10.         // Copy a File.
11.         try { // Attempt to open the files.
12.             fin = new FileInputStream(args[0]);
13.             fout = new FileOutputStream(args[1]);
14.             // ...
15.
```

```
1.                  do {
2.                      i = fin.read();
3.                      if (i != -1) fout.write(i);
4.                  } while (i != -1);
5.              } catch (IOException exc) {
6.                      System.out.println("I/O Error: " + exc);
7.              } finally {
8.                  try { if (fin != null) fin.close();
9.                  } catch (IOException exc) {
10.                     System.out.println("Error Closing Input File");
11.                 }
12.                 try { if (fout != null) fout.close();
13.                 } catch (IOException exc) {
14.                     System.out.println("Error Closing Output File");
15.                 }
16.             }
17.         }
18. }
```

Automatically closing a file

- From JDK 7: Java offers another, more streamlined way to manage resources, such as file streams, by automating the closing process.
  - It is based on another version of the try statement called **trywith-resources** (automatic resource management).
  - Advantage: prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed.
  - Forgetting to close a file can result in memory leaks and could lead to other problems.

```
try (resources-specification) {
        // use the resource
}
```

- resource-specification is a statement that declares and initializes a resource, such as a file.

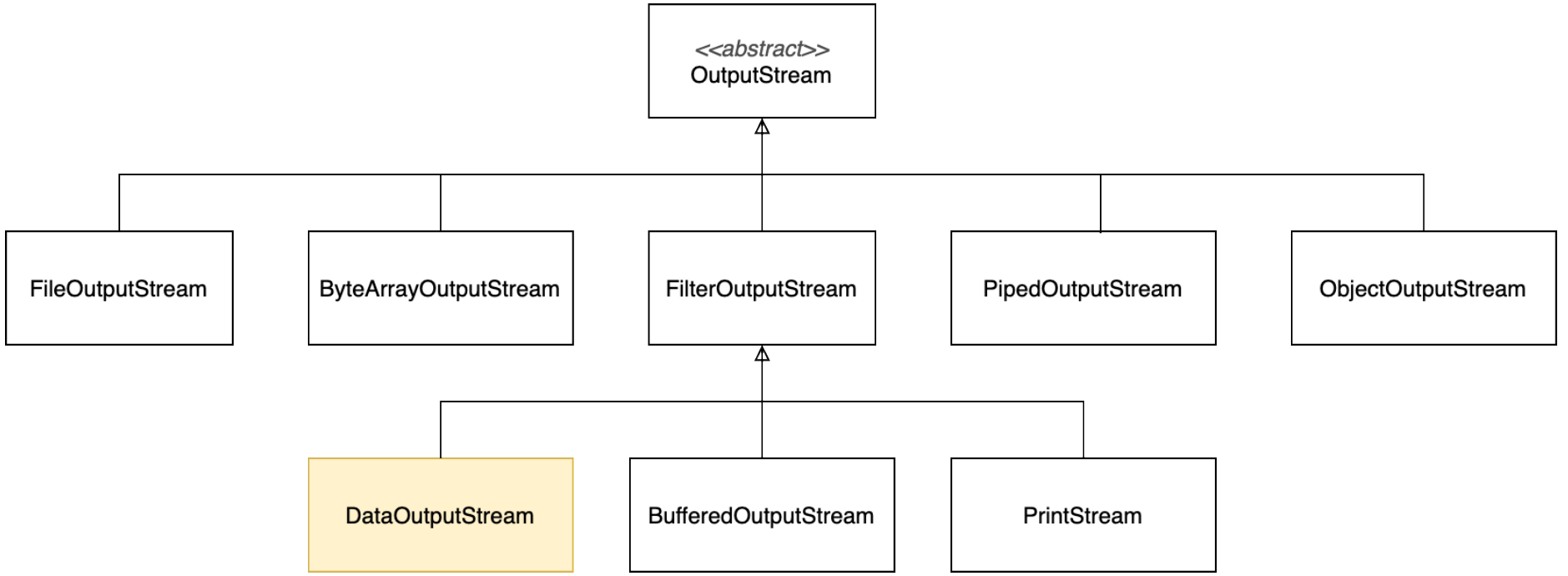
```
1.  import java.io.*;
2.  class ShowFile2 {
3.      public static void main(String args[]) {
4.          int i;
5.          // First, make sure that a file name has been specified.
6.          if (args.length != 1) {
7.              System.out.println("Usage: ShowFile filename");
8.              return;
9.          }
10.         // Use try-with resources to open a file and then automatically close it when the try
11.         block is left.
12.         try (FileInputStream fin = new FileInputStream(args[0])){
13.             do {
14.                 i = fin.read();
15.                 if (i != -1) System.out.print((char) i);
16.             } while (i != -1);
17.         } catch (IOException exc) {
18.                 System.out.println("I/O Error: " + exc);
19.         }
20.     }
21. }
```

```
1.  import java.io.*;
2.  class CopyFile1 {
3.      public static void main(String args[]) throws IOException {
4.          int i;
5.          // First, confirm that both files has been specified.
6.          if (args.length != 2) {
7.              System.out.println("Usage: CopyFile from to");
8.              return;
9.          }
10.         // Open and manage two files via the try statement.
11.         try (FileInputStream fin = new FileInputStream(args[0]);
12.          FileOutputStream fout = new FileOutputStream(args[1])){
13.             do {
14.                 i = fin.read();
15.                 if (i != -1) fout.write(i);
16.             } while (i != -1);
17.         } catch (IOException exc) {
18.             System.out.println("I/O Error: " + exc);
19.         }
20.     }
21. }
22.
```

Reading and writing binary data

- Use DataInputStream and DataOutputStream.

DataOutputStream [1]



DataOutputStream [2]

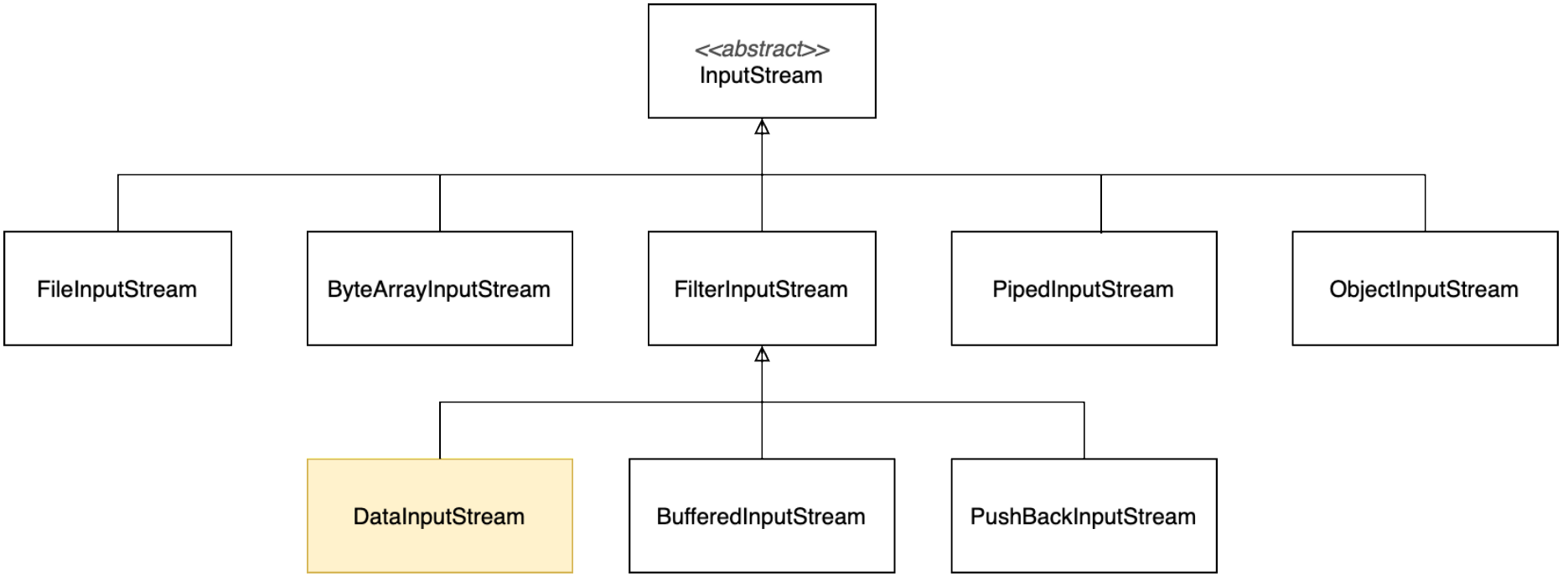
- DataOutputStream implements the DataOutput interface.
  - This interface defines methods that write all of Java's primitive types to a file.
  - This data is written using its internal, binary format, not its human-readable text form.
- Constructor

  DataOutputStream(OutputStream outputStream)
  - outputStream is the stream to which data is written.
  - To write output to a file, you can use the object created by FileOutputStream for this parameter.

Commonly Used Output Methods Defined by DataOutputStream

Output Method	Purpose
void writeBoolean(boolean *val*)	Writes the **boolean** specified by *val*.
void writeByte(int *val*)	Writes the low-order byte specified by *val*.
void writeChar(int *val*)	Writes the value specified by *val* as a **char**.
void writeDouble(double *val*)	Writes the **double** specified by *val*.
void writeFloat(float *val*)	Writes the **float** specified by *val*.
void writeInt(int *val*)	Writes the **int** specified by *val*.
void writeLong(long *val*)	Writes the **long** specified by *val*.
void writeShort(int *val*)	Writes the value specified by *val* as a **short**.

DataInputStream [1]



DataInputStream [2]

- DataInputStream implements the DataInput interface, which provides methods for reading all of Java's primitive types.
- DataInputStream uses an InputStream instance as its foundation, overlaying it with methods that read the various Java data types.
- DataInputStream reads data in its binary format, not its human readable form.
- Constructor:

DataInputStream(InputStream inputStream)

  - inputStream is the stream that is linked to the instance of DataInputStream being created.

Commonly Used Input Methods Defined by DataInputStream

Input Method	Purpose
boolean readBoolean( )	Reads a **boolean**.
byte readByte( )	Reads a **byte**.
char readChar( )	Reads a **char**.
double readDouble( )	Reads a **double**.
float readFloat( )	Reads a **float**.
int readInt( )	Reads an **int**.
long readLong( )	Reads a **long**.
short readShort( )	Reads a **short**.

```
1.  import java.io.*;
2.  class RWData {
3.      public static void main(String args[]) {
4.          int i = 10; double d = 1023.56; boolean b = true;
5.          // Write some values.
6.          try (DataOutputStream dataOut = new DataOutputStream(new
7.          FileOutputStream("testdata"))) {
8.              System.out.println("Writing " + i);
9.              dataOut.writeInt(i);
10.             System.out.println("Writing " + d);
11.             dataOut.writeDouble(d);
12.             System.out.println("Writing " + b);
13.             dataOut.writeBoolean(b);
14.             System.out.println("Writing " + 12.2 * 7.4);
15.             dataOut.writeDouble(12.2 * 7.4);
16.         } catch (IOException exc) {
17.             System.out.println("Write error.");
18.             return;
19.         }
20.         System.out.println();
21.         ...
```

```
1.              ...
2.              // Now, read them back.
3.              try (DataInputStream dataIn = new
4.                 DataInputStream(new FileInputStream("testdata"))) {
5.                     i = dataIn.readInt();
6.                     System.out.println("Reading " + i);
7.
8.                     d = dataIn.readDouble();
9.                     System.out.println("Reading " + d);
10.
11.                    b = dataIn.readBoolean();
12.                    System.out.println("Reading " + b);
13.
14.                    d = dataIn.readDouble();
15.                    System.out.println("Reading " + d);
16.             } catch (IOException exc) {
17.                     System.out.println("Read error.");
18.             }
19.     }
20. }
```

Exercise: Compare two files

Version 1:

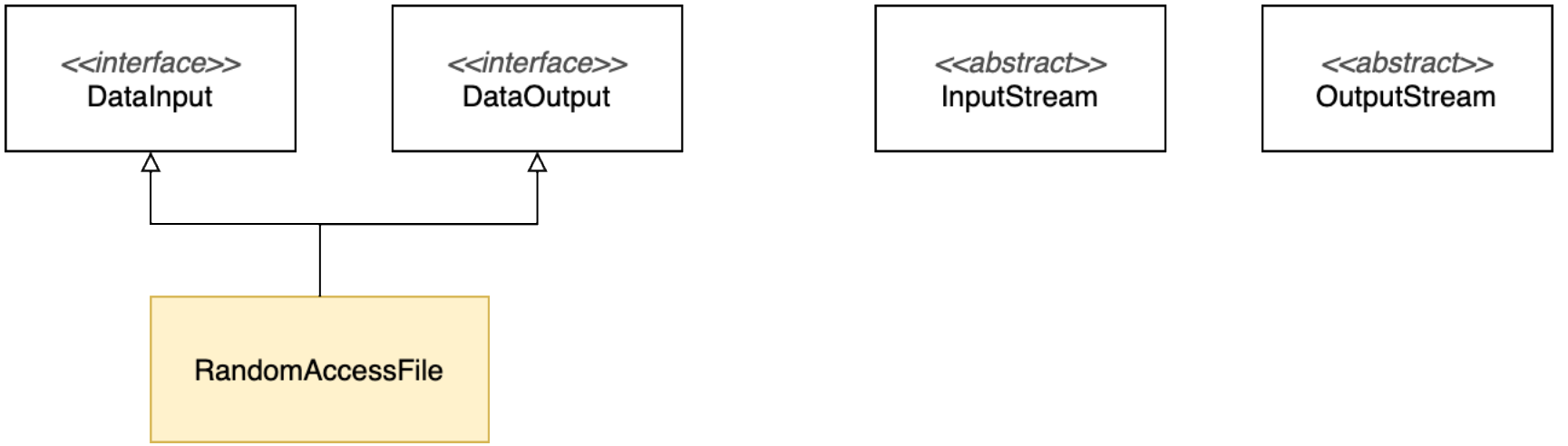
- Write a program that checks whether two files have the same content.

Exercise: Compare two files

Version 2:

- Check if we compare the same file, show message and leave.
- Write a program that checks whether two files have the same content (in ignoring the case of letters).
- If two files differ then the program will display the position where the files differ.

RandomAccessFile [1]



RandomAccessFile [2]

- Java allows you to access the contents of a file in random order → use RandomAccessFile.
- RandomAccessFile is not derived from InputStream or OutputStream. Instead, it implements the interfaces DataInput and DataOutput, which define the basic I/O methods.
- It also supports positioning requests → you can position the file pointer within the file.
- Constructor:

RandomAccessFile(String fileName, String access)

throws FileNotFoundException

  - The name of the file is passed in fileName and access determines what type of file access is permitted.

RandomAccessFile [3]

- To set the current position of the file pointer within the file:

  void seek(long newPos) throws IOException

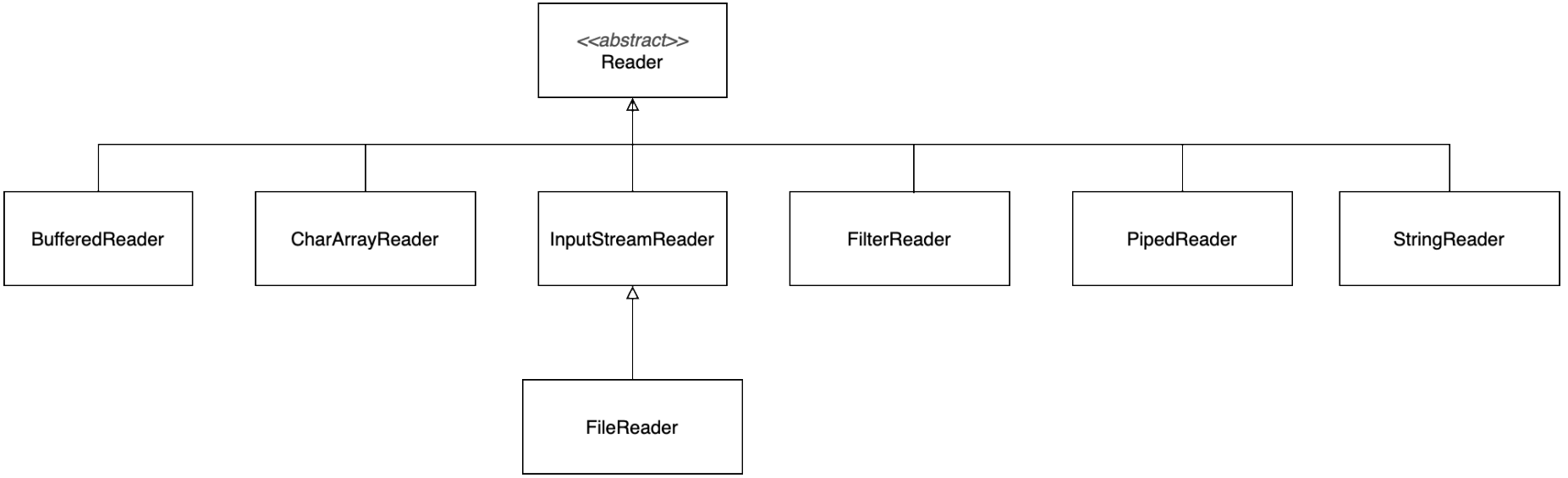
  - newPos specifies the new position, in bytes, of the file pointer from the beginning of the file.
  - After a call to seek(), the next read or write operation will occur at the new file position.
- Because RandomAccessFile implements the DataInput and DataOuput interfaces, methods to read and write the primitive types are available.
  - The read() and write() methods are also supported.

```
1.  import java.io.*;
2.  class RandomAccessDemo {
3.      public static void main(String args[]) {
4.          double data[] ={ 19.4, 10.1, 123.54, 33.0, 87.9, 74.25};
5.          double d;
6.          // Open and use a random access file.
7.          try (RandomAccessFile raf = new
8.                  RandomAccessFile("random.dat", "rw")) {
9.              // Write values to the file.
10.             for (int i = 0; i < data.length; i++) {
11.                 raf.writeDouble(data[i]);
12.             }
13.             // Now, read back specific values
14.             raf.seek(0); // seek to first double
15.             d = raf.readDouble();
16.             System.out.println("First value is " + d);
17.             raf.seek(8); // seek to second double
18.             d = raf.readDouble();
19.             System.out.println("Second value is " + d);
20.             ...
```

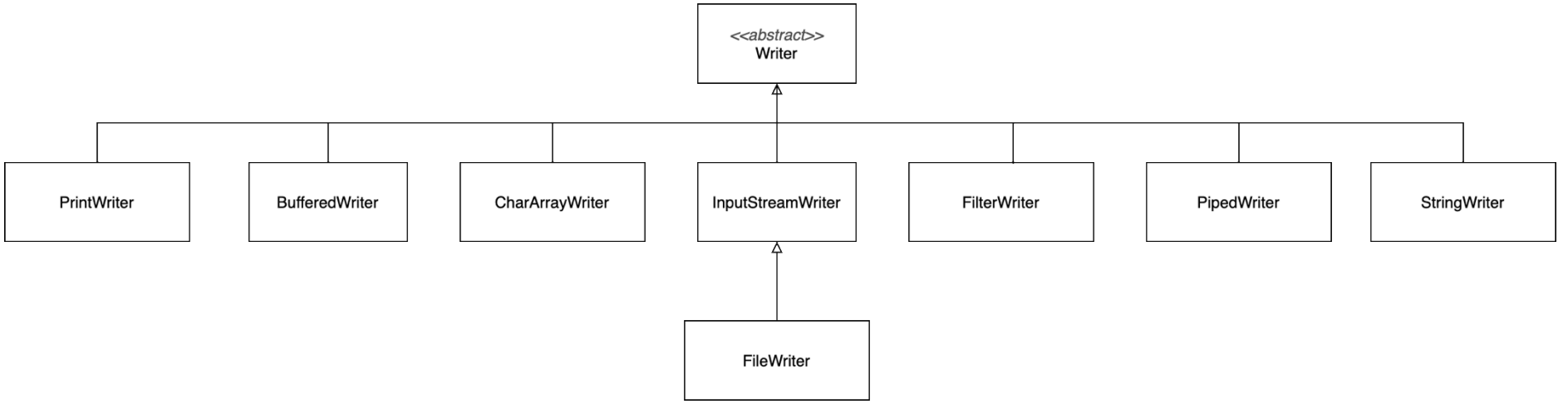
```
1.                  ...
2.                  raf.seek(8 * 3); // seek to fourth double
3.                  d = raf.readDouble();
4.                  System.out.println("Fourth value is " + d);
5.
6.                  System.out.println();
7.
8.                  // Now, read every other value.
9.                  System.out.println("Here is every other value: ");
10.                 for (int i = 0; i < data.length; i += 2) {
11.                       raf.seek(8 * i); // seek to ith double
12.                       d = raf.readDouble();
13.                       System.out.print(d + " ");
14.                 }
15.           } catch (IOException exc) {
16.                       System.out.println("I/O Error: " + exc);
17.           }
18.     }
19. }
```

Character streams

Character - Reader



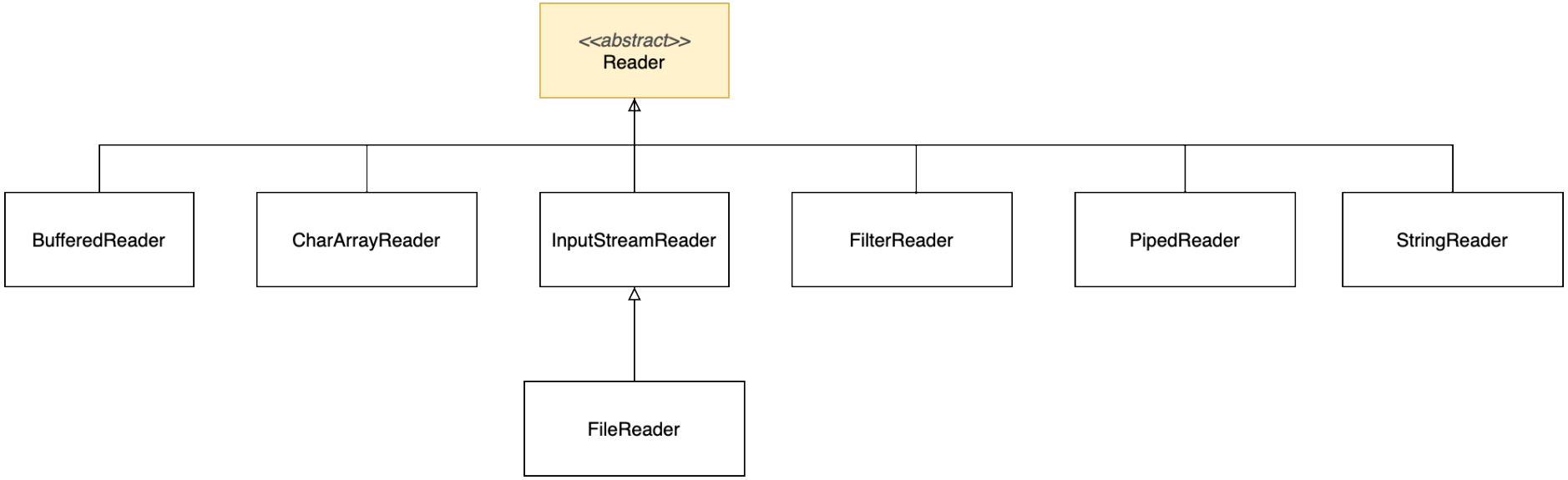
Character - Writer



Using Java's character-based streams

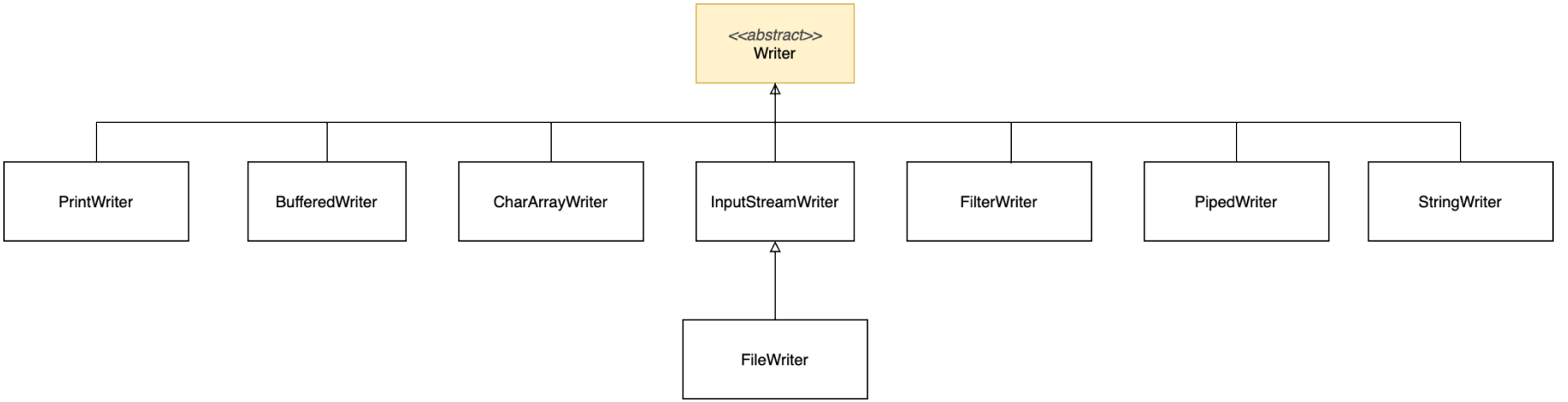
- Java's byte streams are both powerful and flexible but not the ideal way to handle character-based I/O → character stream classes.
- At the top of the character stream hierarchy are the abstract classes `Reader` and `Writer`.
- Most of the methods can throw an `IOException` on error.
- The methods defined by these two abstract classes are available to all of their subclasses → form a minimal set of I/O functions that all character streams will have.

Reader


<<abstract>>
Reader
BufferedReader
CharArrayReader
InputStreamReader
FilterReader
PipedReader
StringReader
FileReader

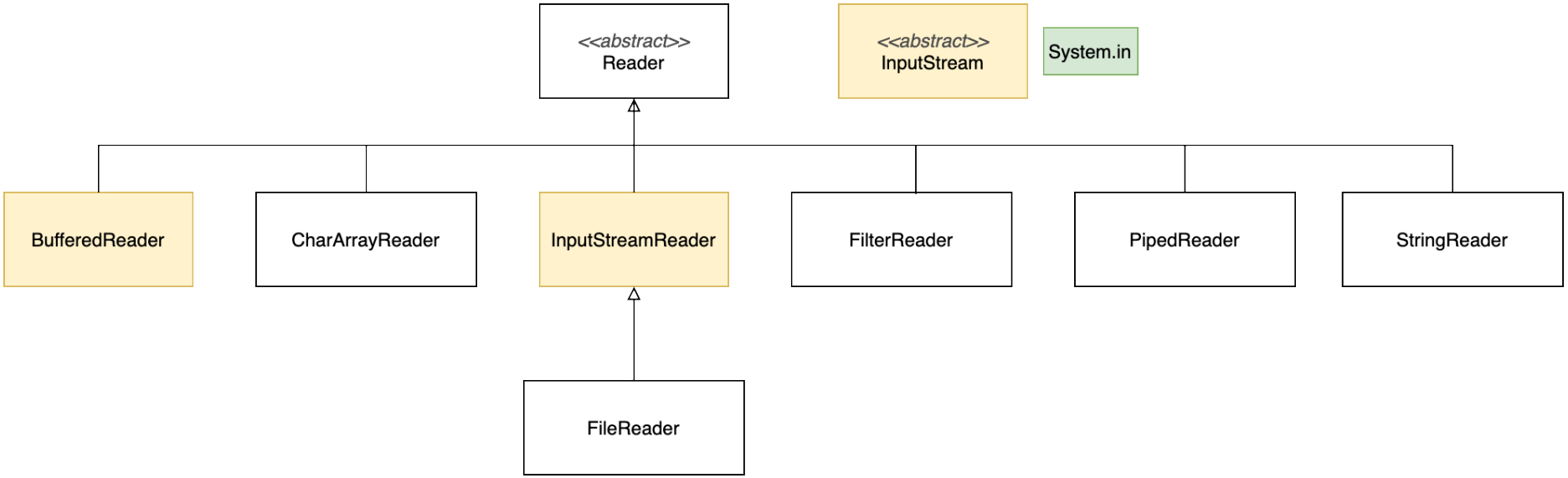
Method	Description
abstract void close( )	Closes the input source. Subsequent read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )**/**reset( )** are supported on this stream.
static Reader nullReader( )	Returns an open, but null reader, which is a reader that contains no data. Thus, the reader is always at the end of the stream and no input can be obtained. The reader can, however, be closed. (Added by JDK 11.)
int read( )	Returns an integer representation of the next available character from the invoking input stream. −1 is returned when an attempt is made to read at the end of the stream.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. −1 is returned when an attempt is made to read at the end of the stream.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. −1 is returned when an attempt is made to read at the end of the stream.
int read(CharBuffer *buffer*)	Attempts to fill the buffer specified by *buffer*, returning the number of characters successfully read. −1 is returned when an attempt is made to read at the end of the stream. **CharBuffer** is a class that encapsulates a sequence of characters, such as a string.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.
long transferTo(Writer *writer*)	Copies the contents of the invoking reader to *writer*, returning the number of characters copied. (Added by JDK 10.)

Writer


<<abstract>>
Writer
PrintWriter
BufferedWriter
CharArrayWriter
InputStreamWriter
FilterWriter
PipedWriter
StringWriter
FileWriter

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*)	Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream. **CharSequence** is an interface that defines read-only operations on a sequence of characters.
Writer append(CharSequence *chars*, int *begin*, int *end*)	Appends the sequence of *chars* starting at *begin* and stopping with *end* to the end of the invoking output stream. Returns a reference to the invoking stream. **CharSequence** is an interface that defines read-only operations on a sequence of characters.
abstract void close( )	Closes the output stream. Subsequent write attempts will generate an **IOException**.
abstract void flush( )	Causes any output that has been buffered to be sent to its destination. That is, it flushes the output buffer.
static Writer nullWriter( )	Returns an open, but null output writer, which is a writer to which no output is written. The writer can, however, be closed. (Added by JDK 11.)
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **char**.
void write(char *buffer*[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *str*, beginning at the specified *offset*.

Console input using character streams [1]



Console input using character streams [2]

- Use InputStreamReader, which converts bytes to characters. To obtain an InputStreamReader object that is linked to System.in, use the constructor:

  InputStreamReader(InputStream inputStream)

  - Since System.in refers to an object of type InputStream, it can be used for inputStream.
- Next, use the object produced by InputStreamReader, construct a BufferedReader using the constructor:

  BufferedReader(Reader inputReader)

  - inputReader is the stream that is linked to the instance of BufferedReader being created.
- Putting it all together, we have:

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

Reading Characters

- Use read() method defined by BufferedReader in the same way as they were read using byte streams.

int read() throws IOException

int read(char data[]) throws IOException

int read(char data[], int start, int max)

throws IOException

Example


```
1.  import java.io.*;
2.  class ReadChars {
3.      public static void main(String args[]) throws IOException {
4.          char c;
5.          BufferedReader br = new BufferedReader(new
6.                  InputStreamReader(System.in));
7.
8.          System.out.println("Enter characters, period to quit.");
9.          // read characters
10.         do {
11.             c = (char) br.read();
12.             System.out.println(c);
13.         } while (c != '.');
    }
}
```

Reading Strings

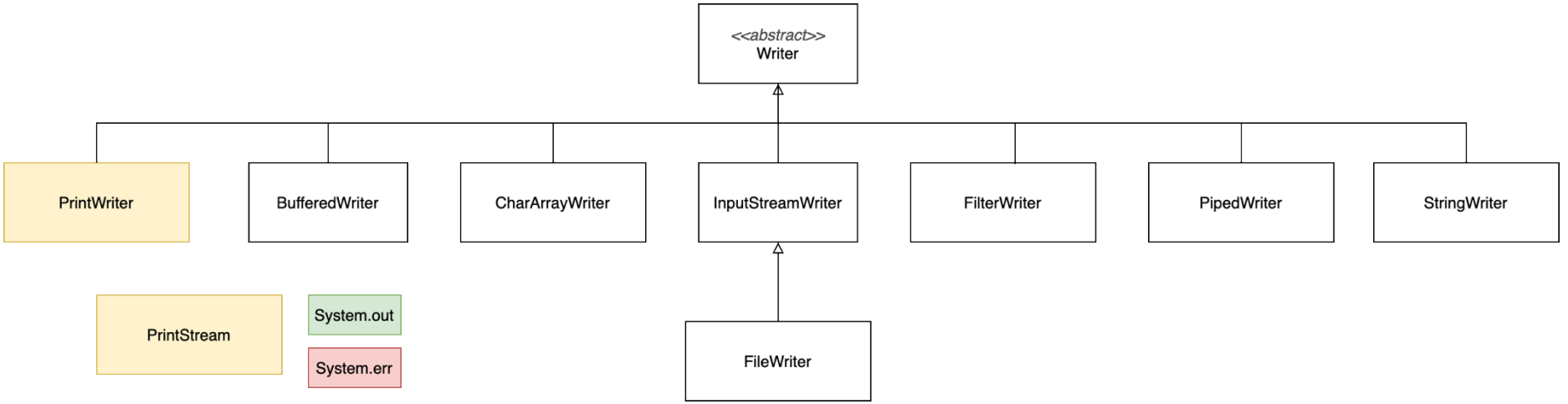
String readLine() throws IOException

- It returns a String object that contains the characters read. It returns null if an attempt is made to read when at the end of the stream.

Example


```
1.  import java.io.*;
2.  class ReadLines {
3.      public static void main(String args[]) throws IOException {
4.          // create a BufferedReader using System.in
5.          BufferedReader br =
6.            new BufferedReader(new InputStreamReader(System.in));
7.          String str;
8.
9.          System.out.println("Enter lines of text.");
10.         System.out.println("Enter 'stop' to quit.");
11.         do {
12.             str = br.readLine();
13.             System.out.println(str);
14.         } while (!str.equals("stop"));
15.     }
16. }
```

Console output using character streams [1]



Console output using character streams [2]

- The preferred method of writing to the console is through a PrintWriter stream (character-based classes).
- Constructor:

```
PrintWriter(OutputStream outputStream,
            boolean flushingOn)
```

  - outputStream is an object of type OutputStream and flushingOn controls whether Java flushes the output stream every time a println() method (among others) is called.
  - If flushingOn is true, flushing automatically takes place.
  - If false, flushing is not automatic.
- PrintWriter supports the print() and println() methods for all types including Object.

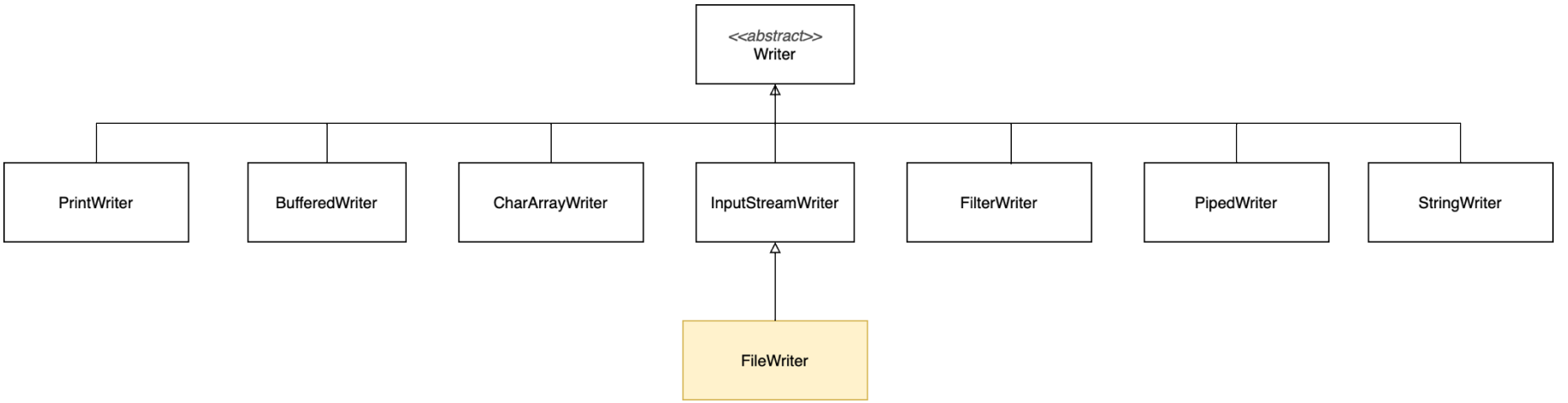
Example


```
1.  //Demonstrate PrintWriter.
2.  import java.io.*;
3.  public class PrintWriterDemo {
4.      public static void main(String args[]) {
5.          PrintWriter pw = new PrintWriter(System.out, true);
6.          int i = 10;
7.          double d = 123.65;
8.
9.          pw.println("Using a PrintWriter.");
10.         pw.println(i);
11.         pw.println(d);
12.
13.         pw.println(i + " + " + d + " is " + (i + d));
14.     }
15. }
```

FILE I/O USING CHARACTER STREAMS

- Use the FileReader and FileWriter classes

Using a FileWriter [1]



Using a FileWriter [2]

- FileWriter creates a Writer that you can use to write to a file. Two commonly used constructors:

FileWriter(String fileName) throws IOException

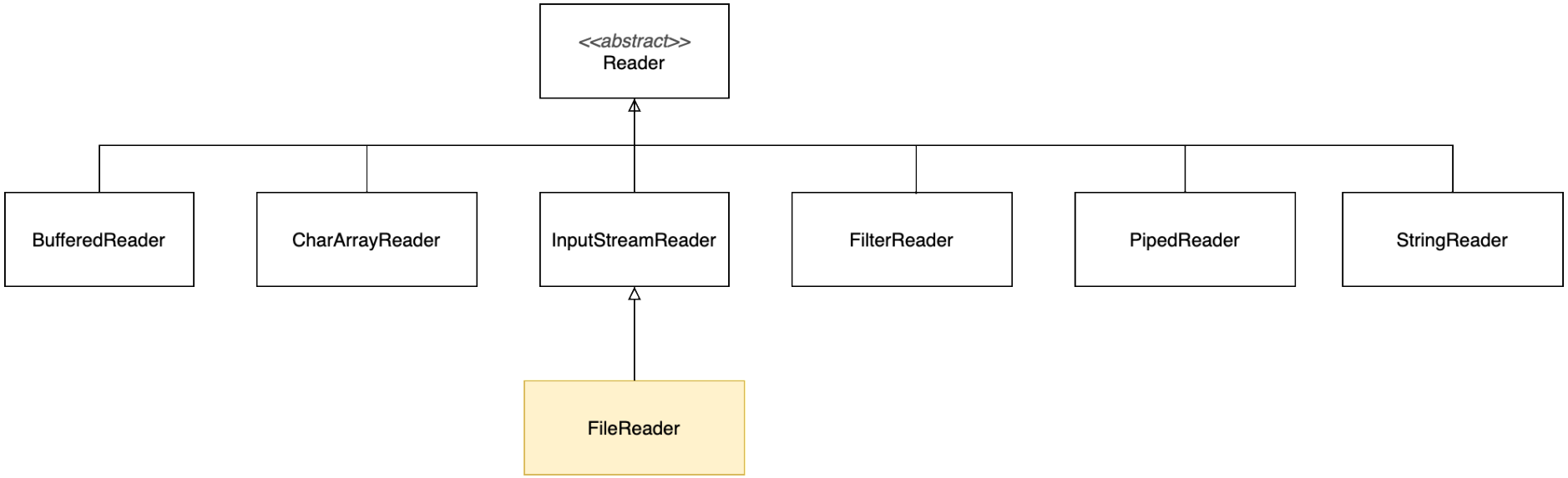
FileWriter(String fileName, boolean append)

throws IOException

  - fileName is the full path name of a file.
  - If append is true, then output is appended to the end of the file. Otherwise, the file is overwritten.
  - FileWriter is derived from OutputStreamWriter and Writer

```
1.  import java.io.*;
2.  class KtoD {
3.      public static void main(String args[]) {
4.          String str;
5.          BufferedReader br = new BufferedReader(new
6.                                          InputStreamReader(System.in));
7.          System.out.println("Enter text ('stop' to quit).");
8.          try (FileWriter fw = new FileWriter("test.txt")) {
9.              do {
10.                 System.out.print(": ");
11.                 str = br.readLine();
12.                 if (str.compareTo("stop") == 0) break;
13.                 str = str + "\r\n"; // add newline
14.                 fw.write(str);
15.             } while (str.compareTo("stop") != 0);
16.         } catch (IOException exc) {
17.           System.out.println("I/O Error: " + exc);
18.       }
19.     }
20. }
```

Using a FileReader [1]



Using a FileReader [2]

- FileReader class creates a Reader that you can use to read the contents of a file. A commonly used constructor:

  FileReader(String fileName)

  throws FileNotFoundException

  - fileName is the full path name of a file.
  - It throws a FileNotFoundException if the file does not exist.
  - FileReader is derived from InputStreamReader and Reader.

```
1.  import java.io.*;
2.  class DtoS {
3.      public static void main(String args[]) {
4.          String s;
5.          /* Create and use a FileReader wrapped in a
6.           BufferedReader. */
7.          try (BufferedReader br = new BufferedReader(
8.                                          new FileReader("test.txt"))) {
9.              while ((s = br.readLine()) != null) {
10.                 System.out.println(s);
11.             }
12.         } catch (IOException exc) {
13.            System.out.println("I/O Error: " + exc);
14.         }
15.     }
16. }
```

Using Java's type wrappers to convert numeric strings

- Java's println() method provides a convenient way to output various types of data to the console →println() automatically converts numeric values into their human-readable form.
- However, methods like read() do not provide a parallel functionality that reads and converts a string containing a numeric value into its internal, binary format.
- Java provides various other ways: **Java's type wrappers**.
- The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean.
  - Offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Conversion methods

Wrapper	Conversion Method
Double	static double parseDouble(String *str*) throws NumberFormatException
Float	static float parseFloat(String *str*) throws NumberFormatException
Long	static long parseLong(String *str*) throws NumberFormatException
Integer	static int parseInt(String *str*) throws NumberFormatException
Short	static short parseShort(String *str*) throws NumberFormatException
Byte	static byte parseByte(String *str*) throws NumberFormatException

Example


```
1.  import java.io.*;
2.  class AvgNums {
3.      public static void main(String args[]) throws IOException{
4.          // create a BufferedReader using System.in
5.          BufferedReader br = new BufferedReader(new
6.                                          InputStreamReader(System.in));
7.          String str; int n; double sum = 0.0; double avg, t;
8.          System.out.print("How many numbers will you enter: ");
9.          str = br.readLine();
10.         try {
11.             n = Integer.parseInt(str);
12.         } catch (NumberFormatException exc) {
13.             System.out.println("Invalid format");
14.             n = 0;
15.         }
16.         ...
```

```
1.              ...
2.              System.out.println("Enter " + n + " values.");
3.              for (int i = 0; i < n; i++) {
4.                      System.out.print(": ");
5.                      str = br.readLine();
6.                      try {
7.                              t = Double.parseDouble(str);
8.                      } catch (NumberFormatException exc) {
9.                              System.out.println("Invalid format");
10.                             t = 0.0;
11.                     }
12.                     sum += t;
13.             }
14.             avg = sum / n;
15.             System.out.println("Average is " + avg);
16.     }
17. }
```

Scanner

- Another way to convert a numeric string into its internal, binary format is to use one of the methods defined by the java.util.Scanner
- Scanner reads formatted input and converts it into its binary form.
- Scanner can be used to read input from a variety of sources, including the console and files → use Scanner to read a numeric string entered at the keyboard and assign its value to a variable.
- Constructor:

Scanner(InputStream from)

Exercise [1]

1. Write a program that counts the number of lines of an input file.
2. Write a program that counts the number of words in an input file.

Exercise [2]

3. Write a program that reads a file (provided) containing list of contacts. Each line has three fields: first name, last name, phone number separated by a tab (\t).
   - Define the class Contact having the three fields mentioned above.
   - Read file and put it in a list of contacts, then add operations on that list: Add new contact, Modify a contact, Delete a contact.
   - Save a list of contacts to an output file.
4. Write a program that reads an input file (provided) containing the grades of students in Java course (Each student is stored in a line, all fields are seperated by a tab \t).
   - Define all necessary classes.
   - Read the input file and put information in a list of Students.
   - Add operations on the list of Students: manage Students, manage grades.
   - Save the list into an outputfile with the average note (Column TB).

QUESTION ?