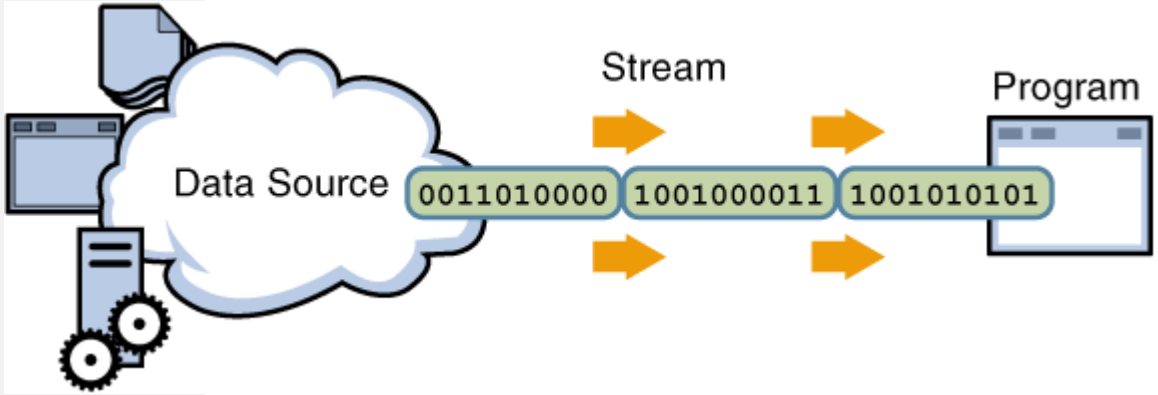
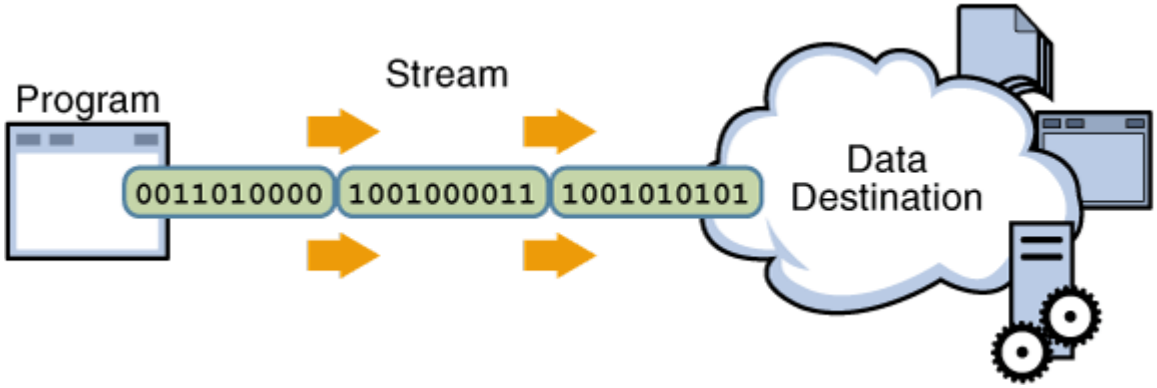
Java I/O

I/O Streams



Reading data using input stream



Writing data using output stream

I/O from Command Line

- Standard streams
  - **`System.out`**
    - Default standard output – *terminal screen*
    - An object of type **`PrintStream`**
  - **`System.err`**
    - Default standard error – *terminal screen*
    - An object of type **`PrintStream`**
  - **`System.in`**
    - Standard input – *keyboard*
    - An object ot type **`InputStream`**

All standard streams can be rerouted to a file stream

`System.setOut, System.setIn, System.setErr`

Write to Standard Output/Error

- Methods to write

```
System.out.println(…);
System.out.print(…);

System.err.println(…);
System.err.print(…);
```

- The **`print()`** and **`println()`** methods are overloaded for most primitive type

```
void println(char);
void println(int);
void println(double);
```

Read from Standard Input

- Read as text (strings)
  - using **`java.util`** package

```
Scanner in = new Scanner(System.in);
String s = in.nextLine();
```

- Read using streams
  - using **`java.io`** package

```
//convert raw byte stream to Unicode character
InputSteamReader ir = new InputStremReader(System.in)
// BufferedReader allows to read a line at a time
BufferedReader in = new BufferedReader(ir);

String s = in.readLine();

in.close();       //Always close the buffered reader
```

Secure Password Entry

- **`Console`** Class
  - Not as convenient as Scanner
    - Only read a line of input at a time
  - Useful for secure password entry
    - **`readPassowrd()`** method suppresses echoing, so the password is not visible on the user's screen
  - Since 1.6 in **`java.io`** package

```
Console cons = System.console();

String username = cons.readLine("Username: ");
char [] password = cons.readPassword("Password: ");      //password invisible
```

Creating a `File` Object

- File only:

```
File myFile = new File("myFile.txt");  //current directory
```

- Directory:

```
File myFile = new File("MyDirName", "myFile.txt");
```

```
File myDir = new File("myDirName");
myFile = new File(myDir, "myFile.txt");
```

`File` object does not allow you to access the contents of the file

File Tests and Utilities

- File Names

```
String getName()
String getPath()
String getAbsolutePath();
String getParent()
Boolean renameTo(File newName)
```

- File Tests
  - `boolean exists()`
  - `boolean canWrite()`
  - `boolean canRead()`
  - `boolean isFile()`
  - `boolean isDirectory()`
  - `boolean isAbsolute()`
- File Information and Utilities
  - `long lastModified()`
  - `long length()`
  - `boolean delete()`
- Directory Utilities
  - `boolean mkdir()`
  - `String[] list()`

File String I/O

- Reading from a file
  - Creating a **`Scanner`** object from a **`File`** object

```
Scanner in = new Scanner(new File("myFile.txt"));

String inLine = in.nextLine();
String aWord = in.next();      //read a single word delimited by whitespace)
int aNumber = in.nextInt();
```

- Writing to a file

```
PrintWriter out = new PrintWriter(new File("myFile.txt"));

out.println();
```

File Stream I/O

- Reading from a file
  - Use the **`FileReader`** class to read characters
  - Use the **`BufferedReader`** class to use the **`readLine()`** method

```
File file = new File("myFile.txt");
BufferedReader in = BufferedReader(new FileReader(file));
String s = in.readLine();
```

- Writing to a file
  - Use the **`FileWriter`** class to write characters
  - Use the **`PrintWriter`** class to use the **`print()`** and **`println()`** methods

```
File file = new File("myFile.txt");
PrintWriter out = new PrintWriter(new FileWriter(file));
out.println("some text");
```

or

```
PrintWriter out = new PrintWriter("myFile.txt");
```

Example: Read File

```
import java.io.*;
public class ReadFile {
    public static void main (String args[]) {
        File file = new File(args[0]); // Create file
        try {
            // Create a buffered reader to read each line from a file.
            BufferedReader in = new BufferedReader(new FileReader(file));
            String s;

            // Read each line from the file and echo it to the screen.
            while ((s = in.readLine()) != null) {
                System.out.println(s);
            }
            // Close the buffered reader, which also closes the file reader.
            in.close();

        } catch (FileNotFoundException e1) {
           // If this file does not exist
           System.err.println("File not found: " + file);

        } catch (IOException e2) {
           // Catch any other IO exceptions.
           e2.printStackTrace();
        }
   }
}
```

Example: Write File

```
import java.io.*;
public class WriteFile {
      public static void main (String args[]) {
          File file = new File(args[0]); // Create file
          try {
              // Create a buffered reader to read each line from standard in.
              BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
              // Create a print writer on this file.
              PrintWriter out = new PrintWriter(new FileWriter(file));
              String s;
              System.out.print("Enter file text.  ");
              System.out.println("[Type cntl-d to stop.]");
              // Read each input line and echo it to the screen.
              while ((s = in.readLine()) != null) {
                  out.println(s);
              }
              // Close the buffered reader and the file print writer.
              in.close();
              out.close();
          } catch (IOException e) {
               // Catch any IO exceptions.
               e.printStackTrace();
          }
      }
}
```

Reading and Writing Binary Data

- **`DataInput`** and **`DataOutput`** interfaces

```
writeChars()
writeChar()
writeByte()
writeInt()
writeFloat()
writeUTF()
…
```

```
readInt()
readChar()
readFloat()
readUTF()
readFully()
…
```

- **`DataInputStream`** and **`DataOutputStream`** classes
  - Implement **`DataInput/DataOutput`** interfaces

```
DataInputStream in = new DataInputStream(new FileInputStream("datafile.dat"));

DataOutputStream out = new DataOutputStream(new FileOutputStream("datafile.dat"));
```

Drawbacks of `java.io.File`

- Many methods didn't throw exceptions when they failed
- The rename method didn't work consistently across platforms
- There was no real support for symbolic links
- More support for metadata was desired
  - file permissions, file owner, and other security attributes
- Accessing file metadata was inefficient

Using `Path` Class

- **`Path`** object contains information about files and directories
  - A file is identified by its path through the file system
  - Since 1.7 in **`java.nio.file`** package

```
String getName()          int getNameCount()
String getName(int)       boolean exists()
void delete()
```

eg:

```
Path filePath = Paths.get("C:\\aFolder\\myFile.txt");
```

- Checking file accessibility

```
checkAccess()
```

eg:

```
try {
    filePath.checkAccess(AccessMode.READ, AccessMode.EXECUTE);
}
catch (IOException e) {…}
```

Reading/Writing Using `Path`

- Reading from **`inputStream`**

```
Path filePath = Paths.get("myFile.dat");

InputStream in = filePath.newInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(in));

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

- Writing to **`OutputStream`**

```
OutputStream out = filePath.newOutputStream(CREATE);
BufferedOutputStream output = new BufferedOutputStream(out);

output.write(theData);
output.flush();
output.close();
```

- Converting between **`File`** and **`Path`**

```
Path input = file.toPath();
File f = input.toFile();
```

Random-access Files

- **`RandomAccessFile`** class allows to find or write data anywhere in a file without reading through all data
  - implement **`DataInput/DataOutput`** interfaces

```
RandomAccessFile in = new RandomAccessFile("myData.dat", "r");
RandomAccessFile inOut = new RandomAccessFile("myData.dat", "rw");
```

- Current position of the file pointer

```
long getFilePointer()
```

- Position the file pointer to a byte position

```
void seek(long position)
```

Access Files Randomly

- Use **`FileChannel`** class
  - Created from **`Path`** object
    - Open/create a file
  - A **`FileChannel`** object is *seekable/readable/writable*

```
Path filePath = Paths.get("myFile.dat");

FileChannel fc = (FileChannel)filePath.newByteChannel(READ);
```

or

```
FileChannel fc = FileChannel.open(filePath, READ);

fc.position();                  //returns this channel's file position
fc.position(newPosition);       //sets this chennal's file position
fc.read(aByteBuffer);
fc.write(aByteBuffer);
```

Types of I/O

- Stream I/O
  - Reads/writes a byte/character/primitive data at a time
  - Each request is handled directly by the underlying OS

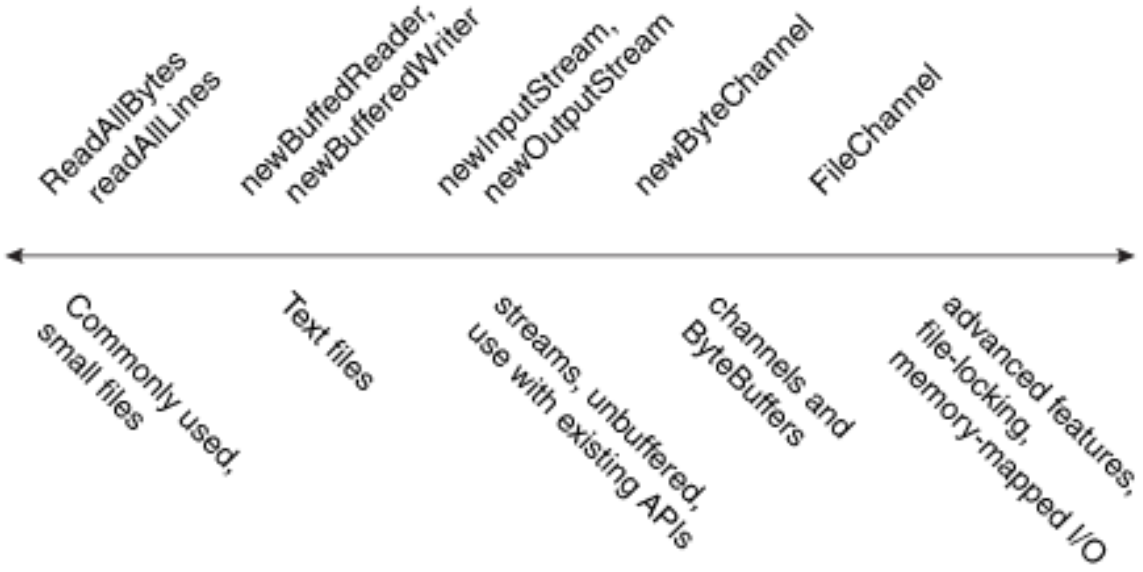
- Buffered stream I/O
  - Reads/writes a line of text at a time
  - Native input API is called only when the buffer is empty/full
  - Unbuffered streams can be converted to buffered streams

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
```

- Channel I/O
  - Reads/writes a buffer at a time
  - Capable to maintain a position in the channel
    - Random access

Files I/O Methods

Since 1.7



From less complex to more complex

Formatting Output

- **`printf()`** method
  - Similar to C **`printf()`** function

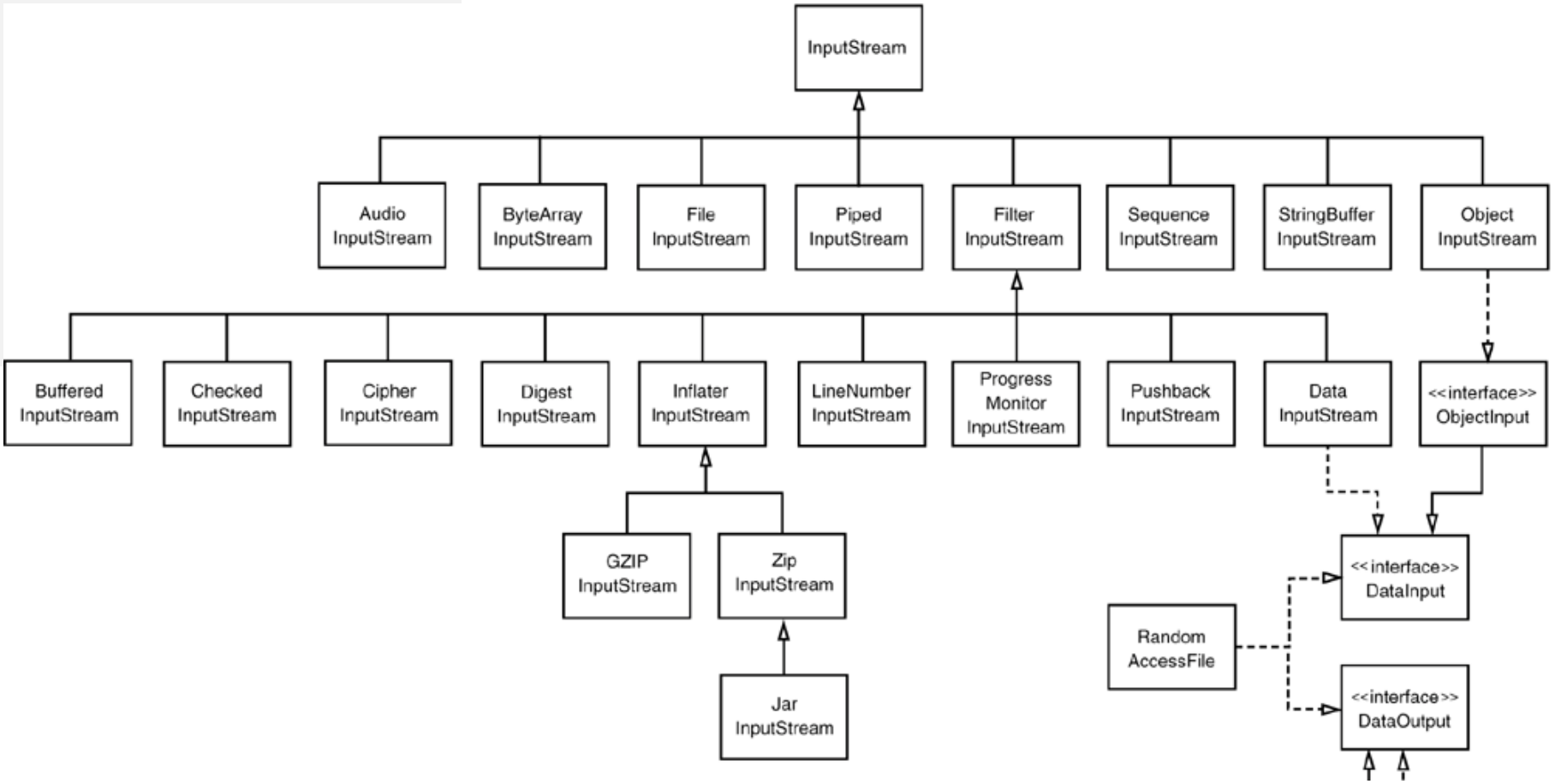
```
double x = 10000.0 / 3.0;

System.out.print(x);                    //prints 3333.3333333333335

System.out.printf("%8.2f", x);          //prints  3333.33

System.out.printf("%,.f", x);           //prints 3,333.33
```

flag conversion character

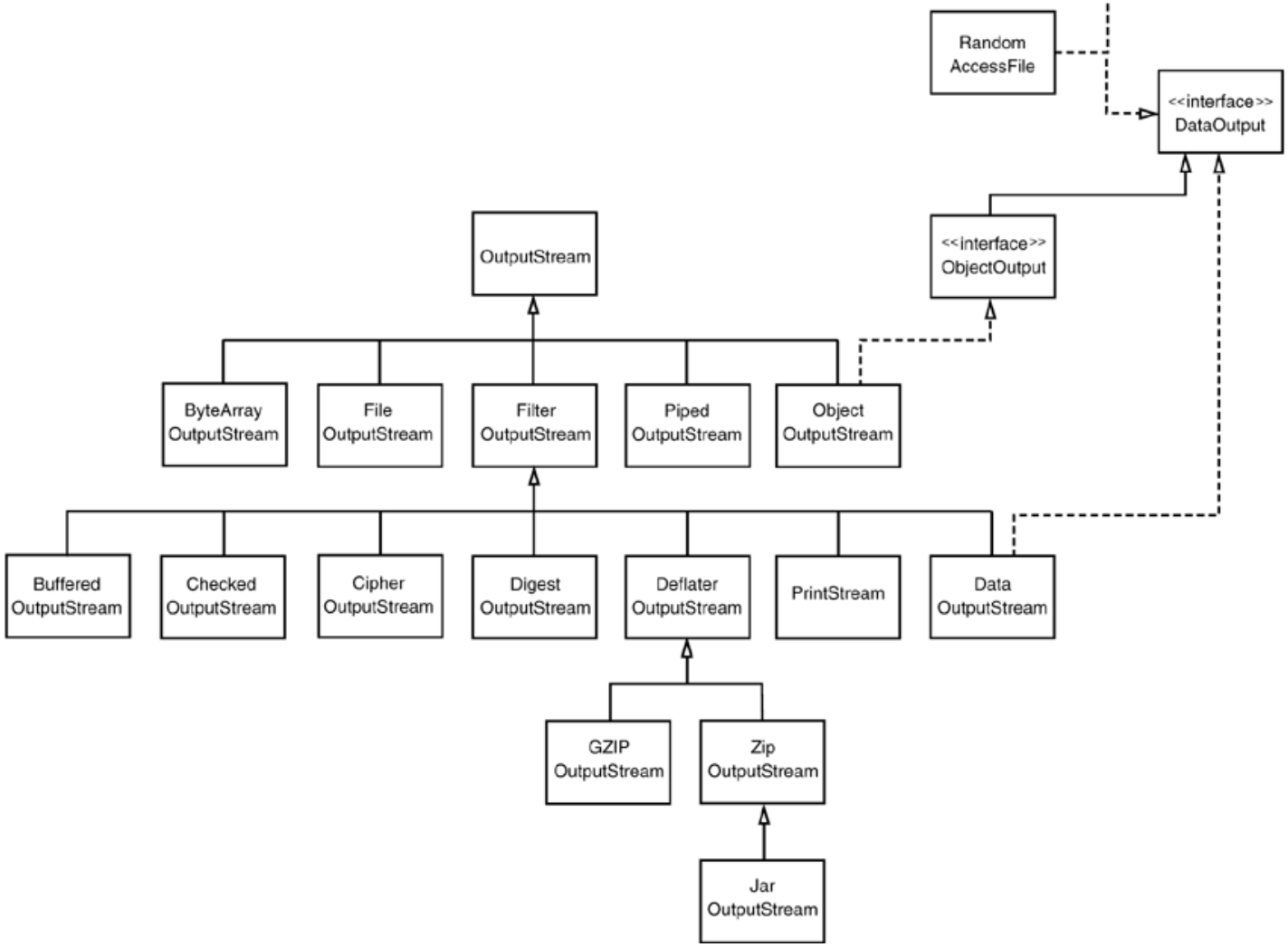
- **`String.format()`** method

```
String message = String.format("%s is %d years old", name, age);
```

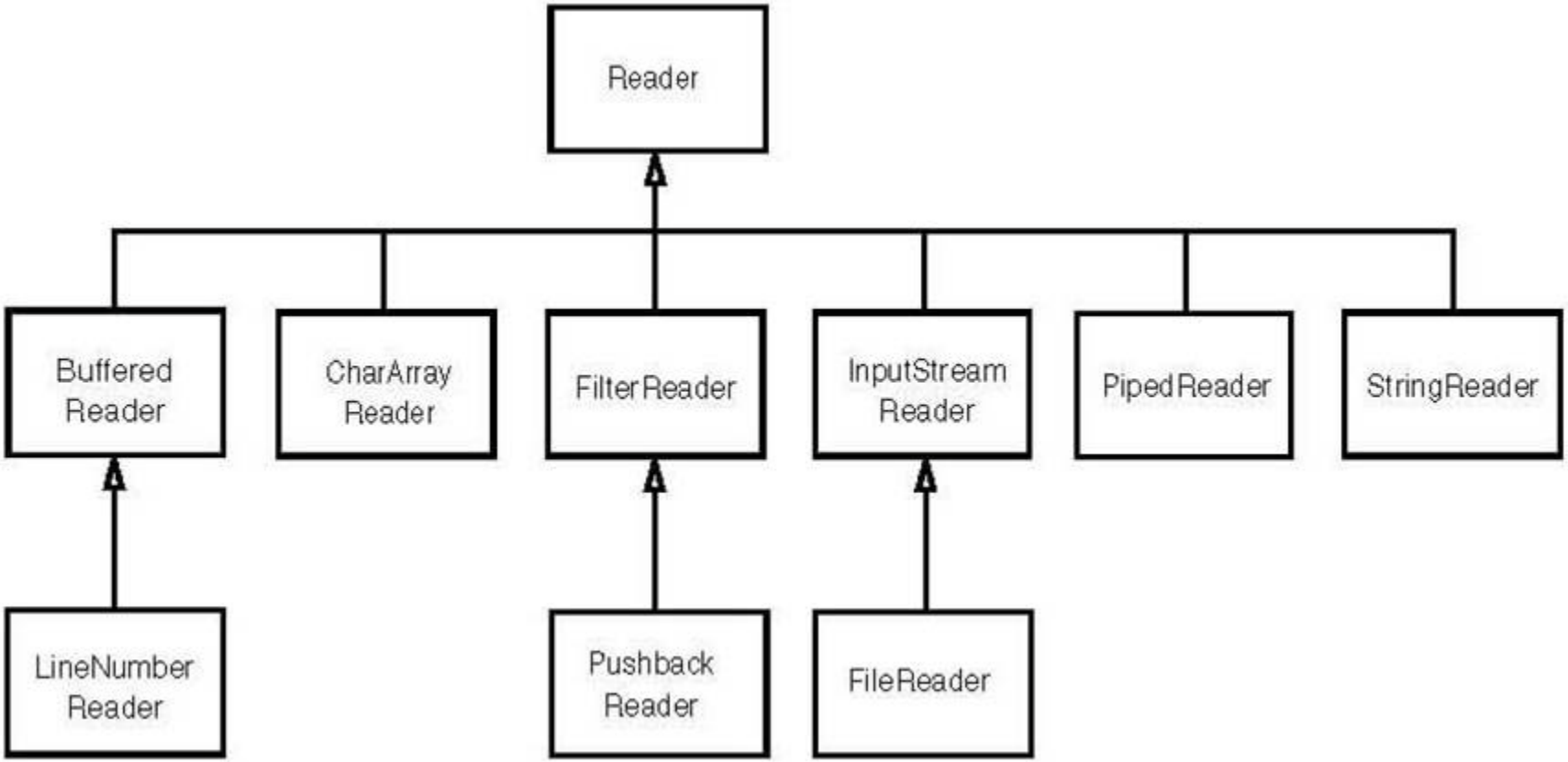
Input Stream Hierarchy



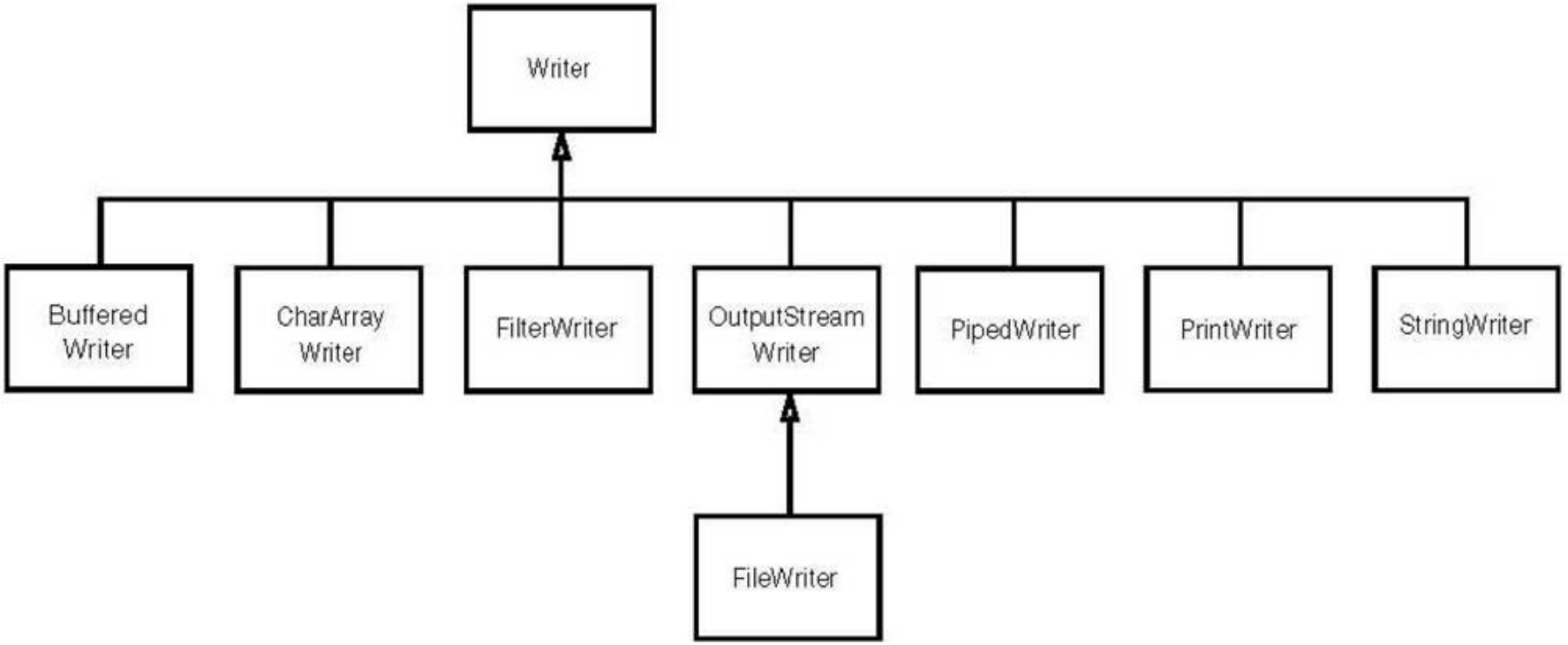
Output Stream Hierarchy



Reader Hierarchy



Writer Hierarchy



I/O Interfaces