Programming in Java

I/O Fundamentals, Console and File I/O

Hua Huang, Ph.D.

Spring 2019

Objectives

- Write a program that uses **command-line arguments** and **system properties**
- Examine the `Properties` class
- Construct **node** and **processing streams**, and use them appropriately
- **Serialize** and **deserialize** objects
- Distinguish **readers** and **writers** from streams, and select appropriately between them
- Read/Write data from/to the **console**
- Describe **files** and **file I/O**
- Use `RandomAccessFiles`

Command-Line Arguments

- Any Java technology application can use **command-line arguments**.
- These **string arguments** are placed on the command line to launch the Java interpreter **after the class name**:

```
java TestArgs arg1 arg2 "another arg"
```

- Each command-line argument is placed in the **`args`** array that is passed to the static `main` method:

```
public static void main(String[] args)
```

Question: how to access the command line arguments in C/C++ program?

Command-Line Arguments

```
public class TestArgs {
  public static void main(String[] args) {
    for ( int i = 0; i < args.length; i++ ) {
      System.out.println("args["+ i+"] is '"+args[i] + "'");
    }
  }
}
```

- Example execution:

```
java TestArgs arg0 arg1 "another arg"
args[0] is 'arg0'
args[1] is 'arg1'
args[2] is 'another arg'
```

System Properties

- System properties are a feature that **replaces** the concept of **environment variables** (which are platform-specific).
- The `System.getProperties` method returns a **`Properties`** object.
- The `getProperty` method returns a **String representing the value of the named property**.

- Use the **-D** option on the command line to **include a new property**.

- Question: What are environment variables??

The `Properties` Class

- The **`Properties`** class implements a mapping of names to values (a **String-to-String** map).
- The `propertyNames` method returns an **Enumeration** of all property names.
- The `getProperty` method returns a **String** representing the value of the named property.
- You can also read and write a properties collection into a file using `load` and `store`.

The Properties Class(Cont.)

```
1  import java.util.Properties;
2  import java.util.Enumeration;
3
4  public class TestProperties {
5    public static void main(String[] args) {
6      Properties props = System.getProperties();
7      props.list(System.out);
8    }
9  }
```

The Properties Class(Cont.)

- The following is an example test run of this program:

```
java -DmyProp=theValue TestProperties
```

- The following is the (partial) output:

```
...
java.runtime.name=OpenJDK Runtime Environment
file.encoding=UTF-8
java.vm.name=OpenJDK 64-Bit Server VM
myProp=theValue
java.vendor.url.bug=http://bugreport.java.com/bugreport/
java.io.tmpdir=/tmp
java.version=10.0.2
...
```

I/O Stream Fundamentals

- A stream is **a flow of data** from a **source or to a sink**.
- A **source stream**(源流) initiates the flow of data, also called an **input stream**.
- A **sink stream**(目标流) terminates the flow of data, also called an **output stream**.
- Sources and sinks are both **node streams**(节点流).
- Types of node streams are **files**, **memory**, and **pipes** between threads or processes.

Fundamental Stream Classes

Stream	Byte Streams	Character Streams
Source streams	`InputStream`	`Reader`
Sink streams	`OutputStream`	`Writer`

Data Within Streams

- Java technology supports **two types of streams**: **character** and **byte**.
- Input and output of **character data** is handled by **readers** and **writers**.
- Input and output of **byte** data is handled by **input streams** and **output streams**:
  - Normally, the term **stream** refers to a **byte stream**.
  - The terms **reader** and **writer** refer to **character streams**.

The `InputStream` Methods

- The three basic **`read`** methods are:
  - `int read()`
  - `int read(byte[] buffer)`
  - `int read(byte[] buffer, int offset, int length)`
- Other methods include:
  - `void close()`
  - `int available()`
  - `long skip(long n)`
  - `boolean markSupported()`
  - `void mark(int readlimit)`
  - `void reset()`

The `OutputStream` Methods

- The three basic `write` methods are:
  - `void write(int c)`
  - `void write(byte[] buffer)`
  - `void write(byte[] buffer, int offset, int length)`
- Other methods include:
  - `void close()`
  - `void flush()`

The `Reader` Methods

- The three basic `read` methods are:
  - `int read()`
  - `int read(char[] cbuf)`
  - `int read(char[] cbuf, int offset, int length)`
- Other methods include:
  - `void close()`
  - `boolean ready()`
  - `long skip(long n)`
  - `boolean markSupported()`
  - `void mark(int readAheadLimit)`
  - `void reset()`

The `Writer` Methods

- The basic `write` methods are:
  - `void write(int c)`
  - `void write(char[] cbuf)`
  - `void write(char[] cbuf, int offset, int length)`
  - `void write(String string)`
  - `void write(String string, int offset, int length)`
- Other methods include:
  - `void close()`
  - `void flush()`

Node Streams

Type	Character Streams	Byte Streams
File	`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`
Memory: array	`CharArrayReader` `CharArrayWriter`	`ByteArrayInputStream` `ByteArrayOutputStream`
Memory: string	`StringReader` `StringWriter`	N/A
Pipe	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`

A Simple Example

- This program performs a copy file operation using a manual buffer:

```
java TestNodeStreams file1 file2
```

```
01  import java.io.*;
02  public class TestNodeStreams {
03    public static void main(String[] args) {
04      try {
05        FileReader input = new FileReader(args[0]);
06        try {
07          FileWriter output = new FileWriter(args[1]);
08          try {
09            char[] buffer = new char[128];
10            int charsRead;
11            // read the first buffer
12            charsRead = input.read(buffer);
```

A Simple Example(Cont.)

```
13              while ( charsRead != -1 ) {
14                // write buffer to the output file
15                output.write(buffer, 0, charsRead);
16                // read the next buffer
17                charsRead = input.read(buffer);
18              }
19            }
20            finally {
21              output.close();
22            }
23          }
24          finally {
25            input.close();
26          }
27        } catch (IOException e) {
28          e.printStackTrace();
29        }
30      }
31  }
```

Buffered Streams

```
01  import java.io.*;
02  public class TestBufferedStreams {
03    public static void main(String[] args) {
04      try {
05        FileReader input = new FileReader(args[0]);
06        BufferedReader bufInput = new BufferedReader(input);
07        try {
08          FileWriter output = new FileWriter(args[1]);
09          BufferedWriter bufOutput= new BufferedWriter(output);
10          try {
11            String line;
12            // read the first line
13            line = bufInput.readLine();
14            while ( line != null ) {
15              // write the line out to the output file
16              bufOutput.write(line, 0, line.length());
17              bufOutput.newLine();
18              // read the next line
19              line = bufInput.readLine();
20            }
21          }
```

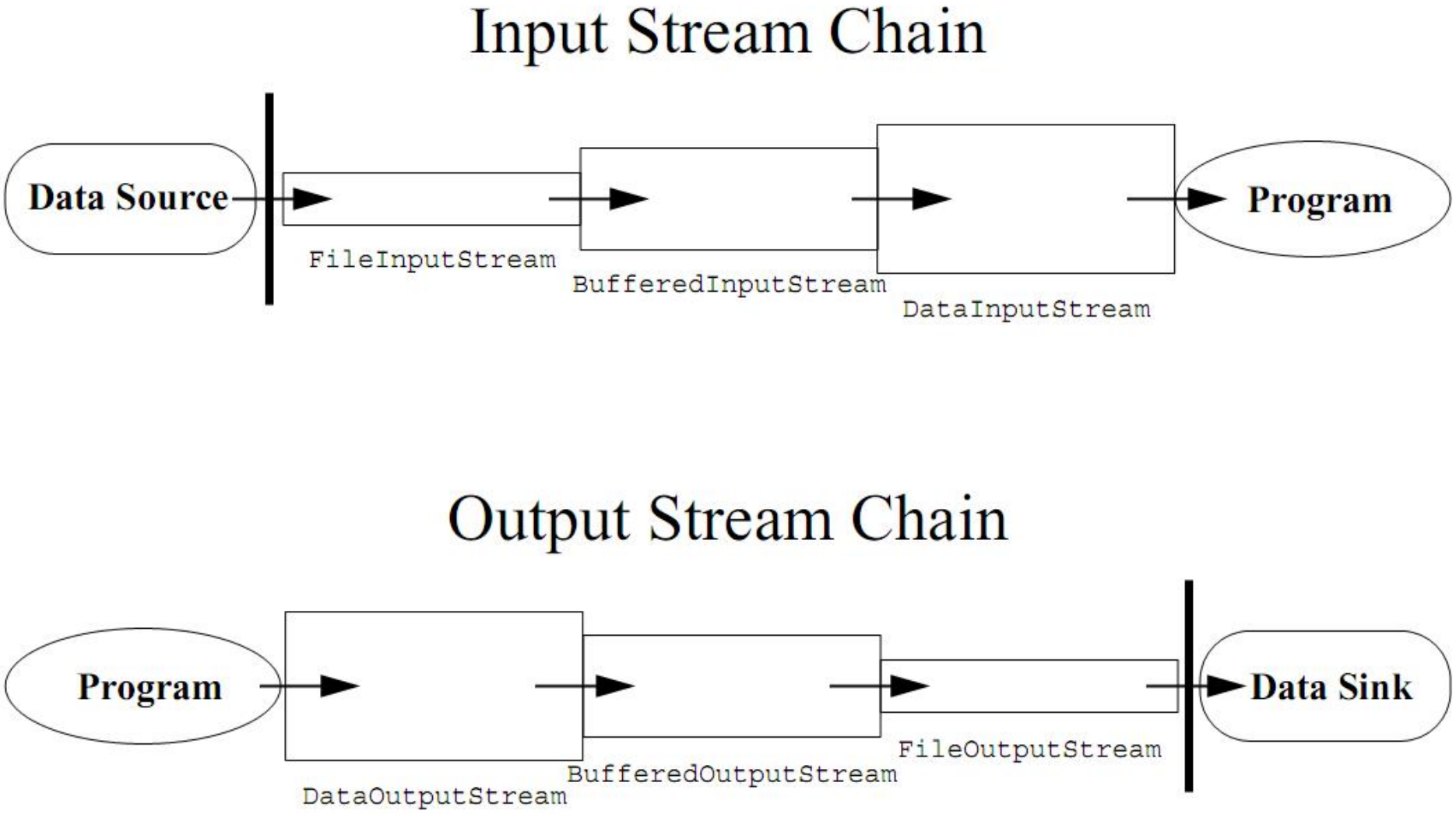
Buffered Streams(Cont.)

```
22            finally {
23              bufOutput.close();
24            }
25          }
26          finally {
27            bufInput.close();
28          }
29        } catch (IOException e) {
30          e.printStackTrace();
31        }
32      }
33    }
```

- This program performs a copy file operation using a built-in buffer:

```
java TestBufferedStreams file1 file2
```

I/O Stream Chaining



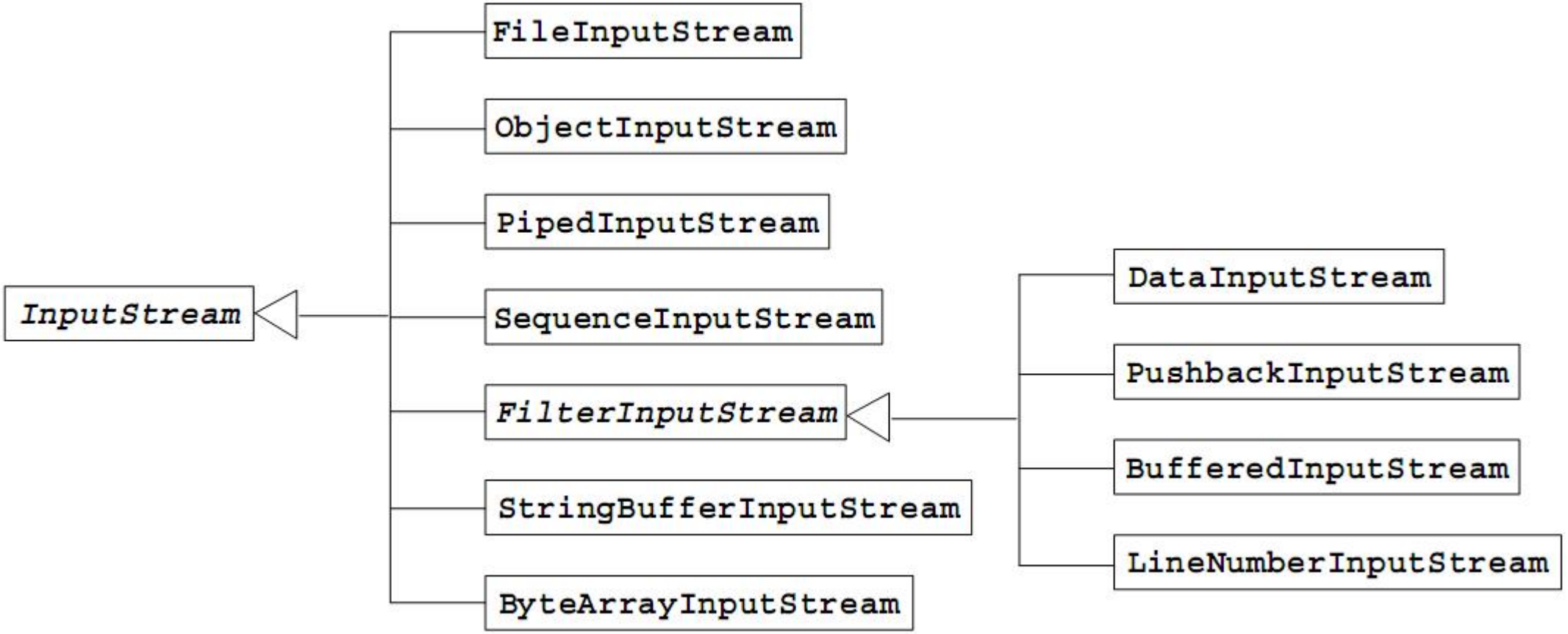
Processing Streams

Type	Character Streams	Byte Streams
Buffering	`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`
Filtering	*`FilterReader`* *`FilterWriter`*	*`FilterInputStream`* *`FilterOutputStream`*
Converting between bytes and character	`InputStreamReader` `OutputStreamWriter`	
Performing object serialization		`ObjectInputStream` `ObjectOutputStream`

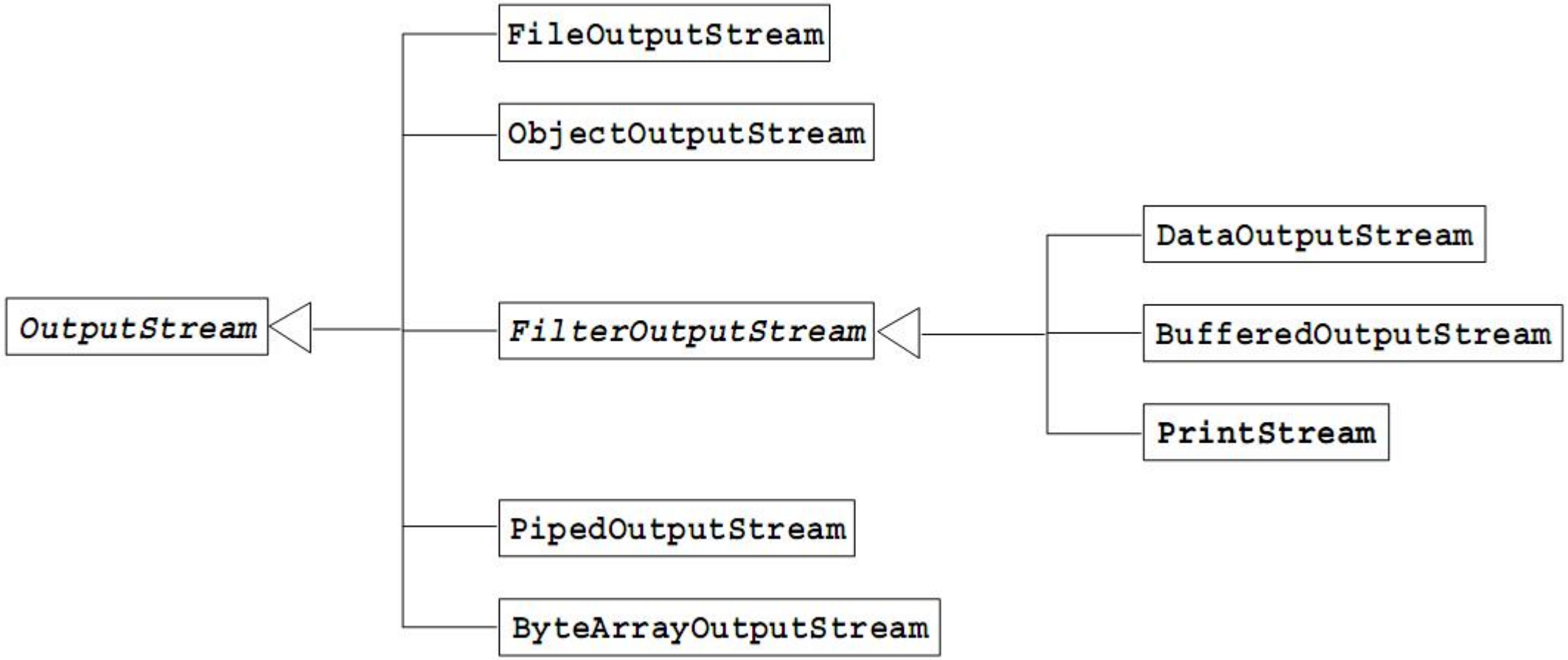
Processing Streams(Cont.)

Type	Character Streams	Byte Streams
Performing data conversion		`DataInputStream` `DataOutputStream`
Counting	`LineNumberReader`	`LineNumberInputStream`
Peeking ahead	`PushbackReader`	`PushbackInputStream`
Printing	`PrintWriter`	`PrintStream`

The `InputStream` Class Hierarchy



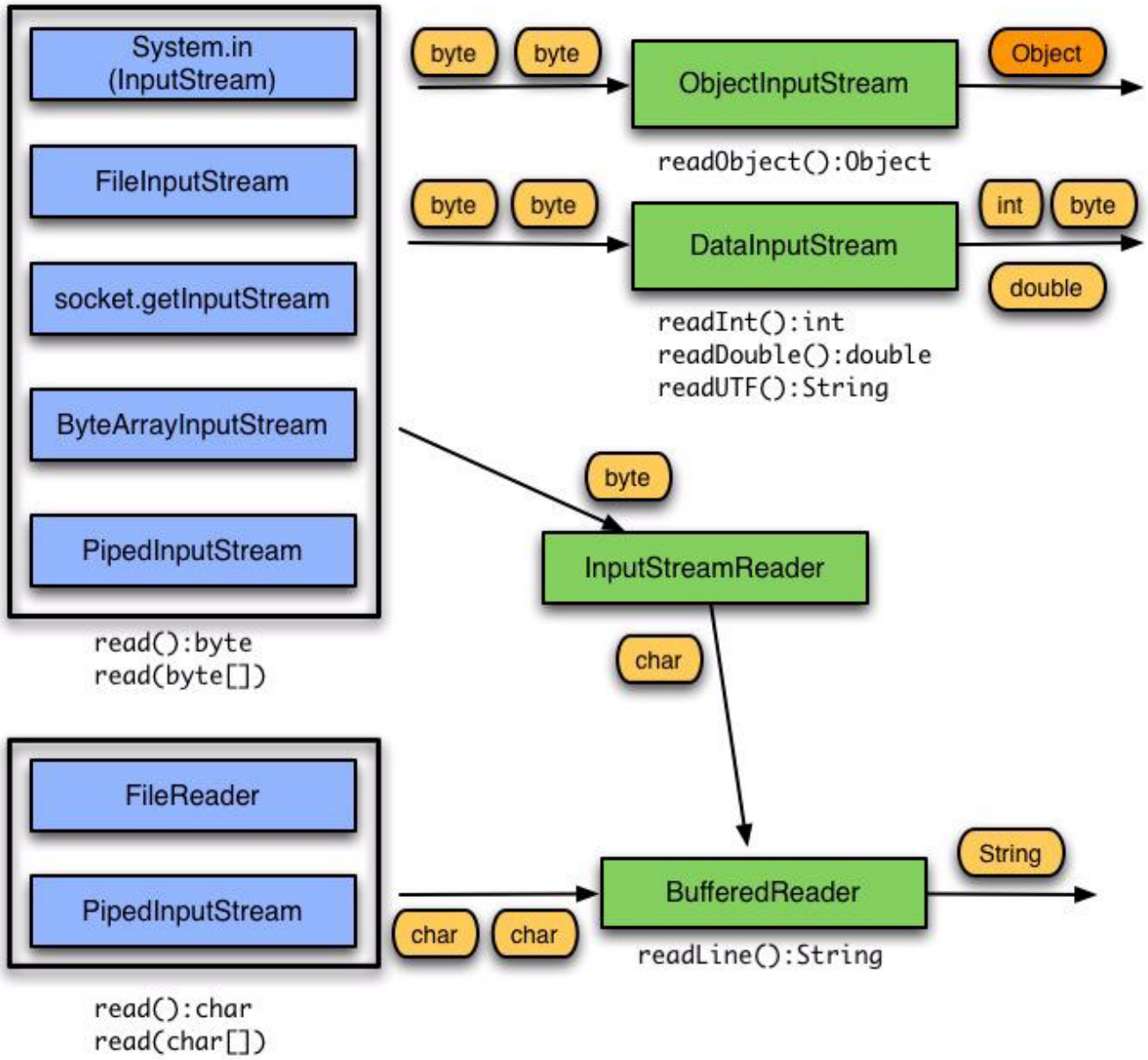
The `OutputStream` Class Hierarchy



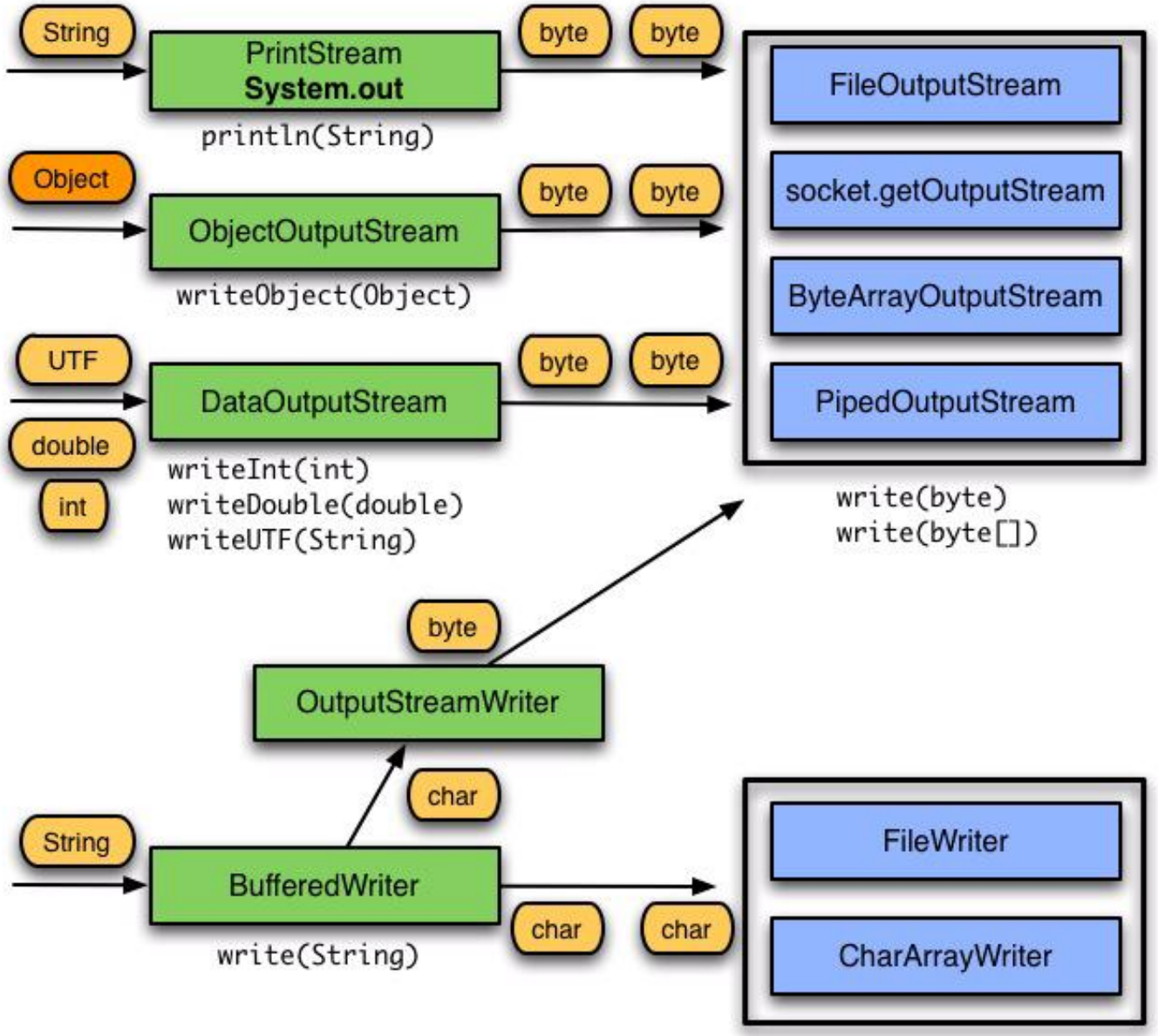
The `ObjectInputStream` and `ObjectOutputStream` Classes

- The Java API provides a standard mechanism (called **object serialization**) that completely automates the process of **writing and reading objects** from streams.
- When **writing an object**, the object output stream writes the **class name**, followed by a **description of the data members** of the class, in the order they appear in the stream, followed by the **values for all the fields** on that object.
- When **reading an object**, the object input stream reads the **name of the class** and the **description of the class** to match against the class in memory, and it reads the **values** from the stream to populate a newly allocation instance of that class.
- **Persistent storage** of objects can be accomplished if files (or other persistent storage) are used as streams.

Input Chaining Combinations: A Review



Output Chaining Combinations: A Review



Serialization

- **Serialization** is a mechanism for **saving the objects as a sequence of bytes** and **rebuilding** them later when needed.
- When an object is serialized, **only the fields of the object are preserved**
- When a field references an **object**, the fields of the referenced object are **also serialized**
- Some object classes are **not serializable** because their fields represent **transient** operating system-specific information. (**static attributes are not serialized**)
- For objects of a specific class to be serializable, the class must implement the `java.io.Serializable` interface

Serialization(Cont.)

```
1  public class MyClass implements Serializable {
2       public transient Thread myThread;
3       private transient String customerID;
4       private int total;
5     }
```

- The field **access modifier** (`public`, `protected`, default, and `private`) **has no effect on the data field being serialized**.
- **`Strings`** represented as file system safe universal character set transformation format (**UTF**) characters.
- The `transient` keyword prevents the data from being serialized.

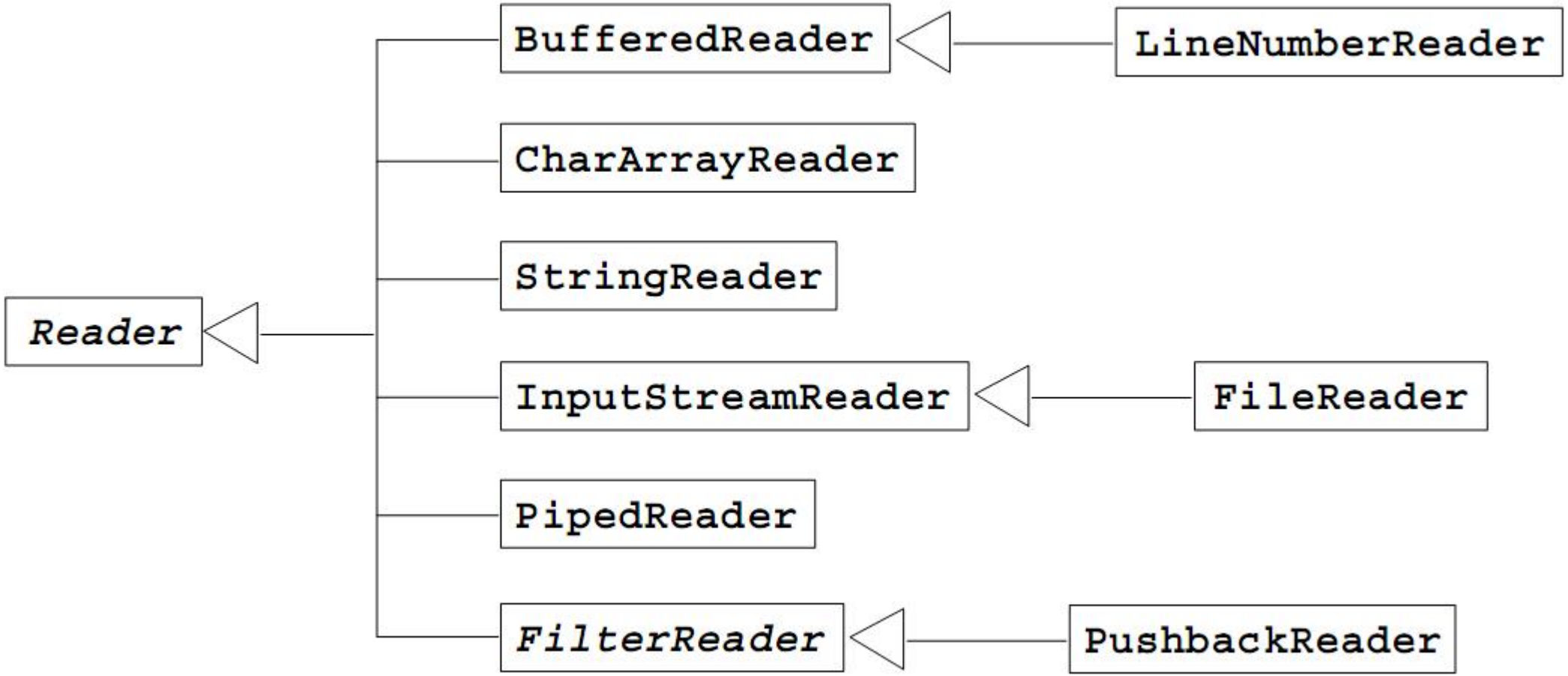
The SerializeDate Class

```
01  import  java.io.*;
02  import  java.util.Date;
03  public  class SerializeDate {
04    SerializeDate() {
05      Date d = new Date ();
06      try {
07        FileOutputStream f =
08          new FileOutputStream ("date.ser");
09        ObjectOutputStream s =
10          new ObjectOutputStream (f);
11        s.writeObject (d);
12        s.close ();
13      } catch (IOException e) {
14        e.printStackTrace ();
15      }
16    }
17    public static void main (String args[]) {
18      new SerializeDate();
19    }
20  }
```

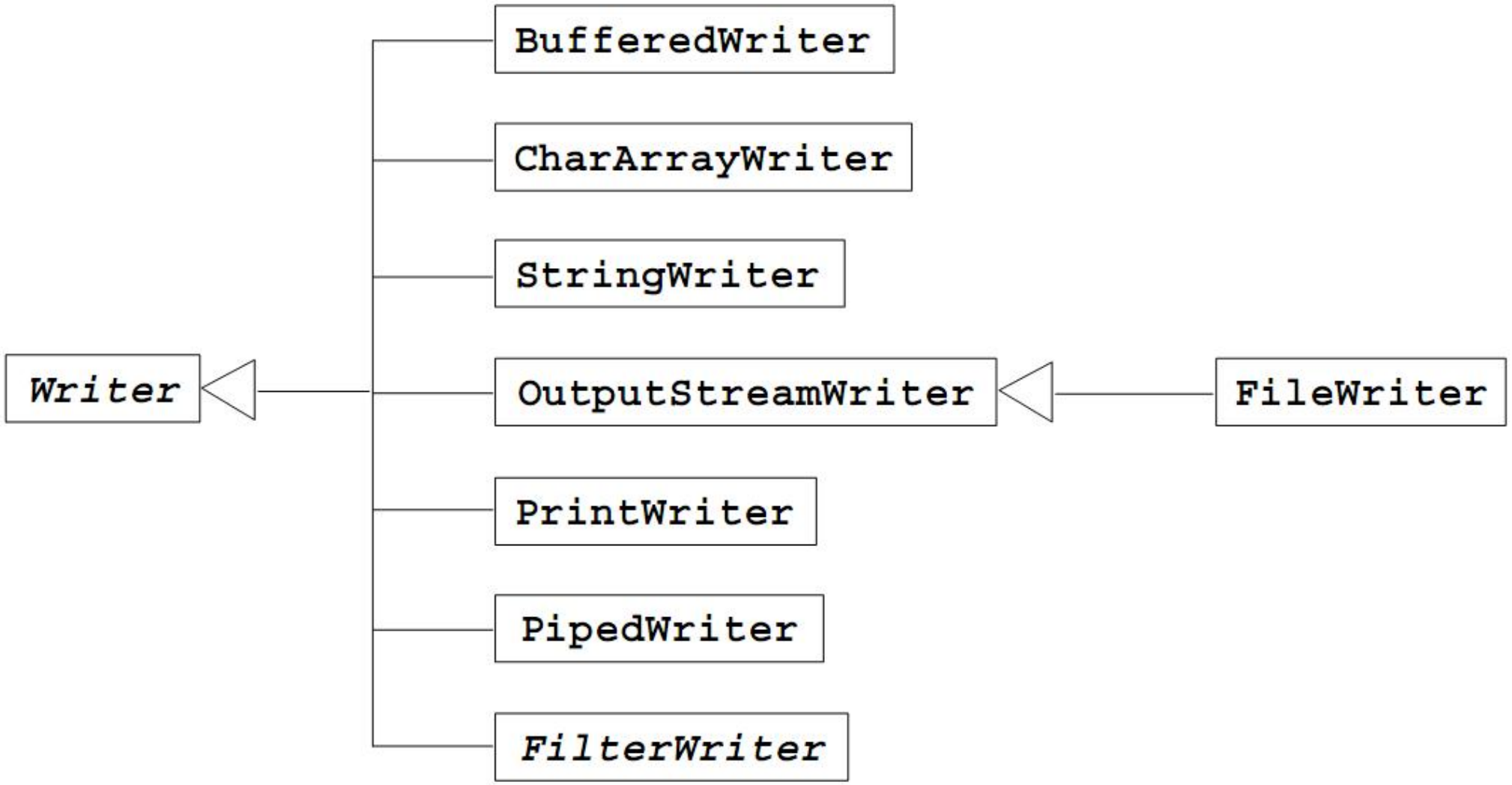
The DeSerializeDate Class

```
01  import java.io.*;
02  import java.util.Date;
03  public class DeSerializeDate {
04    DeSerializeDate () {
05      Date d = null;
06      try {
07        FileInputStream f = new FileInputStream ("date.ser");
08        ObjectInputStream s = new ObjectInputStream (f);
09        d = (Date)s.readObject ();
10        s.close ();
11      } catch (Exception e) {
12        e.printStackTrace ();
13      }
14      System.out.println("Deserialized Date object from date.ser");
15      System.out.println("Date: "+d);
16    }
17    public static void main (String args[]) {
18      new DeSerializeDate();
19    }
20  }
```

The `Reader` Class Hierarchy



The `Writer` Class Hierarchy



Console I/O

- The variable **`System.out`** enables you to write to standard output.
  - **`System.out`** is an object of type **`PrintStream`**.
- The variable **`System.in`** enables you to read from standard input.
  - **`System.in`** is an object of type **`InputStream`**.
- The variable **`System.err`** enables you to write to standard error.
  - **`System.err`** is an object of type **`PrintStream`**.

Writing to Standard Output

- The **`println`** methods print the argument and a **newline** character (**\n**).

  Caution: Don't use hard-coding newline.

- The **`print`** methods print the argument **without a newline** character.
- The **`print`** and **`println`** methods are overloaded for most primitive types (**`boolean`**, **`char`**, **`int`**, **`long`**, **`float`**, and **`double`**) and for **`char[]`**, **`Object`**, and **`String`**.
- The **`print(Object)`** and **`println(Object)`** methods call the **`toString`** method on the argument.

Reading From Standard Input

```
01  import java.io.*;
02  public class KeyboardInput {
03    public static void main (String args[]) {
04      String s;
05    // Create a bufferedreader to rea each line from the keyboard.
06      InputStreamReader ir = new InputStreamReader(System.in);
07      BufferedReader in = new BufferedReader(ir);
08      System.out.println("Unix: ctrl-d\nWindows: ctrl-z to exit");
09      try {// Read each input line and echo it to the screen.
10        s = in.readLine();
11        while ( s != null ) {
12          System.out.println("Read: " + s);
13          s = in.readLine();
14        }
15        in.close();// Close the buffered reader.
16      } catch (IOException e) { // Catch any IO exceptions.
17        e.printStackTrace();
18      }
19    }
20  }
```

Simple Formatted Output

- You can use the formatting functionality as follows:

```
out.printf("name count\n");
String s = String.format("%s %5d%n", user, total);
```

- Common formatting codes are listed in this table.

Code	Description
`%s`	Formats the argument as a string, usually by calling the `toString`() method on the object.
`%d %o %x`	Formats an integer, as a decimal, octal, or hexadecimal value.
`%f %g`	Formats a floating point number. The `%g` code uses scientific notation.
`%n`	Inserts a **newline** character to the string or stream.
`%%`	Inserts the **%** character to the string or stream.

Simple Formatted Input

- The **`Scanner`** class provides a **formatted input** function.
- A **`Scanner`** class can be used with c**`onsole input streams`** as well as **`file`** or **`network streams`**.
- You can read console input as follows:

```
01  import  java.io.*;
02  import  java.util.Scanner;
03  public  class ScannerTest {
04    public static void main(String [] args) {
05      Scanner s = new Scanner(System.in);
06      String param = s.next();
07      System.out.println("the param 1" + param);
08      int value = s.nextInt();
09      System.out.println("second param" + value);
10      s.close();
11    }
12  }
```

Files and File I/O

- The `java.io` package enables you to do the following:
  - Create **`File`** objects
  - Manipulate File objects
  - **Read and write to file streams**

Creating a New `File` Object

- The File class provides several utilities:

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

- **Directories are treated like files** in the Java programming language. You can create a File object that represents a directory and then use it to identify other files, for example:

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

The `File` Tests and Utilities

- File information:
  - `String getName()`
  - `String getPath()`
  - `String getAbsolutePath()`
  - `String getCanonicalPath()`
  - `String getParent()`
  - `long lastModified()`
  - `long length()`
- File modification:
  - `boolean renameTo(File newName)`
  - `boolean delete()`

The File Tests and Utilities(Cont.)

- Directory utilities:
  - `boolean mkdir()`
  - `String[] list()`
- File tests:
  - `boolean exists()`
  - `boolean canWrite()`
  - `boolean canRead()`
  - `boolean isFile()`
  - `boolean isDirectory()`
  - `boolean isAbsolute();`
  - `boolean is Hidden();`

File Stream I/O

- For file input:
  - Use the **`FileReader`** class to read **characters**.
  - Use the **`BufferedReader`** class to use the **`readLine`** method.
- For file output:
  - Use the **`FileWriter`** class to write **characters**.
  - Use the **`PrintWriter`** class to use the **`print`** and **`println`** methods.

File Input Example

```
01  import java.io.*;
02  public class ReadFile {
03    public static void main (String[] args) {
04      File file = new File(args[0]);// Create file
05      try {// Create a buffered reader to read each line from a file.
06        BufferedReader in = new BufferedReader(new FileReader(file));
07        String s;
08        s = in.readLine();// Read each line from the file and echo it.
09        while ( s != null ) {
10          System.out.println("Read: " + s);
11          s = in.readLine();
12        }
13        in.close(); // Close the buffered reader
14      } catch (FileNotFoundException e1) {
15        System.err.println("File not found: " + file); // If not exist
16      } catch (IOException e2) {
17        e2.printStackTrace(); // Catch any other IO exceptions.
18      }
19    }
20  }
```

File Output Example

```
01  import java.io.*;
02  public class WriteFile {
03    public static void main (String[] args) {
04      File file = new File(args[0]);// Create file
05      try {// Create a bufferedreader toline from  read each standard in.
06        InputStreamReader isr = new InputStreamReader(System.in);
07        BufferedReader in = new BufferedReader(isr);
08        PrintWriter out = new PrintWriter(new FileWriter(file)); /
09        String s;
10        System.out.print("Enter file text.");
11        System.out.println("[Type ctrl-d to stop.]");
12        while ((s = in.readLine()) != null) { // Read input line and echo
13          out.println(s);
14        }
15        in.close(); // Close the buffered reader and the file print writer.
16        out.close();
17      } catch (IOException e) {
18        e.printStackTrace(); // Catch any IO exceptions.
19      }
20    }
21  }
```

Creating a Random Access File

- With the file name:

```
myRAFile = new RandomAccessFile(String filename, String mode);
```

- With a File object:

```
myRAFile = new RandomAccessFile(File file, String mode);
```

- Mode is a string, can be **"r"** for read only or **"rw"** for read and write, refer to JDK API for more info.

Questions or Comments?