CHAPTER 16

Managing Input/Output Files in Java

16.1 INTRODUCTION

So far we have used variables and arrays for storing data inside the programs. This approach poses the following problems.

1. The data is lost either when a variable goes out of scope or when the program is terminated. That is, the storage is temporary.
2. It is difficult to handle large volumes of data using variables and arrays.

We can overcome these problems by storing data on *secondary storage devices* such as floppy disks or hard disks. The data is stored in these devices using the concept of *files*. Data stored in files is often called *persistent* data.

A file is a collection of related *records* placed in a particular area on the disk. A record is composed of several fields and a field is a group of characters as illustrated in Fig. 16.1. Characters in Java are *Unicode* characters composed of two *bytes*, each byte containing eight binary digits, 1 or 0.

Storing and managing data using files is known as *file processing* which includes tasks such as creating files, updating files and manipulation of data. Java supports many powerful features for managing input and output of data using files. Reading and writing of data in a file can be done at the level of bytes or characters or fields depending on the requirements of a particular application. Java also provides capabilities to read and write class objects directly. Note that a record may be represented as a class object in Java. The process of reading and writing objects is called *object serialization*. In this chapter, we discuss various features supported by Java for file processing.

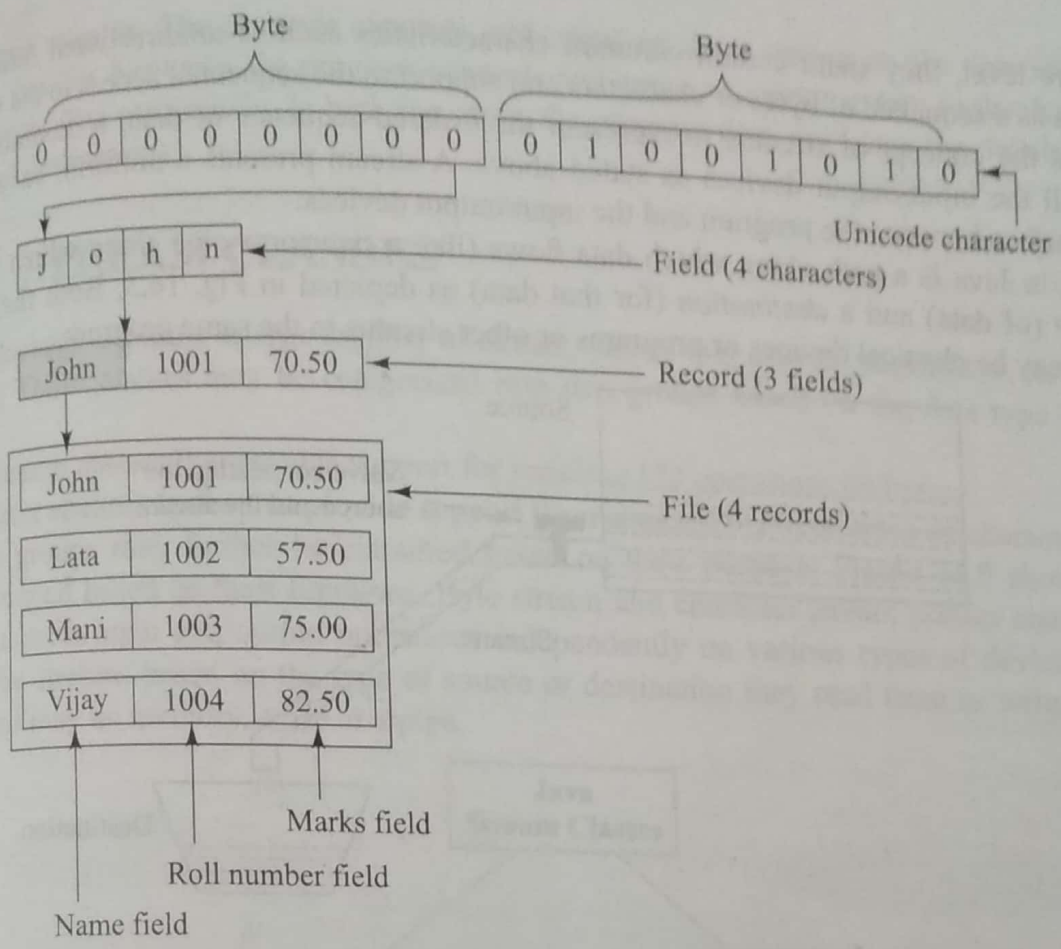

Fig. 16.1 *Data representation in Java files*

16.2 CONCEPT OF STREAMS

In file processing, input refers to the flow of data into a program and output means the flow of data out of a program. Input to a program may come from the keyboard, the mouse, the memory, the disk, a network, or another program. Similarly, output from a program may go to the screen, the printer, the memory, the disk, a network, or another program. This is illustrated in Fig. 16.2. Although these devices look very different at

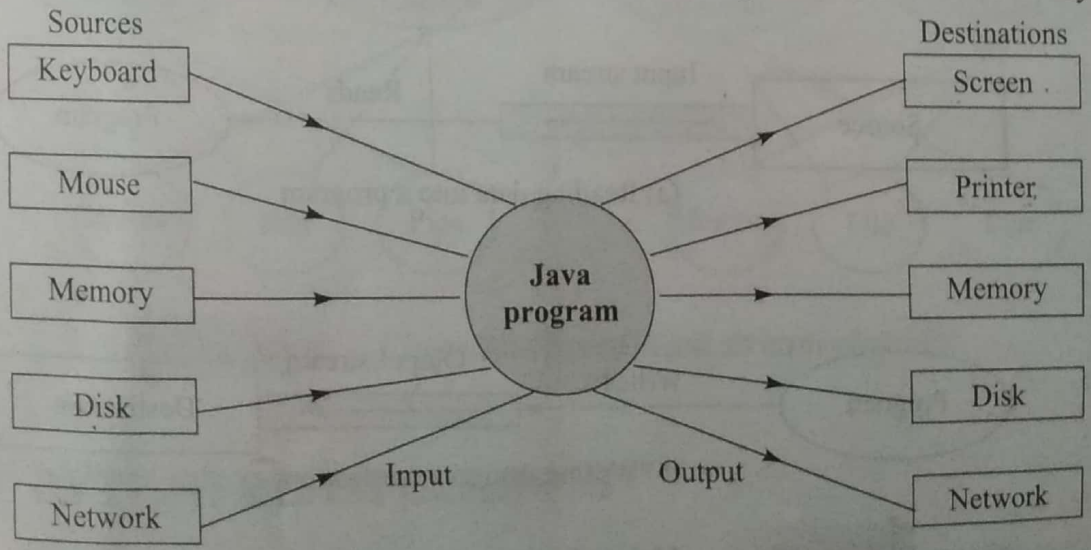

Fig. 16.2 *Relationship of Java program with I/O devices*

the hardware level, they share certain common characteristics such as unidirectional movement of data, treating data as a sequence of bytes or characters and support to the sequential access to the data.

Java uses the concept of streams to represent the ordered sequence of data, a common characteristic shared by all the input/output devices as stated above. A stream presents a uniform, easy-to-use, object-oriented interface between the program and the input/output devices.

A stream in Java is a path along which data flows (like a river or a pipe along which water flows). It has a *source* (of data) and a *destination* (for that data) as depicted in Fig. 16.3. Both the source and the destination may be physical devices or programs or other streams in the same program.

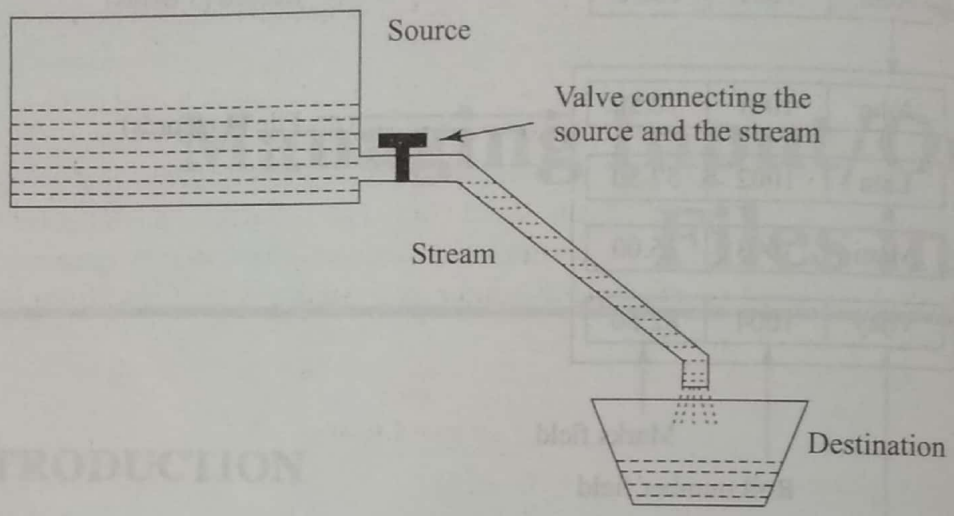

Fig. 16.3 *Conceptual view of a stream*

The concept of sending data from one stream to another (like one pipe feeding into another pipe) has made streams in Java a powerful tool for file processing. We can build a complex file processing sequence using a series of simple stream operations. This feature can be used to filter data along the pipeline of streams so that we obtain data in a desired format. For example, we can use one stream to get raw data in binary format and then use another stream in series to convert it to integers.

Java streams are classified into two basic types, namely, *input stream* and *output stream*. An input stream extracts (i.e. *reads)* data from the source (file) and sends it to the program. Similarly, an output stream takes data from the program and sends (i.e. *writes)* it to the destination (file). Figure 16.4 illustrates the use of

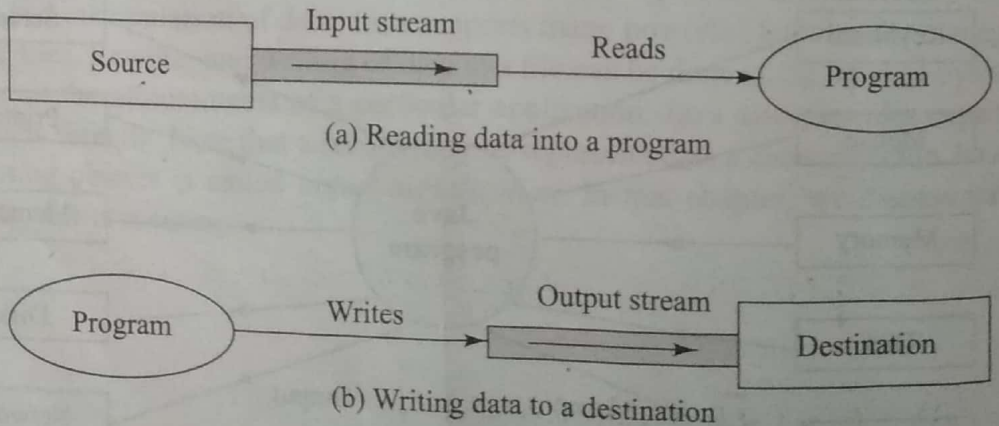

Fig. 16.4 *Using input and output streams*

input and output streams. The program connects and opens an input stream on the data source and then reads the data serially. Similarly, the program connects and opens an output stream to the destination place of data and writes data out serially. In both the cases, the program does not know the details of end points (i.e. source and destination).

16.3 STREAM CLASSES

The java.io package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

1. Byte stream classes that provide support for handling I/O operations on bytes.
2. Character stream classes that provide support for managing I/O operations on characters.

These two groups may further be classified based on their purpose. Figure 16.5 shows how stream classes are grouped based on their functions. Byte stream and character stream classes contain specialized classes to deal with input and output operations independently on various types of devices. We can also cross-group the streams based on the type of source or destination they read from or write to. The source (or destination) may be memory, a file or a pipe.

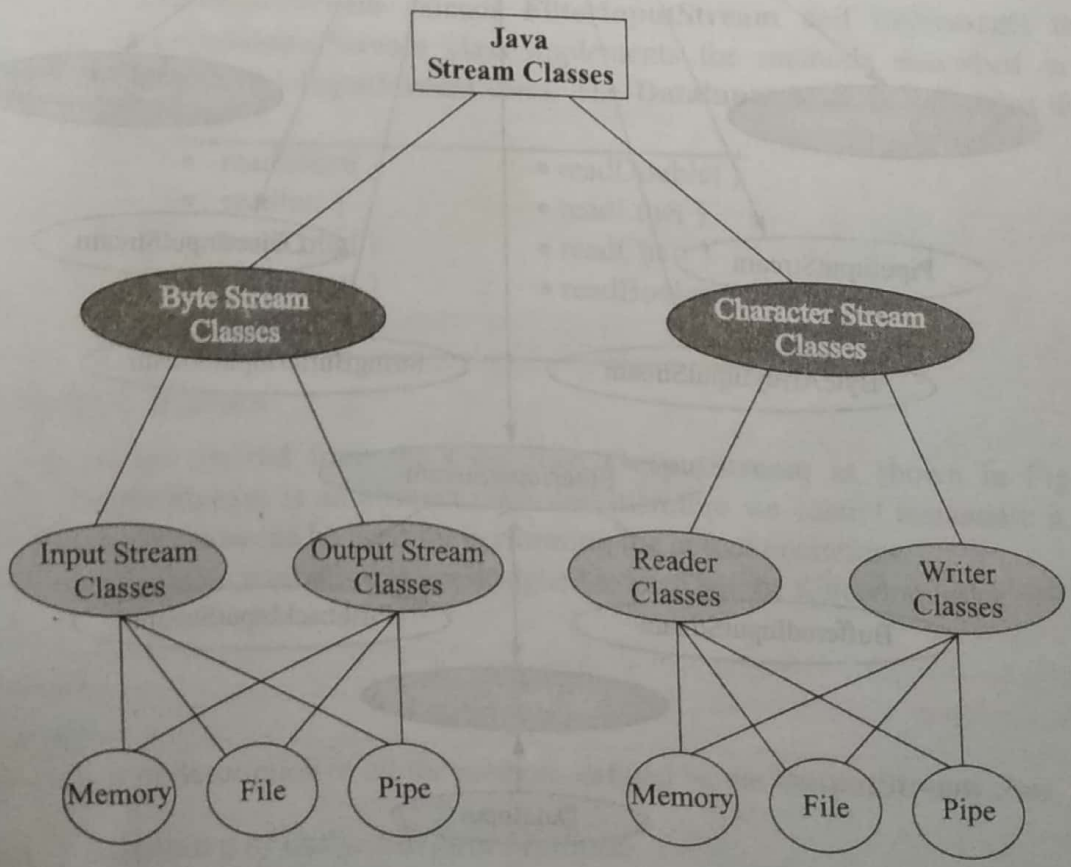

Fig. 16.5 *Classification of Java stream classes*

16.4 BYTE STREAM CLASSES

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Since the streams are unidirectional, they can transmit bytes

in only one direction and, therefore, Java provides two kinds of byte stream classes: *input stream classes* and *output stream* classes.

Input Stream Classes

Input stream classes that are used to read 8-bit bytes include a super class known as **InputStream** and a number of subclasses for supporting various input-related functions. Figure 16.6 shows the class hierarchy of input stream classes.

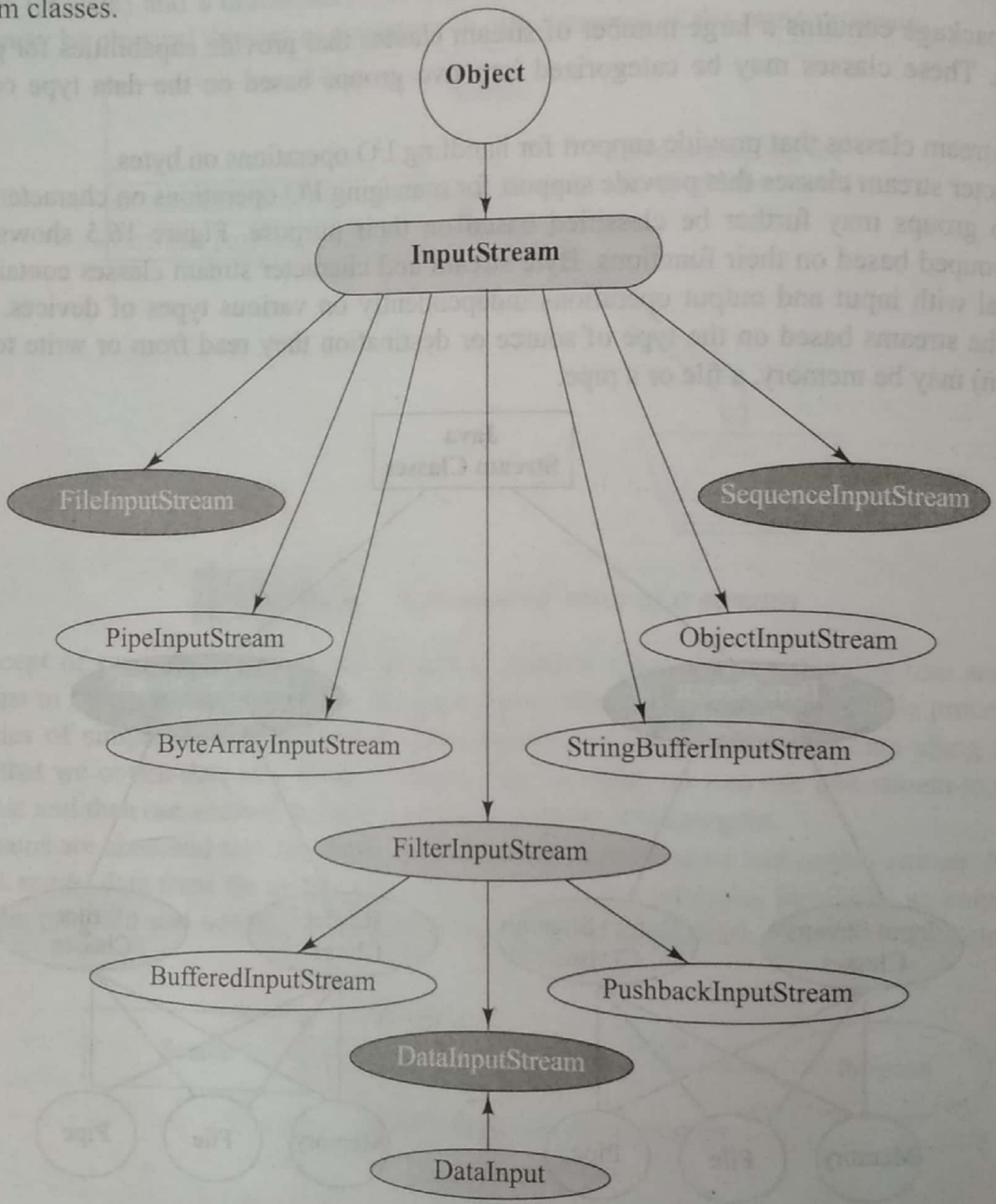

Fig. 16.6 *Hierarchy of input stream classes*

The super class **InputStream** is an abstract class, and, therefore, we cannot create instances of this class. Rather, we must use the subclasses that inherit from this class. The **InputStream** class defines methods for performing input functions such as

- Reading bytes
- Closing streams

- Marking positions in streams
- Skipping ahead in a stream
- Finding the number of bytes in a stream

Table 16.1 gives a brief description of all the methods provided by the **InputStream** class.

Table 16.1 *Summary of InputStream Methods*

Method	*Description*
1. read ( )	Reads a byte from the input stream
2. read (byte b[ ])	Reads an array of bytes into b
3. read (byte b[ ], int n, int m)	Reads m bytes into b starting from nth byte
4. available( )	Gives number of bytes available in the input
5. skip(n)	Skips over n bytes from the input stream
6. reset( )	Goes back to the beginning of the stream
7. close( )	Closes the input stream

Note that the class **DataInputStream** extends **FilterInputStream** and implements the interface **DataInput.** Therefore, the **DataInputStream** class implements the methods described in **DataInput** in addition to using the methods of **InputStream** class. The **DataInput** interface contains the following methods:

- readShort( )
- readInt( )
- readLong( )
- readFloat( )
- readUTF( )
- readDouble( )
- readLine( )
- readChar( )
- readBoolean( )

Output Stream Classes

Output stream classes are derived from the base class **OutputStream** as shown in Fig. 16.7. Like **InputStream,** the **OutputStream** is an abstract class and therefore we cannot instantiate it. The several subclasses of the **OutputStream** can be used for performing the output operations.

The **OutputStream** includes methods that are designed to perform the following tasks:

- Writing bytes
- Closing streams
- Flushing streams

Table 16.2 gives a brief description of all the methods defined by the **OutputStream** class.

Table 16.2 *Summary of OutputStream Methods*

Method	*Description*
1. write ( )	Writes a byte to the output stream
2. write (byte b[ ])	Writes all bytes in the array b to the output stream
3. write (byte b[ ], int n, int m)	Writes m bytes from array b starting from nth byte
4. close( )	Closes the output stream
5. flush( )	Flushes the output stream

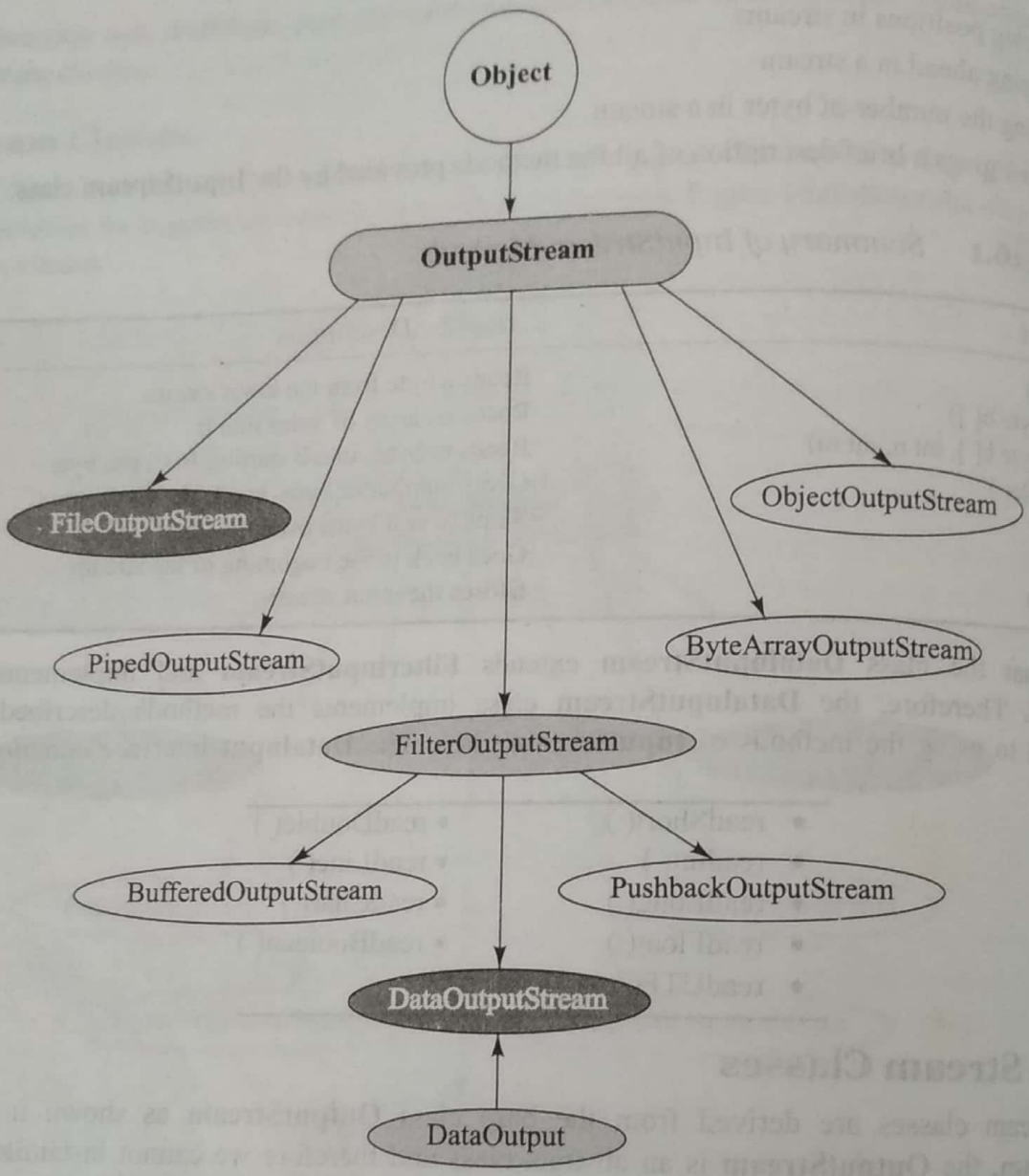

Fig. 16.7 *Hierarchy of output stream classes*

The **DataOutputStream,** a counterpart of **DataInputStream,** implements the interface **DataOutput** and, therefore, implements the following methods contained in **DataOutput** interface.

- writeShort( )
- writeInt( )
- writeLong( )
- writeFloat( )
- writeUTF( )
- writeDouble( )
- writeBytes( )
- writeChar( )
- writeBoolean( )

16.5 CHARACTER STREAM CLASSES

Character stream classes were not a part of the language when it was released in 1995. They were added later when the version 1.1 was announced. Character streams can be used to read and write 16-bit Unicode characters. Like byte streams, there are two kinds of character stream classes, namely, *reader stream* classes and *writer stream* classes.

Reader Stream Classes

Reader stream classes are designed to read character from the files. **Reader** class is the base class for all other classes in this group as shown in Fig. 16.8. These classes are functionally very similar to the input stream classes, except input streams use bytes as their fundamental unit of information, while reader streams use characters.

The **Reader** class contains methods that are identical to those available in the **InputStream** class, except **Reader** is designed to handle characters (see Table 16.1). Therefore, reader classes can perform all the functions implemented by the input stream classes.

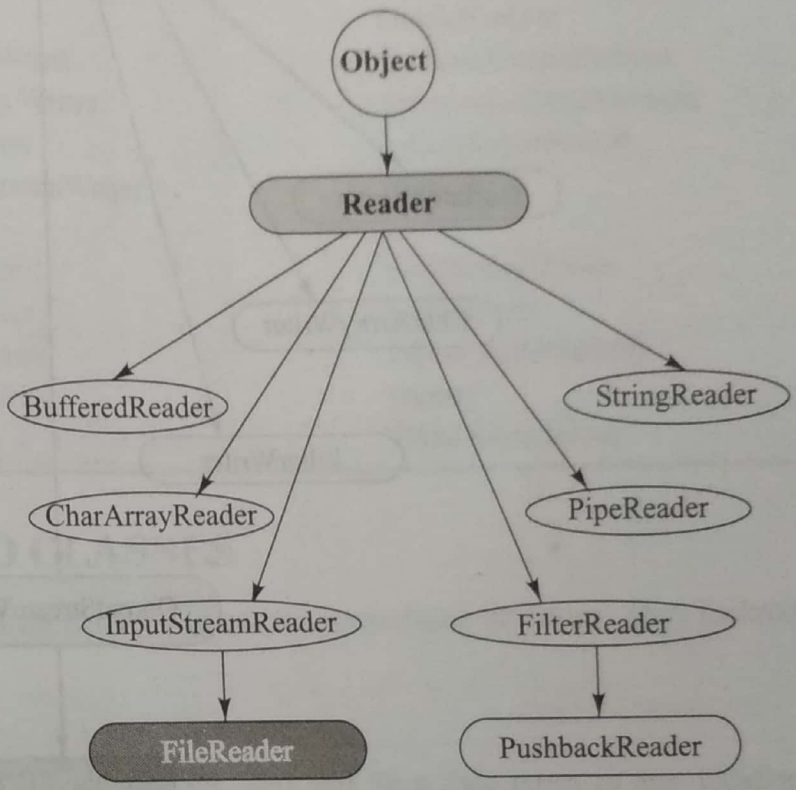

Fig. 16.8 *Hierarchy of reader stream classes*

Writer Stream Classes

Like output stream classes, the writer stream classes are designed to perform all output operations on files. Only difference is that while output stream classes are designed to write bytes, the writer stream classes are designed to write characters.

The **Writer** class is an abstract class which acts as a base class for all the other writer stream classes as shown in Fig. 16.9. This base class provides support for all output operations by defining methods that are identical to those in **OutputStream** class (see Table 16.2).

16.6 USING STREAMS

We have seen briefly various types of input and output stream classes used for handling both the 16-bit characters and 8-bit bytes. Although all the classes are known as i/o classes, not all of them are used for reading and writing operations only. Some perform operations such as buffering, filtering, data conversion, counting and concatenation while carrying out i/o tasks.

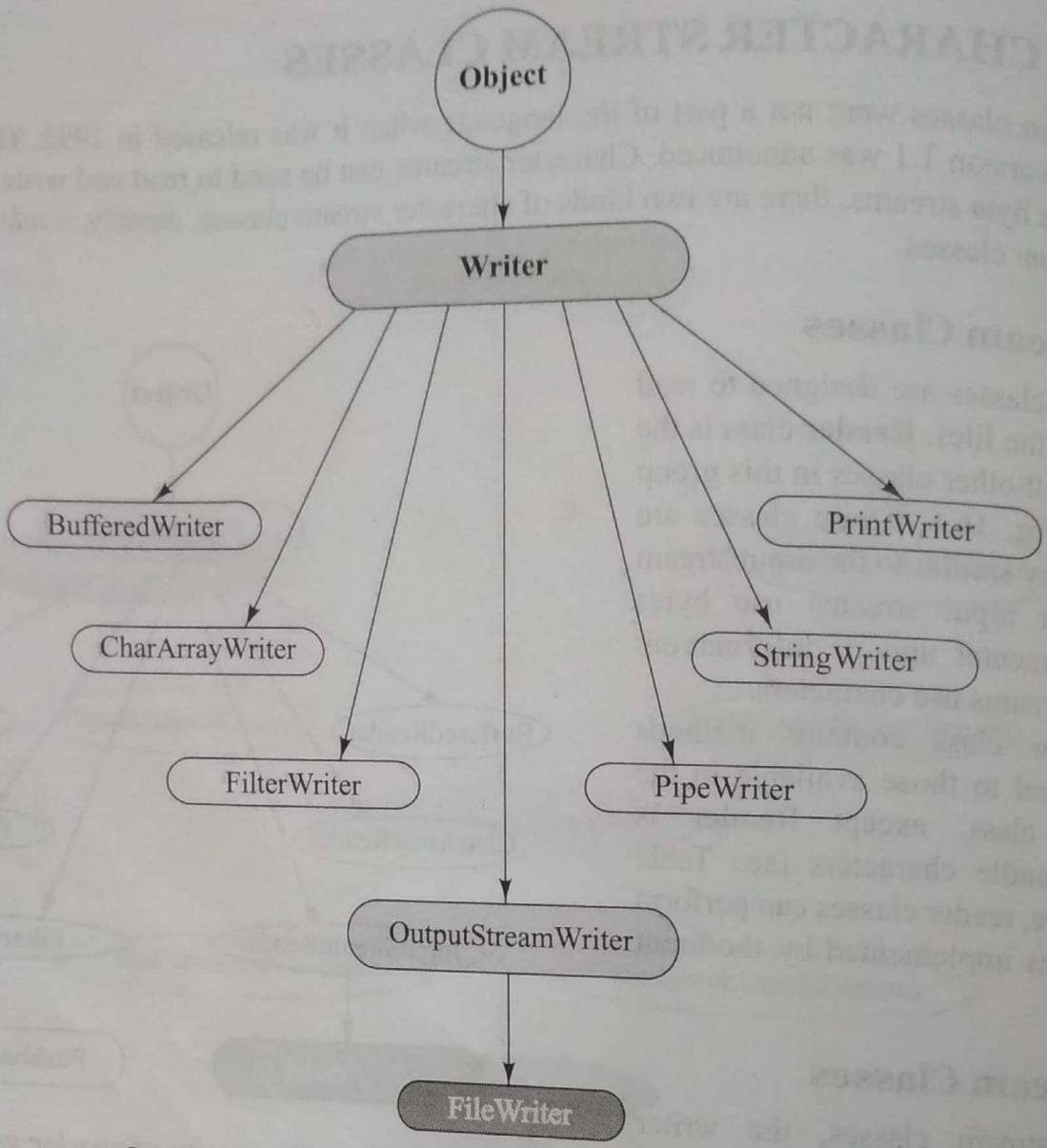

Fig. 16.9 *Hierarchy of writer stream classes*

As pointed out earlier, both the character stream group and the byte stream group contain parallel pairs of classes that perform the same kind of operation but for the different data type. Table 16.3 gives a list of tasks and the character streams and byte streams that are available to implement them.

Table 16.3 *List of Tasks and Classes Implementing Them*

Task	*Character Stream Class*	*Byte Stream Class*
Performing input operations	Reader	InputStream
Buffering input	BufFeredReader	BufferdInputStream
Keeping track of line numbers	LineNumberReader	LineNumberInputStream
Reading from an array	CharArrayReader	ByteArrayInputStream
Translating byte stream into a character stream	InputStreamReader	(none)
Reading from files	FileReader	FileInputStream

(Contd)

Table 16.3 (Contd)

Task	Character Stream Class	Byte Stream Class
Filtering the input	FilterReader	FilterInputStream
Pushing back characters/bytes	PushbackReader	PushbackInputStream
Reading from a pipe	PipedReader	PipedInputStream
Reading from a string	StringReader	StringBufferInputStream
Reading primitive types	(none)	DataInputStream
Performing output operations	Writer	OutputStream
Buffering output	BufferedWriter	BufferedOutputStream
Writing to an array	CharArrayWriter	ByteArrayOutputStream
Filtering the output	FilterWriter	FilterOutputStream
Translating character stream into a byte stream	OutputStreamWriter	(none)
Writing to a file	FileWriter	FileOutputStream
Printing values and objects	PrintWriter	printStream
Writing to a pipe	PipedWriter	PipedOutputStream
Writing to a string	String Writer	(none)
Writing primitive types	(none)	DataOutputStream

16.7 OTHER USEFUL I/O CLASSES

The **java.io** package supports many other classes for performing certain specialized functions. They include among others:

- RandomAccessFile
- StreamTokenizer

The **RandomAccessFile** enables us to read and write bytes, text and Java data types to any location in a file (when used with appropriate access permissions). This class extends **object** class and implements **DataInput** and **DataOutput** interfaces as shown in Fig. 16.10. This forces the **RandomAccessFile** to implement the methods described in both these interfaces.

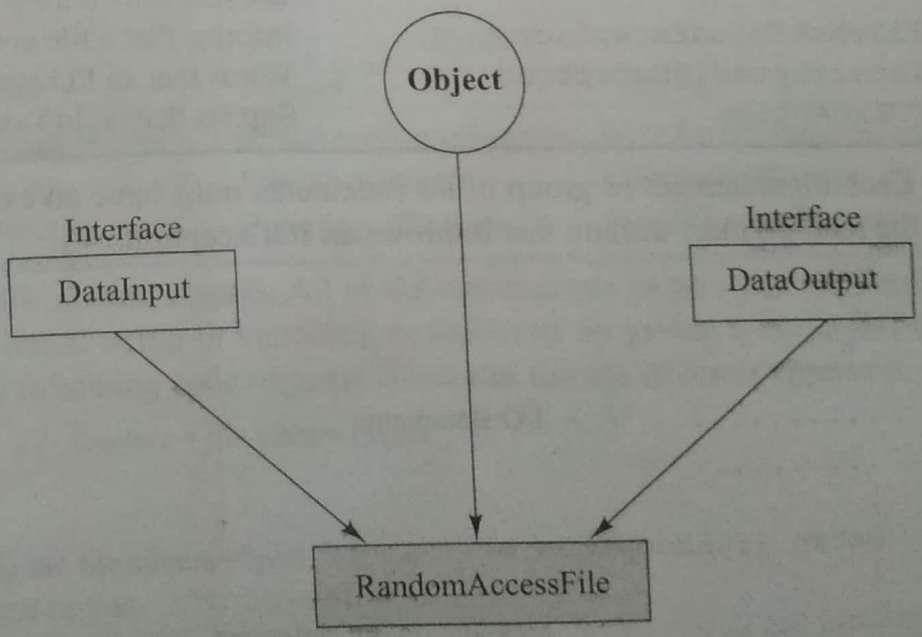

Fig. 16.10 *Implementation of the RandomAccessFile*

The class **Stream Tokenizer,** a subclass of **object** can be used for breaking up a stream of text from an input text file into meaningful pieces called *tokens*. The behaviour of the **StreamTokenizer** class is similar to that of the **StringTokenizer** class (of **java.util** package) that breaks a string into its component tokens.