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.

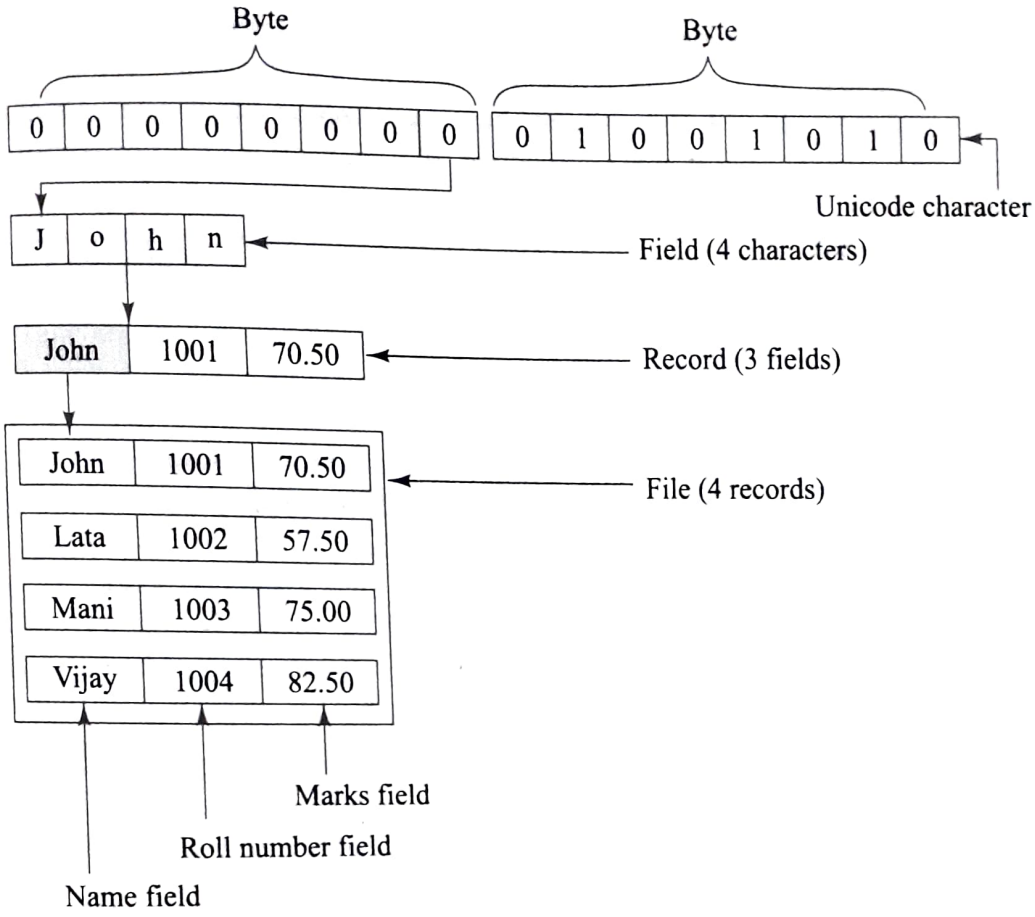


Fig. 16.1 *Data representation in Java files*

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

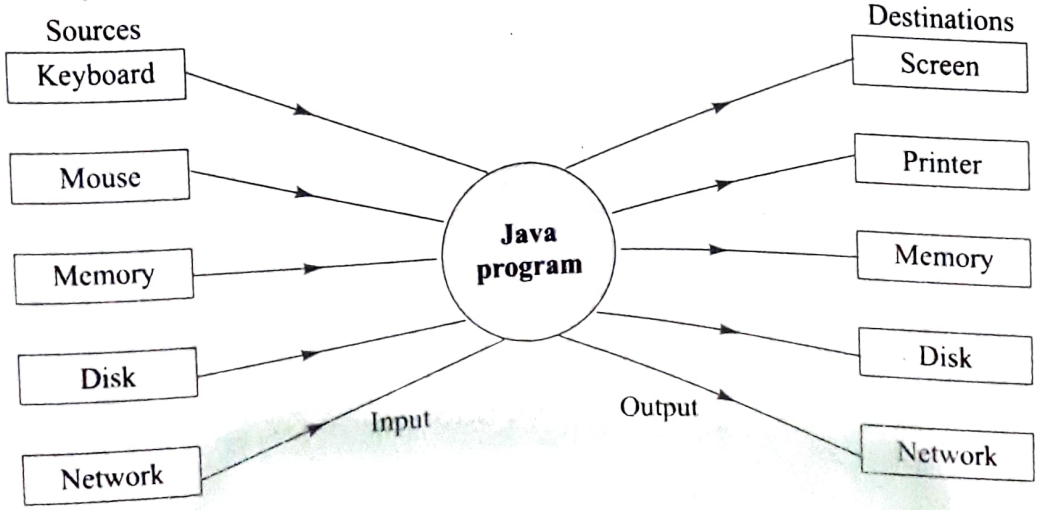


Fig. 16.2 *Relationship of Java program with I/O devices*

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.

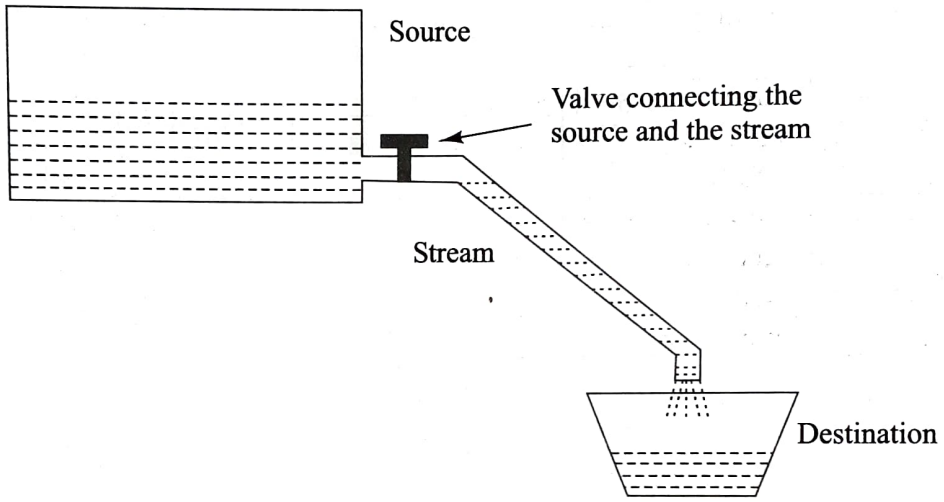
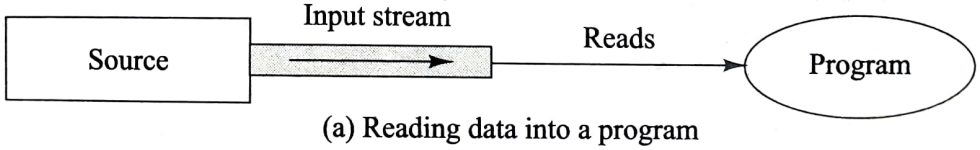


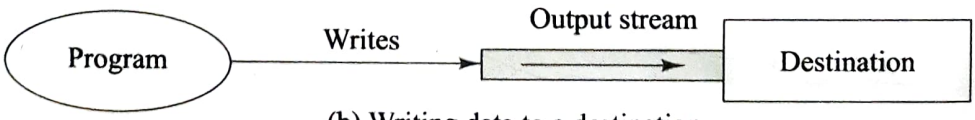
Fig. 16.3 *Conceptual view of a stream*

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



(a) Reading data into a program



(b) Writing data to a destination

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.

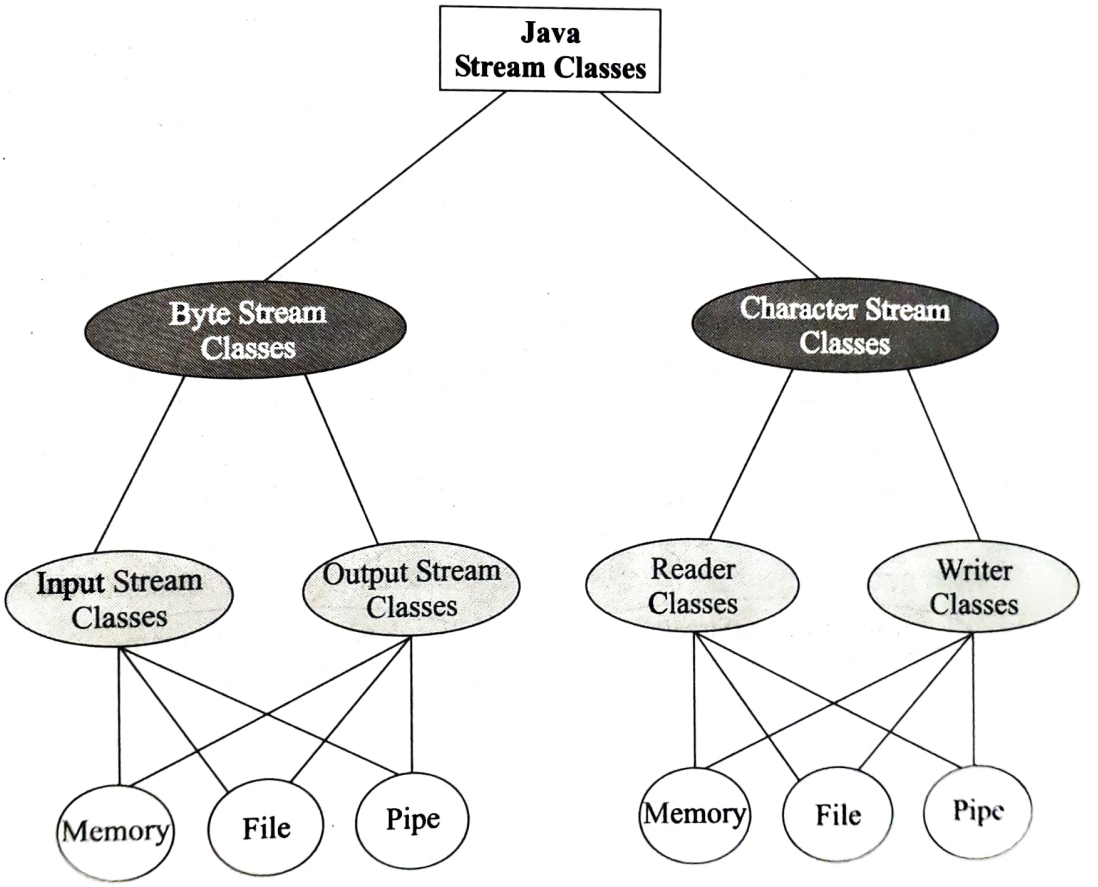


Fig. 16.5 *Classification of Java stream classes*

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.

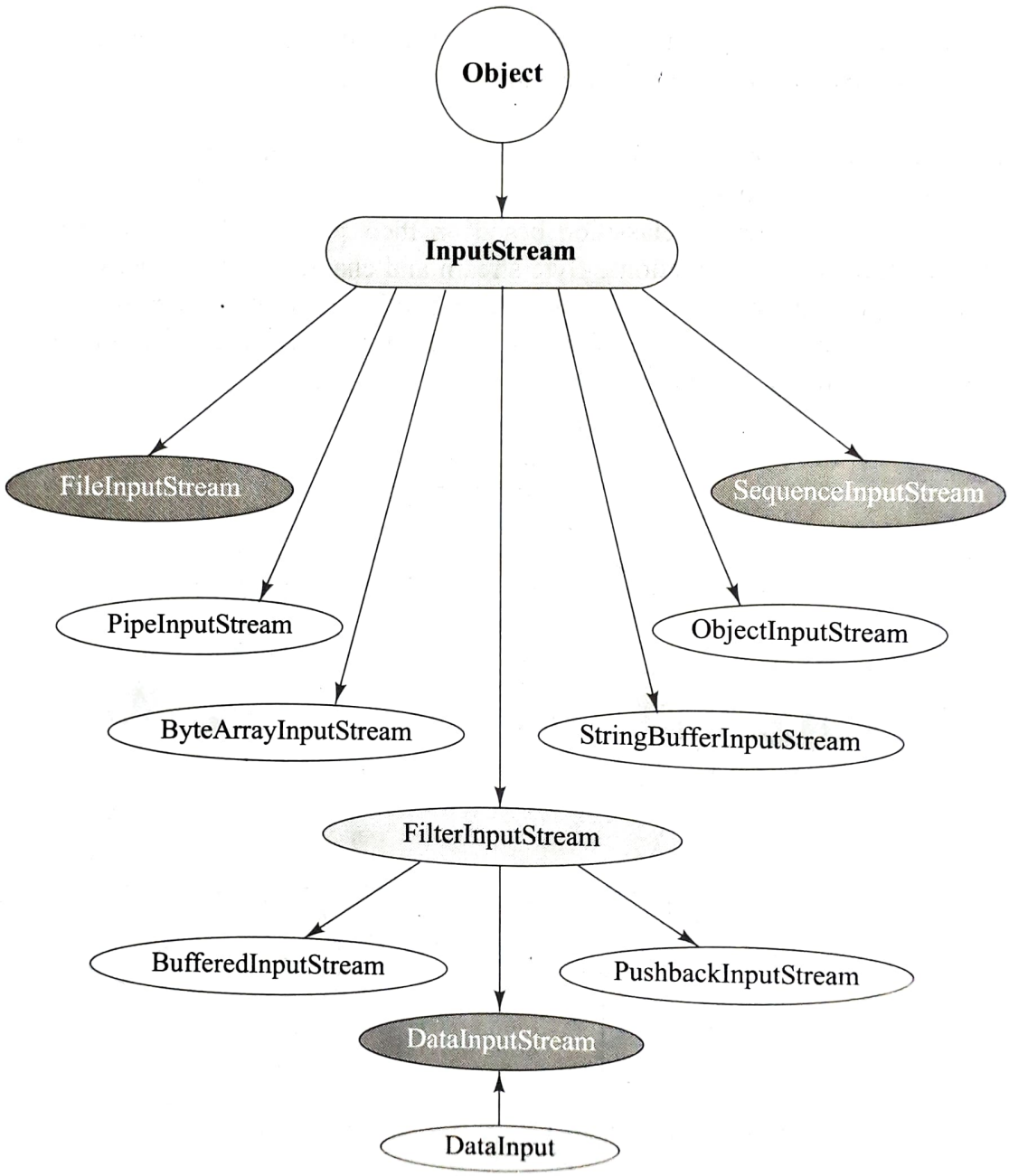


Fig. 16.6 *Hierarchy of input stream classes*

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

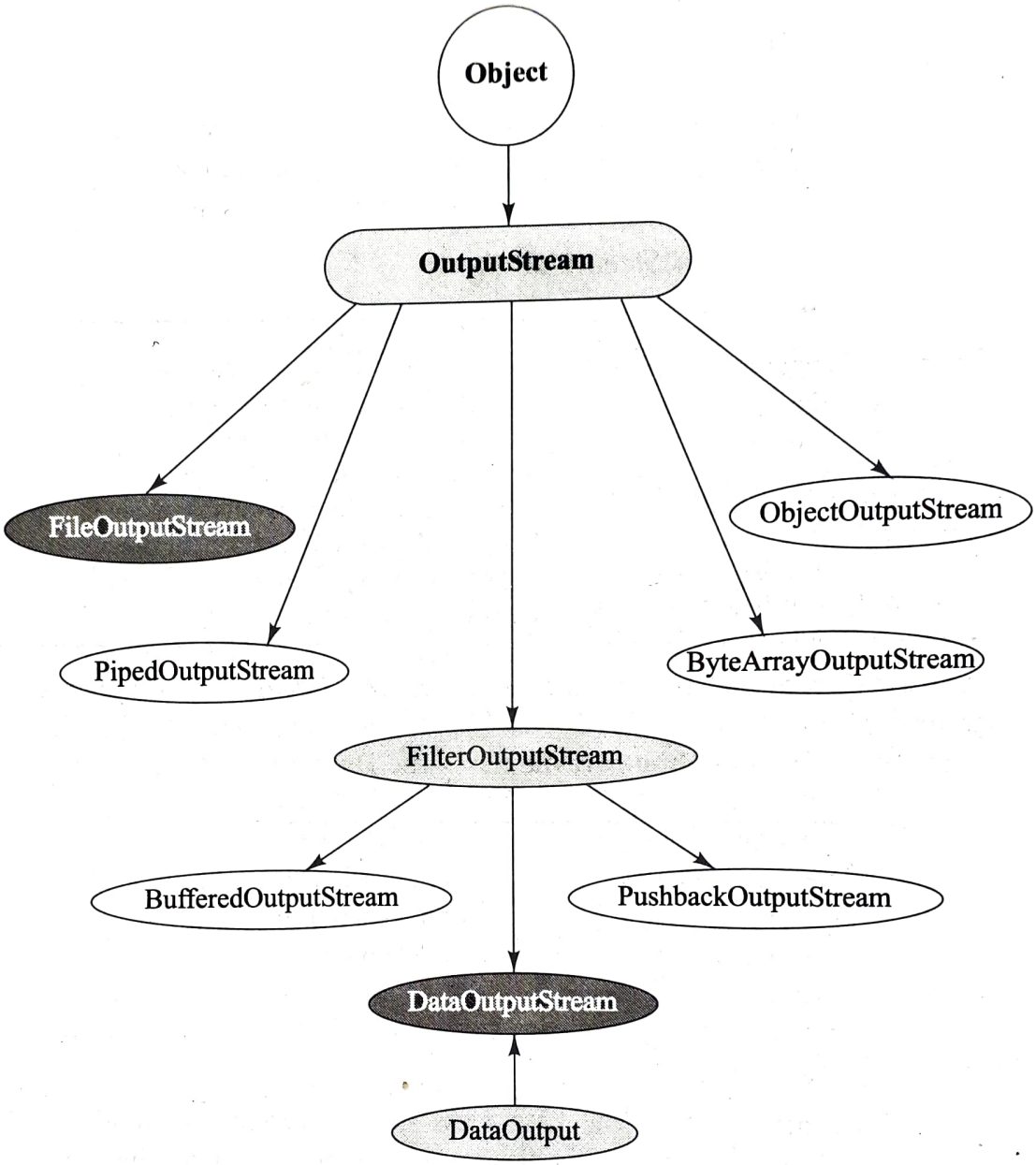


Fig. 16.7 *Hierarchy of output stream classes*

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.

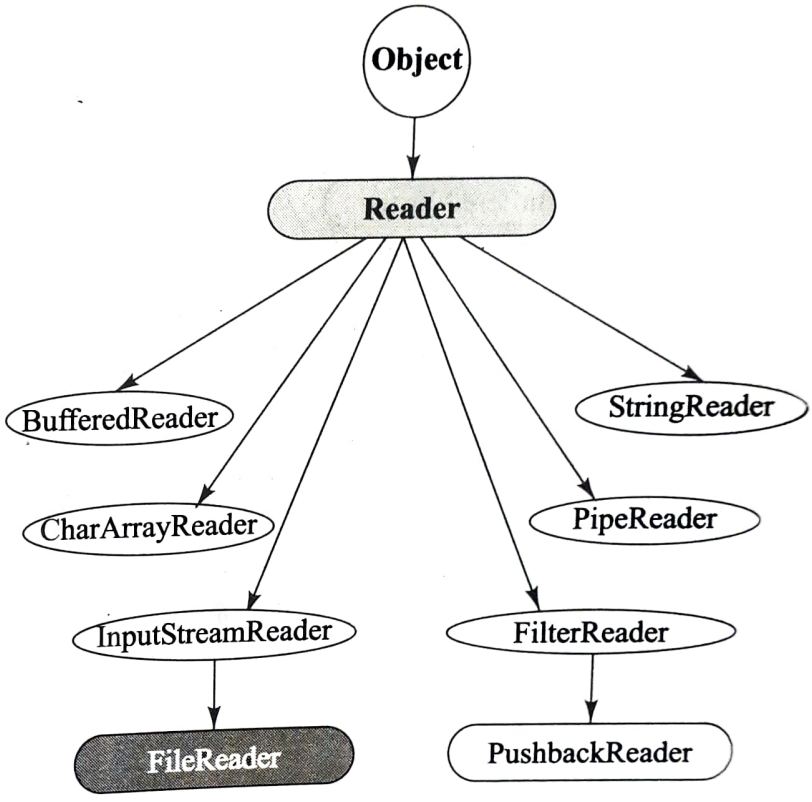


Fig. 16.8 *Hierarchy of reader stream classes*

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.

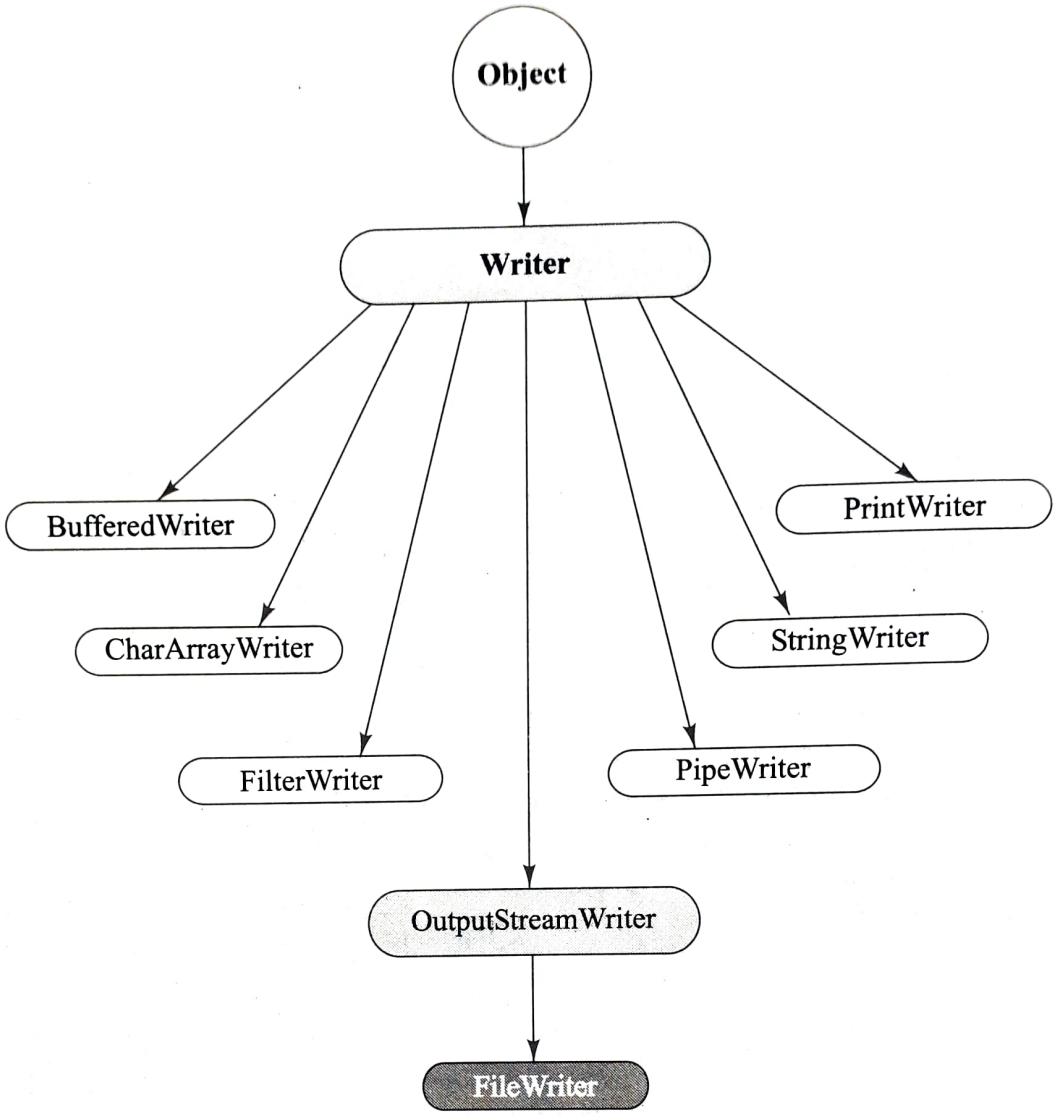


Fig. 16.9 *Hierarchy of writer stream classes*

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.

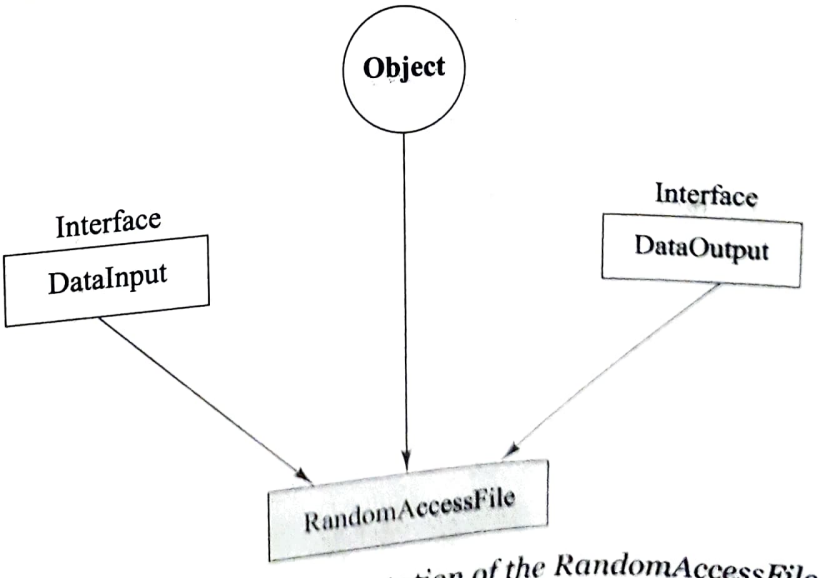


Fig. 16.10 *Implementation of the RandomAccessFile*

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.

16.8 USING THE FILE CLASS

The **java.io** package includes a class known as the **File** class that provides support for creating files and directories. The class includes several constructors for instantiating the **File** objects. This class also contains several methods for supporting the operations such as

- Creating a file
- Opening a file
- Closing a file
- Deleting a file
- Getting the name of a file
- Getting the size of a file
- Checking the existence of a file
- Renaming a file
- Checking whether the file is writable
- Checking whether the file is readable

16.9 INPUT/OUTPUT EXCEPTIONS

When creating files and performing i/o operations on them, the system may generate i/o related exceptions. The basic i/o related exception classes and their functions are given in Table 16.4.

Table 16.4 *Important I/O Exception Classes and their Functions*

I/O Exception class	*Function*
`EOFException`	Signals that an end of the file or end of stream has been reached unexpectedly during input
`FileNotFoundException`	Informs that a file could not be found
`InterruptedIOException`	Warns that an I/O operations has been interrupted
`IOException`	Signals that an I/O exception of some sort has occurred

Each i/o statement or group of i/o statements must have an exception handler around it as shown below or the method must declare that it throws an IOException.

```
try
{
.........
.........     / / I/O statements
.........
}
catch (IOException e)
{
.........     / / Message output statement
}
```

Proper use of exception handlers would help us identify and locate i/o errors more effectively.

16.10 CREATION OF FILES

If we want to create and use a disk file, we need to decide the following about the file and its intended purpose:

- Suitable name for the file
- Data type to be stored
- Purpose (reading, writing, or updating)
- Method of creating the file

A filename is a unique string of characters that helps identify a file on the disk. The length of a filename and the characters allowed are dependent on the OS on which the Java program is executed. A filename may contain two parts, a primary name and an optional period with extension. Examples:

```
input.data          salary
test.doc            student.txt
inventory           rand.dat
```

Data type is important to decide the type of file stream classes to be used for handling the data. We should decide whether the data to be handled is in the form of characters, bytes or primitive type.

The purpose of using a file must also be decided before using it. For example, we should know whether the file is created for reading only, or writing only, or both the operations.

As we know, for using a file, it must be opened first. This is done by creating a file stream and then linking it to the filename. A file stream can be defined using the classes of **Reader/InputStream** for reading data and **Writer/OutputStream** for writing data. The common stream classes used for various i/o operations are given in Table 16.5. The constructors of stream classes may be used to assign the desired filenames to the file stream objects.

Table 16.5 *Common Stream Classes used for I/O Operations*

Source or Destination	*Characters*		*Bytes*	
	Read	*Write*	*Read*	*Write*
Memory	CharArrayReader	CharArrayWriter	ByteArrayInputStream	ByteArrayOutputStream
File	FileReader	FileWriter	FileInputStream	FileOutputStream
Pipe	PipedReader	PipedWriter	PipedInputStream	PipedOutputStream

There are two ways of initializing the file stream objects. All of the constructors require that we provide the name of the file either *directly*, (as a literal string or variable), or *indirectly* by giving a file object that has already been assigned a filename. The following code segment illustrates the use of direct approach.

```
FileInputStream fis;     // Declare a file stream object
try
{
    // Assign the filename to the file stream object
    fis = new FileInputStream ("test.dat") ;
    ....
}
    catch (IOException e)
....
....
```

The indirect approach uses a file object that has been initialized with the desired filename. This is illustrated by the following code.

```
.....
.....
File inFile;                              // Declare a file object
InFile = new File ("test.dat");           // Assign the filename to
                                          // the file object

FileInputStream fis;
try
{
      // Give the value of the file object
      // to the file stream object
      fis = new FileInputStream (inFile);
      .....
 }
 catch (.....)
 .....
 .....
```

The code above includes five tasks:

- Select a filename
- Declare a file object
- Give the selected name to the file object declared
- Declare a file stream object
- Connect the file to the file stream object
- Both the approaches are illustrated in Fig. 16.11.

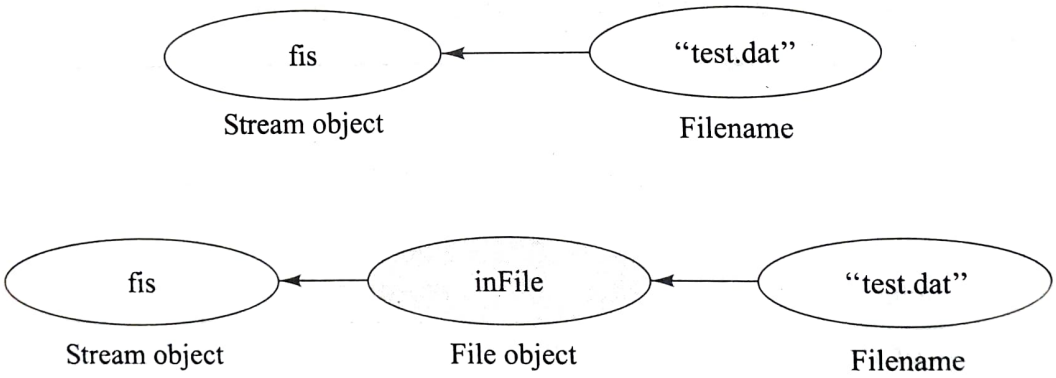


Fig. 16.11 *Hierarchy of writer stream classes*

16.11 READING/WRITING CHARACTERS

As pointed out earlier, subclasses of **Reader** and **Writer** implement streams that can handle characters. The two subclasses used for handling characters in files are **FileReader** (for reading characters) and **FileWriter** (for writing characters). Program 16.1 uses these two file stream classes to copy the contents of a file named "input.dat" into a file called "output.dat".

Program 16.1 *Copying characters*

```
// Copying characters from one file into another
import java.io.*;
class CopyCharacters
{   public static void main (String args [ ] )
    {
        // Declare and create input and output files
        File inFile = new File ("input.dat") ;
        File outFile = new File ("output.dat") ;
        FileReader ins = null;      // Creates file stream ins
        FileWriter outs = null;     // Creates file stream outs
        try
    {
        ins = new FileReader (inFile);       // Opens inFile
        outs = new FileWriter (outFile);     // Opens outFile
        // Read and write till the end
        int ch;
        while ( (ch = ins.read( ) ) != - 1)
        {
            outs.write (ch) ;
        }
    }
    catch (IOException e)
    {
        System.out.println (e) ;
        System.exit (- 1) ;
    }
    finally // Close files
    {
        try
        {
            ins.close ( ) ;
            outs.close ( );
        }
        catch (IOException e) { }
    }
}
```

This program is very simple. It creates two file objects **inFile** and **outFile** and initializes them with "input.dat" and "output.dat" respectively using the following code:

```
File inFile = new File ("input.dat");
File outFile = new File ("output.dat");
```

The program then creates two file stream objects ins and outs and initializes them with "null" as follows:

```
FileReader ins = null;
FileWriter outs = null;
```

These streams are then connected to the named files using the following code:

```
ins = new FileReader (inFile);
outs = new FileWriter (outFile);
```

This connects **inFile** to the **FileReader** stream **ins** and **outFile** to the **FileWriter** stream **outs**. This essentially means that the files "input.dat" and "output.dat" are opened. The statements

```
ch = ins.read ( )
```

reads a character from the **inFile** through the input stream **ins** and assigns it to the variable **ch.** Similarly, the statement

```
outs.write (ch) ;
```

writes the character stored in the variable **ch** to the **outFile** through the output stream **outs.** The character-1 indicates the end of the file and therefore the code

```
while ( (chains.read( ) ) != -1)
```

causes the termination of the while loop when the end of the file is reached. The statements

```
ins.close ( ) ;
outs.close ( ) ;
```

enclosed in the **finally** clause close the files created for reading and writing. When the program catches an I/O exception, it prints a message and then exits from execution.

The concept of using file streams and file objects for reading and writing characters in Program 16.1 is illustrated in Fig. 16.12.

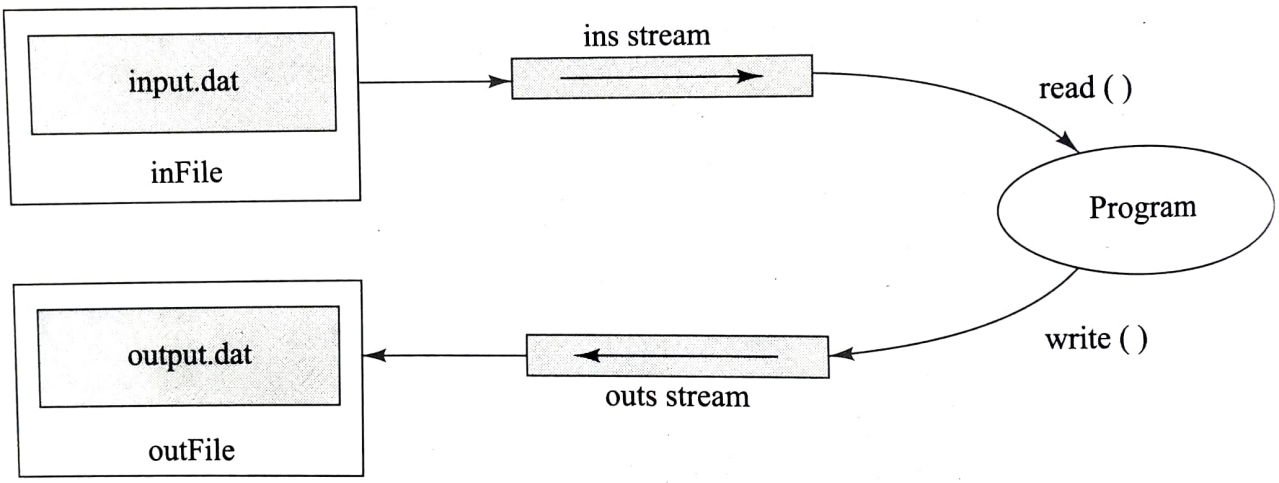


Fig. 16.12 *Reading from and writing to files*

16.12 READING/WRITING BYTES

In Program 16.1 we have used **FileReader** and **FileWriter** classes to read and write 16-bit characters. However, most file systems use only 8-bit bytes. As pointed out earlier, Java i/o system provides a number of classes that can handle 8-bit bytes. Two commonly used classes for handling bytes are **FileInputStream** and **FileOutputStream** classes. We can use them in place of **FileReader** and **FileWriter.**

Program 16.2 demonstrates how **FileOutputStream** class is used for writing bytes to a file. The program writes the names of some cities stored in a byte array to a new file named "city.txt". We can verify the contents of the life by using the command

```
type city.txt
```

Program 16.2 *Writing bytes to a file*

```
// Writing bytes to a file
import java.io.*;
class Write Bytes
{
    public static void main (String args [ ] )
    {
        // Declare and initialize a byte array
        byte cities [ ] = {'D', 'E', 'L', 'H', 'I', *\n\ 'M', 'A', 'D',
            'R', 'A', 'S', '\n', 'L', 'O', 'N', 'D', 'O', 'N', *\n'};
        // Create an output file stream
        FileOutputStream outfile = null;
        try
        {
            // Connect the outfile stream to "city.txt"
            outfile = new FileOutputStream ("city.txt") ;
            // Write data to the stream
            outfile.write (cities) ;
            outfile.close ( ) ;
        }
        catch (IOException ioe)
        {
            System.out.println (ioe) ;
            System.exit (-1) ;
        }
    }
}
```

```
type city.txt
    DELHI
        MADRAS
            LONDON
```

Note that a instantiating a **FileOutputStream** object with the name of the file creates and opens the file. We may also supply the filename as a command line argument at the time of execution.

Remember, there are several forms of **write( )** method. The one we have used here writes the entire byte array to the file. Finally, we close the file opened for writing.

Program 16.3 shows how **FileInputStream** class is used for reading bytes from a file. The program reads an existing file and displays its bytes on the screen. Remember, before we run this program, we must first create a file for it to read. We may use this program to read the file "city.txt" created in Program 16.2.

Program 16.3 *Reading bytes from a file*

```
// Reading bytes from a file
import java.io.*;
class ReadBytes
{
    public static void main (String args [ ] )
    {
        // Create an input file stream
        FileInputStream infile = null;
        int b;
        try
```

```
        {
            // Connect infile stream to the required file
            infile = new FileInputStream (args [ 0 ] ) ;
            // Read and display data
            while ( (b = infile.read ( ) ) != -1)
            {
                System.out.print ( ( char) b) ;
            }
            infile.close ( ) ;
        }
        catch (IOException ioe)
        {
            System.out.println (ioe) ;
        }
    }
}
```

Note that the program requires the filename to be given as a command line argument. Program 16.3 displays the following when we supply the filename "city.txt".

```
Prompt>java ReadBytes city.txt
    DELHI
        MADRAS
            LONDON
```

Another example code given in Program 16.4 uses both **FileInputStream** and **FileOutputStream** classes to copy files. We need to provide a source filename for reading and a target filename for writing. In the example code, we have supplied file names directly to the constructors while creating file streams. We may also supply them as command line arguments. Note that the file "in.dat" already exists and contains the following text.

```
Java programming for Internet.
Javascript for Web page development.
Perl for Server-side scripting.
```

Program 16.4 *Copying bytes from one file to another*

```
// Copying bytes from one file to another
import java.io.*;
class CopyBytes
{
    public static void main (String args [ ] )
    {
        // Declare input and output file streams
        FileInputStream infile = null;          // Input stream
        FileOutputStream outfile = null;        // Output stream
        // Declare a variable to hold a byte
        byte byteRead;
        try
        {
            // Connect infile to in.dat
            infile = new FileInputStream ("in.dat") ;
            // Connect outfile to out.dat
            outfile = new FileOutputStream ("out.dat") ;
            // Reading bytes from in.dat and
```

```
            // writing to out.dat
            do
            {
                byteRead = (byte) infile.read ( ) ;
                outfile.write (byteRead) ;
            }
            while (byteRead != -1) ;
        }
        catch (FileNotFoundException e)
        {
            System.out.println ("File not found") ;
        }
        catch (IOException e)
        {
            System.out.println (e.getMessage ( ) ) ;
        }
        finally // Close files
        {
            try
            {
                infile.close ( ) ;
                outfile.close ( ) ;
            }
            catch (IOExcedption e) { }
        }
    }
}
```

The command `type out.dat` will produce the following output:

```
Java programming for Internet.
Javascript for Web page development.
Perl for Server-side scripting.
```

Program 16.4 creates **infile** and **outfile** streams for handling the input/output operations. The program then continuously reads a byte from "in.dat" file (using **infile** stream) and writes it to "out.dat" file (using **outfile** stream) until the end of file condition is reached. We should avoid writing to an existing file. We may use the **exists( )** method in the **File** class to check whether the named file already exists. Example:

```
File fout    =    new File ("out.dat") ;
if (fout.exists ( ) )
return ( ) ;
```

This program could also be written using **FileReader** and **FileWriter** classes.

16.13 HANDLING PRIMITIVE DATA TYPES

The basic input and output streams provide read/write methods that can only be used for reading/ writing bytes or characters. If we want to read/write the primitive data types such as integers and doubles, we can use filter classes as wrappers on existing input and output streams to filter data in the original stream. The two filter classes used for creating "data streams" for handling primitive types are **DataInputStream** and **DataOutputStream.** These classes use the concept of multiple inheritance as illustrated in Fig. 16.13 and therefore implements all the methods contained in both the parent class and the interface.

A data stream for input can be created as follows:

```
FileInputStream fis = new FileInputStream (infile) ;
DataInputStream dis = new DataInputStream (fis) ;
```

These statements first create the input file stream fis and then create the input data stream **dis.** These statements basically wrap **dis** on **fis** and use it as a "filter". Similarly, the following statements create the output data stream **dos** and wrap it over the output file stream **fos.**

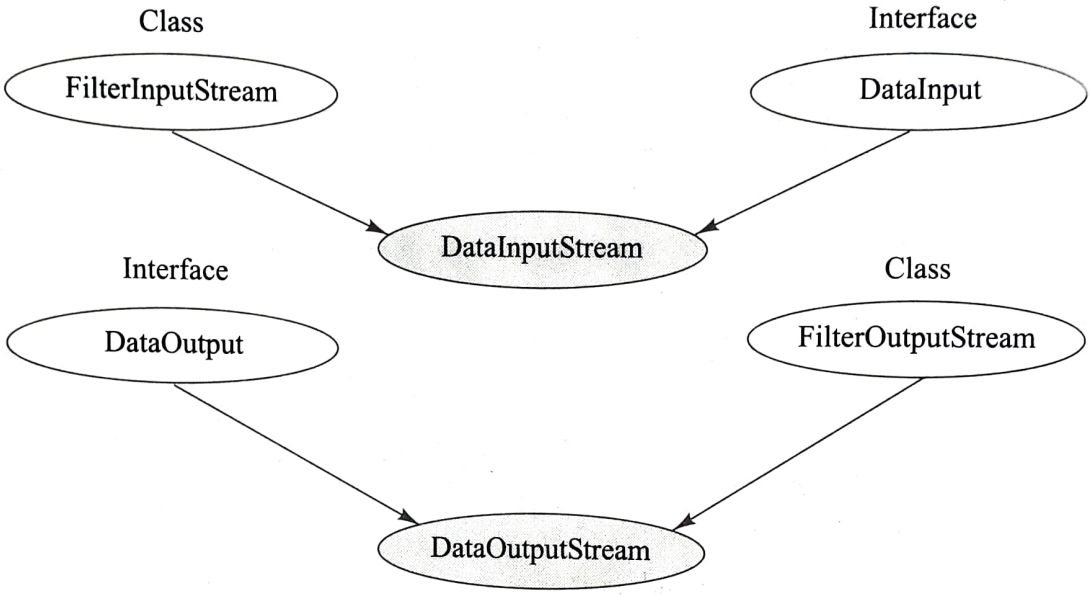


Fig. 16.13 *Hierarchy of data stream classes*

```
FileOutputStream fos = new FileOutputStream (outfile) ; DataOu-
tputStream dos = new DataOutputStream (fos) ;
```

Note that the file objects **infile** and **outfile** must be initialized with appropriate file names before they are used. We may also use file names directly in place of file objects.

Program 16.5 demonstrates the use of data streams for reading and writing primitive data types. The program first creates "prim.dat" file and then writes a few primitive data types into it using data output stream. At the end of writing, the streams are closed.

The program also creates a data input stream, and connects it to "prim.dat" file. It then reads data from the file and displays them on the screen. Finally, it closes the streams. Note that the **main** method declares that it throws IOException and therefore we need not use **try**..... **catch** statements.

Program 16.5 *Reading and writing primitive data*

```
// Reading and writing primitive data
import java.io.*;
Class ReadWritePrimitive
{
    public static void main (String args [ ] ) throws IOException
    {
        File primitive = new File ("prim.dat") ;
        FileOutputStream fos = new FileOutputStream (primitive) ;
        DataOutputStream dos = new DataOutputStream (fos) ;
        // Write primitive data to the "prim.dat" file
```

```
        dos.writeInt (1999) ;
        dos.writeDouble (375.85) ;
        dos.writeBoolean (false) ;
        dos .writeChar ( 'X' ) ;
        dos.close ( ) ;
        fos.close ( ) ;
        // Read data from the "prim.dat" file
        FileInputStream fis = new FileInputStream (primitive) ;
        DataInputStream dis = new DataInputStream (fis) ;
        System.out.println (dis.readInt( ) ) ;
        System.out.println (dis.readDouble ( ) ) ;
        System.out.println (dis.readBoolean ( ) ) ;
        System.out.println (dis.readChar ( ) ) ;
        dis.close ( ) ;
        fis.close ( ) ;
    }
}
```

Program 16.5 displays the following:

```
1999
375.85
false
X
```

The data streams used in Program 16.5 and their functions are illustrated in Fig. 16.14.

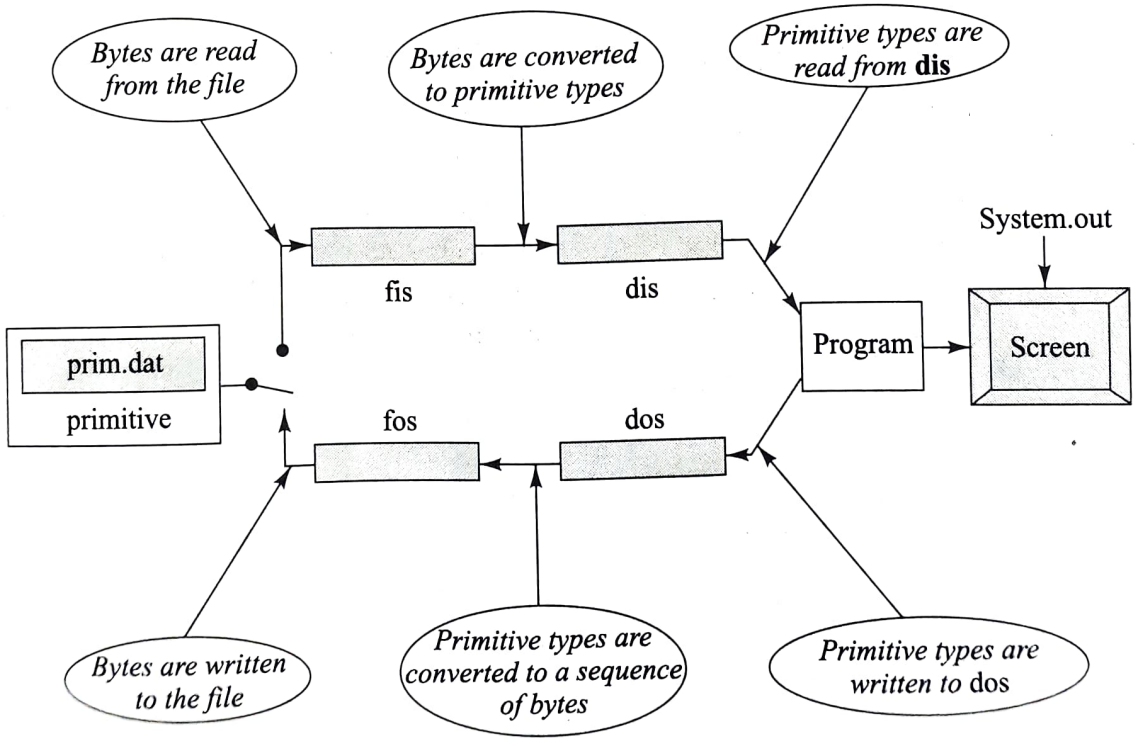


Fig. 16.14 *Data streams used in Program 16.5*

Another example code shown in Program 16.6 generates random integers and stores them in a file named "rand.dat". The program then reads the integers from the file and displays on the screen.

Program 16.6 *Using a single file for storing and retrieving*

```
//Storing and retrieving integers using data streams
//on a single file
import java.io.*;
class ReadWriteIntegers
{
    public static void main (String args [ ] )
    {
        //Declare data streams
        DataInputStream dis = null;              //  Input stream
        DataOutputStream dos = null;             //  Output stream
        //Construct a file
        File intFile = new File ("rand.dat") ;
        //Writing integers to rand.dat file
        try
        {
            //Create output stream for intFile file
            dos = new DataOutputStream (new
            FileOutputStream (intFile) ) ;
            for (int i = 0; i<20;i++)
                dos.writeInt ( (int) (math.random ( ) *100) ) ;
        }
        catch (IOException ioe)
        {
            System.out.printIn (ioe.getMessage ( ) ) ;
        }
        finally
        {
            try
            {
                dos.close ( ) ;
            }
            catch (IOException ioe) { }
        }
        //Reading integers from rand.dat file
        try
        {
            //Create input stream for intFile file
            dis = new DataInputStream (new
                    FileInputStream (intFile) ) ;
            for (int i=0; i < 20; i++)
            {
                int n = dis.readInt ( ) ;
                System.out.print (n + " ") ; }
    }
    catch (IOException ioe)
    {
        System.out.println (ioe.getMessage ( ) ) ; }
        finally
        {
            try
            {
                dis.close ( ) ;
```

```
            }
            catch (IOException ioe) { }
        }
    }
}
```

Output of Program 16.6:

```
78 62 54 56 55 48 48 35 13 64 13 90 10 78 91 42 9 44 84 66
```

16.14 CONCATENATING AND BUFFERING FILES

It is possible to combine two or more input streams (files) into a single input stream (file). This process is known as *concatenation* of files and is achieved using the **SequenceInputStream** class. One of the constructors of this class takes two **InputStream** objects as arguments and combines them to construct a single input stream.

Java also supports creation of buffers to store temporarily data that is read from or written to a stream. The process is known as *buffered ilo* operation. A buffer sits between the program and the source (or destination) and functions like a filter. Buffers can be created using the **BufferedInputStream** and **BufferedOutputStream** classes.

Program 16.7 *Example of concatenation and buffering*

```
//Concatenating and buffering files
import java.io.*;
class SequenceBuffer
{
    public static void main (String args [ ] )
    throws IOException
    {
        // Declare file streams
        FileInputStream file1 = null;
        FileInputStream file2 = null;
        // Declare file3 to store combined files
        SequenceInputStream file3 = null;
        // Open the files to be concatenated
        file1 = new FileInputStream ("text1.dat") ;
        file2 = new FileInputStream ("text2.dat") ;
        // Concatenate file1 and file2 into file3
        file3 = new SequenceInputStream (file1, file2) ;
        // Create buffered input and output streams
        BufferedInputStream inBuffer =
            new BufferedInputStream (file3);
        BufferedOutputStream outBuffer =
            new BufferedOutputStream (System.out) ;
        // Read and write till the end of buffers
        int ch;
        while ((ch = inBuffer.read ())!= -1)
        {
            outBuffer.write ((char)ch);
        }
        inBuffer.close ( );
```

```
        outBuffer.close ( );
        fiIel. close ( );
        file2. close ( );
    }
}
```

Program 16.7 illustrates the process of concatenation as well as buffering. The program creates two objects of class **FileInputStream** for the files "textl.dat" and "text2.dat". The two **FileInputStream** objects **file1** and **file2** are used as arguments to the **SequenceInputStream** constructor to obtain a single input stream object **file3.** The file **file3** now contains the contents of **file1** and **file2.**

The program now creates an input buffer named **inBuffer** and connects it to **file3** and creates and output buffer named **outBuffer** and connects it to **system.out** that represents screen. It then uses a while loop to read all bytes in the input buffer and display them through the output buffer.

The entire process of concatenation, buffering and displaying the contents of two independent files is illustrated in Fig. 16.15.

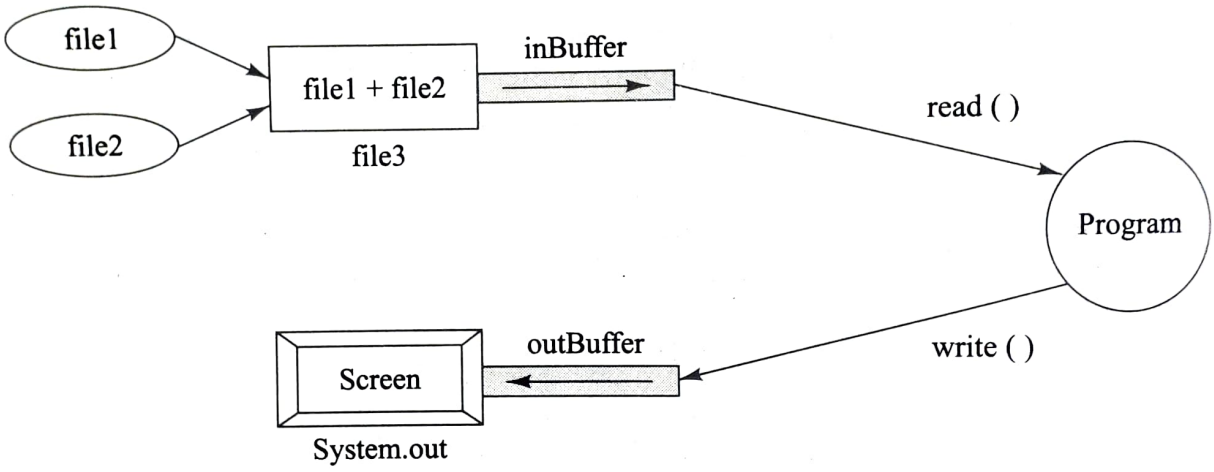


Fig. 16.15 *Illustration of concatenation and buffering*

Given the contents of "text1.dat" and "text2.dat" as follows:

Contents of "text1.dat":

```
Java (tm) Development Kit
    Version 1.2
Binary Code License
```

Contents of "text2.dat":

```
This binary code license ("License") contains rights and restrictions
associated with use of the accompanying software and documentation
("Software"). Read the License carefully before installing the Soft-
ware. By installing the Software you agree to the terms and conditions
of this License.
```

Then, the output of Program 16.7 would be:

```
Java (tm) Development Kit

    Version 1.2
```

```
                Binary Code License
The binary code license ("License") contains rights and restrictions
associated with use of the accompanying software and documentation
("Software"). Read the License carefully before installing the Soft-
ware. By installing the Software you agree to the terms and conditions
of this License.
```

16.15 RANDOM ACCESS FILES

So far we have discussed files that can be used either for "read only" or for "write only" operations and not for both purposes simultaneously. These files are read or written only sequentially and, therefore, are known as *sequential* files.

As stated earlier, the **RandomAccessFile** class supported by the **Java.io** package allows us to create files that can be used for reading and writing data with random access. That is, we can "jump around" in the file while using the file. Such files are known as *random access* files.

A file can be created and opened for random access by giving a mode string as a parameter to the constructor when we open the file. We can use one of the following two mode strings:

- " r" for reading only
- " rw" for both reading and writing

An existing file can be updated using the "rw" mode.

Program 16.8 demonstrates how a random access is created and used for both reading and writing data. Random access files support a pointer known as *file pointer* that can be moved to arbitrary positions in the file prior to reading or writing. The file pointer is moved using the method **seek( )** in the **RandomAccessFile** class. When the file is opened by the statement

```
file = new RandomAccessFile ("rand.dat", "rw");
```

the file pointer is automatically positioned at the beginning of the file.

Program 16.8 *Reading/Writing using a random access file*

```
//Writing and reading with random access
import java.io.*;
class RandomIO
{
    public static void main (string args [ ] )
    {
        RandomAccessFile file = null;
        try
        {
            file = new RandomAccessFile ("rand.dat", "rw") ;
            // Writing to the file
            file.WriteChar ('X') ;
            file.writeInt (555) ;
            file.writeDouble (3.1412) ;

            file.seek (0) ; // Go to the beginning
            // Reading from the file
            System.out.println (file.readChar ( ) ) ;
            System.out.println (file.readInt ( ) ) ;
```

```
            System.out.println (file.readDouble ( ) ) ;
            file. seek (2) ; // Go to the second item
            System.out.println (file.readInt ( ) ) ;

            // Go to the end and append false to the file
            file.seek (file.length ( ) ) ;
            file.writeBoolean (false) ;

            file.seek (4) ;
            System.out.println (file.readBoolean ( ) ) ;
            file.close ( ) ;
        }
        catch (IOException e) { System.out.println (e) ; }
    }
}
```

The program opens a random access file and then performs the following operations.

1. Writes three items of data

```
X
555
3.1412
```

2. Brings the file pointer to the beginning
3. Reads and displays all the three items

```
X
555
3.1412
```

4. Takes the pointer to the second item and then reads and displays the second item in the file.

```
555
```

5. Places the pointer at the end using the method **length( )** and then adds a Boolean item to the file. (Now there are four items in the file and the pointer is at the end, that is, beyond the fourth item).
6. Finally, takes the pointer to the fourth item and displays it.
7. At the end, closes the file.

The output on the screen would appear as follows:

```
X
555
3.1412
555
false
```

Program 16.9 shows how we could append items to an existing file using the **RandomAccessFile** class. This program opens the file "city.text" created earlier by Program 16.2 and then appends MUMBAI to the end.

Program 16.7 *Appending to an existing file*

```
// Appending to a text file using random access
import java.io.*;
class RandomAccess
{
    static public void main (String args [ ] )
```

```
        RandomAccessFile rFile;
        try
        {
            rFile = new RandomAccessFile ("city.txt", "rw") ;
            rFile.seek (rFile.length ( ) ) ; //Go to the end
            rFile.writeBytes ("MUMBAI/n") ; //Append MUMBAI
            rFile.close ( ) ;
        }
        catch (IOExceptionioe)
        {
            System.out.println (ioe) ;
        }
    }
}
```

16.16 INTERACTIVE INPUT AND OUTPUT

In all the examples discussed so far, we generated data for writing to files within the programs or used data from other files stored in the memory. What if the data is to be provided through the keyboard? We can do this in Java by using an object of the **DataInputStream** class. The process of reading data from the keyboard and displaying output on the screen is known as *interactive i/o.* There are two types of interactive i/o. First one is referred to as simple interactive i/o which involves simple input from the keyboard and simple output in a pure text form. The second type is referred to as graphical interactive i/o which involves input from various input devices and output to a graphical environment on frames and applets.

We have used data stream object to read data from the keyboard in Chapter 4 and graphical interactive input and output for applets in Chapter 14. In this section we shall consider how to use interactive i/o while handling files.

Simple Input and Output

The **System** class contains three i/o objects, namely **System.in, System.out,** and **System.err** where **in, out,** and **err** are static variables. The variable **in** is of **InputStream** type and the other two are of **PrintStream** type. We can use these objects to input from the keyboard, output to the screen, and display error messages. **System** class is stored in **java.lang** package which is imported into a Java program automatically.

To perform keyboard input for primitive data types, we need to use the objects of **DataInputStream** and **StringTokenizer** classes. The following code illustrates the reading of an integer value from the keyboard.

```
static DataInputStream din =
                    new DataInputStream (System.in) ;
static String Tokenizer st;
.........
.........
st = new StringTokenizer (din.readLine) ;
int code = Integer.parseint (st. nextToken) ;
```

The first line of code wraps **din** over the input stream object **System.in** thus enabling the object **din** to read data from the keyboard. For example, the method call

```
din.readLine ( )
```

will fetch an entire string (up to a newline which will be discarded) from the console. The statement

```
st = new StringTokenizer (din.readLine ( ) ) ;
```

initializes the **StringTokenizer** object **st** with the string read by the **readLine( )** method. Finally, the line

```
int code = Integer.parseInt (st.nextToken) ;
```

takes the string, converts it into the corresponding integer value, and then assigns the result to the integer type variable **code.** A similar algorithm may be used for reading all the other primitive type data from the console.

Program 16.10 demonstrates how data is read from the keyboard for writing to a file and how the data is read back from the file for display on the screen.

Program 16.10 *Interactive input and output*

```
//Creating files interactively from keyboard input
import java.util.*;
//For using StringTokenizer class
import java.io*;
class Inventory
{
     static DataInputStream din = new DataInputStream (System.in) ;
     static StringTokenizer st;

     public static void main (String args [ ] ) throws IOException
     {
          DataOutputStream dos = new DataOutputStream (new
                                 FileOutputStream ("invent.dat") ) ;
          //  Reading from console
          System.out.println ("Enter code number") ;
          st = new StringTokenizer (din.readLine ( ) ) ;
          int code = Integer.parseInt (st.nextToken ( ) ) ;

          System.out.println ("Enter number of items") ;
          st = new StringTokenizer (din.readLine ( ) ) ;
          int items = Integer.parseInt (st.nextToken ( ) ) ;

          System.out.println ("Enter cost") ;
          st = new StringTokenizer (din.readLine ( ) ) ;
          double cost = new Double (st.nextToken ( ) ). doubleValue ( );

          //Writing to the file "invent.dat"
          dos.writeInt (code) ;
          dos.writeInt (items) ;
          dos .writeDouble (cost) ;
          dos.close ( ) ;

          //Processing data from the file
          DataInputStream dis = new DataInputStream (new
                                FileInputStream ("invent.dat") ) ;
          int codeNumber = dis.readInt ( ) ;
          int totalItems = dis.readInt ( ) ;
          double itemCost = dis.readDouble ( ) ;
          double totalCost = total Items * itemCost; dis.close ( ) ;

          // Writing to console
          System.out.println ( ) ;
```

```
        System.out.println ("Code Number: " + codeNumber) ;
        System.out.println ("Item Cost: " + itemCost) ;
        System.out.println ("Total Items: " + totalItems) ;
        System.out.println ("Total Cost: " + totalCost) ;
    }
}
```

The screen would look as follows when the program is executed.

```
Enter code number
1001
Enter number of items
193
Enter cost
452
Code Number   : 1001
Item cost :   452.0
Total Items : 193
Total Cost :  87236.0
```

Graphical Input and Output

Program 16.11 creates a simple sequential student file interactively using window frames. The program uses the **TextFiled** class to create text fields that receive information from the user at the keyboard and then writes the information to a file. A record of information contains roll number, name, and marks obtained by a student in a test.

Program 16.11 *Creating a file using text fields in windows*

```
import java.io.*;
import java.awt.*;
class StudentFile extends Frame
{
    // Defining window components
    TextField number, name, marks;
    Button enter, done;
    Label numLabel , namelabel, markLabel;
    DataOutputStream dos;
    // Initialize the Frame
    public StudentFile ( )
    {
        super ("Create Student File") ;
    }
    // Setup the window
    public void setup ( )
    {
        resize (400, 200) ;
        setLayout (new GridLayout (4, 2) ) ;
        // Create the components of the Frame
        number = new TextField (25) ;
        numLabel = new Label ("Roll Number") ;
        name = new TextField (25) ;
```

```
        nameLabel = new Label ("Student name") ;
        marks = new TextField (25) ;
        markLabel = new Label ("Marks") ;
        enter = new Button ("ENTER") ;
        done = new Button ("DONE") ;

        // Add the components to the Frame
        add (numLabel) ;
        add (number) ;
        add (nameLabel) ;
        add (name) ;
        add (markLabel) ;
        add (marks) ;
        add (enter) ;
        add (done) ;
        // Show the Frame
        show ( ) ;
       // Open the file
        try
        {
             dos = new DataOutputStream (
                       new FileOutputStream ("student.dat") ) ;
        }
        catch (IOException e)
        {
             System.err.println (e.toString ( ) ) ;
             System.exit (1) ;
        }
}
// Write to the file
public void addRecord ( )
{
     int num;
     Double d;
     num = (new Integer (number.getText ( ) ) ). intValue ( ) ;
     try
     {
        dos.writeInt(num);
        dos.writeUTF(name.getText ( ) ) ;
        d = new Double (marks.getText ( ) ) ;
        dos.writeDouble (d. doubleValue ( ) ) ;
    }
    catch (IOException e) { }

    // Clear the text fields
    number.setText (" ") ;
    name.setText (" ") ;
    marks.setText (" ") ;

 }

// Adding the record and clearing the TextFields

public void cleanup ( )
```

```
{
  if (! number.getText ( ) . equals (" ") )
     {
     addRecord ( ) ;
}
try
{
     dos.flush ( ) ;
     dos.close ( ) ;
          }
          catch (IOException e) { }
     }
     // Processing the event
     public boolean action (Event event, object o)
     {
          if (event.teg instanceof Button)
          {
               if (event.arg.equals ("ENTER") )
               {
                    addRecord ( ) ;
                    return true;
               }
          }
          return super.action (event, o) ;
     }
     public boolean handleEvent (Event event)
     {
          if (event.get instanceof Button)
          {
               if (event.arg.equals ("DONE") )
               {
                    cleanup ( ) ;
                    System.exit (0) ;
                    return true;
               }
          }
          return super.handleEvent (event) ;
     }
     // Execute the program
     public static void main (String args [ ] )
     {
          StudentFile student = new StudentFile ( ) ;
          student.setup ( ) ;
     }
}
```

The program uses classes **Frame, TextField, Button,** and **Label of java.awt** package to create the window and the text fields required to receive a student record. The method **setup( )** does the job of setting up the window. The method **addRecord( )** writes the information to the "student.dat" file created earlier.

When we execute the program, a window appears on the screen to enable us to enter data (Fig. 16.16).

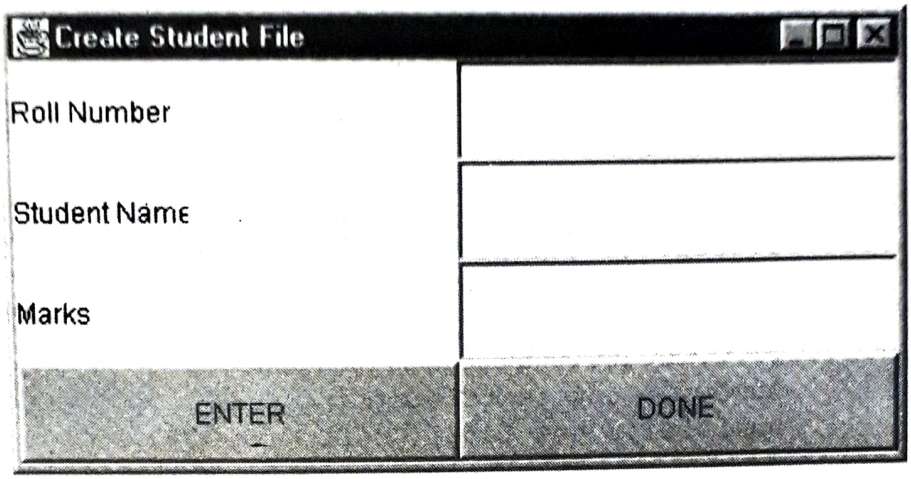


Fig. 16.16 *Screen output produced by Program 16.11*

After entering data in the appropriate text fields, we must click the **ENTER** button to write data to the file. This invokes the **addRecord( )** which performs the task of writing to the file. When we clock the "DONE" button, the program will call the method **cleanup( )** which will close the file stream and terminate the program execution.

Program 16.12 reads the data stored in "student.dat" file by the previous program. The program opens the file for reading and sets up a window that is similar to the one created for writing. When we click the **NEXT** button, the program reads a record from the file and displays its content in the text fields of the window. A clock on the **DONE** button closes the file stream and then terminates the execution.

Program 16.12 *Reading a file using text fields*

```
import java.io.*;
import java.awt.*;
class ReadStudentFile extends Frame
{
     // Defining window components
     TextField number, name, marks;
     Button next,done;
     Label numLabel, nameLabel, markLabel ;
     DataInputStream dis;
     boolean moreRecords = true;

     // Initialize the Frame
     public ReadStudentFile ( )
     {
          super ("Create Student File") ;
     }

     // Setup the window
     public void setup ( )
     {
          resize (400, 200) ;
          setLayout (new GridLayout (4, 2) ) ;

          // Create the components of the Frame
          number = new TextField (25) ;
          numLabel =new Label ("Roll Number") ;
          name = new TextField (25) ;
```

```
    nameLabel = new Label ("Student Name") ;
    marks = new TextField (25) ;
    markLabel = new Label ("Marks") ;
    next = new Button ("NEXT") ;
    done = new Button ("DONE") ;

    // Add the components to the Frame
    add (numLabel) ;
    add (number) ;
    add (nameLabel) ;
    add (name) ;
    add (markLabel) ;
    add (marks) ;
    add (next) ;
    add (done) ;

    // Show the Frame
    show ( )

    // Open the file
    try
    {
        dis = new DatainputStream (
        new FileinputStream ("student.dat") ) ;
    }
    catch (IOException e)
    {
            System.err.println (e.toString ( ) ) ;
            System.exit (1) ;
    }
}

// Read from the file
public void readRecord ( )
{
    int n;
    String s;
    double d;
    try
    {
        n = dis.readInt ( ) ;
        s = dis.readUTF ( ) ;
        d = dis.readDouble ( ) ;
        number.setText (String.valueOf (n) ) ;
        name.setText (String.valueOf (s) ) ;
        marks.setText (String.valueOf (d) ) ;
    }
    catch (EOFException e)
    {
        moreRecords = false;
    }
    catch (IOException ioe)
    {
        System.out.println ("IO Error") ;
        System.exit (1) ;
    }
}
```

```
// Closing the input file
public void cleanup ( )
{
     try
     {
          dis.close ( ) ;
     }
     catch (IOException e) { }
}

// Processing the event
public boolean action (Event event, Object o)
{
     if (event.target instanceof Button)
     {
     if (event.arg.equals ("NEXT") )
               readRecord ( ) ;
     }
     return true;
}
public boolean handleEvent (Event event)
{
     if (event.target instanceof Button)
{
     if (event.arg.equals ("DONE") ||
               moreRecords == false)
     {
          cleanup ( ) ;
          System.exit (0) ;
          return true;
     }
     }
     return super.handleEvent (event) ;
}
// Execute the program
public static void main (String args [.] )
{
     ReadStudentFile student = new ReadStudentFile ( ) ;
     student.set up ( ) ;
}

}
```

16.17 OTHER STREAM CLASSES

Java supports many other input/output streams that we might find useful in some situations. A brief discussion of some of these streams is given as follows.

Object Streams

We have seen in this chapter how we can read and write characters, bytes, and primitive data types. It is also possible to perform input and output operations on objects using the object streams. The object streams are created using the **ObjectInputStream** and **ObjectOuptutStream** classes. In this case, we may declare records

as objects and use the object classes to write and read these objects from files. As mentioned in the beginning, this process is known as *object serialization.*

Piped Streams

Piped streams provide functionality for threads to communicate and exchange data between them. Figure 16.17 shows how two threads use *pipes* for communication. The write thread sends data to the read thread through a pipeline that connects an object of **PipedInputStream** to an object of **PipedOutputStream.** The objects **inputPipe** and **outputPipe** are connected using the **connect ( )** method.

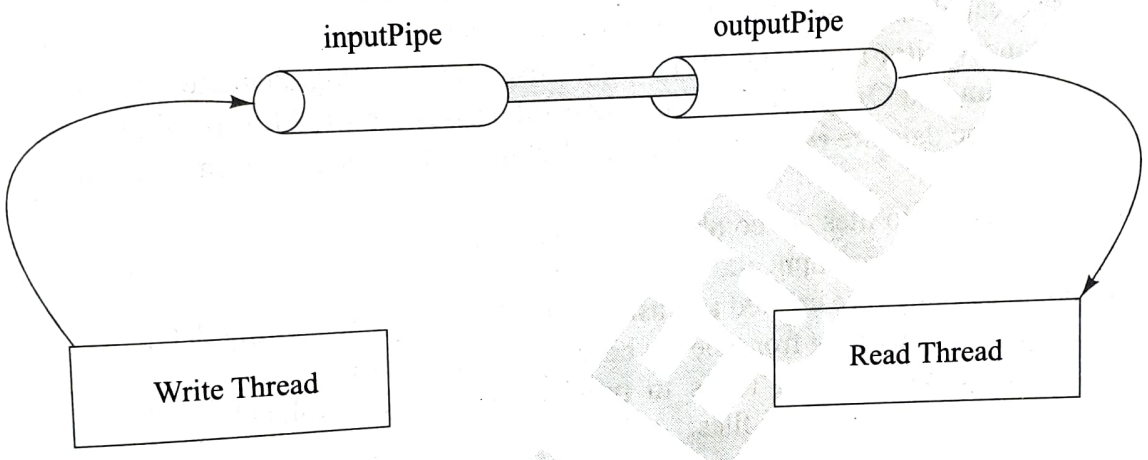


Fig. 16.17 *Threads using pipes to communicate*

Pushback Streams

The pushback streams created by the classes **PushbackInputStream** and **PushbackReader** can be used to push a single byte or a character (that was previously read) back into the input stream so that it can be reread. This is commonly used with parsers. When a character indicating a new input token is read, it is pushed back into the input stream until the current input token is processed. It is then reread when processing of the next input token is initiated.

Filtered Streams

Java supports two abstract classes, namely, **FilterInputStream** and **FilterOutputStream** that provide the basic capability to create input and output streams for filtering input/output in a number of ways. These streams, known as *filters,* sit between an input stream and an output stream and perform some optional processing on the data they transfer. We can combine filters to perform a series of filtering operations as shown in Fig. 16.18. Note that we used **DataInputStream** and **DataOutputStream** as filters in the Program 16.5 for handling primitive type data.

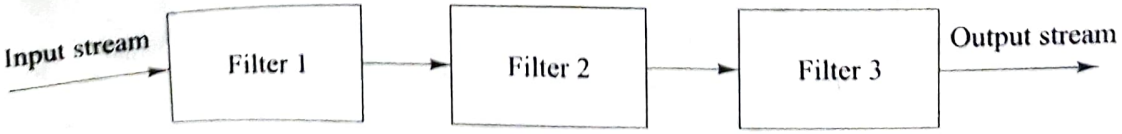


Fig. 16.18 *The concept of using filters*

16.18 SUMMARY

In this chapter we have learned how to work with files for storing and retrieving data. We have discussed in detail the following:

- How the concept of streams are used for handling all input and output operations.
- How stream classes provide capabilities for processing files.
- How stream classes are classified into different groups to handle different data types.
- How the classes in each group are hierarchically related.
- What are the basic methods used for input and output operation.
- How files are created and opened for input and output operations.
- How **Reader** and **Writer** and their subclasses are used for handling characters in files.
- How **InputStream** and **OutputStream** and their subclasses are used for handling bytes in files.
- How primitive type data are read or written to files using **DataInputStream** and **DataOutputStream** classes.
- How the contents of two files are combined into a single file.
- How the buffers are used in input and output operations.
- How random access files are created and used for both reading and writing data.
- How to read data interactively from the keyboard and write to files.
- How to use windows and text fields to provide interactive graphical display while performing the input and output operations on files.

KEY TERMS

Files, Persistent Data, Unicode, Secondary Storage, File Processing, Object Serialization, Records, Bytes, Fields, Stream, Source, Destination, Input Stream, Output Stream, Reader Stream, Writer Stream, Tokens, Concatenation, Buffering, Sequential File, Random Access File, File Pointer, Interactive Input, Graphical Interactive I/O, Windows, Text Fields, Pipes, Filters.

REVIEW QUESTIONS

16.1 What is a file? Why do we require files to store data?
16.2 What is a stream? How is the concept of streams used in Java?
16.3 What are input and output streams? Explain them with illustrations.
16.4 What is a stream class? How are the stream classes classified?
16.5 Describe the major tasks of input and output stream classes.
16.6 Distinguish between
(a) InputStream and Reader classes
(b) OutputStream and Writer classes
16.7 Describe the functions of the **File** class?
16.8 Describe the most commonly used classes for handling i/o related exceptions.
16.9 State the steps involved in creating a disk file.
16.10 What is meant by initializing a file stream object? What are the ways of doing it? Give example code for each of them.
16.11 Which streams must always be used to process external files? Why?

16.12 What is a random access file? How is it different from a sequential file? Why do we need a random access file?
16.13 Create a **DataInputStream** for the file named "student.dat".
16.14 Create a **RandomAccessFile** stream for the file "student.dat" for updating the student information in the file.
16.15 Write statements to create a file stream that concatenates two existing files.
16.16 Can we open an existing file for writing? If not, why?
16.17 How would you check whether a file to be opened for writing already exists?
16.18 While reading a file, how would you check whether you have reached the end of the file?
16.19 Write statements to create data streams for the following operations:
(a) Reading primitive data from a file
(b) Writing primitive data to a file
16.20 Describe, through appropriate statements, how a double type value is read from the keyboard interactively.
16.21 Write a program that will count the number of characters in a file.
16.22 Modify the above program so that it will also count the number of words, and lines in the file.
16.23 Rewrite Program 16.1 using the **FileInputStream** and **FileOutputStream** classes.
16.24 Rewrite Program 16.4 using the **FileReader** and **FileWriter** classes.
16.25 Write a program to create a sequential file that could store details about five products. Details include product code, cost, and number of items available and are provided through the keyboard.
16.26 Write a program to read the file created in Review Question 16.25 and compute and print the total value of all the five products.
16.27 Rewrite the program of Review Question 16.25 using a random access file so that we can add more products to the file, if necessary.
16.28 Write a program that will print the details of the alternate products stored in the random access file of Review Question 16.27.

DEBUGGING EXERCISES

16.1 Find errors in the following code which writes data of one file to another file.

```
import java.io.*;
public class file1
{
    public static void main(String args[])
    {
        try
        {
            FileReader fr=new FileReader("in.dat" ) ;
            FileWriter fw=new FileWriter"out.dat");
            int ch;
            while((ch=fr.read())!=-1)
            {
                fr.write(ch) ;
            }
        }
        catch(Exception ex)
        {
            System.out.println(ex);
        }
    }
}
```

16.2 Debug the following code for reading a file using FileInputStream class.

```
import java.io.*;
class file2
{
    public static void main(String args[])
    {
        if (args.length ==1)
        {
            try
            {
                FileInputStream fstream = new FileInputStream(args[0]);
                DataInputStream in = new DataInputStreamO;
                while (in.available() !=0)
                {
                    System.out.println(in.readLine());
                }
                in.close( );
            }
            catch (Exception e)
            {
                System.err.println("File input Error");
            }
        }
    else
        System.out.println ( )"Invaiid parameters");
    }
}
```

16.3 Find the compile-time error in the program given below.

```
import java.io.*;
class FileOutput
{
    public static void main(String args[])
    {
        FileOutputStream out;
        PrintStream p;
        try
        {
            out = new FileOutputStream( );
            p = new PrintStream( out );
            p.println ("This is written to a file");
            p.close( );
        }
        catch (Exception e)
        {
            System.err.println ("Error writing to file");
        }
    }
}
```

16.4 The program throws an exception. Correct the code to make it run successfully.

```
import java.io.*;
```

```
public class randomAccess
{
   public static void main(String[] args) throws IOException
   {
        RandomAccessFile raf = new RandomAccessFile ("random.text",
"rwasd");
        try
        {
             Writer out = new OutputStreamWriter(new
             FileOutputStream(raf.getFD( )), "UTF-8");

             out.write"Programming in C");
             (out.flush( );
             raf.seek(12);
             out.write ("Java");
             out.flush( );
         }
         finally
         {
              raf .close( );
         }
    }
}
```

16.5 Correct the code for reading a file byte by byte.

```
import java.io.*;
class ByteRead
{
    public static void main(String args[]) throws IO Exception
    {
         int i ;
         FileInputStream fin=new FileInputStream("c:\\input.text");
         do
         {
              i=fin.readByte();
              System.out.println((char)i);
         }
         while( != -1);
         fin.close()
         fin = null;
    }
}
```

CHAPTER 17

Java Collections

17.1 INTRODUCTION

The *collections framework* which is contained in the **java.util** package is one of Java's most powerful subsystems. The collections framework defines a set of interfaces and their implementations to manipulate collections, which serve as a container for a group of objects such as a set of words in a dictionary or a collection of mails. The collections framework also allows us to store, retrieve, and update a set of objects. It provides an API to work with the data structures, such as lists, trees, maps, and sets. In this chapter, we shall discuss the interfaces, classes, and algorithms available in the collections framework.

17.2 OVERVIEW OF INTERFACES

The collections framework contains many interfaces, such as Collection, Map, and Iterator. Other interfaces of the framework extend these interfaces. The interfaces available in the collections framework can be structured as shown in Fig. 17.1. The interfaces List and Set are the subinterfaces of the Collection interface. The SortedMap interface is the subinterface of the Map interface. The ListIterator interface is the subinterface of the Iterator interface. Brief description of these interfaces is provided in Table 17.1.

Table 17.1 *Description of Interfaces*

Interface	*Description*
Collection	collection of elements.
List(extends Collection)	sequence of elements.
Queue (extends Collection)	special type of list
Set(extends Collection)	collection of unique elements.
SortedSet(extends Set)	sorted collection of unique elements.
Map	collection of key and value pairs, which must be unique.
SortedMap(extends Map)	sorted collection of unique key value pairs.
Iterator	object used to traverse through a collection.
ListIterator (extends Iterator)	object used to traverse through the sequence.

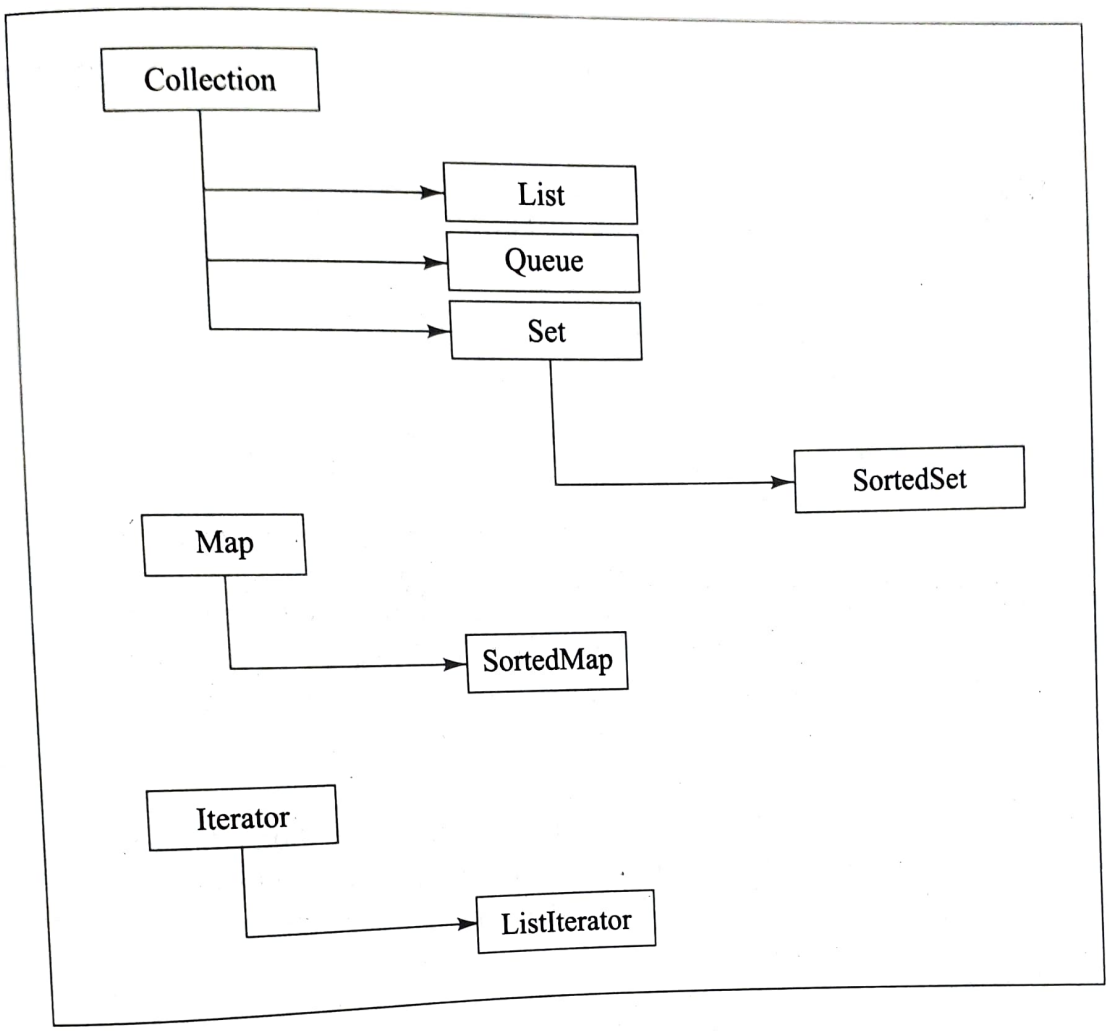


Fig. 17.1 *Interfaces defined in the Collections Framework*

The Collection Interface

All collection classes must implement the Collection interface. The Collection interface defines some methods, which enable us to access the objects of a collection. Table 17.2 describes these methods.

Table 17.2 *Methods Defined in the Collection Interface*

Methods	*Description*
add(object o)	Returns true if the object is added to the specified collection.
addAll(collection c)	Returns true if the entire object in the collection is added to the specified collection.
clear( )	Removes all elements from the specified collection.
contains(object o)	Returns true if the collection contains the specified element.
containsAll(collection c)	Returns true if the collection contains all the elements in the specified collection.
equals(object o)	Returns true if the specified object matches with the object in the collection.
hashCode( )	Returns the hashcode for the collection.
isEmpty( )	Return true if the collection is empty.
iterator( )	Returns an iterator over the elements in the collection.
Methods	*Description*

(*Contd*)

Table 17.2 (*Contd*)

Methods	*Description*
remove(object o)	Returns true, if the specified element is present in the collection and removes the object from the collection.
removeAll(collection c)	Returns true, if all the elements in collection c is removed from the specified collection.
retainAll(collection c)	Returns true, if all the elements in Collection c is retained in the specified collection.
size( )	Returns the number of elements in the collection.
toArray( )	Returns an array containing all of the elements in the collection.
toArray(object[] a)	Returns an array of object if the array contains all the elements in the specified collection.

The methods, **add( )**, **addAll( )**, and **remove( )** that modify the collection objects, will throw an exception, **UnsupportedOperationException,** when the collection does not support the respective operation. The methods of the Collection interface will throw the **ClassCastException** exception, when we add an incompatible element to a collection. For example, some collections may not support null elements.

The Set Interface

The Set interface extends the Collection interface and it contains the methods that are inherited from the Collection interface. The Set interface does not allow the use of duplicate elements in a collection. Hence, the **add( )** method returns false, if we add the duplicate element to the collection.

The List Interface

The List interface contains an ordered sequence of elements available in a collection. It allows duplicate elements in the List. The List interface inherits the methods of the Collection interface. In addition to these methods the List interface also contains the methods described in Table 17.3.

Table 17.3 *Methods Defined in the List Interface*

Method	*Description*
add (int index, object o)	Adds the element, o in the specified index of the list.
addAll (int index, collection c)	Adds all the elements of collection, c in the specified index of the list.
get (int index)	Returns the element available in the specified index of the list.
indexOf (object o)	Returns the index of object o in the list. If there are more than one occurrence of object o, the method returns the index of the first occurrence. If the object o is not available in the list, the method returns –1.
lastIndexOf (object o)	Returns the last index of the object o in the list. If the object o is not available in the list, the method returns –1.
listIterator ( )	Returns a list iterator of the elements.
listIterator (int index)	Returns a list iterator of the elements starting from the specified index of a list.
remove (int index)	Removes the element at the specified index of the list.
set (int index, object o)	Replaces the element in the specified index with the specified element.
subList (int startindex, int endindex)	Returns the elements available from the specified startindex to the endindex.

The SortedSet Interface

The SortedSet interface is used to sort the elements of a collection in ascending order. The SortedSet interface extends the Set interface, which in turn extends the Collection interface. The SortedSet interface does not allow duplicate elements in a set. In addition to the methods defined by the Set interface, the SortedSet interface contains the methods listed in Table 17.4.

Table 17.4 *Methods Defined in the SortedSet Interface*

Methods	*Description*
comparator( )	Returns the comparator object. If the elements in the SortedSet are in ascending order then it returns null.
first( )	Returns the first element from the SortedSet.
headset(Object toElement)	Returns the number of elements less than that of the elements specified using the toElement object. The elements are returned from the sorted set.
last( )	Returns the last element from the SortedSet.
subSet(Object FromElement, Object ToElement)	Returns the elements between the range specified by the objects, FromElement and ToElement. Here the returned set includes the FromElement and excludes the ToElement.
tailSet(Object FromElement)	Returns the elements from a sorted set that are greater than or equal to the FromElement.

The Queue Interface

The Queue interface extends Collection interface and declares the behaviour of a queue, which is often a first-in, first-out list. In a queue, elements can only be removed from the head of the queue. The Queue interface defines a few methods as listed in Table 17.5.

Table 17.5 *Methods Defined in Queue*

Methods	*Description*
element( )	Returns the element at the head of the queue. The element is not removed.
offer(Object o)	Attempts to add an element to the queue. Returns true if added, false otherwise.
peek( )	Returns the element at the head of the queue. The element is not removed.
poll( )	Returns the element at the head of the queue after removing the element.
remove( )	Returns the element at the head of the queue after removing the element.

Note:
1. The mothods **element( )** and **peek( )** are similar but, if the queue is empty, **element( )** throws the exception **NoSuchElementException** while **peek( )** returns **null.**
2. The methods **poll( )** and **remove( )** perform the same job but, when the queue is empty, **poll( )** returns **null** while **remove( )** throws the exception **NoSuchElementException.**

The Map Interface

The Map interface maps unique key elements to their values. For example, in a mail server, each mail id is mapped to a unique password. The Map interface allows us to view the elements of a collection as set of keys, collection of values, and the mappings of key-value pairs. Table 17.6 describes the methods of the Map interface.

Table 17.6 *Methods Defined in the Map Interface*

Methods	*Description*
clear( )	Removes all the mappings from a map.
containsKey(Object key)	Returns true if a map contains mapping for the specified key.
contains Value(Object value)	Returns true if the specified value maps with one or more key in a map interface.
entrySet( )	Returns the key-value pair contained in this map.
equals(Object o)	Returns true if the specified object maps with an object of a map interface.
get(Objectkey)	Returns the value, which is mapped to the specified key.
isEmpty( )	Returns true if a map contains no key-value mapping.
keySet( )	Returns the keys in a map. If we remove a key from the map, the corresponding value will also be removed.
put(Object key, Object value)	Maps the specified key with the specified value.
putAll(Map t)	Copied all the specified key value pair from a specified map to the map with which we are currently working.
remove(Object key)	Removes the specified key from the map.
size( )	Returns the number of key-value mappings available in a map.

The SortedMap Interface

The SortedMap interface extends the Map Interface. The SortedMap interface contains elements in ascending order. In this, the sorting is based on the keys. The functionality of the SortedMap is analogous to the functionality of the SortedSet interface. This Map interface is implemented in the TreeMap class. Table 17.7 describes the methods of the SortedMap Interface.

Table 17.7 *Methods Defined in the SortedMap Interface*

Methods	*Description*
comparator( )	Returns the comparator of the sorted map. It returns null, if the sorted map uses natural ordering for their keys.
firstKey( )	Returns the first key of the sorted map.
headMap(Object end)	Returns the keys of the sorted map that are less than the specified end object.
lastKey( )	Returns the last key of the sorted map.
subMap(Object start, Object end)	Returns the keys of the sorted map that are greater than or equal to the specified start object and less than are equal to the specified end object.
tailMap (Object start)	Returns the keys of the sorted map that are greater than or equal to the start object.

The Iterator Interface

The Iterator interface enables us to sequentially traverse and access the elements contained in a collection. The elements of a collection can be accessed using the methods defined by the Iterator interface. Table 17.8 describes the methods of the Iterator interface.

Table 17.8 *Methods Defined in the Iterator Interface*

Method	*Description*
hasNext( )	Returns true if the collection contains more than one element.
next( )	Returns the next element from the collection.
remove( )	Remove the current element from the collection.

The **next( )** method of the Iterator interface returns the next element if it is available in the collection. If there is no such element in the collection, it will throw the exception **NoSuchElementException.** The **remove( )** method throws the exception **IllegalStateException** when there is no element in the collection. Using the Iterator interface, we can insert elements only at the end of a list. If we need to insert elements at the required location of the list, we should use the ListIterator interface, which extends the Iterator interface. The ListIterator interface contains a method named **add(int index, Object obj).** This method allows us to add an element at the required location based on the index value.

17.3 OVERVIEW OF CLASSES

The classes available in the collections framework implement the collection interface and the subinterfaces. These classes also implement the Map and Iterator interfaces. Table 17.9 lists out the classes and their corresponding implementations.

Table 17.9

Class	*Name of the interface*
AbstractCollection	Collection
AbstractList	List
AbstractQueue	Queue
AbstractSequentialList	List
LinkedList	List
ArrayList	List, Cloneable, and Serializable
AbstractSet	Set
EnumSet	Set
HashSet	Set
PriorityQueue	Queue
TreeSet	Set
Vector	List, Cloneable, and Serializable
Stack	List, Cloneable, and Serializable
Hashtable	Map, Cloneable, and Serializable

The AbstractCollection Class

The AbstractCollection class implements the Collection interface. Therefore it contains all the methods available in the Collection Interface. We use the AbstractCollection class to implement a collection, which cannot be modified. For example, a collection containing the months of a year cannot be modified.

We can also implement collection using the AbstractCollection class by overriding the **add(object o)** method and implementing the **remove (object o)** method. The constructor for this class is represented as:

```
Protected     AbstractCollection()
```

The AbstractList Class

The AbstractList class extends the AbstractCollection class and implements the List interface. We use the AbstractList class to access the data randomly. For example, we can use the index values to access the elements of an array at random. The Constructor for this class is represented as:

```
Protected AbstractList( )
```

The AbstractList class extends the methods, such as **add(Object o), clear(), iterator()** of the AbstractCollection class. It also inherits the methods from the List interface.

Note: The AbstractSequentialList class is used to access the elements of an array sequentially.

The ArrayList Class

The ArrayList class extends the AbstractList class and implements the interfaces, such as List, Cloneable and Serializable. Using the ArrayList class, we can use dynamic array in Java applications. The dynamic array is an array in which the array size is not fixed in advance. Therefore, we can change the size of an array at run time using the ArrayList class. Every instance of the ArrayList class is allowed to store a set of elements in the list. The capacity increases automatically as we add elements to the list.

The constructor of the ArrayList takes three forms:

- **ArrayList():** Creates an empty list. The capacity of the list is initialized to ten.
- **ArrayList(Collection c):** Creates a list to which the elements of the specified collection are added.
- **ArrayList(int capacity):** Creates an empty list. The capacity of the list is initialized to the specified value.

The ArrayList class inherits the methods from the List interface. The elements of an array can be accessed directly using the **get()** and **set()** methods. The **add()** method is used to add an element to the array list and the **remove()** method is used to remove an element from the array list. Program 17.1 that illustrates the usage of the **add()** and **remove()** methods of the ArrayList class:

Program 17.1 *Using the methods of the ArrayList Class*

```
import java.util.*;
public class ArrayListExample
{
    public static void main(String args[])
    {
        ArrayList arraylist=new ArrayList();
        System.out.println("Initial size of arraylist" +arraylist.size());
        arraylist.add("A");
        arraylist.add("B");
        arraylist.add("C");
        arraylist.add("D");
        System.out.println( "Size of arraylist after adding the element"+
            arraylist size());
        System.out.println("Contents of arraylist"+arraylist);
        arraylist.add(2, "E");
```

```
    {
        list.addLast(obj);
    }
    public Object bottom()
    {
        return list.getLast();
    }
    public Object pop()
    {
        return list.removeFirst();
    }
    public static void main(Strings args[])
    {
        Car myCar;
        Bird myBird;
        MyStack s = new MyStack();
        s.pushl (new Car());
        s.push2(new Bird());
        myCar = (Car)s.pop();
        System.out.println("The first element in the list:" +myCar);
        myBird=(Bird)s.bottom();
        System.out.println("The last element in the list:" +myBird);
    }
}
class Car
{
    String carl, car2, car3, car4;
    Car()
    {
        carl="Benz";
        car2="Toyoto";
        car3="Qualis";
        car4="Santro";
    }
}
class Bird
{
    String birdl, bird2, bird3;
    Bird()
    {
        birdl="parrot";
        bird2="duck";
        bird3="raven";
    }
}
```

The output of Program 17.2 is:

```
The first element in the list: Benz
The last element in the list: raven
```

The HashSet Class

The HashSet class extends the AbstractSet class and implements the Set interface. The AbstractSet class itself extends the AbstractCollection class. The HashSet class is used to create a collection and store it in a

hash table. Each collection refers to a unique value called hash code. The hash code is used as an index to associate with the object, which is stored in the hash table. This type of storing information in a hash table is called *hashing*. Constructors for the HashSet class are:

- **HashSet():** Constructs an empty HashSet
- **HashSet(Collection c):** Initializes the HashSet using the element c
- **HashSet(int capacity):** Initializes the capacity of the HashSet
- **HashSet(int capacity, float fillratio):** Initializes the capacity and the fill ratio of the HashSet

The value of the fill ratio ranges from 0.0 to 1.0. This value is used to set the initial size of the hash set. If the number of elements is greater than the capacity of the hash set, the size of the hash set is expanded automatically by multiplying the capacity with the fill ratio. The default value of the fill ratio is 0.75.

The HashSet class inherits the methods of its parent classes and the methods of the implemented interface. Program 17.3 illustrates the use of methods of the HashSet class. It is important to note that **HashSet** does not guarantee the order of its elements. Elements may be stored in any order.

Program 17.3 *Using the methods of the HashSet Class*

```
import java.util.*;
class HashSetExample
{
    public static void main(String args[])
    {
        HashSet hs=new HashSet();
        hs.add("D")
        hs.add("A");
        hs.add("C");
        hs.add("B");
        hs.add("E");
        System.out.println("The elements available in the hash set are: "+hs);
    }
```

The output of Program 17.3 is:

```
The elements available in the hash set are: [D, A, C, B, E]
```

The TreeSet Class

The TreeSet Class implements the Set interface. The sorted elements are stored in a tree structure. This class allows us to access and retrieve the elements from a tree in less time. The constructor for the TreeSet class can take the following forms:

- **TreeSet( ):** Builds an empty tree set. This empty constructor will sort the elements in ascending order.
- **TreeSet(Collection c):** Builds a tree set, which contains a collection of elements.
- **TreeSet(Comparator comp):** Builds a tree set based on the specified comparator.
- **TreeSet(SortedSet s):** Builds a tree set, which contains elements in the specified order.

Program 17.4 illustrates the usage of the TreeSet class.

Program 17.4 *Using the TreeSet Class*

```
import java.util.*;
public class TreeSetExample
{
    public static void main(String args[])
    {
        TreeSet ts=new TreeSet();
        Ts.add("B");
        Ts.add("C");
        Ts.add("A");
        Ts.add("E");
        Ts.add("D");
        System.out.println("The elements in the TreeSet are: "+ ts);
    }
}
```

The output of Program 17.4 is:

```
The elements in the TreeSet are: [A, B, C, D, E]
```

Note that the elements are automatically arranged in sorted order.

The Vector Class

The Vector class extends the AbstractList class and implements the interfaces, such as List, Cloneable, and Serializable. The Vector class is similar to the ArrayList class except that the Vector class is synchronized. As the Vector class is synchronized, multiple threads cannot access the vector objects simultaneously. Only one thread can access the Vector object at the specific time. The Vector class implements the dynamic array. The constructors of the Vector class are:

- **Vector():** Serves as a default constructor with an array size of 10.
- **Vector(int size):** Builds a vector with the specified size.
- **Vector(int size, int increment):** Builds a vector of the specified size. The specified increment value is used to increase the size of the vector. If the increment value is not specified, then the size of the vector is doubled for each allocation.
- **Vector(Collection c):** Builds a vector that consists of elements available in the specified collection c.

Table 17.11 illustrates the methods and the task performed by these methods of the Vector class:

Table 17.11 *Methods Defined in the Vector Class*

Methods	*Description*
addElement(Object element)	Adds the specified element to the end of the vector and increments the size of the vector by one.
capacity( )	Returns the current capacity of the vector.
contains(Object element)	Returns true if the specified object is present in the vector.
containsAll(Collection c)	Returns true if the vector contains all the elements specified in the collection.
elementAt(int index)	Returns an element at the specified index.
ensureCapacity(int minimumcapacity)	Sets the specified minimum capacity to the size of the vector.
Methods	*Description*

(Contd)

Table 17.11 (*Contd*)

Methods	*Description*
get(int index)	Returns the object, which is available in the specified position of the vector.
setElementAt(Object e,int index)	Replaces the element at the specified index with specified element.
setSize(int newsize)	Sets the size of the vector to the specified size, newsize.
size()	Returns the size of the vector.
toString()	Return a String representation of this vector.

Program 17.5 *Example of using the Vector Class*

```
import java.util.Iterator;
import java.util.Vector;
public class VectorExample
{
    public static void main(String[] args)
    {
        Vector fruits = new Vector();
        fruits.add("Apple");
        fruits.add("Orange");
        fruits.add("Grapes");
        fruits. add("Pine");
        Iterator it = fruits.iterator();
        while(it.hasNext())
            System.out.printIn(it.next());
    }
}
```

The output of Program 17.5 is:

```
Apple
Orange
Grapes
Pine
```

The Stack Class

The Stack class extends the Vector class. In addition to the inherited methods from the Vector class, the Stack class contains some methods to perform operations, such as push, pop, peek, and search. The constructor of Stack class can be represented as:

```
Stack()
```

The above constructor is used to create an empty stack. The Stack class uses the First In Last Out (FILO) mechanism. Table 17.12 describes some of the methods of the Stack class:

Table 17.12 *Methods Defined in the Stack Class*

Methods	*Description*
empty()	Returns true if the stack is empty.
peek()	Returns the element at the top of the Stack.
pop()	Removes the element at the top of the Stack and returns the element that is removed from the Stack.
push()	Adds an item to the top of the Stack.

For example, consider the Program 17.6 that adds string values to a stack.

Program 17.6 *Example of using the Stack Class*

```
import java.util.*;
public class stackex
{
    public static void main(String args[])
    {
        Stack st=new Stack();
        st.push("Java");
        st.push("latest");
        st.push("Edition");
        st.push("-fifth");
        System.out.println("The elements in the Stack: "+st);
        System.out.println("The element at the top: "+st.peek());
        System.out.println("The element poped out of the stack: "+st.pop());
        System.out.println("The element in a stack after pop out an element: "+st);
        System.out.println("The  result  of  searching:  "+st.search("r  u");
    }
}
```

The output of Program 17.6 is:

```
The element in the Stack: [Java, latest, Edition, -fifth]
The element at the top: -fifth
The element popped out of the Stack: -fifth
The element in a stack after pop out an element: [Java, latest,
Edition]
The result of searching: -1
```

The Hashtable Class

The Hashtable class implements the interfaces, such as Map, Cloneable, and Serializable. The hashtable is used to store values in the form of map key with value. The key should implement the hashcode and equals methods to store and retrieve values in a hashtable. The constructors of the Hashtable class are:

- **Hashtable():** Creates an empty hashtable with the default initial capacity as 11 and the loadFactor as 0.75.
- **Hashtable(int initialCapacity):** Creates an empty hashtable with the specified initial capacity and the default loadFactor.
- **Hashtable(int initialCapacity, float loadFactor):** Creates an empty hashtable, which has the specified initial capacity and loadFactor.
- **Hashtable(Map m):** Creates a new hashtable with the specified map.

Program 17.7 *Using the Hashtable Class*

```
import java.util.Enumeration;
import java.util.Hashtable;
public class HashTableExample
{
```

```
public static void main(String[] args)
{
    Hashtable ht = new Hashtable();
    ht.put("Item1", "Apple");
    ht.put("Item2", "Orange");
    ht.put("Item3", "Grapes");
    ht.put("Item4", "Pine");
    Enumeration e = ht.keys();
    while(e.hasMoreElements())
    {
        String str = (String) e. nextElement() ;
        System.out.println(ht.get(str));
    }
}
}
```

The output of the above program is:

```
Apple
Orange
Grapes
Pine
```

17.4 OVERVIEW OF ALGORITHMS

The collections framework supports several algorithms that allow us to operate on collections. We can use these algorithms to sort, shuffle, manipulate, and search a set of elements in a collection. Some of the algorithms available in the collections framework are:

- Sorting
- Shuffling
- Manipulating
- Searching

These algorithms are contained in the Collections class. Table 17.13 shows some algorithms defined in the Collection class.

Table 17.13 *Algorithius Defined in the Collections Class*

Methods	*Description*
binarySearch(Listl,Object v)	Searches the specified object in the specified list. It returns the position of the specified object in the list. If the specified object is not in the list, then it returns -1.
copy(List src, List dest)	Copies elements from one list to another.
disjoint(Collection cl, Collection c2)	Returns true if no common element is available in the two specified collections.
frequency(Collection c, Object o)	Returns the number of elements that equal to the specified object in the specified collection.
indexOfSubList(List src, List dest)	Returns the index of the first occurrence of the specified destination list in the specified source list and -1 if the source list does not contain the occurrence.

(*Contd*)

Table 17.13 (*Contd*)

Methods	*Description*
lastIndexOfSubList(List src, List dest)	Returns the index of the last occurrence of the specified destination list in the specified source list and -1 if the source list does not contain the occurrence.
replaceAll(List l, old_val, new_val)	Replaces all occurrences of the old value in the list with the new value.
reverse(List l)	Reverses the order of elements in the list.
shuffle(List l, Random r)	Shuffles the elements in the list by using r as a source to generate random values.
swap(List1, index i1, index i2)	Interchanges the elements in the specified indexes of the list.

The Sort Algorithm

The sort algorithm enables us to arrange the elements of a list in a certain order. The ordering depends on the type of elements. If the list contains a set of string elements, the sorting is done alphabetically. If the list contains a set of numeric elements, the elements are arranged in ascending order.

The Shuffle Algorithm

The shuffle algorithm shuffles the elements of a list such that the current order of the list is destroyed. This algorithm arranges the elements using all possible permutations. For example, we can use this algorithm to shuffle objects in a memory game.

Manipulating Algorithms

The Collections class provides algorithms to perform operations, such as fill, reverse, copy, swap, and add on a list of elements on a collection. The reverse operation reverses the order of elements in the list. The fill operation replaces the elements of a list with a specified elements. The copy operation copies the elements of one list to another. The swap operation swaps the specified elements in a list. The addAll operation adds the specified elements to a list.

The Search Algorithm

The search algorithm allows us to search an element in a collection. Here we use the binary search algorithm. To find an element from a list, we need to traverse the entire list. We can use the binary search algorithm in a sorted list. The steps to find an element from a list using binary search algorithm are:

Step 1: Sort the elements in the collection.
Step 2: Find the middle element of the collection.
Step 3: Compare the specified element with the middle element of the collection.
Step 4: If the middle element is greater than the specified element, traverse the first half of the list. Else traverse the second half of the list.

The **binarySearch()** method of the Collections class implement the binary search algorithm. We need to pass the collection and the element to be searched from the specified collection as arguments to the **binarySearch()** method.

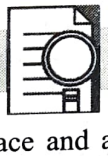
If the element in the collection is not available in the sorted order, we need to pass the comparator as an additional argument to the **binarySearch()** method. This comparator is used to sort the elements of the collection according to the condition specified in the comparator.

Program 17.8 *Use of binarySearch method*

```
Import java.util.*;
Class algorithmdemo
{
    public static void main(String args[])
    {
        LinkedList l=new LinkedList();
        l.add(new String("Java"));
        l.add(new String("is"));
        l.add(new String("platform"));
        l.add(new String("Independent"));
        Comparator r=Collections.reverseOrder();
        Collections.sort(l, r);
        Iterator iter=l.iterator();
        System.out.println("List sorted in reverse order");
        While(iter.hasNext())
        System.out.println(iter.next() + "");
        Collections, shuffle(l) ;
        Iter= l.iterator();
        System.out.println("List shuffled : ");
        While(iter.hasNext())
        System.out.println(iter.next() + "");
        System.out.println();
        System.out.println("Minimum :" + Collections.min(l)) ;
        System.out.println("Maximum:" + Collection.max(l));
    }
}
```

The output for the above program is:

```
List sorted in reverse order: Java is platform independent
List shuffled:
Minimum: independent
Maximum: platform
```



DEBUGGING EXERCISES

17.1 The following code creates an object of List interface and adds and removes items from it. Will this code compile successfully?

```
import java.util.*;
public class ListInterfaceExample
{
        public static void main(String[] args)
        {
            List<String> list;
            list = new ArrayList<String>();
            list.add(l"a");
            list.add(0."b");
            list.add(1, "c");
            list.add(1,"d");
```

```
            list.add(3,"d");
            System.out.println("List is "+list);
            int size = list.size() ;
            Object element = list.get(list.size()-1 );
            System.out.println("Element at "+list.size()+" location is
                    "+element);
            element = list.getItem(());
            System.out.println("Element at 0 location is "+element);
            Collections.sort(list);
            Collections.sort(list, String.CASE_INSENSITIVE_ORDER);
            System.out.println("List after sort is "+list);
            boolean b = list.remove("c");
            element = list.delete(0);
            System.out.println("List after removal of c and 1st element" + list;
    }
}
```

Ans: No, list object does not contain any function with the name of getItem and delete.

```
import java.util.*;
public class ListInterfaceExample
{
    public static void main(String[] args)
    {
        List<String> list;
        list = new ArrayList<String>();
        list.add("a");
        list.add(0,"b");
        list.addd,(1,"c");
        list.add(1,"d");
        list.add(3,"d");
        System.out.println("List is "+list);
        int size = list.size();
        Object element = list.get(list.size()-1 );
        System.out.println("Element at "+list.size()+ "location is
                "+element);
        element = list.get(0);
        System.out.println("Element at 0 location is "+element);
        Collections.sort(list);
        Collections.sort(list, String.CASE_INSENSITIVE_ORDER);
        System.out.println("List after sort is "+list);
        boolean b = list.remove("c");
        element = list.remove(0);
        System.out.println("List after removal of c and 1st element
            "+list);
    }
}
```

17.2 The following code will play with the objects of Set interface. Will this code compile successfully?

```
import java.util.*;
public class SetsExample
{
   public static void main(String[] args)
   {
        Set<String> set1 = new HashSet<String>();
        set1.add("a");
        set1.add("b");
        set1.add("c");
        System.out.println("Set1 is "+set1);
        set1.remove("c");
        System.out.println("Set1 after removing c is "+set1);
        int size = set1.size();
        System.out.println("Size of set1 is "+size);
        set1.add("a");
        size = set1.size();
        System.out.println("Size of seti after adding duplicate item is
            "+size);
        boolean b = set1.isContains("a");
        System.out.println("Is Set1 contains a "+b);
        System.out.println("Is Set1 contains c "+set1.contains("c"));
        Set<String> set2 = new HashSet<String>();
        set2.add("e");
        set2.add("d");
        set2.add("f" );
        System.out.println("Set2 is "+set2);
        set2.add(set1);
        System.out.println("Set2 is after merging set1 elements"+set2);
        set2.removeAll(set1);
        System.out.println("Set2 is after deleting set1 elements"+set2);
        set2.addAll(set1);
        set2.retainAll(set1);
        System.out.println("Set2 is after deleting all elements except
            set1 elements"+set2);
   }
}
```

Ans: No, there is no function with the name of isContains in set interface and add function does not add a complete set into another set. Add function only add a single element in a set.

```
import java.util.*;
public class SetsExample
{
   public static void main(String[] args)
   {
        Set<String> set1 = new HashSet<String>();
        set1.add("a");
```

```
        set1.add("b");
        set1.add("c");
        System.out.println("Set1 is "+set1);
        set1.remove("c");
        System.out.println("Set1 after removing c is "+set1);
        int size = set1.size();
        System.out.println("Size of set1 is "+size);
        set1.add("a");
        size = set1.size();
        System.out.println("Size of set1 after adding duplicate item is
            "+size);
        boolean b = set1.contains("a");
        System.out.println("Is Set1 contains a "+b);
        System.out.println("Is Set1 contains c "+set1.contains("c"));
        Set<String> set2 = new HashSet<String>();
        set2.add("e");
        set2.add("d");
        set2.add("f");
        System.out.println("Set2 is "+set2);
        set2.addAll(set1);
        System.out.println("Set2 is after merging set1 elements "+set2);
        set2.removeAll(set1);
        System.out.println("Set2 is after deleting set1 elements "+set2);
        set2.addAll(set1);
        set2.retainAll(set1);
        System.out.println("Set2 is after deleting all elements except set1
        elements"+set2);
    }
}
```

18.3 The following code will create a vector object and convert it to array. What will be the size and capacity of v after execution of this program?

```
import java.util.*;
class VectorExample
{
    public static void main (String [ ] args)
    {
        Vector<String> v = new Vector<String>(15);
        int i;
        System.out.println("starting...");
        for( i=0;i<2;i++)
        {
            v.add(args[i]);
            System.out.println(args[i]+" added to vector");
        }
        System.out.println ("Converting to array");
        String[] list = new String[v.size()];
        v.copyInto(list);
        System.out.println ("Printing array");
```

```
        for (i=0;i < v.size ( );i++)
        {
            System.out.println ("Element at "+i+" location is "+ list [i]);
        }
    }
}
```

Ans: 17.4 Size of the v will be 2 and the capacity of the v will be 15.

The following code will create a Hashtable object and add some values in it and add it into vector to perform sort using Collections.sort method. Will this code execute successfully? If yes, what will be the output of this?

```
import java.util.*;
public class HashTableExamples
{
    public static void main(String[] s)
    {
        Hashtable<String,Integer> hash= new Hashtable<String,Integer> (4);
        String ob = "ABC";
        Integer in = new Integer(563);
        hash.put (ob,in);
        ob= "XYZ";
        in = new Integer(129);
        hash.put ( ob ,in);
        ob = "MNO";
        in = new Integer (6564);
        hash.put (ob ,in);
        System.out.println(hash);
        Vector<String> v = new Vector<String>(hash.keySet());
        Collections.sort(v);
        for (Enumeration e = v.elements(); e.hasMoreElements();)
        {
            String key = (String)e.nextElement();
            Integer val = (Integer)hash.get(key);
            System.out.println("Key: " + key + " Val: " + val);
        }
    }
}
```

Ans: 17.5 Yes, this will execute successfully and display sorted data on string values of hashtable.

The following code will create a String type array to perform binary search for word "Hello". Will this code compile successfully?

```
public class BinarySearchExample
{
    public int binarySearch(String[] sorted, String key)
    {
        int first = 0;
        int last = sorted.length;
        while (first < last)
        {
            int mid = (first + last) / 2;
            if (key.compareTo(sorted[mid]) < 0)
            {
```

```
                last = mid;
            }
            else if (key.compareTo(sorted[mid]) > 0)
            {
                first = mid + 1;
            }
            else
            {
                return mid;
            }
        }
        return -(first + 1);
    }
    public static void main(String[] args)
    {
        int i=binarySearch(args,"Hello");
        if(i<0)
            System.out.println("Not found");
        else
            System.out.println("Found at "+(i+1)+ "location.");
    }
}
```

Ans: No, this will give a compile time error in function main which is referencing a non-static method binarySearch. binarySearch method should be static to call it from static function main without creating any object of this class.

```
public class BinarySearchExample
{
    public static int binarySearch(String[] sorted, String key)
    {
        int first = 0;
        int last = sorted.length;
        while (first < last)
        {
            int mid = (first + last) / 2;
            if (key.compareTo(sorted[mid]) < 0)
            {
                last = mid;
            }
            else if (key.compareTo(sorted[mid]) > 0)
            {
                first = mid + 1;
            }
            else
            {
                return mid;
            }
        }
        return -(first + 1);
    }
```

```
    public static void main(String[] args)
    {
        int i=binarySearch(args,"Hello");
        if(i<0)
            System.out.println("Not found");
        else
            System.out.println("Found at "+(i+1)+" location.");
    }
}
```