Chapter -6
File Handling

Introduction:

When we use variables and array for storing data inside the programs. We face two Problems:

1) The data is lost either when a variable goes out of scope or when the program is terminated. 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 into secondary storage devices. We can store data using concept of files. Data stored in file is often called persistent data.

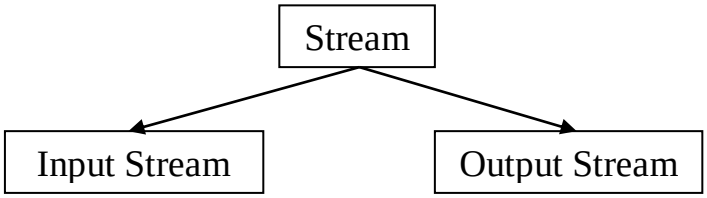
- **A file is collection of related records placed area on the disk. A record is composed of several fields. Field is a group of characters.**
- Storing and managing data using file is known as file processing which includes tasks such as creating files, updating files and manipulation of data.
- Reading and writing of data in a file can be done at the level of bytes or characters or fields depending on the requirement of application.java provides capabilities to read and write class object directly.
- The process of reading and writing objects is called **serialization.**

Concept of stream

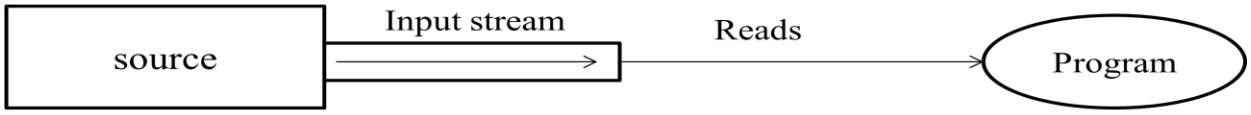
- In file processing, inputrefers to the flow of data into a program and **output** means the flow of data out of a program.
- Input to a program may come from the keyboard, mouse, memory, disk a network or another program.
- Output from a program may go to the screen, printer, memory disk, network or another program.
- Input and output share certain common characteristics like unidirectional movement of data, treating data as a sequence of bytes or characters and support to sequential access to data.
- Java uses concept of **stream** to representordered sequence of data, a common characteristics shared by all input/output devices.
- A **stream** presents 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 pipe along which water flows). It has source (of data) and destination (for that data).Both the

source and destination may be physical devices or programs or other streams in same program.

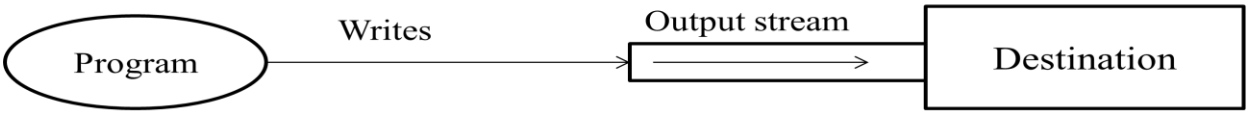
- The concept of sending data from one stream to another has made streams in java a powerful tool for file processing.
- We can build a complex file processing sequence using a series of simple stream operation. This feature is used to filter data along the pipeline of streams so that we obtain data in desired format.
- Java stream classified into basic type as follow:



- Input stream: it extracts (i.e. reads) data from the source (file) and sends it to the program.
- Output stream: it takes data from the program and sends (i.e. writes) it to the destination (file).



(a) Reading data into a program



(b) Writing data to a destination

Stream classes

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

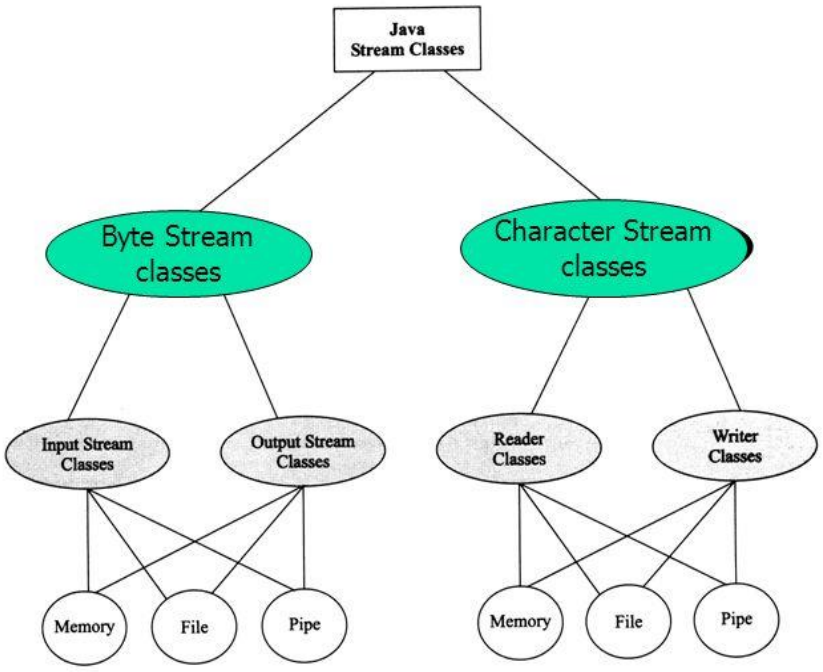
1. **Byte stream classes** that provides support for handling I/O operations on bytes.

2. **Character stream**classes that provide support for managing I/O operation on characters

. These groups mayfurther be classified based on their functions. Byte stream and character stream classes contain specialized classes to deal with input and output

operations independently on various type of devices. We can also cross-group the streams based on the type of source or destination they from or write to. The source (or destination) may be memory, a file or a pipe.

Classification of Java Stream Classes



Classification of Java stream classes

52

1) Byte stream classes

- Byte stream classes have been designed to provide functional features for creating and manipulating streams and files reading and writing bytes.
- Since the steams are unidirectional, they can transmit bytes in only one direction and, therefore,
- Java provides two kinds of byte stream classes:
  1) **input stream classes**
  2) **output stream classes.**

1) Input Stream Classes

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

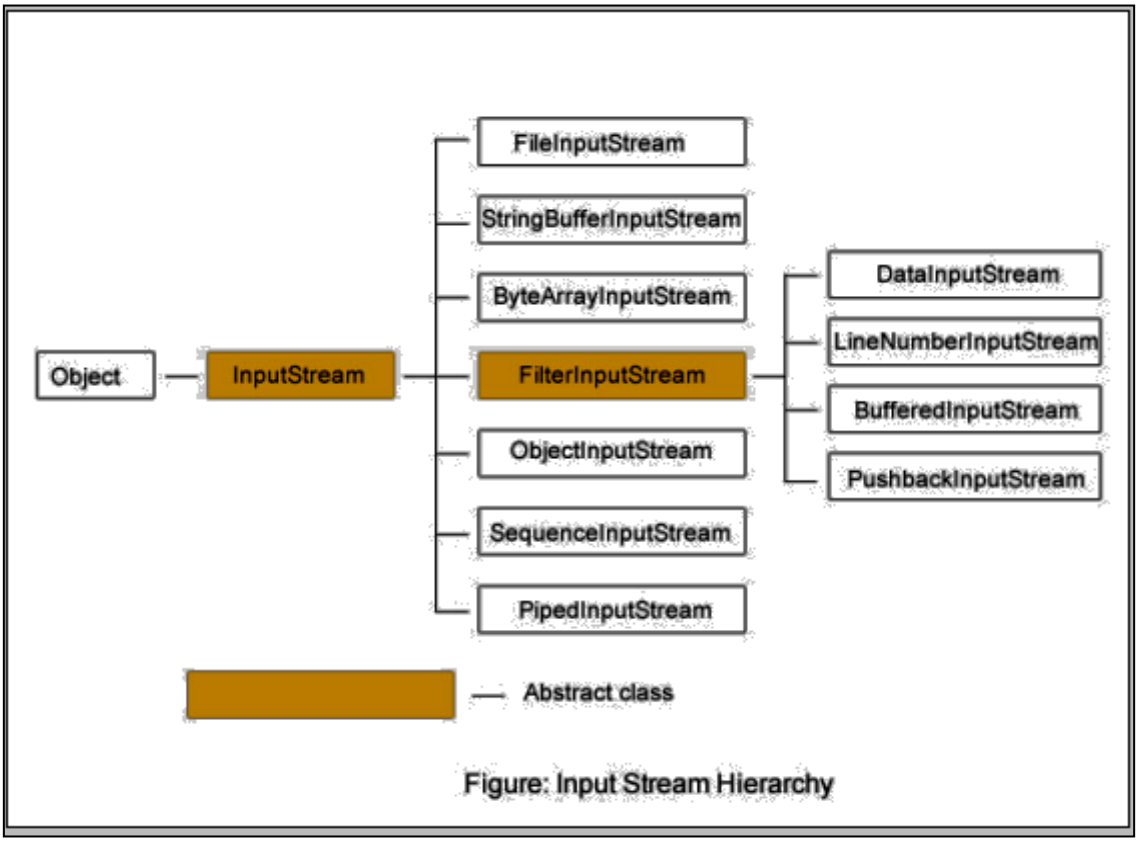


Figure: Input Stream Hierarchy

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

- Reading bytes
- Closing streams
- Marking position in streams
- Skipping ahead in a stream
- Finding the number of bytes in a stream

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[],intn,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 method of InputStream class. The DataInput interface contains the following method.

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

2) OutputStream Classes

- Output stream classes are derived from the base class OutputStream as shown in figure. 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.

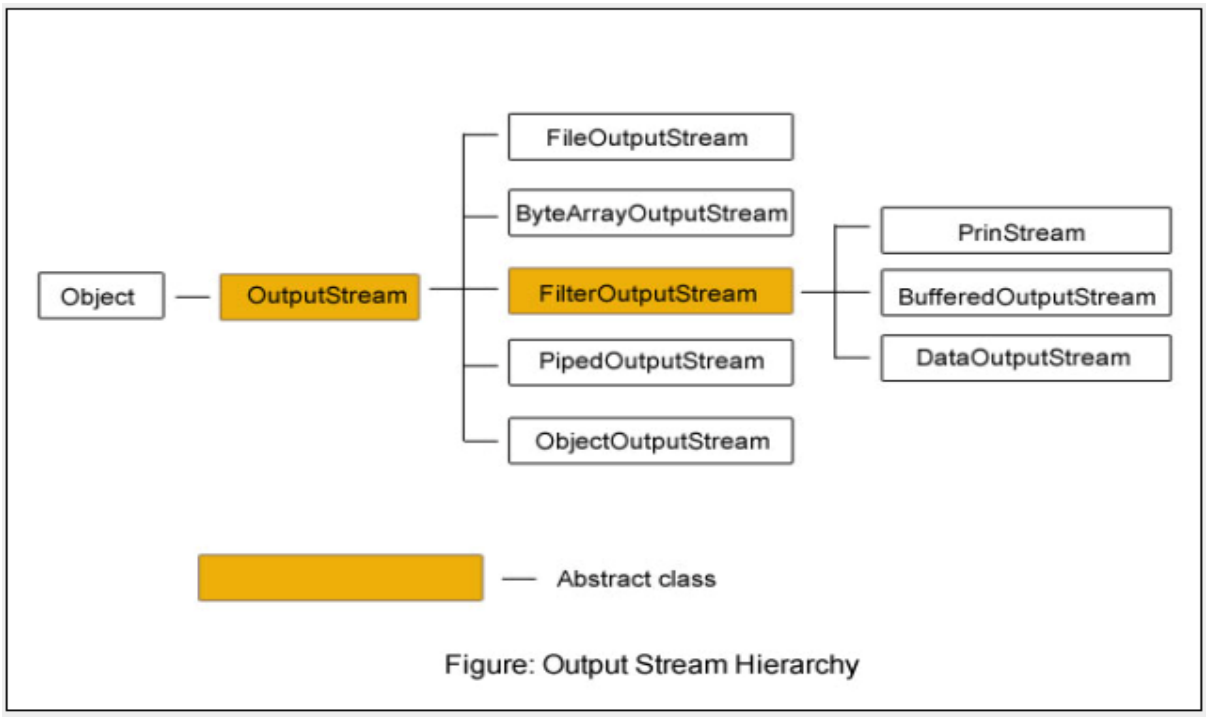


Figure: Output Stream Hierarchy

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

- Writing bytes
- Closing stream
- Flushing stream

Summary of output stream 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[],intn,int m)	Writes m bytes from array b starting from nth byte
4	close()	Close the output stream
5	flush()	Flushes the output stream

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

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

2) 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, reader classes and writer stream classes.

1) 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 figure.
- These classes are functionally very similar to the input stream classes, except input stream use bytes as their fundamental unit of information, while reader stream use characters.

- The Reader class contains method they are identical to those available in the InputStream class, except Reader is designed to handle characters (see Table 6.1) Therefore, reader classes can perform all the functions implemented by the input stream classes.

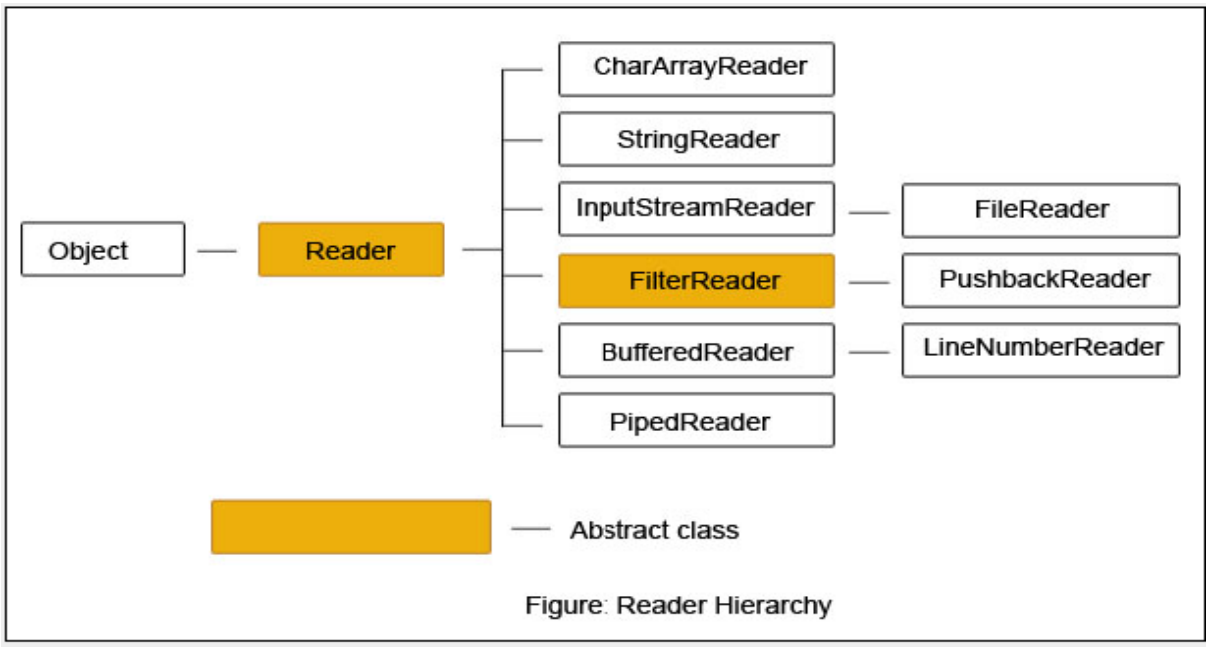


Figure: Reader Hierarchy

2)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 other writer stream classes as shown in figure. This base class provides support for all output operations by defining method that are identical to those in **OutputStream**class(see Table 6.2)

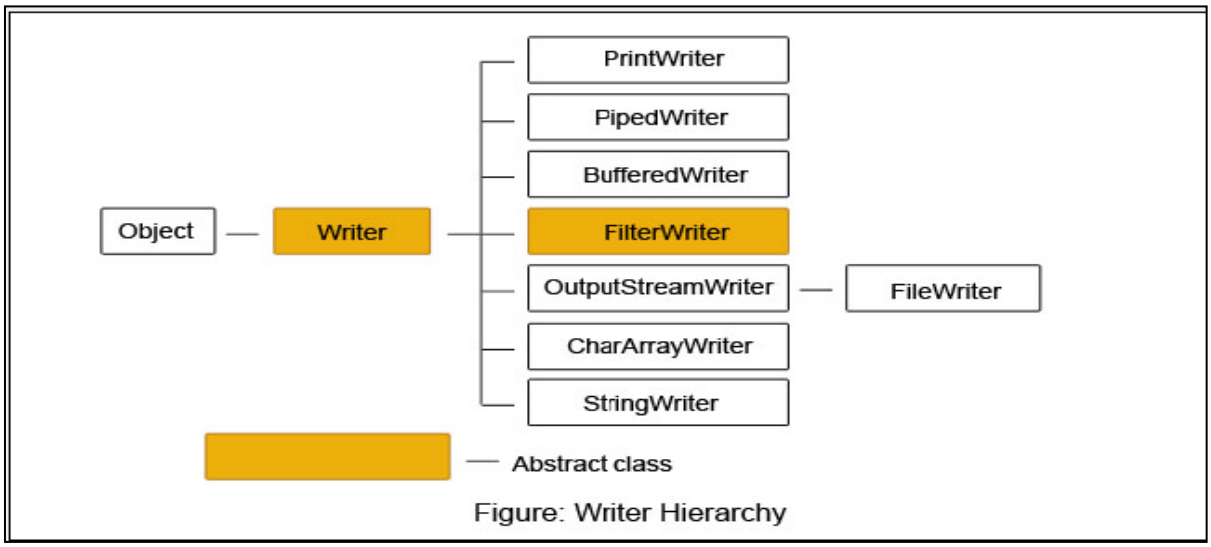


Figure: Writer Hierarchy

Using streams

- We have 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.
- As pointed out earlier, both the character stream group and the byte stream group contain parallel pairs of classes that perform the same of operations but for the different data type.

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
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

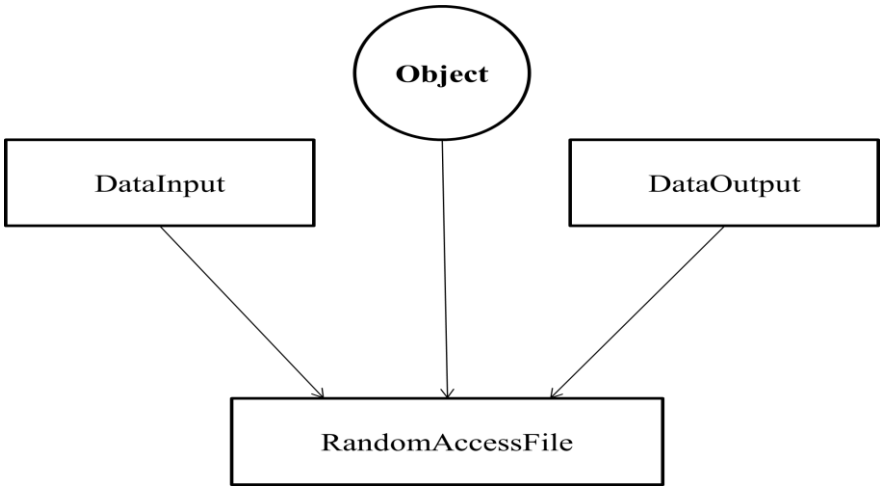
Other useful classes

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

- RandomAccessFile :

  The RandomAccessFile enables us to read and write bytes, text and java data types to any location in a file. This class extends **object** class and implements **DataInput**and **DataOutput**interface
- StreamTokenizer :

  The class **StreamTokenizer,** 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 **StringTokenizer**class (of java.util package) that breaks string into its component tokens.



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.

File class provides methods for operations like:

- 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

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:

	I/O exception class	Function
1	EOFException	Signals that an end of file or end of stream has been reached unexpectedly during input
2	FileNotFoundException	Informs that a file could not be found
3	InteruuptedIOException	Warns that an I/O operations has been interrupted
4	IOException	Signals that an I/O exception of some sort has occurred

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 character 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. Example:

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, weshould know whether the file is created for reading 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 class of **Reader/InputStream**for reading data and**Writer/OutputStream**for

writing data. The common stream classes used for various i/o operations given in table . The constructors of stream classes may be used to assign the desired filenames to the Stream objects.

Source/destination	Characters	
	Read	Write
Memory	CharArrayReader	CharArrayWriter
File	FileReader	FileWriter
Pipe	PipedReader	PipedWriter

Source/destination	Bytes	
	Read	Write
Memory	ByteArrayInputStream	ByteArrayoutputStream
File	FileInputStream	FileoutputStream
Pipe	PipedInputStream	PipedoutputStream

There are two ways of Italianizing the file Streamobjects. All of the constructors require that we provide the name of the file either directly or indirectly by giving a file object that has already assigned a file name. The following code segment illustrates the use of direct approach.

```
FileInputStreamfis;
try
{
//Assign the filename to the file stream object
fis = new FileInputStream (“test.txt”);
.....
}
catch (IOException e)
.....
.....
```

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

```
.....
```

```
.............
File inFile;
InFile = new file (“test.txt”);
FileInputStreamfis;
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 a selected name to the file object declared.
- Declare a file stream object.
- Connect the file to the stream object.

Example-1:
Write a program which creates file and writes byte into that file.

```
import java.io.*;

public class WriteByte
{
        public static void main(String args[])
        {
        File fl=new File(“input.txt”);              \\ to create new file
        FileOutputStreamoutfile = null;
        byte cities[] = {'I',’ ‘,’'L','O','V','E',’ ‘,'I','N','D','I’,’A'};
                try
                {
                        outfile = new FileOutputStream(f1);
                        outfile.write(cities);
                }
                catch(IOException e)
                {
                        System.out.println(e);
                        System.exit(-1);
                }
                System.out.println("Write Byte");
                System.out.println("Thank You...!!!");
        }
}
```

Output :

```
Write Byte
Thank You!!!
```

Program-2

```
import java.io.*;

importjava.util.*;

public class WriteByte_1

{

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

```
        {
                FileOutputStreamoutfile = null;
                //String  s=args[0];          // to input string from command line
                Scanner sc=new Scanner(System.in);
                String s=sc.nextLine();


                byte b1[] = s.getBytes();
                try
                {
                        outfile = new FileOutputStream("in.txt");
                        outfile.write(b1);
                }
                catch(IOException e)
                {
                        System.out.println(e);
                        System.exit(-1);
                }
                System.out.println("Write Byte");
                System.out.println("Thank You...!!!");
        }
}
```

Example-2:

Write a program which reads byte from file.

```
import java.io.*;
public class ReadingByte
{
	public static void main(String args[])
	{
		FileInputStreaminfile = null;
		int b;
		try
		{
			infile = new FileInputStream("input.txt");
			while((b = infile.read()) != -1)
			{
				System.out.println((char)b);
			}
			infile.close();
		}
		catch(IOException e)
		{
			System.out.println("Sorry..!! File Not Found...!!!");
		}
	}
}
```

Output :

```
I LOVE INDIA
```

Reading/writing characters

As pointed out earlier, subclass of Reader and Writer implement streams that can handle characters. The two subclasses used for handling characters in files are **FileReader**and **FileWriter**.

The concept of using file streams and file object for reading and writing characters in program is illustrated in fig:

Example-3
Write a program which creates file and writes characterinto that file.

```
import java.io.*;
classCharacterWrite
{
        public static void main(String args[])
        {
                File f1=new File("input1.txt");
                FileWriterfw  = null;
                try
                {
                        fw=new FileWriter(f1);
                        fw.write("ahmedabad  \n");
                        fw.write("  baroda    \n");
                        fw.close();
                }
                catch(FileNotFoundException e)
                {
                        System.out.println("Sorry..!! File Not Found...!!!");
                }
                catch(IOException e)
                {
                        System.out.println(e.getMessage());
                }
        System.out.println(" write operation done!!");
        }
}
```

Output:
write operation done

Example-4:
Write a program which reads character from file.

```
import java.io.*;

classReadchar
{
	public static void main(String args[])
	{
		FileReaderfr =null;
		try
		{
			fr  = new FileReader("input.txt");
			intch;
			while((ch = fr.read()) != -1)
			{
				System.out.print((char)ch);

			}
			System.out.println("Reading complete");
			fr.close();
		}
		catch(FileNotFoundException e)
		{
			System.out.println("Sorry..!! File Not Found...!!!");
		}
		catch(IOException e)
		{
			System.out.println(e.getMessage());
		}
	}
}
```

Output:

ahmedabad

baroda

Reading complete

Example -5:

Write a program to read one byte at a time from a file and copy it into another fileimmediately.

```
import java.io.*;
classCopyByte
{
	public static void main(String args[])
	{
		try
		{
		byte b=0;
		FileInputStreaminfile  = new FileInputStream("in.txt");
		FileOutputStreamoutfile = new FileOutputStream("out.txt");
			while(byteread != -1)
			{
				b = (byte)infile.read();
				outfile.write(b);
			}
		System.out.println("Byte Copied From in.txt to out.txt FIle ");
		}
		catch(FileNotFoundException e)
		{
			System.out.println("Sorry..!! File Not Found...!!!");
		}
		catch(IOException e)
		{
			System.out.println(e.getMessage());
		}
	}
}
```

Output :

Byte Copied From in.txt to out.txt File

Example -6:

Write a program to merge two files in third file.

```
import java.io.*;

classFileMergeDemo
{
	public static void main(String args[])
	{
		try
		{
		FileInputStream file1 = new FileInputStream("File1.txt");
		FileInputStream file2 = new FileInputStream("File2.txt");
	SequenceInputStream file3 = new SequenceInputStream(file1, file2);
	BufferedInputStream br1 = new BufferedInputStream(file3);
	BufferedOutputStream br2 = new BufferedOutputStream(System.out);
			intch;
			while((ch = br1.read())!=-1)
			{
				br2.write((char)ch);
			}
			br1.close();
			br2.close();
			file1.close();
			file2.close();
			System.out.println("Merge Two File Sucessfully ");
		}
		catch(IOException e)
		{
			System.out.println("Sorry..!! File Not Found...!!!");
		}
	}
}
```

Output

```
Vpmpldrp
```

Write an application to rename a file. Use the renameTo() method of File to accomplish

/*this task. The first command line argument is the old filename and the second is the newfilename.
*/

```
import java.io.*;

classFileRenameDemo
{
        public static void main(String args[])
        {
                File f1 = new File(args[0]);
                File f2 = new File(args[1]);
                f1.renameTo(f2);
        System.out.println("Rename File " +f1+" To "+f2+" Sucessfully ");
        }
}
```

Output :

```
javacFileRenameDemo.java

javaFileRenameDemo  input1.txt  abc.txt
```