File Management in Java

PREPARED BY:

CHINTAN A GAJJAR

INFORMATION TECHNOLOGY DEPARTMENT

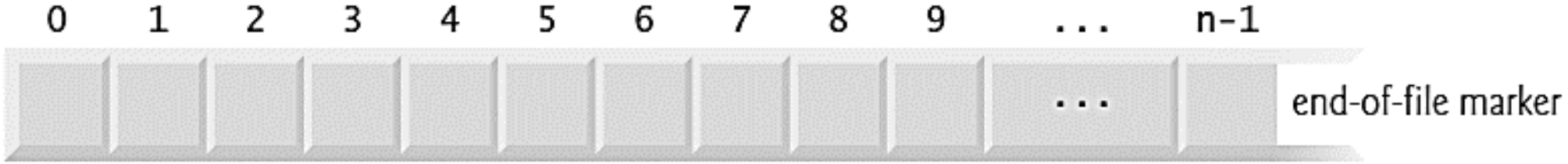
DR S & S S GHANDHY COLLEGE OF ENGINEERING & TECHNOLOGY, SURAT

Introduction to File

- Data stored in variables and arrays is *temporary*—it's lost when a local variable goes out of scope or when the program terminates. For long-term retention of data, even after the programs that create the data terminate, computers use **files**.
- You use files every day for tasks such as writing a document or creating a spreadsheet. Computers store files on **secondary storage devices**, including hard disks, flash drives, DVDs and more. Data maintained in files is **persistent data**—it exists beyond the duration of program execution.

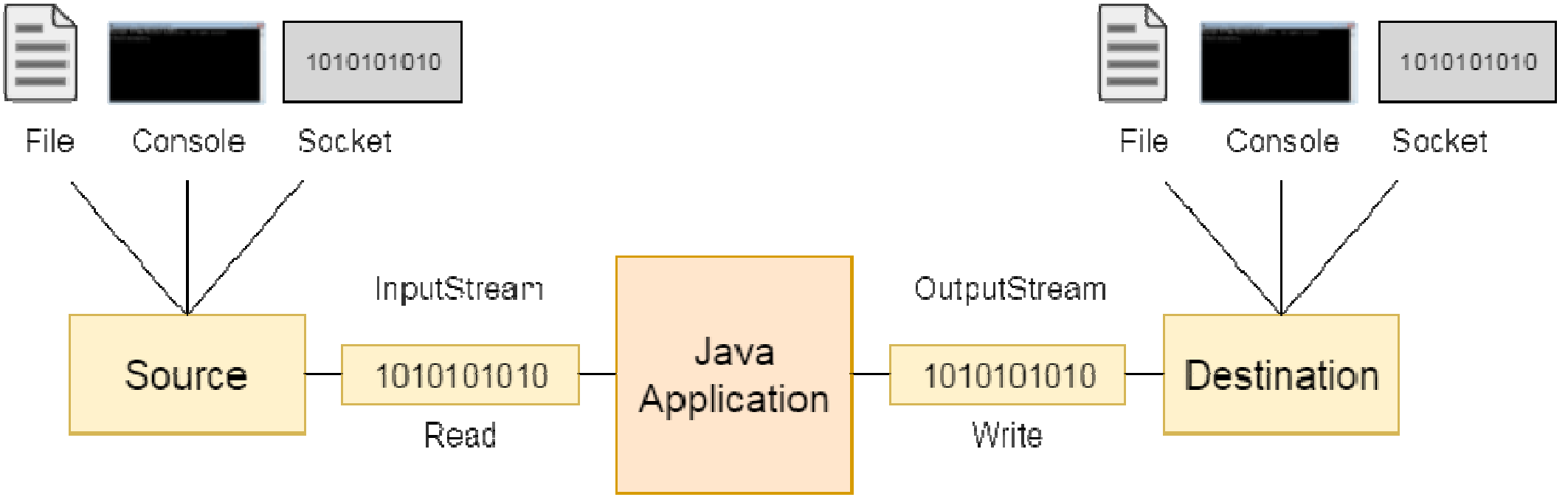
Stream

- Java views each file as a sequential **stream of bytes.** Stream facilitate transporting data from one place to another. Java programs perform I/O through streams.



- Different streams are needed to send or receive data through different sources, such as keyboard, file or network socket.
- A **stream** can be defined as a sequence of data. It can be categorized as ‘input streams’ and ‘output stream’.

Stream in Java

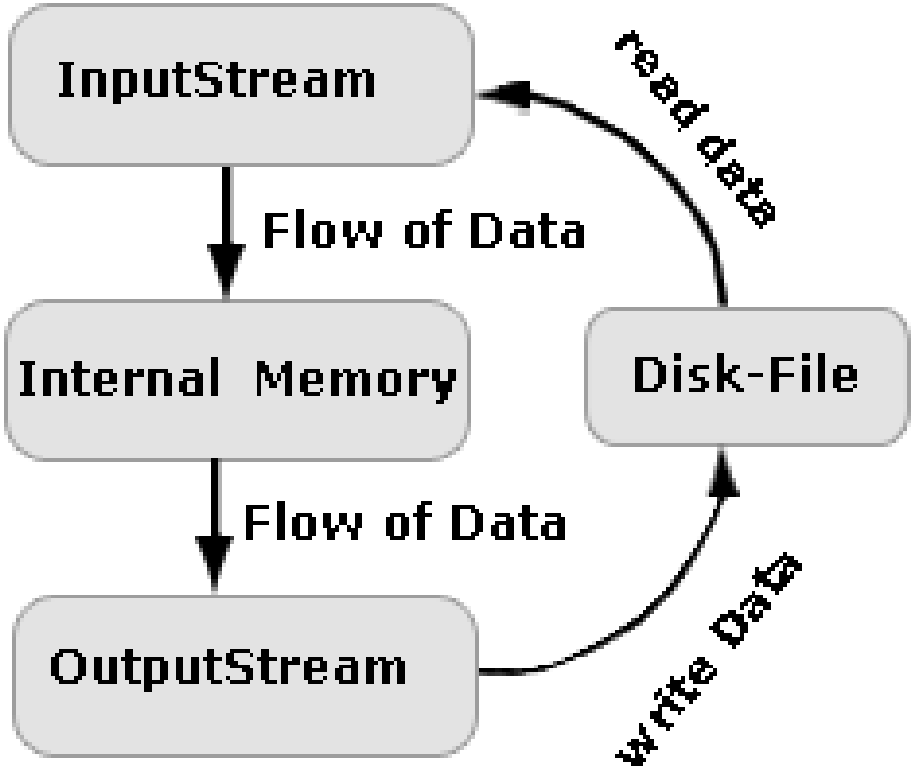


Stream

- Input streams are the steams which receive or read data from keyboard, file or network socket.
- Output streams are the streams which send or write data to the console, file or network socket.
- In java, all the streams are represented by classes in `java.io` package.

Files and Streams

- Java uses **streams** to handle I/O operations through which the data is flowed from one location to another. For example,
  an **InputStream** can flow the data from a disk file to the internal memory and
  an **OutputStream** can flow the data from the internal memory to a disk file. The disk-file may be a text file or a binary file.
- When we work with a text file, we use a **character** stream where one character is treated as per byte on disk. When we work with a binary file, we use a **binary** stream (i.e. **byte** stream).



3 streams

- When a Java program begins executing, it creates three stream objects that are associated with devices automatically:

1. **System.in:**

   This represents `InputStream` object, which by default represents standard input device, i.e. keyboard.

2. **System.out:**

   This represents `PrintStream` object, which by default represents standard output device, i.e. monitor.

3. **System.err:**

   This field also represents `PrintStream` object, which by default represents monitor. But it is used to display error messages.

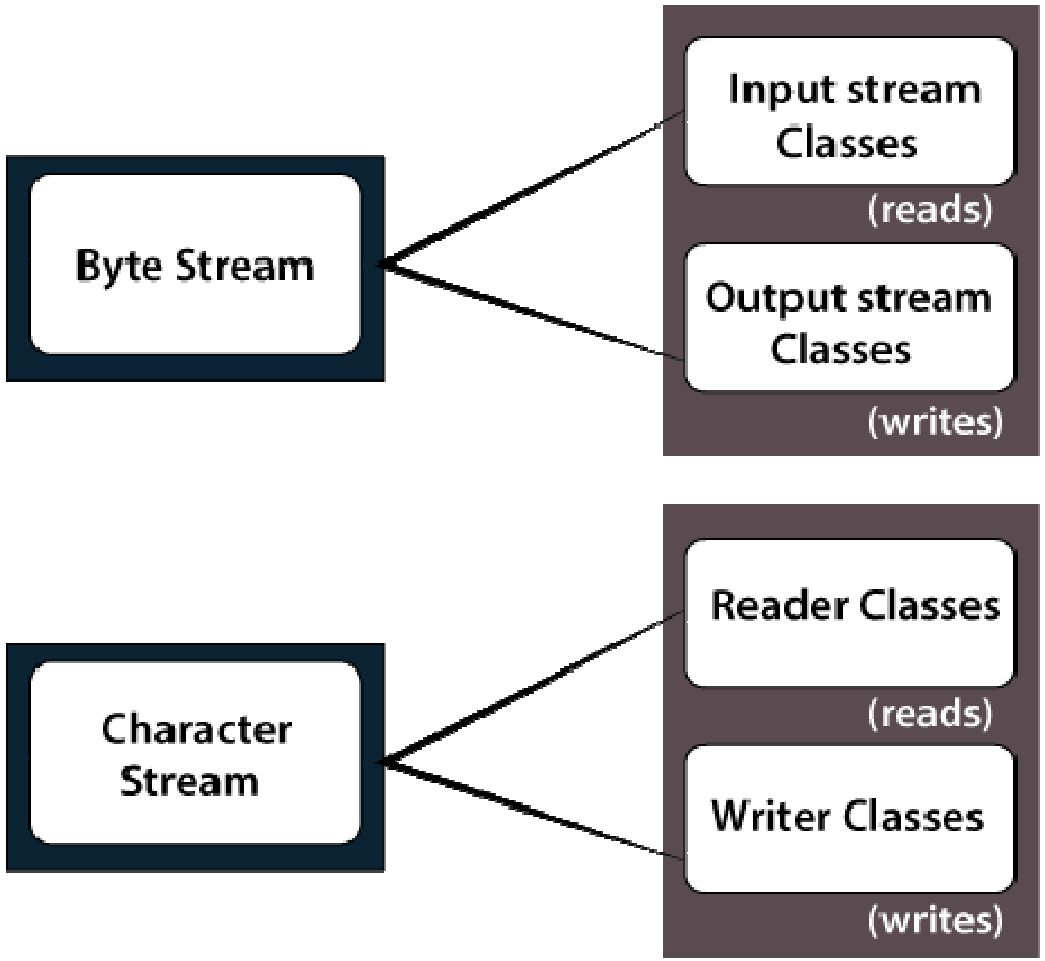
Categories of stream

- File streams can be used to input and output data as **bytes** or **characters**.

1. **Byte-based streams** output and input data in its *binary* format—a char is two bytes, an int is four bytes, a double is eight bytes, etc.

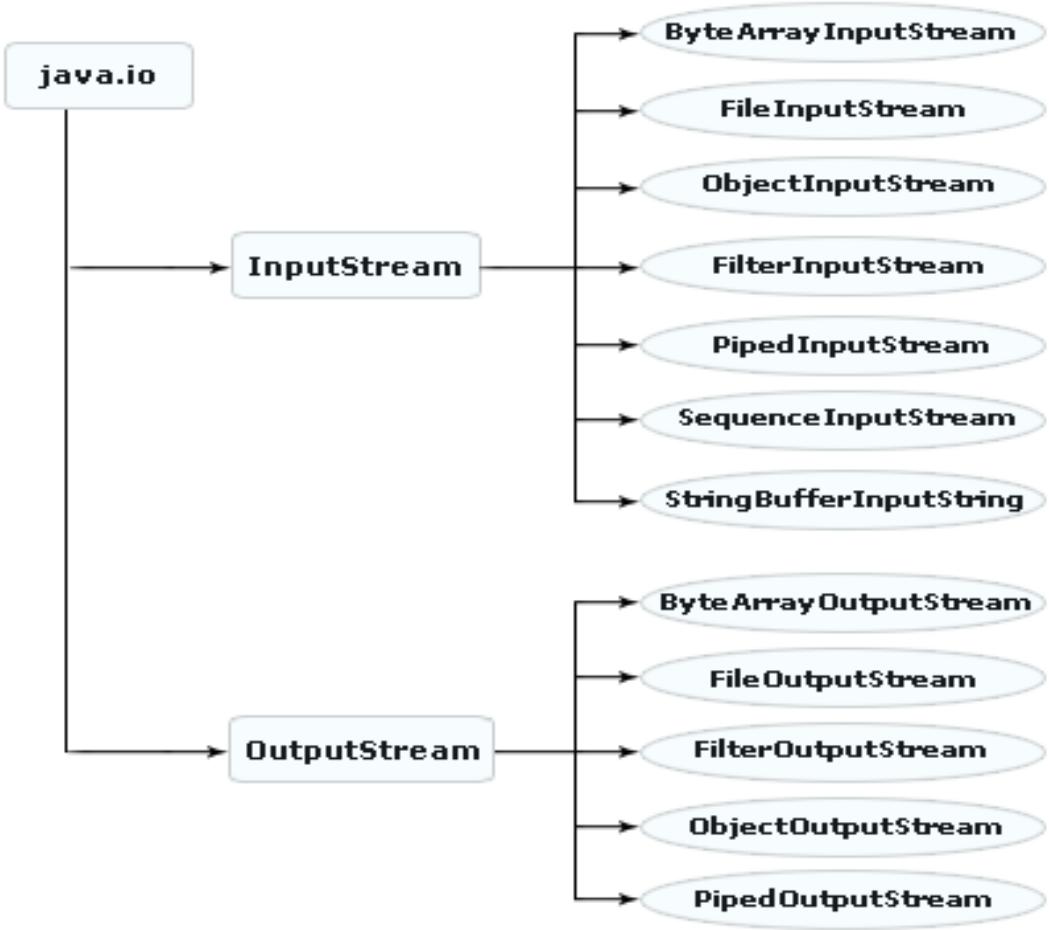
2. **Character-based streams** output and input data as a *sequence of characters* in which every character is two bytes—the number of bytes for a given value depends on the number of characters in that value. For example, the value 2000000000 requires 20 bytes (10 characters at two bytes per character) but the value 7 requires only two bytes (1 character at two bytes per character).

Categories of stream



Byte Stream

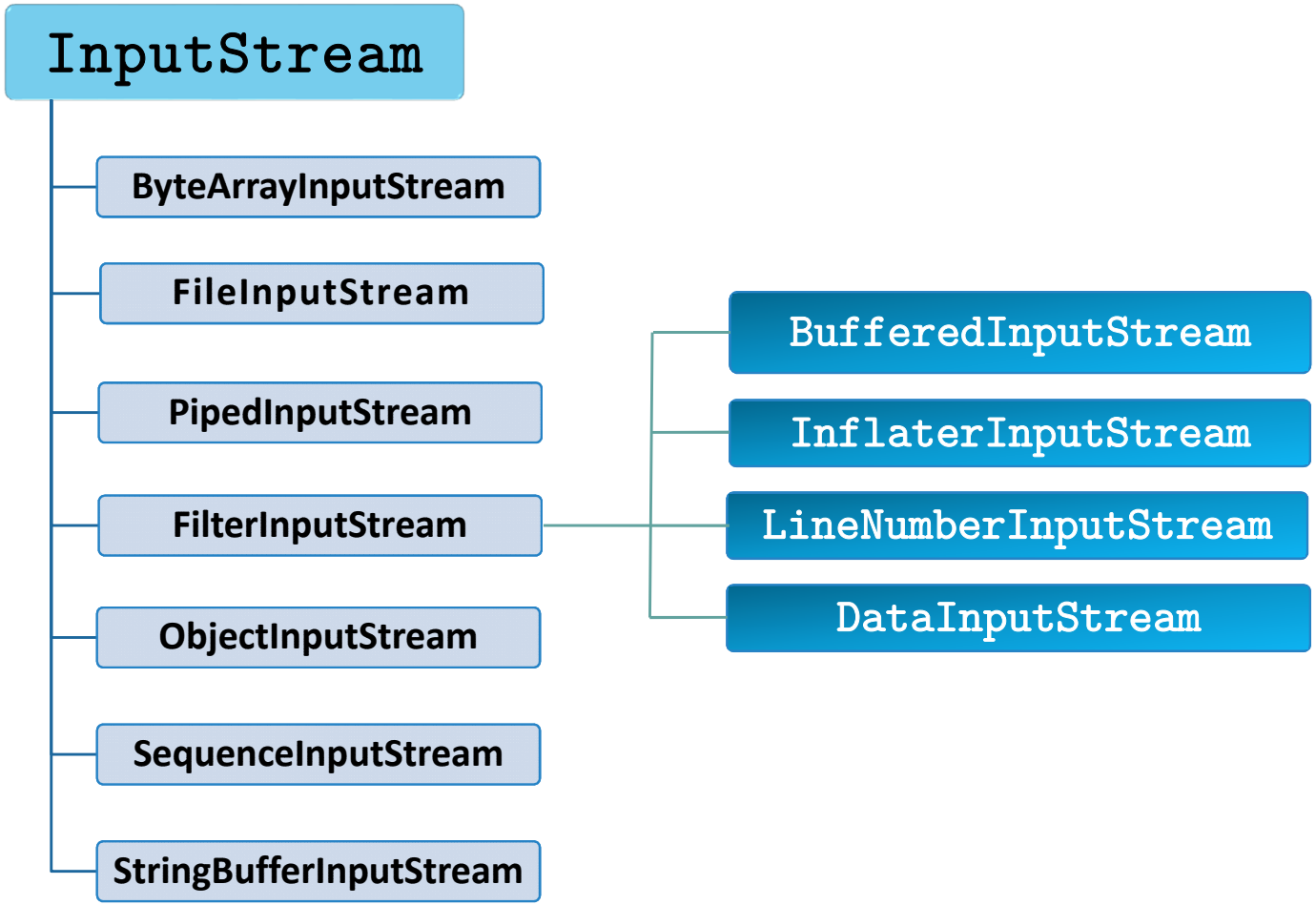
Byte Stream Classes



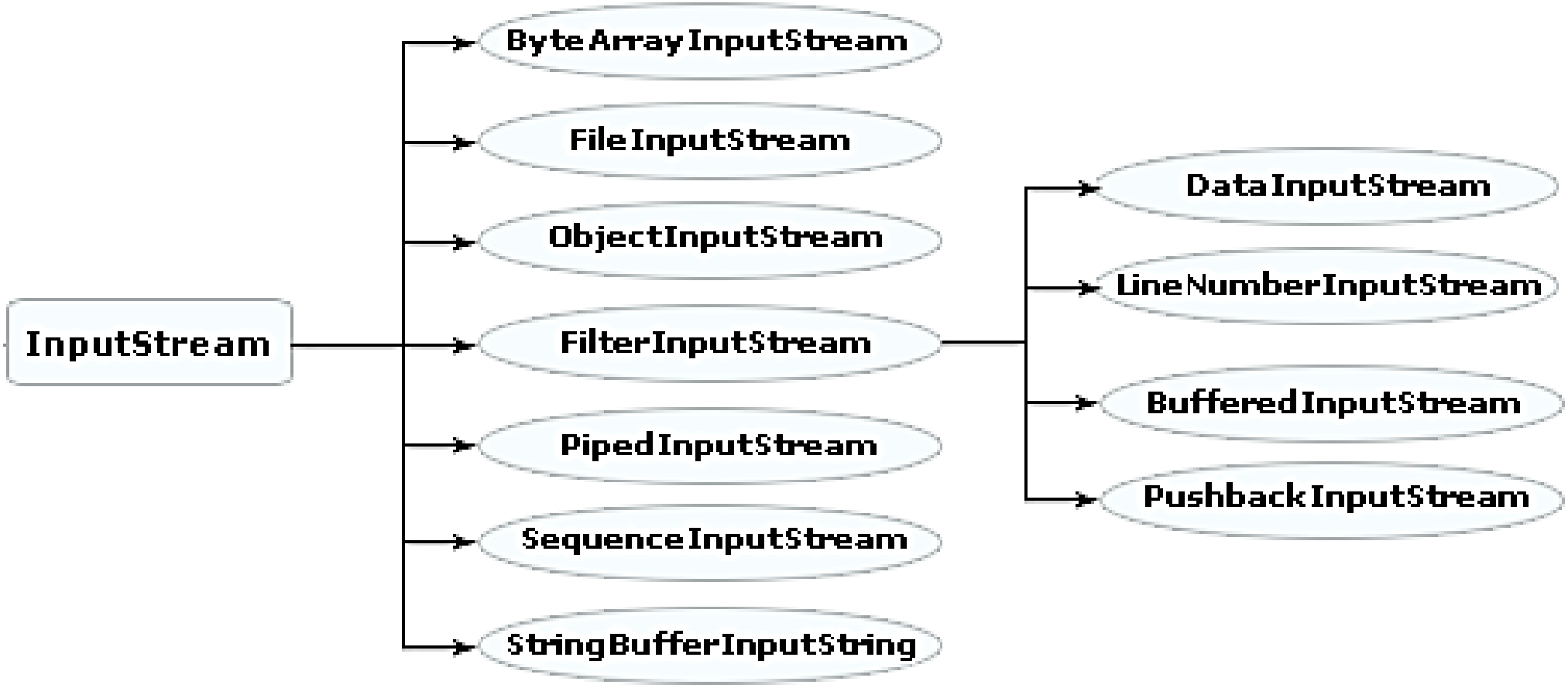
Byte stream

- Byte streams are used to handle any characters(text), images, audio and video files.
- For example, to store an image file (.jpg or .gif), we should go for a byte stream.
- Java byte streams are used to perform **input and output of 8-bits / 1 byte** at a time from **binary file.** The binary files are read by programs that understand the file's specific content and its ordering.
- **InputStream** and **OutputStream** class are most common used classes of byte stream**.**

InputStream (part of Byte Stream)



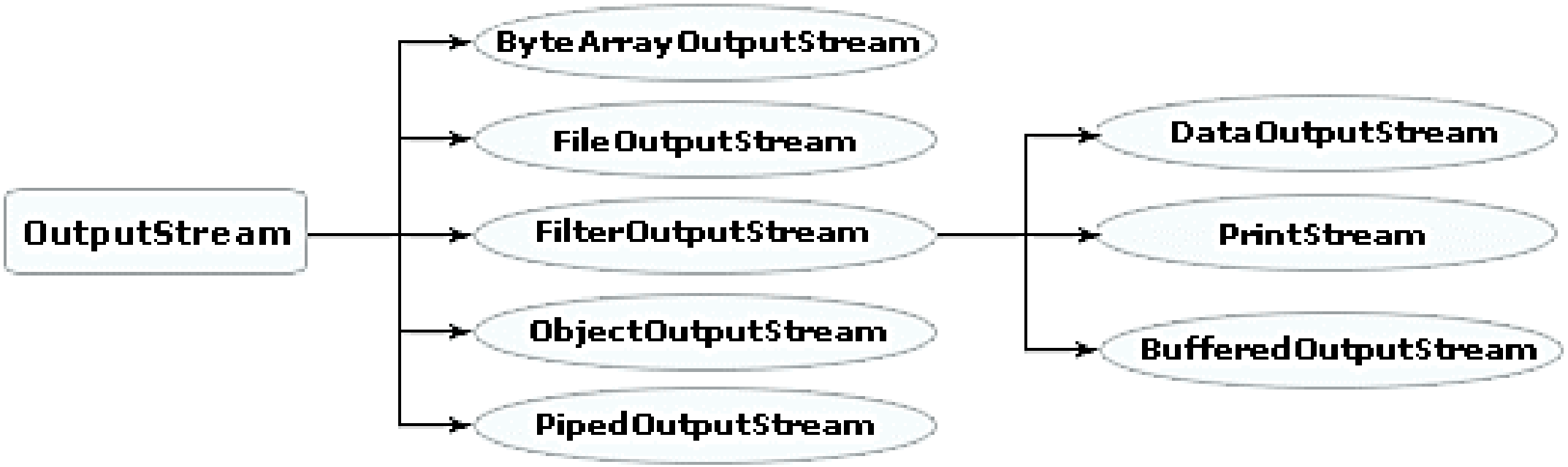
Hierarchy of InputStream



Methods of `InputStream`

Method	Description
read ( )	Returns an integer representation of the next available byte of input. – 1 is returned when the end of the file is encountered.
available( )	Returns the number of bytes of input currently available for reading.
close ( )	Closes the input source. Further read attempts will generate an **IOException**.
mark (int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
marksupported( )	Returns **true** if **mark( )** / **reset( )** are supported by the invoking stream.
reset ( )	Resets the input pointer to the previously set mark.
skip (long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

Hierarchy of `OutputStream` (part of Byte Stream)



Methods of `OutputStream`

Method	Description
write ( )	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with an expression without having to cast it back to **byte**.
flush ( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
close ( )	Closes the input source. Further write attempts will generate an **IOException**.

FileInputStream

FileInputStream

- The **FileInputStream** class creates an input stream that you can use to read bytes from a file. FileInputStream is used for **reading data from the files.**
- It is a subclass of the InputStream class and provides a convenient way to read binary data from a file in a platform-independent way.
- While using FileInputStream class, file must already exist if file does not exist it will generate **error(Exception)**.
- **FileInputStream** overrides six of the methods in the abstract class **InputStream**.

FileInputStream

- There are two way for using file with FileInputStream.

```
FileInputStream f1 = new FileInputStream("C:\\hello.txt");
```

Here, directly File path is specified while creating the FileInputStream object.

```
File f = new File("C:\\java\\hello.txt");
FileInputStream f1 = new FileInputStream(f);
```

In the second method, first object of File is created and then FileInputStream object is created and File object passed to the FileInputStream as a argument.

Methods of `FileInputStream`

Method	Description
FileInputStream (string name)	Creates a new FileInputStream object given the name of the file to be read. (It is constructor.)
read ( )	Reads a byte of data from this input stream. –1 is returned when the end of the file is encountered.
available( )	Returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream.
close ( )	Closes this file input stream and releases any system resources associated with the stream.
skip ()	Skips over and discards n bytes of data from the input stream

Example-1 of FileInputStream

```
import java.io.FileInputStream;
public class FIS_char {
     public static void main(String args[]){
         try{
              FileInputStream fin=new FileInputStream("D:\\test.txt");
             int i = fin.read();   //read single character
             System.out.print((char)i);
             fin.close();
         }
         catch(Exception e){
              System.out.println(e);
         }
     }
}
```

First we have to create a file "test.txt" and then try to read from it.

Example-2 of FileInputStream

```
import java.io.FileInputStream;
public class FIS_example {
     public static void main(String args[]){
         try{
             FileInputStream fin=new FileInputStream("D:\\test.txt");
             int i=0;
             while((i = fin.read())!= -1){          //read upto EOF
                   System.out.print((char)i);
             }
             fin.close();
         }
         catch(Exception e){
              System.out.println(e);
         }
     }
}
```

First we have to create a file "test.txt" and then try to read from it.

FileOutputStream

FileOutputStream

- The **FileOutputStream** class creates an output stream that you can use to write bytes into a file. FileOutputStream is used for **writing data into the files.**
- It is a subclass of the OutputStream class and provides a convenient way to write binary data into a file in a platform-independent way.
- If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter. But you can write byte-oriented as well as character-oriented data. If you attempt to open a read-only file, an exception will be thrown.

FileOutputStream

- There are two way for using file with **FileOutputStream**.

```
FileOutputStream f = new FileOutputStream("C:/hello.txt");
```

Here, directly File path is specified while creating the FileOutputStream object.

```
File f = new File("C:/java/hello.txt");
FileOutputStream f1 = new FileOutputStream(f);
```

In the second method, first object of File is created and then FileOutputStream object is created and File object passed to the FileOutputStream as a argument.

Methods of `FileOutputStream`

Method	Description
FileOutputStream (string name)	Creates a new FileOutputStream object given the name of the file to be written. (It is constructor.)
write ( )	Writes a byte of data to the file. –1 is returned when the end of the file is encountered.
flush ( )	Flushes the output stream, forcing any buffered data to be written to the file.
close ( )	Closes this file output stream and releases any resources associated with it.

Example-1 of FileOutputStream

```
import java.io.FileOutputStream;
public class FOSExample {
    public static void main(String args[]){
        try{
            FileOutputStream fout = new FileOutputStream("D:\\test.txt");
            fout.write(65);    //ASCII of A is 65
            fout.close();
            System.out.println("Write Successful.");
            }
            catch(Exception e){
                System.out.println(e);
            }
    }
}
```

Here, the file “test.txt” is created in D: drive and A is written into the file.

Example-2 of FileOutputStream

```
import java.io.FileOutputStream;
public class FOS_String {
    public static void main(String args[]){
        try{
            FileOutputStream fout = new FileOutputStream("D:\\test.txt");
            String S = "Welcome to Java Programming";
            byte b[] = S.getBytes();//converting string into byte array
            fout.write(b);
            fout.close();
            System.out.println("Write Successful.");
            }
            catch(Exception e){
                System.out.println(e);
            }
    }
}
```

Here, the file “test.txt” is created in D: drive and String is written into the file.

Character Stream

Character stream

- Character or text streams can always store and retrieve data in the form of characters (or text) only.
- It means character streams are more suitable for handling text files like the ones we create in Notepad.
- They are not suitable to handle the images, audio or video files.
- A numeric value in a binary file(byte stream) can be used in calculations, whereas the character 5 is simply a character that can be used in a string of text, as in "I have 5 fruits".

Character stream classes

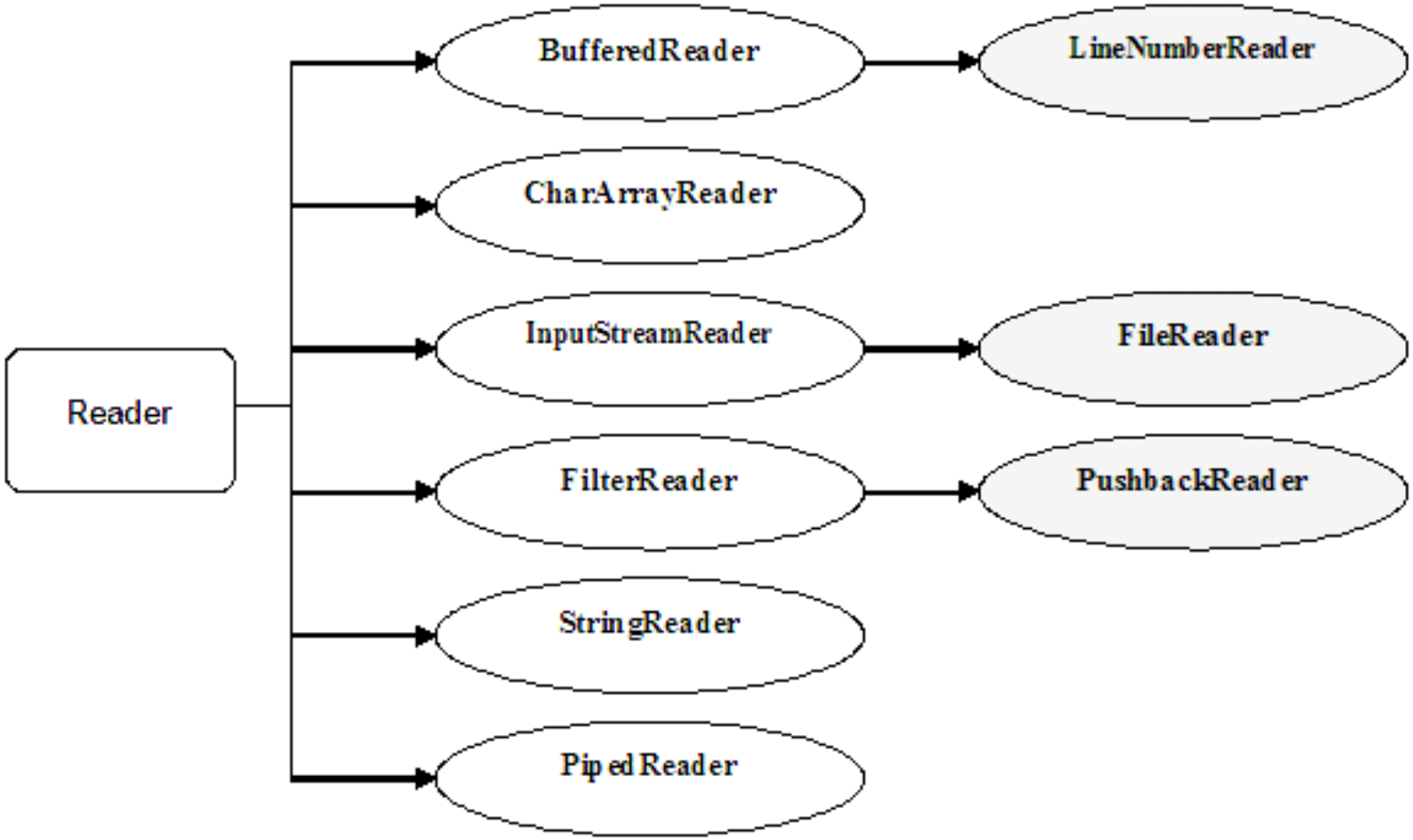
- Java offers another type of streams called Character Streams, which are used to read from the input device and write to the output device in units of 16-bit (Unicode) characters. In some cases, character streams are more efficient than byte streams.
- Character streams are defined by using two class hierarchies. At the top there are two abstract classes, **Reader** and **Writer**. These abstract classes handle **Unicode character** (16-bit) streams. Java has several concrete subclasses of each of these.
- Two of the most important methods are `read`() and `write`(), which read and write characters of data, respectively. These methods are overridden by derived stream classes.

Reader class

`Reader` class

- The **Reader** class contains methods that are identical to those available in the InputStream class, except Reader is designed to handle characters. Therefore, Reader classes can perform all the functions implemented by the InputStream classes.
- Java **Reader** is an abstract class for reading character streams. The only methods that a subclass must implement are `read`(char[], int, int) and `close`(). Most subclasses override some of the methods to provide higher efficiency, additional functionality, or both.

Hierarchy of `Reader` Class



Methods of Reader

Method	Description
Reader ( )	It creates a new character-stream reader whose critical sections will synchronize on the reader itself. (It is constructor.)
read ( )	It reads a single character.
read(char[] cbuf)	It reads characters into an array.
mark ( )	It marks the present position in the stream.
markSupported()	It tells whether this stream supports the mark() operation.
close ( )	It closes the stream and releases any system resources associated with it.
skip ()	It skips characters.

Example of Reader

```
import java.io.*;
class ReaderDemo{
    public static void main(String args[]){
        try {
            Reader R = new FileReader("D:\\temp\\test.txt");
            int data = R.read();
            while (data != -1) {
                System.out.print((char)data);
                data = R.read();
            }
            R.close();
        }
        catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
```

FileReader class

Methods of `FileReader` class

Method	Description
FileReader (String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.(It is constructor.)
read ( )	It is used to return a character in ASCII form. It returns -1 at the end of file.
close ( )	It is used to close the FileReader class.

Example of FileReader

```
import java.io.*;
public class FileReaderDemo {
    public static void main(String args[]) {
        try {
          FileReader fr = new FileReader("D:\\temp\\test2.txt");
          int i;
          while((i = fr.read())!=-1)
             System.out.print((char)i);
          fr.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

BufferedReader class

`BufferedReader` class

- Java **`BufferedReader`** class is used to read the text from a character-based input stream.
- It makes the performance faster.
- It inherits Reader class.
- Declaration:

```
public class BufferedReader extends Reader
```

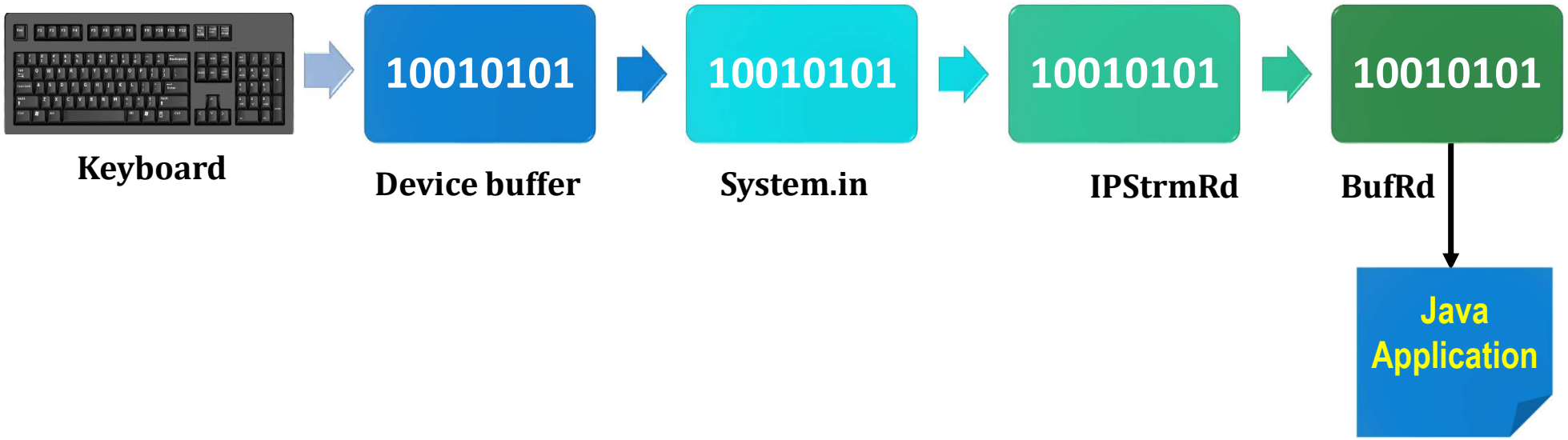
Methods of `BufferedReader` class

Method	Description
BufferedReader (Reader Rd)	It is used to create a buffered character input stream that uses the default size for an input buffer. (It is constructor.)
read ( )	It is used for reading a single character.
int read (char[] buf, int off, int len)	It is used for reading characters into a portion of an array.
String readLine()	It is used for reading a line of text.
long skip(long n)	It is used for skipping the characters.
close ( )	It closes the input stream and releases any of the system resources associated with the stream.

Example of BufferedReader class

```
import java.io.*;
public class BufferedReaderExample {
    public static void main(String args[]) throws Exception {
        FileReader fr = new FileReader("D:\\testout.txt");
        BufferedReader br = new BufferedReader(fr);
        int i;
        while((i=br.read())!=-1){
            System.out.print((char)i);
        }
        br.close();
        fr.close();
    }
}
```

Reading from console by InputStreamReader & BufferedReader



Read from Keyboard

```
import java.io.*;
public class BufferedReaderConsole{
public static void main(String args[])throws Exception{
    InputStreamReader IpStrmRd = new InputStreamReader(System.in);
    BufferedReader BufRd = new BufferedReader(IpStrmRd);
    System.out.print("Enter your name: ");
    String name = BufRd.readLine();
    System.out.println("Welcome "+name);
  }
}
```

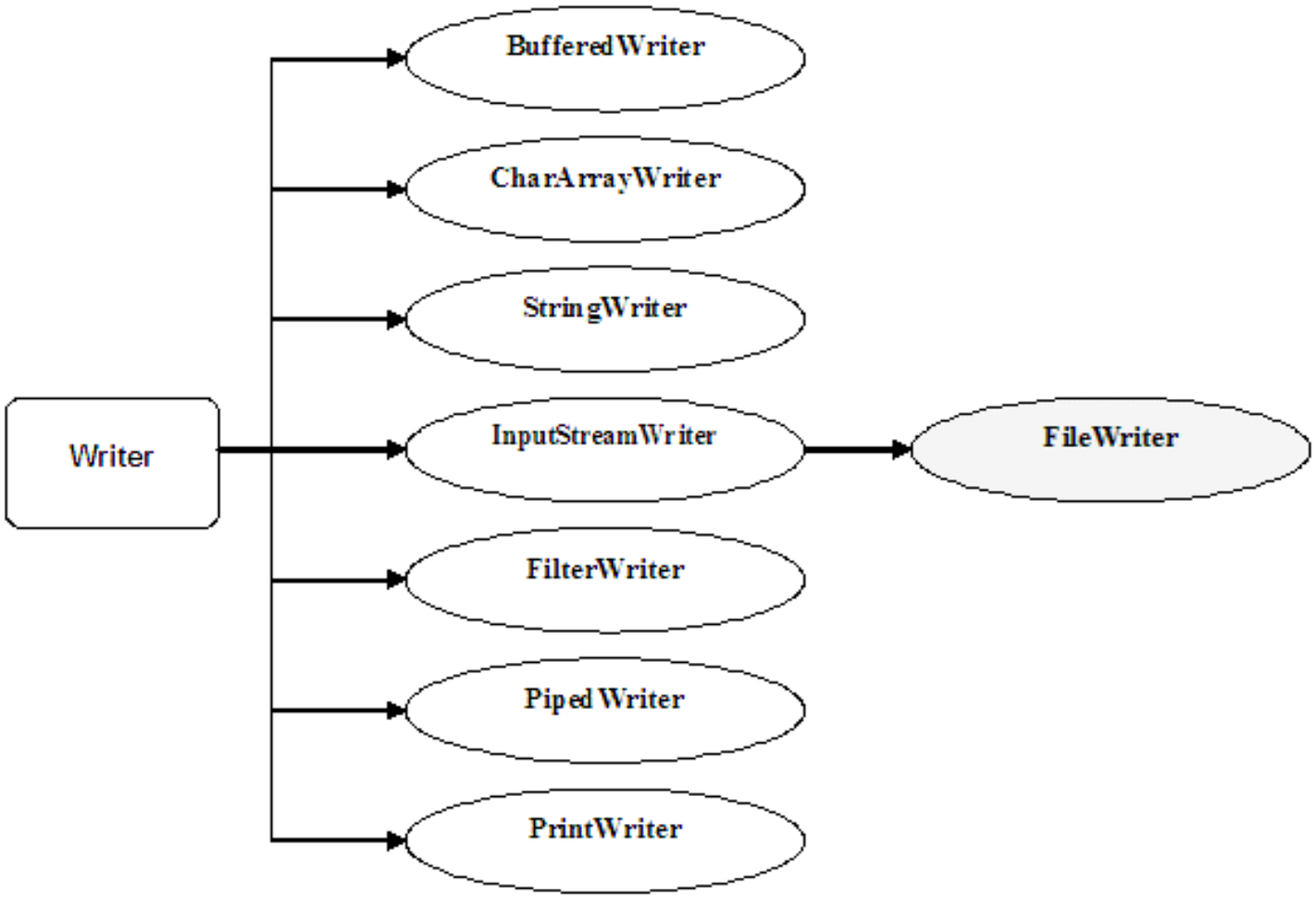
Enter your name: Chintan Gajjar
Welcome Chintan Gajjar

Writer class

Writer class

- The **Writer** class contains methods that are identical to those available in the OutputStream class, except Writer is designed to handle characters. Therefore, Writer classes can perform all the functions implemented by the OutputStream classes.
- Java **Writer** class is an abstract class for writing to character stream. The methods that a subclass must implement are `write`(char[], int, int), flush(), and `close`(). Most subclasses will override some of the methods defined here to provide higher efficiency, functionality or both.

Hierarchy of `Writer` class



Methods of `Writer` class

Method	Description
Writer ( )	It creates a new character-stream writer whose critical sections will synchronize on the writer itself. (It is constructor.)
write (int c)	It writes a single character.
write (char[] buf)	It writes an array of characters.
append(char c)	It appends the specified character to this writer.
append(CharSequence csq)	It appends the specified character sequence to this writer
close ( )	It closes the stream, flushing it first.
flush ()	It flushes the stream.

Example of `Write` class

```
import java.io.*;
public class WriterDemo {
    public static void main(String args[]) {
        try {
            Writer w = new FileWriter("D:/temp/output.txt");
            String data= "I am an Indian.";
            w.write(data);
            w.close();
            System.out.println("File written successfully.");
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

FileWriter class

Methods of `FileWriter` class

Method	Description
FileWriter (String file)	Creates a new file. It gets file name in string.(It is constructor.)
write (String t)	It is used to write the string into FileWriter.
write (char[] c)	It is used to write char array into FileWriter.
write (char c)	It is used to write the char into FileWriter.
close ( )	It is used to close the FileWriter.
flush ()	It is used to flushes the data of FileWriter.

Example of `FileWriter` class

```
import java.io.*;
public class FileWriterDemo {
    public static void main(String args[]){
        try{
            FileWriter fw = new FileWriter("D:\\TESTfile.txt");
            fw.write("Welcome to Java Programming.");
            fw.close();
            System.out.println("Write operation Successful");
        }
        catch(Exception e){
             System.out.println(e);
        }
    }
}
```

BufferedWriter class

`BufferedWriter` class

- Java **`BufferedWriter`** class is used to provide buffering for Writer instances. It makes the performance faster.
- It inherits Writer class.
- The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.
- Declaration:

```
public class BufferedWriter extends Writer
```

Methods of `BufferedWriter` class

Method	Description
BufferedWriter (Writer wrt)	It is used to create a buffered character output stream that uses the default size for an output buffer.(It is constructor.)
void newLine( )	It is used to add a new line by writing a line separator.
write (int c)	It is used to write a single character.
write (char[] c, int off, int len)	It is used to write a portion of an array of characters.
write (String s, int off, int len)	It is used to write a portion of a string.
close ( )	It is used to closes the input stream
flush ()	It is used to flushes the input stream.

Example of BufferedWriter class

```
import java.io.*;
public class BufferedWriterExample {
public static void main(String[] args) throws Exception {
        FileWriter writer = new FileWriter("D:\\testout.txt");
        BufferedWriter buffer = new BufferedWriter(writer);
        buffer.write("Welcome to Java programming.");
        buffer.close();
        System.out.println("Buffered Write Successful.");
    }
}
```

File class

File Class

- Although most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system.
- That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself.
- A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.
- A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.

File class

- The File class from the java.io package, allows us to work with files.
- The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.
- To use the File class, create an object of the class, and specify the filename or directory name:

```
import java.io.File;
File myObj = new File("filename.txt");
```

Methods of `File` class

- The File class has many useful methods for creating and getting information about files.

Method	Description
`createNewFile()`	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
`getName()`	It returns the name of the file or directory denoted by this abstract pathname.
`getParent()`	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
`isFile()`	It tests whether the file denoted by this abstract pathname is a normal file.
`isDirectory()`	It tests whether the file denoted by this abstract pathname is a directory.

Methods of `File` class

Method	Description
`canWrite()`	It tests whether the application can modify the file denoted by this abstract pathname.
`canRead()`	It tests whether the application can read the file denoted by this abstract pathname.
`canExecute()`	It tests whether the application can execute the file denoted by this abstract pathname.
`getName()`	It returns the name of the file or directory denoted by this abstract pathname.
`getParent()`	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
`mkdir()`	It creates the directory named by this abstract pathname.

Example-1 of `File` class method

```
import java.io.File;
import java.io.IOException;
public class FileCreate {
    public static void main(String args[]) {
          try {
            File fp = new File("JavaFile.txt");
            if(fp.createNewFile())
                System.out.println("New File is created!");
            else System.out.println("File already exists.");
        }
        catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example-2 of `File` class methods

```
import java.io.File;
class FileClassDemo {
    public static void main(String args[]) {
      File f1 = new File("D:/temp/test.txt");
      System.out.println("File Name: " + f1.getName());
      System.out.println("Path: " + f1.getPath());
      System.out.println("Abs Path: " + f1.getAbsolutePath());
      System.out.println("Parent: " + f1.getParent());
      System.out.println(f1.exists() ? "File exists" : "File does not exist");
      System.out.println(f1.canWrite() ? "is writeable" : "is not writeable");
      System.out.println(f1.canRead() ? "is readable" : "is not readable");
      System.out.println("is "+(f1.isDirectory()?"":"not" + " a directory"));
      System.out.println(f1.isFile()?"is normal file":"might be a named pipe");
      System.out.println("File size: " + f1.length() + " Bytes");
    }
}
```

Output

```
File Name: test.txt
Path: \test.txt
Abs Path: C:\test.txt
Parent: \
File does not exist
is not writeable
is not readable
is not a directory
might be a named pipe
File size: 0 Bytes
```

```
File Name: test.txt
Path: D:\temp\test.txt
Abs Path: D:\temp\test.txt
Parent: D:\temp
File exists
is writeable
is readable
is not a directory
is normal file
File size: 95 Bytes
```

Directories

- A directory is a **File** that contains a list of other files and directories. When you create a **File** object that is a directory, the **isDirectory( )** method will return **true**.

- In this case, we can call **list( )** on that object to extract the list of other files and directories inside.

```
String[ ] list( )
```

- The list of files is returned in an array of **String** objects.

Example of Directory

```
import java.io.File;
class Directory {
   public static void main(String args[]) {
      String dirname = "D:/temp";
      File f1 = new File(dirname);
      if (f1.isDirectory()) {
          System.out.println("Directory of " + dirname);
          String S[] = f1.list();
          for (int i=0; i < S.length; i++) {
              File f = new File(dirname + "/" + S[i]);
              if (f.isDirectory())
                  System.out.println(S[i] + " is a directory");
              else System.out.println(S[i] + " is a file");
              }
          }
      else System.out.println(dirname + " is not a directory");
   }
}
```

```
Directory of D:/temp
Hello.txt is a file
Hello1.txt is a file
test.txt is a file
testdir is a directory
testout.txt is a file
```