

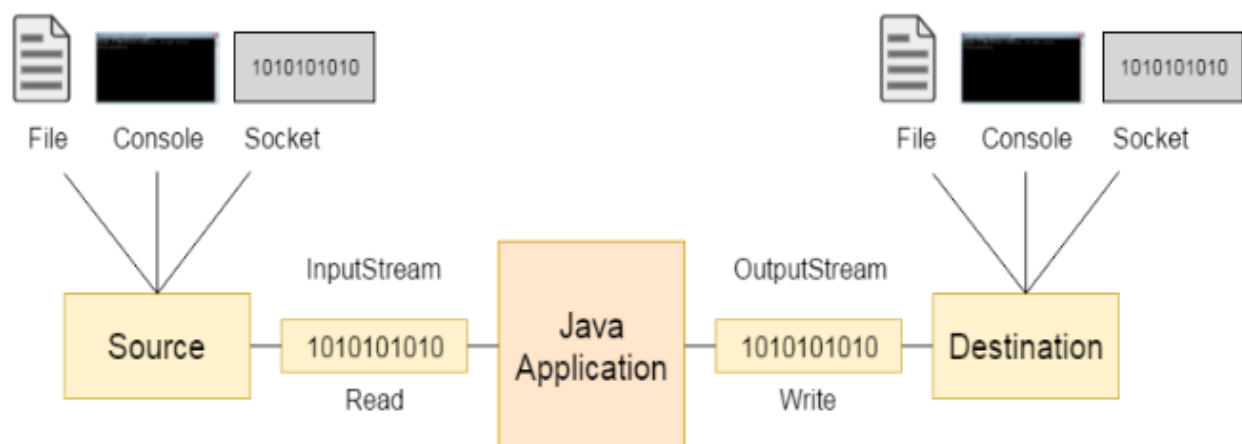
CHAPTER-11

EXPLORING JAVA.IO.*;

Introduction

A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of data flow from one place to another place. It is just like a water-pipe where water flows. Like a water-pipe carries water from one place to another, a stream carries data from one place to another place. A stream can carry data from keyboard to memory or from memory to monitor/printer or from memory to a file. A stream is always required if we want to move data from one place to another.

Basically, there are two types of streams: input streams and output streams. Input streams are those streams which receive or read data coming from some other place (keyboard/file/Network socket). Output streams are those streams which send or write data to some other place (monitor/printer/file/Network socket).



All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same

I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

All streams are represented by classes in **java.io** (input and output) package. This package contains a lot of classes, all of which can be classified into two categories: input streams and output streams.

Now, let us see how a stream works. We know an input stream reads data. So, to read data from keyboard, we can attach the keyboard to an input stream, so that the stream can read the data typed on the keyboard. Keyboard is represented by a field, called in **System** class. When we write **System.in**, we are representing a standard input device, i.e. keyboard, by default. System class is found in **java.lang** (language) package and has three fields. These three streams are created automatically for us as shown below. All these fields represent some type of stream:

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 standard output device, i.e. monitor.

Note that both **System.out** and **System.err** can be used to represent the monitor and hence any of these two can be used to send data to the monitor. **System.out** is used to display normal messages and results, whereas **System.err** is used to display error messages.

So, in this way streams represent I/O devices in Java. Even if we change the keyboard or monitor, we can still use the same streams to handle those devices.

Here are some basic points about I/O:

- Data in files on your system is called persistent data because it persists after the program runs.
- Files are created through streams in Java code.
- A stream is a linear, sequential flow of bytes of input or output data.
- Streams are written to the file system to create files.
- Streams can also be transferred over the Internet.
- Three standard streams are created for us automatically:
- **System.out** - standard output stream. (Default device is Console)
- **System.in** - standard input stream. (Default device is keyboard)
- **System.err** - standard error. (Default device is Console)

Another classification of streams is **byte stream**, **text stream** and **Object Stream**.

The **java.io** package contains a large number of classes that deal with Java input and output. Most of the classes consist of:

- **Byte streams** that are subclasses of **InputStream** or **OutputStream**.
- **Character streams** that are subclasses of **Reader** and **Writer**

The **Reader** and **Writer** classes read and write 16-bit Unicode characters. **InputStream** reads 8-bit bytes, while **OutputStream** writes 8-bit bytes.

As their class name suggests, **ObjectInputStream** and **ObjectOutputStream** transmit entire objects. **ObjectInputStream** reads objects; **ObjectOutputStream** writes objects.

Unicode is an international standard character encoding that is capable of representing most of the world's written languages. In Unicode, two bytes make a character. Using the 16-bit Unicode character streams makes it easier to internationalize your code. As a result, the software is not dependent on one single encoding.

11.1 What to use

There are a number of different questions to consider when dealing with the **java.io** package:

- What is data format: text or binary?
- Do you want random access capability?
- Are you dealing with objects or non-objects?
- What are your sources and sinks for data?
- Do you need to use filtering?

11.1.1 Text file or Binary File

Files can store data either in text format or a binary format. Lets explained each of them with examples for your easy understanding.

Binary files

Most of the files that we see in our computer system are called binary files.

Example:

- **Document files:** .pdf, .doc, .xls etc.
- **Image files:** .png, .jpg, .gif, .bmp etc.
- **Video files:** .mp4, .3gp, .mkv, .avi etc.
- **Audio files:** .mp3, .wav, .mka, .aac etc.
- **Database files:** .mdb, .accde, .frm, .sqlite etc.
- **Archive files:** .zip, .rar, .iso, .7z etc.
- **Executable files:** .exe, .dll, .class etc.

All binary files follow a specific format. We can open some binary files in the normal text editor but we can't read the content present inside the file. That's because all the binary files will be encoded in the binary format, which can be understood only by a computer or machine.

For handling such binary files we need a specific type of software to open it.

For Example, You need Microsoft word software to open .doc binary files. Likewise, you need a pdf reader software to open .pdf binary files and you need a photo editor software to read the image files and so on.

Advantage of binary files is that they are more efficient to process than text files.

Text files

Text files don't have any specific encoding and it can be opened in normal text editor itself.

For example, Java source programs are stored in text files and can be read by a text editor but Java classes are stored in binary files and are read by the JVM.

Example:

- **Web standards:** html, XML, CSS, JSON etc.
- **Source code:** c, app, js, py, java etc.
- **Documents:** txt, tex, RTF etc.
- **Tabular data:** csv, tsv etc.
- **Configuration:** ini, cfg, reg etc.

If we use binary data, such as integers or doubles, then use the **InputStream** and **OutputStream** classes. If you are using text data, then the **Reader** and **Writer** classes are right.

11.1.2 Random access

Do you want random access to records? Random access allows you to go anywhere within a file and be able to treat the file as if it were a collection of records.

The **RandomAccessFile** class permits random access. The data is stored in binary format.

Using random access files improves performance and efficiency.

11.1.3 Object or non-object

Are you inputting or outputting the attributes of an object? If the data itself is an object, then use the **ObjectInputStream** and **ObjectOutputStream** classes.

11.1.4 Sources and sinks for data

What is the source of your data? What will be consuming your output data, that is, acting as a sink? You can input or output your data in a number of ways: sockets, files, strings, and arrays of characters.

Any of these can be a source for an **InputStream** or **Reader** or a sink for an **OutputStream** or **Writer**.

11.1.5 Filtering

Do you need filtering for your data? There are a couple ways to filter data.

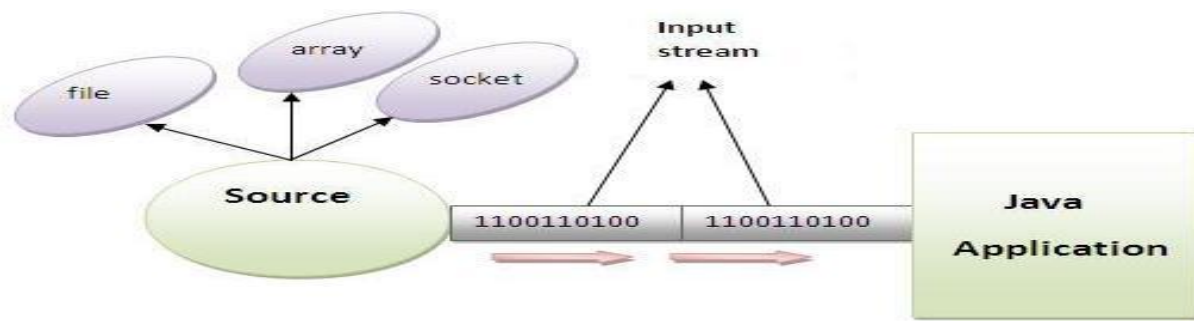
Buffering is one filtering method. Instead of going back to the operating system for each byte, you can use an object to provide a buffer.

Checksumming is another filtering method. As you are reading or writing a stream, you might want to compute a checksum on it. A checksum is a value you can use later on to make sure the stream was transmitted properly.

11.2 Basic Streams

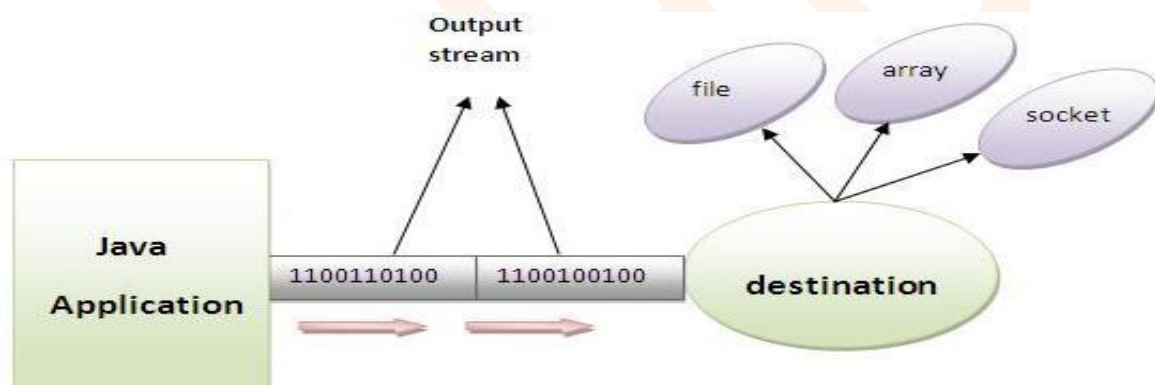
InputStream/Reader

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.



OutputStream/Writer

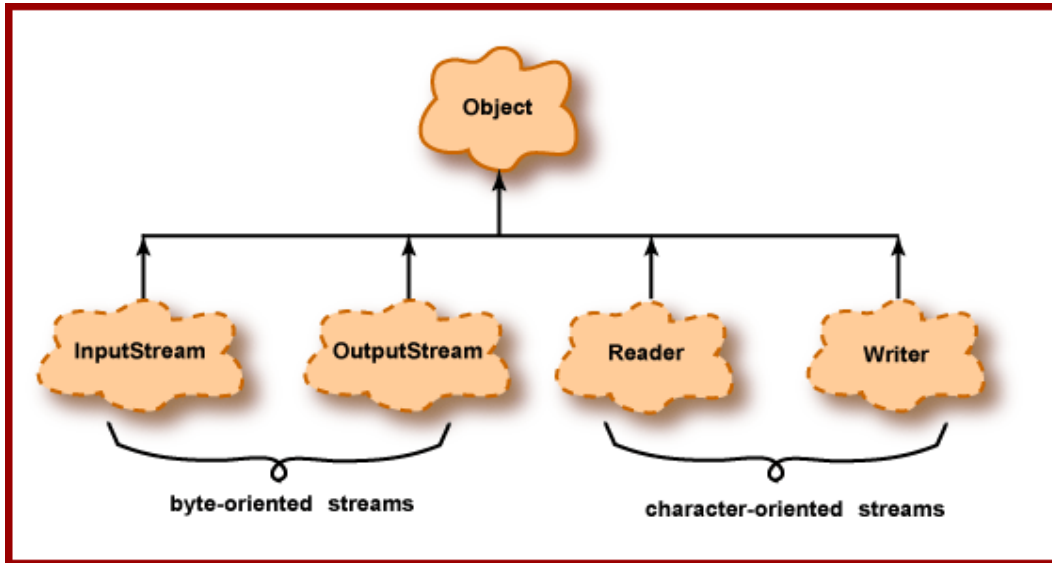
Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.



All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection.

11.3 I/O Stream Class Hierarchy

Basic I/O Hierarchy



The diagram shows the top of the hierarchy for the java.io package. The dotted clouds are *abstract classes*. They act as base classes for specialized streams (to be discussed shortly).

Streams are byte-oriented or character-oriented. Each type has input streams and output streams.

1. Byte-oriented streams.
 - Intended for general-purpose input and output.
 - Data may be primitive data types or raw bytes.
2. Character-oriented (Text Stream) streams.
 - Intended for character data.
 - Data is transformed from/to 16 bit Java char used inside programs to the UTF format used externally.

UTF stands for "Unicode Transformation Format". Usually a UTF text file is identical to an ASCII text file. ASCII is the standard way to represent characters that most computers have used for the past forty years. A file created with a text editor is usually an ASCII text file.

A UTF text file can include non-ASCII characters such as Cyrillic, Greek, and Asian characters. By reading and writing UTF files, Java programs can process text from any of the World's languages.

11.4 Byte Stream Class Hierarchy

It handle data in the form of bits and bytes. Byte streams are used to handle any characters (text), images, audio and video files. For example, to store an image file (.gif or .jpg), we should go for a byte stream.

Byte streams are defined within two class hierarchies, one for input and one for output and represent byte stream classes which provide the tools to read and write binary data as a sequence of *bytes*.

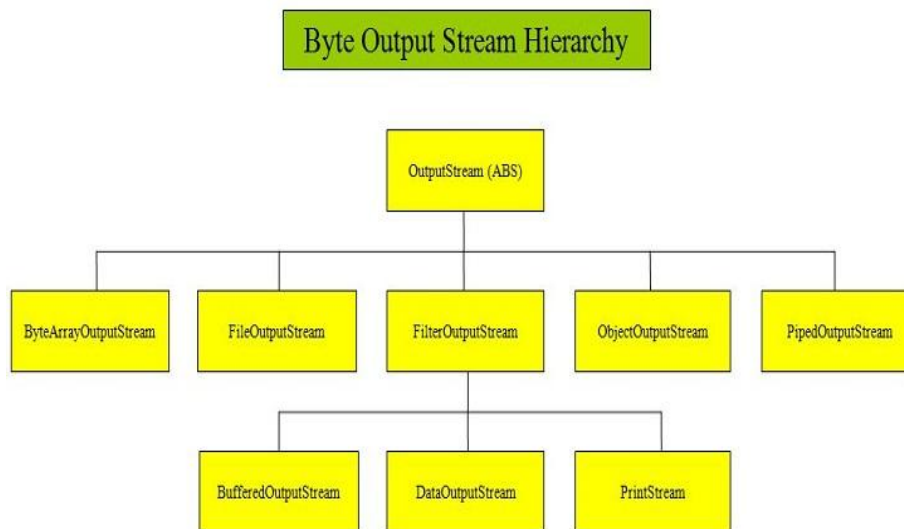
- The **OutputStream** class is the *abstract superclass* of all byte output streams
- The **InputStream** class is the *abstract superclass* of all byte input streams

These classes define the characteristics that are common to byte input and byte output streams, which are implemented in the concrete subclasses of each hierarchy.

11.4.1 Byte Output Stream Hierarchy

The diagram below shows most of the classes in the *byte output stream hierarchy* of which **OutputStream** class is the *abstract*

superclass. Some subclasses of the **FilterOutputStream** class are not shown.



Class	Description
<u>OutputStream</u>	Abstract byte stream superclass which describes this type of output stream.
ByteArrayOutputStream	Output byte stream that writes data to a byte array.
<u>FileOutputStream</u>	Output byte stream that writes bytes to a file in a file system.
FilterOutputStream	Output byte stream that implements OutputStream.
BufferedOutputStream	Output byte stream that writes bytes to a buffered output stream.
<u>DataOutputStream</u>	Output stream to write Java primitive data types.
PrintStream	Convenience output byte stream to add functionality to another stream, an

Class	Description
	example being to print to the console using <code>print()</code> and <code>println()</code> .
<u>ObjectOutputStream</u>	Output stream to write and serialize objects for reading using <code>ObjectInputStream</code> .

Now we have got an overview of the class hierarchies involved in Java I/O its time to investigate these classes in more detail.

Now, we take a closer look at some of the *byte stream classes* that are available for our use in the **java.io** package and how we use them. The *byte stream classes* are the original input and output streams which were shipped with JDK 1.0. This is the reason why **System.in**, **System.out** and **System.err** use the *InputStream* and *PrintStream* types from the *byte stream classes* and not the *character stream classes* as you would expect.

The java.io.OutputStream Class

At the top of the *byte output stream hierarchy* is the **java.io.OutputStream** *abstract superclass* which defines a basic set of output functions that all *byte output stream classes* have. These methods allow us to flush and write bytes within the *byte output stream* as well as close the stream when we finish our processing. All methods within the **java.io.OutputStream** class can *throw* an **IOException**.

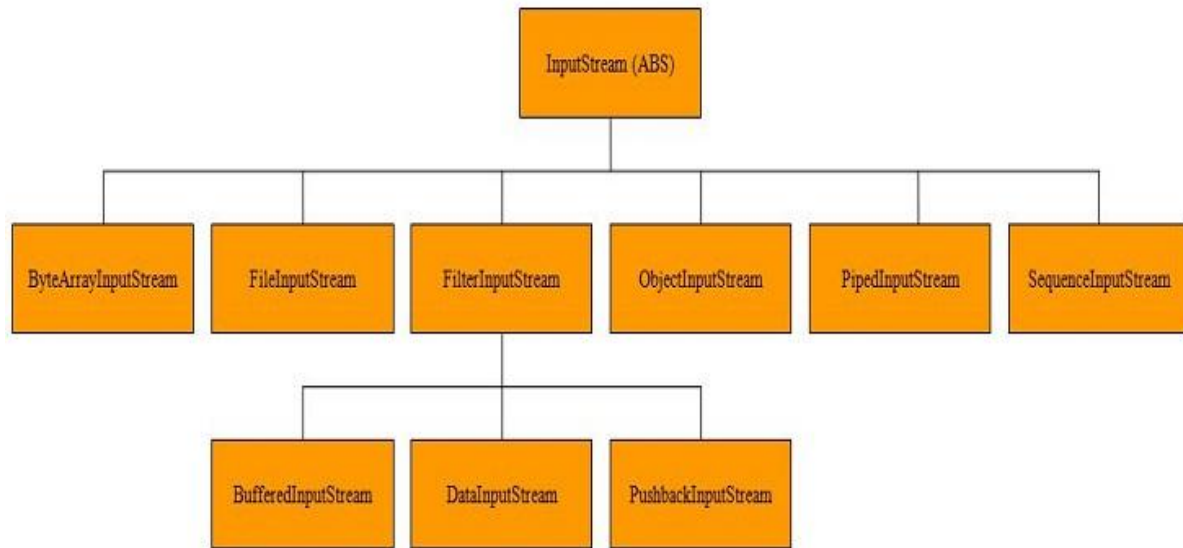
The table below shows the declarations of all the methods in the **java.io.OutputStream** class:

Method Declaration	Description
public void close()	Closes output stream and release any system resources associated with it.
public void flush()	Flushes output stream and forces any buffered output bytes to be written out.
public abstract void write(int b)	Writes a single byte of data specified by b to the output stream.
<u>public void write(bytes[] b)</u>	Writes b bytes of data from b array.
public void write(bytes[] b, int off, int len)	Writes len bytes of data from b array starting from off.

11.4.2 Byte Input Stream Hierarchy

The diagram below shows most of the classes in the *byte input stream hierarchy* of which InputStream class is the *abstract superclass*. Some subclasses of the FilterInputStream class are not shown.

Byte Input Stream Hierarchy



Class	Description
<u>InputStream</u>	Abstract byte stream superclass which describes this type of input stream.
ByteArrayInputStream	Input byte stream that reads bytes from an internal byte array.
<u>FileInputStream</u>	Input byte stream that reads bytes from a file in a file system.
FilterInputStream	Input byte stream that implements InputStream.
BufferedInputStream	Input byte stream that reads bytes into an internal buffer before use.

Class	Description
<u>DataInputStream</u>	Input stream to reads Java primitive data types.
PushbackInputStream	Input byte stream containing functionality to return bytes to the input stream.
<u>ObjectInputStream</u>	Input stream to read and deserialize objects output and serialized using ObjectOutputStream.
SequenceInputStream	Concatenation of two or more input streams read sequentially.

InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Commonly used methods of **InputStream** class

Method	Description
1) public abstract int read()throws IOException:	reads the next byte of data from the input stream.It returns -1 at the end of file.
2) public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException:	is used to close the current input stream.

11.4.3 FileInputStream and FileOutputStream (File Handling)

FileInputStream and **FileOutputStream** classes are used to read and write data in file. In another words, they are used for file handling in java.

FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

FileOutputStream class:

A *FileOutputStream* is an output stream for writing data to a file.

If you have to write primitive values then use **FileOutputStream**.

Program 1: Write a program to read data from the keyboard and write it to a text file using byte stream classes.

```
//Creating a text file using byte stream classes

import java.io.*;
class FileHandlingExample1
{ public static void main(String args[]) throws IOException
{ //attach keyboard to DataInputStream
  DataInputStream dis = new DataInputStream (System.in);
  //attach the file to FileOutputStream
  FileOutputStream fout = new FileOutputStream
("myfile.txt");
  //read data from DataInputStream and write into
  FileOutputStream
```



```

    char ch;
    System.out.println ("Enter @ at end : " );
    while( (ch = (char) dis.read() ) != '@' )
        fout.write (ch);
    fout.close ();
}
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Create1.java
D:\JQR>java Create1
Enter @ at end :
I am writing first line
I am writing second line
@
D:\JQR>

```

In this program, we read data from the keyboard and write it to *myfile.txt* file. This program accept data from the keyboard till the user types @ when he does not want to continue.

Here, we stored three lines of text into *myfile.txt*. To view the contents of *myfile.txt*, we can use any text editor program.

If program1 is run again, the old data of *myfile.txt* will be lost and any recent data is only stored into the file. If we do not want to lose the previous data of the file, and just append the new data at the end of already existing data, then we should open the file by writing *true* along with the filename as:

Takes

```

FileOutputStream fout = new  
FileOutputStream("myfile.txt", true);

```

When the above statement is used – even though the program is run several times – all previous data will be preserved and new data will be added to the old data.

How to Improve the Efficiency of Program1 using BufferedOutputStream?

Let us estimate how much time it takes to read 100 characters from the keyboard and write all of them into a file. Let us assume that we read data from the keyboard into memory using **DataInputStream** and it takes 1 second to read 1 character into memory. Let us assume that this character is written into the file by **FileOutputStream** by spending another 1 second. So, for reading and writing a single character, a total of 2 seconds are used. Thus, to read and write 100 characters, it takes 200 seconds. This is wasting a lot of time.

On the other hand, if Buffered classes are used, they provide a buffer (temporary block of memory), which is first filled with characters and then all the characters from the buffer can be at once written into the file. Buffered classes should be used always in connection to other stream classes. For example, **BufferedOutputStream** can be used along with **FileOutputStream** to write data into a file.

In Program1, we can attaché FileOutputStream to BuffereOutputStream as

```
FileOutputStream fout = new  
FileOutputStream("myfile.txt", true);
```

```
BuffereOutputStream bout = new  
BufferedOutputStream(fout,1024)
```

Here, the buffer size is declared as 1024 bytes. If the buffer size is not specified, then a default buffer size of 512 bytes is used.

Program 2: Write a program to improve the efficiency of writing data into a file using BufferedOutputStream.

```
//Creating a text file using byte stream classes
import java.io.*;
public class BufferedOutputStreamEx
{
    public static void main(String args[]) throws
IOException
    {
        //attached keyboard to DataInputStream
        DataInputStream dis = new
DataInputStream(System.in);
        //attach myfile to FileOutputStream
        FileOutputStream fout = new
FileOutputStream("myfile.txt", true);
        //attached FileOutputStream to
BufferedOutputStream
        BufferedOutputStream bout = new
BufferedOutputStream(fout,1024);
        System.out.println("Enter text (@ at the end:");
        char ch;
        while((ch = (char) dis.read())!='@')
            bout.write(ch);
        bout.close();// close the file
    }
}
```

OUTPUT:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Create2.java
D:\JQR>java Create2
Enter @ at end :
This is new line in my file
@
D:\JQR>type myfile
I am writing first line
I am writing second line
This is new line in my file
D:\JQR>
```

Program 3: Write a program to read data from the keyboard and write it to a text file and read data back from that text file using byte stream classes.

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
class FileHandlingExample2
{ public static void main(String args[]) throws IOException
{ //attach keyboard to DataInputStream
  DataInputStream dis = new DataInputStream (System.in);
  //attach the file to FileOutputStream
  FileOutputStream fout = new FileOutputStream ("myfile");
  //read data from DataInputStream and write into
  FileOutputStream
  char ch;
  System.out.println ("Enter @ at end : " );
  while( (ch = (char) dis.read() ) != '@' )
    fout.write (ch);

  fout.close ();

  try {
    FileInputStream fis = new FileInputStream("myfile"); //
    Create a stream to read from FileOutputStream.txt
```

```

        System.out.println("FileInputStream fis has been
opened.");
        int i;
        do
        {
            i = fis.read();
            if (i != -1)
            { // The -1 value denotes the end of file condition
                System.out.println("We read in character " + (char)i);
            }
        } while (i != -1);

        fis.close();
    }
    catch (FileNotFoundException ex) {
        System.out.println("Exception opening
FileOutputStream.txt : " + ex);
    }
    catch (IOException ex) {
        System.out.println("We caught exception to fis: " + ex);
    }
}
}

```

In program3, I used **FileInputStream** to read data from file in the form of sequence of bytes.

First, we should attach the file to a FileInputStream as:

```
FileInputStream fin = new FileInputStream("myfile.txt")
```

This will enable us to read data from the file. Then, to read data from the file, we should read data from the **FileInputStream** as:

```
ch = fin.read();
```

When the `read()` method reads all the characters from the file, it reaches the end of the file. When there is no more data available to read further, the `read` method returns `-1`.

Then, we should attach the monitor to some output stream, example `PrintStream`, so that the output stream will send data to the monitor. For displaying the data, we can use `System.out` which is nothing but `PrintStream` object.

Finally, we read data from the `FileInputStream` and write it to `System.out`. This will display all the file data on the screen.

11.5 Character Streams

Character streams are defined within two class hierarchies, one for input and one for output:

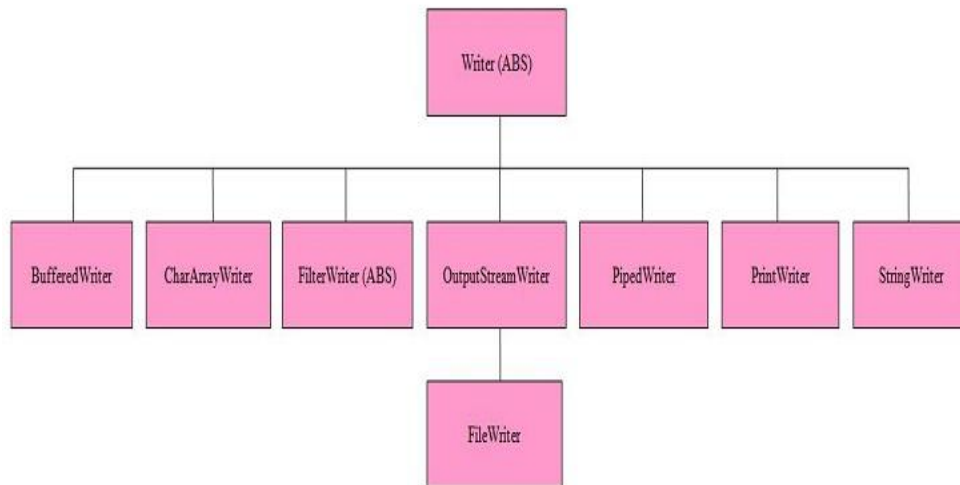
- The **Writer** class is the *abstract superclass* of all character output streams
- The **Reader** class is the *abstract superclass* of all character input streams

These classes define the characteristics that are common to character input and character output streams, which are implemented in the concrete subclasses of each hierarchy.

Character Output Stream Hierarchy

The diagram below shows the classes in the *character output stream hierarchy* of which the `Writer` class is the *abstract superclass*.

Character Output Stream Hierarchy



Class	Description
<u>Writer</u>	Abstract character stream superclass which describes this type of output stream.
<u>BufferedWriter</u>	Buffered output character stream.
CharArrayWriter	Character buffer output stream.
FilterWriter	Abstract character stream for writing filtered streams.
OuputStreamWriter	Output Stream that acts as a bridge for encoding byte streams from character streams.
<u>FileWriter</u>	Output stream for writing characters to a file.

Class	Description
PipedWriter	Piped character output stream.
<u>PrintWriter</u>	Convenience output character stream to add functionality to another stream, an example being to print to the console using print() and println().
StringWriter	Output stream for writing characters to a string.

11.5.1 The java.io.Writer Class

At the top of the character output stream hierarchy is the **java.io.Writer** *abstract superclass* which defines a basic set of output functions that all *character output stream classes* have. These methods allow us to append, flush and write bytes within the *character output stream* as well as close the stream when we finish our processing. All methods within the java.io.Writer class can *throw* an IOException.

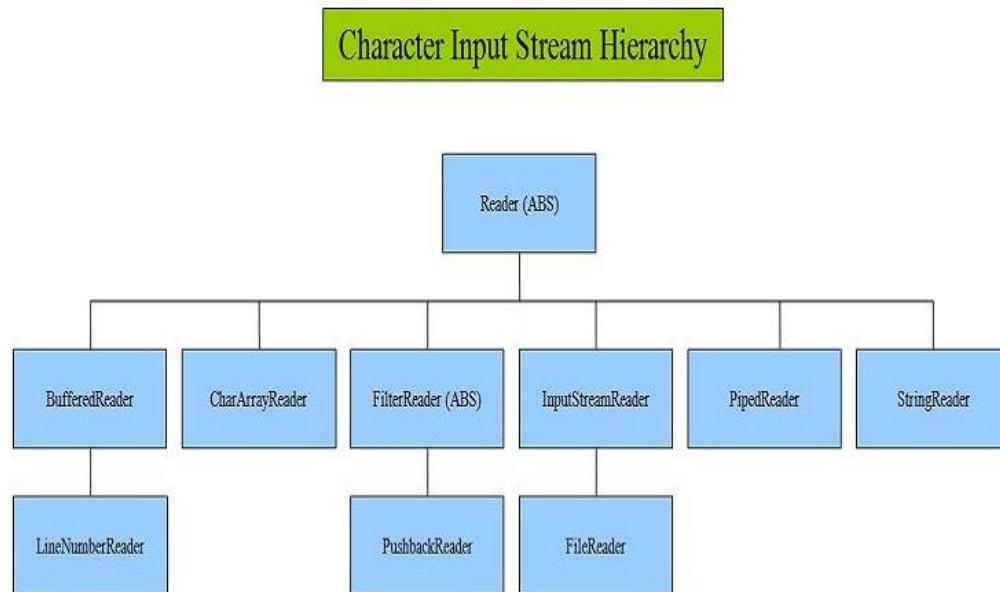
The table below shows the declarations of all the methods in the java.io.Writer class:

Method Declaration	Description
public Writer append(char c)	Appends the specified character to the end of the invoking Writer stream and returns a reference to the invoking Writer stream.
public Writer	Appends the specified character

Method Declaration	Description
append(CharSequence chars, int start, int end)	sequence specified by the start and end arguments to the end of the invoking Writer stream and returns a reference to the invoking Writer stream.
public abstract void close()	Closes output stream and release any system resources associated with it.
public abstract void flush()	Flushes output stream and forces any buffered output bytes to be written out.
public void write(int ch)	Writes a single character of data to the output stream.
public void write(char[] chbuff)	Writes an array of characters.
public void write(String str)	Writes a string of data specified by str to the output stream.
public void write(String str, int offset, int length)	Writes part of a string of data specified by str starting at offset for length.

Character Input Stream Hierarchy

The diagram below shows the classes in the *character input stream hierarchy* of which the Reader class is the *abstract superclass*..



Class	Description
<u>Reader</u>	Abstract character stream superclass which describes this type of input stream.
<u>BufferedReader</u>	Buffered input character stream.
LineNumberReader	Input character stream that keeps a count of line numbers.
CharArrayReader	Character buffer input stream.
FilterReader	Abstract character stream for reading filtered streams.

Class	Description
PushbackReader	Character stream reader containing functionality to return characters to the input stream.
InputStreamReader	Input Stream that acts as a bridge for decoding byte streams into character streams.
FileReader	Input stream for reading characters from a file.
PipedReader	Piped character input stream.
StringReader	Input stream for reading characters from a string.

11.5.2 The java.io.Reader Class

At the top of the character input stream hierarchy is the java.io.Reader *abstract superclass* which defines a basic set of input functions that all *character input stream classes* have. These methods allow us to read, ready, reset and skip characters within the *character input stream* as well as close the stream when we finish our processing. All methods within the java.io.Reader class can *throw* an IOException.

The table below shows the declarations of all the methods in the java.io.Reader class:

Method Declaration	Description
public abstract void	Closes input stream and release any

Method Declaration	Description
close()	system resources associated with it.
public abstract int read()	Reads next character of data from input stream or -1 if end of file encountered.
public void ready()	Returns true if stream is ready to read from.
public boolean reset()	Reset the input pointer to the last mark() position.
public long skip(long skipChars)	Skip number of characters specified by skipChars and return skipChars.

11.5. 3 The java.io.BufferedReader Class

We saw in the Byte Stream Classes how we can read bytes from the console using the [read\(\)](#) and [read\(bytes\[\]\)](#) methods. This worked fine because we only used characters from the standard 8-bit *ASCII character set* which has a range of 0 to 127. If we had tried to run our example with any character outside this range we would have got some very strange results and you might want to go back to the code we wrote and try inputting a character such as á. This will return -96131 followed by lots of zeros or the ? character when cast to a char type. For more verbose applications and proper internationalisation we need to use the full *Unicode character set* which has a range 0 to 65,536 and for this we need to use a *character input stream*.

So to read characters from the console we need a class to convert *byte streams* into *character streams* and from looking at the [character input stream hierarchy](#) we can see that the

InputStreamReader class acts as a *bridge* for decoding *byte streams* into *character streams* which is exactly what we want.

Read Characters From The Console

Lets see how we could use the BufferedReader class to get a sequence of characters from the keyboard:

Program4 : Read character sequence from the keyboard using BufferedReader

```
import java.io.*; // Import all file classes from java.io package

class BufferedReaderExample1
{
    public static void main(String[] args) throws IOException
    {
        String str;
        // Create a BufferedReader wrapped an
        InputStreamReader stream
        InputStreamReader is = new
        InputStreamReader(System.in)
        BufferedReader br = new BufferedReader(is);
        System.out.println("BufferedReader br has been
        opened.");
        do
        {
            str = br.readLine();
            System.out.println("I typed in: " + str + ". Type in 'end'
            to quit");
        } while (!str.equals("end"));
        System.out.println("I typed in 'end'");
    }
}
```

The java.io.FileWriter Class

Program 5: Write a program to create a text file using character or text stream classes.

```
//Creating a text file using character (text) stream classes
import java.io.*;
class Create3
{ public static void main(String args[]) throws IOException
  { String str = "This is an Institute" + "\n You are a
student"; // take a String
  //Connect a file to FileWriter
  FileWriter fw = new FileWriter ("textfile");
  //read chars from str and send to fw
  for (int i = 0; i<str.length() ; i++)
    fw.write (str.charAt (i) );
  fw.close ();
}
}
```

Output:

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window shows the following commands and output:
D:\JQR>javac Create3.java
D:\JQR>java Create3
D:\JQR>type textfile
This is an Institute
You are a student
D:\JQR>

In this example of using the **java.io.FileWriter** class we will create a File object in our working directory. We then accept user input until 'end' is entered using the **BufferedReader** class and write each user entry to a file using the **FileWriter** class:

Program 6: Write a program to read a text file using character or text stream classes.

//Reading data from file using character (text) stream classes

```

import java.io.*;
class Read3
{
    public static void main(String args[]) throws IOException
    {
        //attach file to FileReader
        FileReader fr = new FileReader ("textfile");
        //read data from fr and display
        int ch;
        while ((ch = fr.read()) != -1)
            System.out.print ((char) ch);
        //close the file
        fr.close ();
    }
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Read3.java
D:\JQR>java Read3
This is an Institute
You are a student
D:\JQR>

```

Program 7: Write a program to read and write a text file using character or text stream classes.

import java.io.*; // Import all file classes from java.io package

```

class FileReadWriterExample1 {
    public static void main(String[] args) throws IOException {
        FileWriter fw;

        String str;
        // Open FileWriter.txt and output user input to it
        try {
            fw = new FileWriter("FileWriter.txt"); // Create a stream
            // to write to FileWriter.txt
            System.out.println("FileWriter fw has been opened.");
        }
    }
}

```

```

        // Create a BufferedReader wrapped in
InputStreamReader stream
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("BufferedReader br has been
opened.");
        do {
            str = br.readLine();
            //System.out.println("I typed in: " + str + ". Type in
'end' to quit");
            // Check for stop condition
            if (str.compareTo("end") == 0)
            {
                break;
            }
            str = str + "\n";

            fw.write(str);
        } while (str.compareTo("end") !=0);

        System.out.println("I typed in 'end'");
        fw.close();
    }
    catch (IOException ex) {
        System.out.println("Exception opening FileWriter.txt : "
+ ex);
    }
}

```

```

        FileReader fr = new FileReader(("FileWriter.txt")); // Our
FileReader

```

```

        System.out.println("FileReader fr has been opened.");
        int x;
        do
        {

```



```

        x = fr.read();
        System.out.println("We read in character " + (char)x);
    }while( x!= -1);

    fr.close();

}
}

```

Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similar way Java provides following three standard streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**.

Following is a simple program which creates **InputStreamReader** to read standard input stream until the user types a "q":

```

import java.io.*;
public class ReadConsole
{
    public static void main(String args[]) throws IOException

```

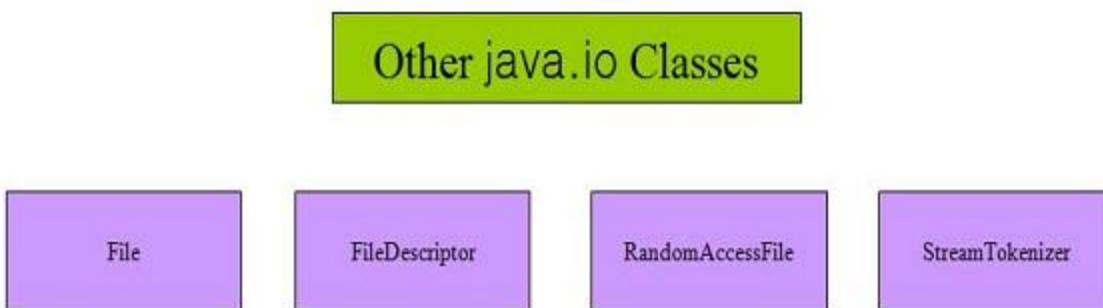
```

    {
        try
        {
            InputStreamReader cin = new
InputStreamReader(System.in);
            System.out.println("Enter characters, 'q' to quit.");
            char c;
            do
            {
                c = (char) cin.read();
                System.out.print(c);
            } while(c != 'q');
        }finally
        {
            cin.close();
        }
    }
}

```

11.6 Other Java I/O Classes

The diagram below shows some other pertinent classes in the java.io package not covered in the byte and character streams above:



Class	Description
<u>File</u>	Abstract representation of file and directory pathnames.
RandomAccessFile	Allows reading and writing of bytes to a random access file.
StreamTokenizer	Input stream to be parsed into 'tokens'.

The java.io.File Class

Let talk about the File class that exists within the Java.io package and how we use objects of this class to represent an actual file and directory pathname that may or may not exist already on a hard drive. A File object doesn't contain the file in question or any data associated with the file, it just acts like a pointer to said file and can be a relative or absolute pathname:

1. **Relative URL** - The common use of a relative URL is by omitting the protocol and server name, as documents generally reside on the same server. So this would be `directoryName/fileName.extension`.
For example the relative url `images/j5icon.jpg`
2. **Absolute URL** - An absolute url is the complete address of the resource.
For example the absolute url
`d:/Mahesh/IO/info/images/j5icon.jpg`

Before we get look at some of the methods of the File class, lets clarify what we mean by a File object pointing to a file:

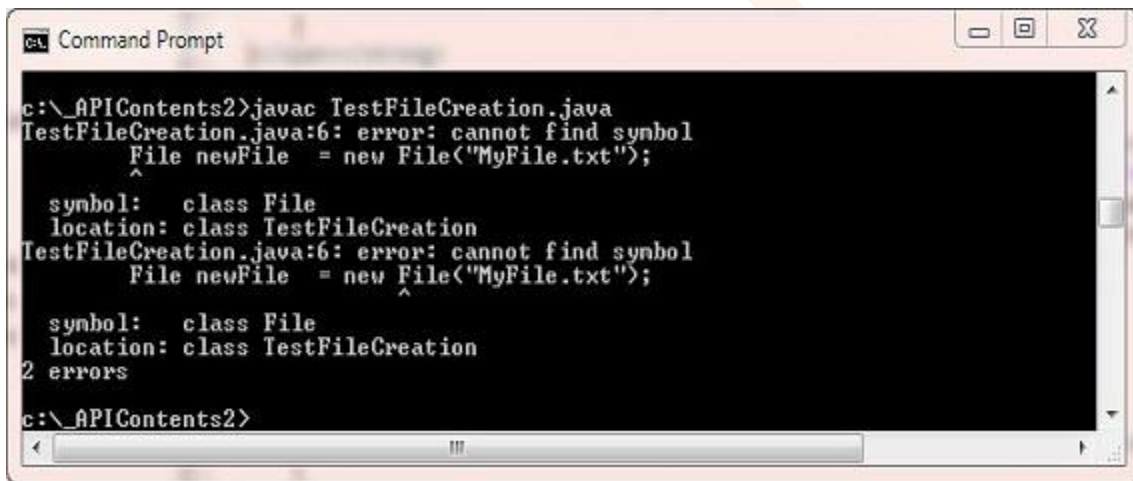
```
/*
  Create a file object that points to a file
*/
```

```

class TestFileCreation
{
    public static void main(String[] args)
    {
        File newFile = new File("MyFile.txt");
    }
}

```

Save, compile and run the TestFileCreation test class in directory **c:_APIContents2** in the usual way.



```

c:\_APIContents2>javac TestFileCreation.java
TestFileCreation.java:6: error: cannot find symbol
    File newFile = new File("MyFile.txt");
    ^
  symbol:   class File
  location: class TestFileCreation
TestFileCreation.java:6: error: cannot find symbol
    File newFile = new File("MyFile.txt");
    ^
  symbol:   class File
  location: class TestFileCreation
2 errors
c:\_APIContents2>

```

The above screenshot shows the output of compiling the TestFileCreation class. The compiler cannot find the File object type. This is because the File class exists within the java.io package and so we need to either qualify usage as in:

```

/*
  Create a file object that points to a file
*/
class TestFileCreation
{
    public static void main(String[] args)
    {

```

```

        java.io.File newFile = new java.io.File("MyFile.txt");
// Qualifying
    File gives clean compile
}
}

```

Or we can import the java.io package as in the code below. We will use the import keyword over qualifying the names in our examples as we think it is easier and less error-prone:

```

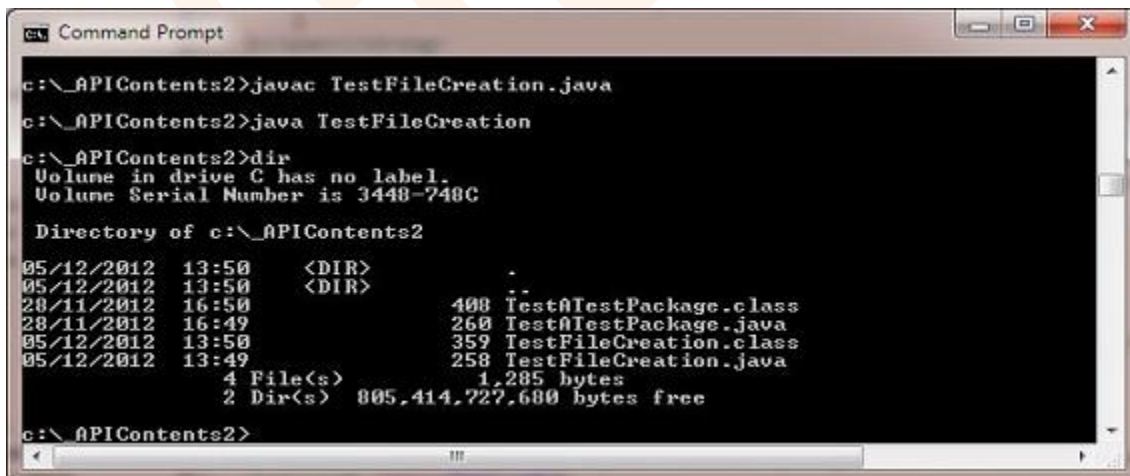
/*
    Create a file object that points to a file
*/
import java.io.File; // Just import the file class from java.io
package

```

```

class TestFileCreation {
    public static void main(String[] args) {
        File newFile = new File("MyFile.txt");
    }
}

```



```

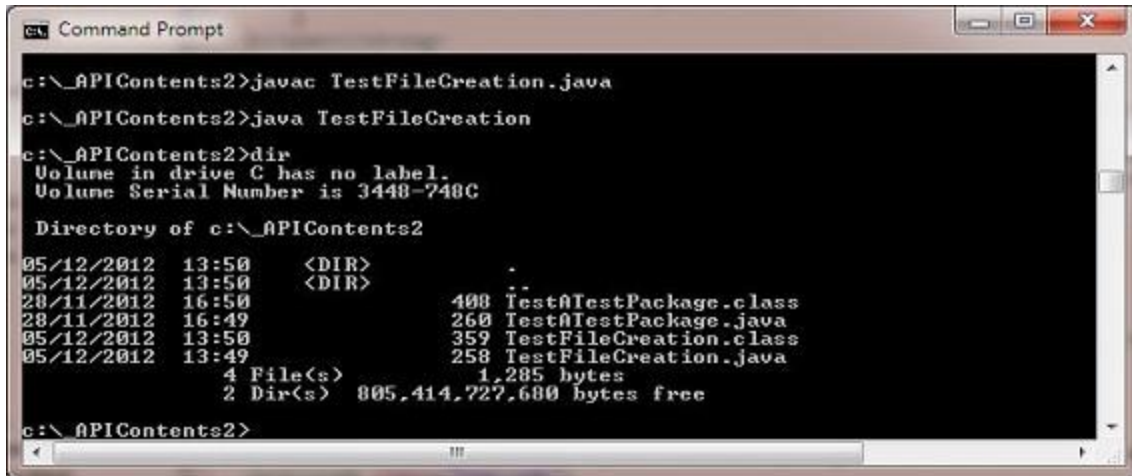
c:\_APIContents2>javac TestFileCreation.java
c:\_APIContents2>java TestFileCreation
c:\_APIContents2>dir
Volume in drive C has no label.
Volume Serial Number is 3448-748C

Directory of c:\_APIContents2

05/12/2012  13:50    <DIR>          .
05/12/2012  13:50    <DIR>          ..
28/11/2012  16:50                408 TestATestPackage.class
28/11/2012  16:49                260 TestATestPackage.java
05/12/2012  13:50                359 TestFileCreation.class
05/12/2012  13:49                258 TestFileCreation.java
               4 File(s)              1,285 bytes
               2 Dir(s)  805,414,727,680 bytes free

c:\_APIContents2>

```



```
Command Prompt
c:\_APIContents2>javac TestFileCreation.java
c:\_APIContents2>java TestFileCreation
c:\_APIContents2>dir
Volume in drive C has no label.
Volume Serial Number is 3448-748C

Directory of c:\_APIContents2

05/12/2012  13:50    <DIR>          .
05/12/2012  13:50    <DIR>          ..
28/11/2012  16:50             408 Test0TestPackage.class
28/11/2012  16:49             260 Test0TestPackage.java
05/12/2012  13:50             359 TestFileCreation.class
05/12/2012  13:49             258 TestFileCreation.java
               4 File(s)              1,285 bytes
               2 Dir(s)  805,414,727,680 bytes free

c:\_APIContents2>
```

The above screenshot shows the output of running the TestFileCreation class and then doing a **dir** command in the **c:_APIContents2** directory. As you can see we haven't created a physical file called **MyFile.txt**, we just made a File object to point to a file that may exist called **MyFile.txt** in the relative path we are in.

This time we will create a physical file and directory and use some File methods:

```
/*
  Create a file object that points to a directory
*/
import java.io.File; // Just import the file class from java.io
package
import java.io.IOException; // Import the IOException class
from java.io package
```

```
class TestFileCreation2
{
    public static void main(String[] args)
    {
        File newFile = new File("MyFile.txt");
        if (newFile.exists())
```

```

    { // Check to see if Myfile.txt already exists
      System.out.println("Myfile.txt already exists.");
    }
    else
    {
      System.out.println("Try to create Myfile.txt");
      try
      {
        newFile.createNewFile(); // Create a file called
Myfile.txt
        System.out.println("We created Myfile.txt");
      }
      catch (IOException ex)
      {
        System.out.println("We caught exception: " + ex);
      }
    }
    File newDir = new File("JavaIO");
    newDir.mkdir(); // Create a directory called JavaIO
    if (newDir.isDirectory()) { // true as we created a
directory
      System.out.println("We created a directory.");
      System.out.println(newDir.getAbsolutePath());
      System.out.println(newDir.getParent());
    }
  }
}

```

Save, compile and run the TestFileCreation2 test class in directory **c:_APIContents2** in the usual way.

```
Command Prompt

c:\_APIContents2>javac TestFileCreation2.java
c:\_APIContents2>java TestFileCreation2
Try to create Myfile.txt
We created Myfile.txt
We created a directory.
c:\_APIContents2\JavaIO
null

c:\_APIContents2>dir
Volume in drive C has no label.
Volume Serial Number is 3448-748C

Directory of c:\_APIContents2

06/12/2012  12:38    <DIR>          .
06/12/2012  12:38    <DIR>          ..
06/12/2012  12:38    <DIR>          JavaIO
06/12/2012  12:38             0 MyFile.txt
28/11/2012  16:50       408 TestATestPackage.class
28/11/2012  16:49       260 TestATestPackage.java
05/12/2012  13:50       359 TestFileCreation.class
05/12/2012  13:51       183 TestFileCreation.java
06/12/2012  12:38       1,225 TestFileCreation2.class
06/12/2012  12:38       1,239 TestFileCreation2.java
              7 File(s)              3,674 bytes
              3 Dir(s)  805,412,048,896 bytes free

c:\_APIContents2>
```

Directories in Java:

A directory is a File which can contains a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail check a list of all the methods which you can call on **File** object and what are related to directories.

Creating Directories:

There are two useful **File** utility methods, which can be used to create directories:

- The **mkdir()** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory:

```
import java.io.File;

public class CreateDir {
    public static void main(String args[]) {
        String dirname = "/tmp/user/java/bin";
        File d = new File(dirname);
        // Create directory now.
        d.mkdirs();
    }
}
```

Compile and execute above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories:

You can use **list()** method provided by **File** object to list down all the files and directories available in a directory as follows:

```
import java.io.File;

public class ReadDir {
    public static void main(String[] args) {

        File file = null;
        String[] paths;

        try{
            // create new file object
            file = new File("/tmp");
```

```

        // array of files and directory
        paths = file.list();

        // for each name in the path array
        for(String path:paths)
        {
            // prints filename and directory name
            System.out.println(path);
        }
    }catch(Exception e){
        // if any error occurs
        e.printStackTrace();
    }
}
}

```

This would produce following result based on the directories and files available in your **/tmp** directory:

How to delete files with certain extension only

In Java, you can implements the [FilenameFilter](#), override the accept(File dir, String name) method, to perform the file filtering function.

In this example, we show you how to use FilenameFilter to list out all files that are end with “**.txt**” extension in folder “**c:\\folder**“, and then delete it.

```
package com.mkyong.io;
```

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

```
public class FileChecker {
```

```
    private static final String FILE_DIR = "c:\\folder";
```

```
    private static final String FILE_TEXT_EXT = ".txt";
```

```

public static void main(String args[]) {
    new FileChecker().deleteFile(FILE_DIR,FILE_TEXT_EXT);
}

public void deleteFile(String folder, String ext){

    GenericExtFilter filter = new GenericExtFilter(ext);
    File dir = new File(folder);

    //list out all the file name with .txt extension
    String[] list = dir.list(filter);

    if (list.length == 0) return;

    File fileDelete;

    for (String file : list){
        String temp = new StringBuffer(FILE_DIR)
            .append(File.separator)
            .append(file).toString();
        fileDelete = new File(temp);
        boolean isdeleted = fileDelete.delete();
        System.out.println("file : " + temp + " is deleted : " +
isdeleted);
    }
}

//inner class, generic extension filter
public class GenericExtFilter implements FilenameFilter {

    private String ext;

    public GenericExtFilter(String ext) {
        this.ext = ext;
    }
}

```

```

        public boolean accept(File dir, String name) {
            return (name.endsWith(ext));
        }
    }
}

```

How to find files with certain extension only

A [FilenameFilter](#) example, it will only display files that use “.jpg” extension in folder “c:\\folder”.

```

package com.mkyong.io;

import java.io.*;

public class FindCertainExtension {

    private static final String FILE_DIR = "c:\\folder";
    private static final String FILE_TEXT_EXT = ".jpg";

    public static void main(String args[]) {
        new FindCertainExtension().listFile(FILE_DIR,
        FILE_TEXT_EXT);
    }

    public void listFile(String folder, String ext) {

        GenericExtFilter filter = new GenericExtFilter(ext);

        File dir = new File(folder);

        if(dir.isDirectory()==false){
            System.out.println("Directory does not exists
: " + FILE_DIR);
            return;
        }
    }
}

```

```

    }

    // list out all the file name and filter by the
extension
    String[] list = dir.list(filter);

    if (list.length == 0) {
        System.out.println("no files end with : " +
ext);
        return;
    }

    for (String file : list) {
        String temp = new
StringBuffer(FILE_DIR).append(File.separator)
        .append(file).toString();
        System.out.println("file : " + temp);
    }
}

// inner class, generic extension filter
public class GenericExtFilter implements
FilenameFilter {

    private String ext;

    public GenericExtFilter(String ext) {
        this.ext = ext;
    }

    public boolean accept(File dir, String name) {
        return (name.endsWith(ext));
    }
}
}

```

How to rename file in Java

Java comes with **renameTo()** method to rename a file. However, this method is really platform-dependent: you may successfully rename a file in *nix but failed in Windows. So, the return value (true if the file rename successful, false if failed) should always be checked to make sure the file is rename successful.

File.renameTo() Example

```
package com.mk Yong.file;

import java.io.File;

public class RenameFileExample
{
    public static void main(String[] args)
    {
        File oldfile =new File("oldfile.txt");
        File newfile =new File("newfile.txt");

        if(oldfile.renameTo(newfile)){
            System.out.println("Rename succesful");
        }else{
            System.out.println("Rename failed");
        }
    }
}
```

How to copy file in Java

Java didn't comes with any ready make file copy function, you have to manual create the file copy process. To copy file, just

convert the file into a bytes stream with **FileInputStream** and write the bytes into another file with **FileOutputStream**.

The overall processes are quite simple, just do not understand why Java doesn't include this method into the **java.io.File** class.

File copy example

Here's an example to copy a file named "Afile.txt" to another file named "Bfile.txt". If the "Bfile.txt" is exists, the existing content will be replace, else it will create with the content of the "Afile.txt".

```
package com.mkkyong.file;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class CopyFileExample
{
    public static void main(String[] args)
    {
        InputStream inStream = null;
        OutputStream outStream = null;

        try{

            File afile =new File("Afile.txt");
            File bfile =new File("Bfile.txt");

            inStream = new FileInputStream(afile);
```

```

        outputStream = new FileOutputStream(bfile);

        byte[] buffer = new byte[1024];

        int length;
        //copy the file content in bytes
        while ((length = inputStream.read(buffer)) > 0){

            outputStream.write(buffer, 0, length);

        }

        inputStream.close();
        outputStream.close();

        System.out.println("File is copied successful!");

    }catch(IOException e){
        e.printStackTrace();
    }
}
}

```

How to move file to another directory in Java

Java.io.File does not contains any ready make move file method, but you can workaround with the following two alternatives :

1. File.renameTo().
2. Copy to new file and delete the original file.

In the below two examples, you move a file “**C:\\folderA\\Afile.txt**” from one directory to another directory with the same file name “**C:\\folderB\\Afile.txt**”.

1. File.renameTo()


```

package com.mkyong.file;

import java.io.File;

public class MoveFileExample
{
    public static void main(String[] args)
    {
        try{

            File afile =new File("C:\\folderA\\Afile.txt");

            if(afile.renameTo(new File("C:\\folderB\\" +
afile.getName()))){
                System.out.println("File is moved successful!");
            }else{
                System.out.println("File is failed to move!");
            }

        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

2. Copy and Delete

```

package com.mkyong.file;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class MoveFileExample
{

```

```

public static void main(String[] args)
{

    InputStream inStream = null;
    OutputStream outStream = null;

    try{

        File afile =new File("C:\\folderA\\Afile.txt");
        File bfile =new File("C:\\folderB\\Afile.txt");

        inStream = new FileInputStream(afile);
        outStream = new FileOutputStream(bfile);

        byte[] buffer = new byte[1024];

        int length;
        // copy the file content in bytes
        while ((length = inStream.read(buffer)) > 0){

            outStream.write(buffer, 0, length);

        }

        inStream.close();
        outStream.close();

        // delete the original file
        afile.delete();

        System.out.println("File is copied successful!");

    }catch(IOException e){
        e.printStackTrace();
    }
}

```

```
}
```

Example 1: Write a program to count number of character, word and lines in a text file.

```
import java.io.*;  
public class Count  
{  
    public static void main(String[] args) throws  
IOException  
    {  
        int ch;  
        boolean prev = true;  
        int char_count =0;  
        int word_count =0;  
        int line_count =0;  
        FileInputStream fin = new  
FileInputStream("Myfile.txt");  
        while((ch = fin.read())!= -1)  
        {  
            if (ch != ' ')  
                ++char_count;  
            if(!prev && ch == ' ')  
                ++word_count;  
            if(ch == ' ')  
                prev = true;  
            else  
                prev = false;  
            if(ch == '\n')  
                ++line_count;
```

```

    }
    char_count -= line_count*2;
    word_count += line_count;
    System.out.println("No. of chars= " + char_count);
    System.out.println("No. of words= " +
word_count);
    System.out.println("No. of lines= " + line_count);
    fin.close();
}
}

```

11.7 Serialization of objects:

So far, we wrote some programs where we stored only text into the files and retrieved same text from the files. These text files are useful when we do not want to perform any calculations on the data. What happens if we want to store some structured data in the files? For example, we want to store some employee details like employee identification number (*int* type), name (*String* type), salary (*float* type) and date of joining the job (*Calendar* type) in a file. This data is well structured and got different types. To store such data, we need to create a class Employee with the instance variable id, name, sal, doj as shown here:

Class Employee implements Serializable

```

{
    //Instance Variable
    int id;
    String name;
    float sal;
    Calendar doj;
}

```

Then create an object to this class and store actual data into that object. Later, this should be stored into the file using *ObjectOutputStream*. Please observe that the *Serializable* interface should be implemented by the class whose objects are to be stored into the file. This is the reason why *Employee* class implements *Serializable* interface.

To store the *Employee* class object into a file, follow these steps:

First, attach *objfile* to *FileOutputStream*. This helps to write data into *objfile*.

```
FileOutputStream fos = new FileOutputStream("objfile");
```

Then, attach *FileOutputStream* to *ObjectOutputStream*.

```
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

Now, *ObjectOutputStream* can write objects using *writeObject()* method to *FileOutputStream*, which stores them into the *objfile*.

Storing objects into file like this is called 'serialization'. The reverse process where objects can be retrieved back from a file is called 'de-serialization'.

Serializable interface is an empty interface without any numbers in it. It does not contain any methods also. Such an interface is called "marking interface" or "tagging interface". Marking interface is useful to mark the objects of class for a special purpose. For example, '*Serializable*' interface marks the class objects as 'serializable' so that they can be written into a file. If *Serializable* interface is not implemented by the class, then writing that class object into a file will lead to *NotSerializableException*.

De-serialization is the process of reading back the objects from a file.

To read *Employee* class object from *objfile*, follow these steps:

Attach *objfile* to *FileInputStream*. This helps to read objects from *objfile*.

```
FileInputStream fis = new FileInputStream("objfile");
```

Attach *FileInputStream* to *ObjectInputStream*. This help *ObjectInputStream* gets the objects from *FileInputStream*.

```
ObjectInputStream ois = new ObjectInputStream(fis);
```

Now, read objects from *ObjectInputStream* using ***readObject()*** method as:

```
Employee e = (Employee) ois.readObject();
```

Write a program to create *Employ* class whose objects is to be stored into a file.

```
import java.io.*;
import java.util.*;
public class ObjectSerialization
{
    public static void main (String args[]) throws
    IOException, ClassNotFoundException
    { BufferedReader br = new BufferedReader (new
InputStreamReader (System.in));
        FileOutputStream fos = new FileOutputStream
        ("empfile");
        ObjectOutputStream oos = new ObjectOutputStream (
        fos );
        System.out.println("Writing object into a File(i.e.
        serialization) ");
        System.out.print ("Enter how many objects : ");
        int n = Integer.parseInt(br.readLine () );
```

```

    for(int i = 0;i<n;i++)
    {
        Employ e1 = Employ.getData ();
        oos.writeObject (e1);
    }
    oos.close ();
    fos.close ();
    System.out.println("Reading object from File(i.e.De-
serialization) ");
    FileInputStream fis = new FileInputStream ("empfile");
    ObjectInputStream ois = new ObjectInputStream (fis);
    try
    { Employ e;
        while ( (e = (Employ) ois.readObject() ) != null)
            e.display ();
    }
    catch(EOFException ee)
    {
        System.out.println ("End of file Reached...");
    }
    finally
    { ois.close ();
        fis.close ();
    }
}
}

```

```

class Employ implements Serializable
{
    private int id;
    private String name;
    private float sal;
    private Date doj;
    Employ (int i, String n, float s, Date d)

```

```

        {
            id = i;
            name = n;
            sal = s;
            doj = d;
        }
        void display ()
        {
            System.out.println (id+ "\t" + name + "\t" + sal + "\t" +
doj);
        }
        static Employ getData() throws IOException
        {
            BufferedReader br = new BufferedReader (new
InputStreamReader (System.in));
            System.out.print ("Enter employ id : ");
            int id = Integer.parseInt(br.readLine());
            System.out.print ("Enter employ name : ");
            String name = br.readLine ();
            System.out.print ("Enter employ salary : " );
            float sal = Float.parseFloat(br.readLine ());
            Date d = new Date ();
            Employ e = new Employ (id, name, sal, d);
            return e;
        }
    }
}

```