Chapter - 14

The java.io Package

FileDescriptor	Instances of the file descriptor class serve as an opaque handle to the underlying machine-specific structure representing an open file, an open socket, or another source or sink of bytes.
FileInputStream	A FileInputStream obtains input bytes from a file in a file system.
FileOutputStream	A file output stream is an output stream for writing data to a File or to a FileDescriptor.
FilePermission	This class represents access to a file or directory.
FileReader	Convenience class for reading character files.
FileWriter	Convenience class for writing character files.
FilterInputStream	A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
FilterOutputStream	This class is the superclass of all classes that filter output streams.
FilterReader	Abstract class for reading filtered character streams.
FilterWriter	Abstract class for writing filtered character streams.
InputStream	This abstract class is the superclass of all classes representing an input stream of bytes.
InputStreamReader	An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.
LineNumberReader	A buffered character-input stream that keeps track of line numbers.
ObjectInputStream	An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream.

ObjectInputStream. GetField	Provide access to the persistent fields read from the input stream.
ObjectOutputStream	An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
ObjectOutputStream. PutField	Provide programmatic access to the persistent fields to be written to ObjectOutput.
ObjectStreamClass	Serialization's descriptor for classes.
ObjectStreamField	A description of a Serializable field from a Serializable class.
OutputStream	This abstract class is the superclass of all classes representing an output stream of bytes.
OutputStreamWriter	An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into
PipedInputStream	A piped input stream should be connected to a piped output stream; the piped input stream then provides whatever data bytes are written to the piped output stream.
PipedOutputStream	A piped output stream can be connected to a piped input stream to create a communications pipe.
PipedReader	Piped character-input streams.
PipedWriter	Piped character-output streams.
PrintStream	A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently.
PrintWriter	Prints formatted representations of objects to a text-output stream.
PushbackInputStream	A PushbackInputStream adds functionality to another input stream, namely the ability to "push back" or "unread" one byte.
PushbackReader	A character-stream reader that allows characters to be pushed back into the stream.

RandomAccessFile	Instances of this class support both reading and writing to a random access file.
Reader	Abstract class for reading character streams.
SequenceInputStream	A SequenceInputStream represents the logical concatenation of other input streams.
SerializablePermission	This class is for Serializable permissions.
StreamTokenizer	The StreamTokenizer class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time.
StringReader	A character stream whose source is a string.
StringWriter	A character stream that collects its output in a string buffer, which can then be used to construct a string.
Writer	Abstract class for writing to character streams.

Before we start the stream classes we first discuss about the **File** class because this class we use in some stream classes.

The File Class

The **File** class is one of the classes of **java.io**. package. This class does not operate on streams. The File class can not be used to create files or modify the text in the file. The File class can be used to obtain or change the properties of the files such as filename, path of file, length of file, permissions, etc. This class contains various methods for describing the properties of the file and the following tables contain constructors and methods of File class.

Constructors	Meaning
File(File fileobj, String filename)	Creates a new File instance from fileobj and filename string.
File(String directorypath)	Creates a new File instance by converting the given directorypath string into an abstract pathname.
File(String directorypath, String filename)	Creates a new File instance from directorypath string and a filename string.

File(URI uriobj)	Creates a new File instance by converting the given file: URI into an abstract pathname

Methods	Meaning
boolean canExecute()	Tests whether the application can execute the file denoted by this abstract pathname.
boolean canRead()	Tests whether the application can read the file denoted by this abstract pathname.
boolean canWrite()	Tests whether the application can modify the file denoted by this abstract pathname.
int compareTo(**File** pathname)	Compares two abstract pathnames lexicographically.
boolean createNewFile()	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
static **File** createTempFile(**String** prefix, **String** suffix)	Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
static **File** createTempFile(**String** prefix, **String** suffix, **File** directory)	Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
boolean delete()	Deletes the file or directory denoted by this abstract pathname.
void deleteOnExit()	Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
boolean equals(**Object** obj)	Tests this abstract pathname for equality with the given object.
boolean exists()	Tests whether the file or directory denoted by this abstract pathname exists.
File getAbsoluteFile()	Returns the absolute form of this abstract pathname.

String getAbsolutePath()	Returns the absolute pathname string of this abstract pathname.
long getFreeSpace()	Returns the number of unallocated bytes in the partition **named** by this abstract path name.
String getName()	Returns the name of the file or directory denoted by this abstract pathname.
String getParent()	Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
File getParentFile()	Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
long getTotalSpace()	Returns the size of the partition **named** by this abstract pathname.
long getUsableSpace()	Returns the number of bytes available to this virtual machine on the partition **named** by this abstract pathname.
boolean isAbsolute()	Tests whether this abstract pathname is absolute.
boolean isDirectory()	Tests whether the file denoted by this abstract pathname is a directory.
boolean isFile()	Tests whether the file denoted by this abstract pathname is a normal file.
boolean isHidden()	Tests whether the file named by this abstract pathname is a hidden file.
long lastModified()	Returns the time that the file denoted by this abstract pathname was last modified.
long length()	Returns the length of the file denoted by this abstract pathname.
String[]list()	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
String[]list(**FilenameFilter** filter)	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.

File[]listFiles()	Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
File[]listFiles(**FileFilter** filter)	Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]listFiles(**FilenameFilter** filter)	Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
tatic **File**[]listRoots()	List the available filesystem roots.
boolean mkdir()	Creates the directory named by this abstract pathname.
boolean mkdirs()	Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
boolean renameTo(**File** dest)	Renames the file denoted by this abstract pathname.
boolean setReadable(boolean readable)	A convenience method to set the owner's read permission for this abstract pathname.
boolean setReadable(boolean readable, boolean ownerOnly)	Sets the owner's or everybody's read permission for this abstract pathname.
boolean setReadOnly()	Marks the file or directory named by this abstract pathname so that only read operations are allowed.
boolean setWritable(boolean writable)	A convenience method to set the owner's write permission for this abstract pathname.
boolean setWritable(boolean writable, boolean ownerOnly)	Sets the owner's or everybody's write permission for this abstract pathname.

String toString()	Returns the pathname string of this abstract pathname.
URI toURI()	Constructs a file: URI that represents this abstract pathname.

A program to demonstrate the some methods of File class

Bin>edit file1.java

```
import  java.io.File;
class file1
{
   public static void main(String argv[])
   {
         File f=new File("c:\\program
               files\\java\\jdk1.5.0_5\\bin\\A.java");
         System.out.print("\n FileName="+f.getName());
         System.out.print("\n Parent="+f.getParent());
         System.out.print("\n Path="+f.getPath());
         System.out.print("\nSize="+f.length());
         System.out.print("\n Can write="+f.canWrite());
         System.out.print("\n Can read="+f.canRead());
         System.out.print("\n Is file+"+f.isFile());
         System.out.print("\n Is directory="+
                              f.isDirectory());
         System.out.print("\n Is hidden="+f.isHidden());
         System.out.print("\n Last Modified : "+
                              f.lastModified());
   }
}
```

Bin>javac file1.java
Bin>java file1

File Name=A. java
Parent= C:\Program files\java\jdk1.5.0_5\bin
Path= C:\ Program files\java\jdk1.5.0_5\bin\A.java
Size=0
Can write= false
Can read= false
Is file+false
Is directory= false

Is hidden= false
Last modified:0

delete()

The **delete()** method of File class deletes the file or directory denoted by abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. The signature of the **delete()** method is.

public boolean delete()

If **delete()** method deletes the file or directory successfully then it returns **true** otherwise **false**.

renameTo()

The **renameTo()** method of File class can be used to rename the file denoted by this abstract pathname. If the file is renamed successfully then this method returns **true** otherwise returns **false**

The return value should always be checked to make sure that the rename operation was successful. The signature of **renameTo()** method is:

public boolean renameTo(File destobj)

destobj is the destination File object with which the file should be renamed.

Bin>edit file2.java

```
import java.io.File;
class file2
{
   public static void main(String argv[])
   {
         File f1=new File("c:\\program files\\java\\
                  jdk1.5.0_5\\bin\\A.java");
         File f2=new File("c:\\program files\\java\\
                  jdk1.5.0_5\\bin\\B.java");
         boolean flag;
         flag= f1.renameTo(new File File("c:\\program
               files\\java\\jdk1.5.0_5\\bin\\Z.java"));

         if(flag==true)
```

```
                System.out.print("\n A.java file
                                  successfully renamed");

            else
                System.out.print("\n file not renamed");
            flag=f2.delete();

            if(flag==true)
                System.out.print("\n B.java successfully
                                  deleted");

            else
                System.out.print("\n file not deleted");

        }
}
```

Bin>javac file2.java
Bin>java file2

File not renamed
File not deleted

list ( )

The **list()** method of **File** class returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

If this abstract pathname does not denote a directory, then this method returns null. Otherwise an array of strings is returned, one for each file or sub-directory in the directory. Each string is a file or directory name rather than a complete path.

There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order. The signature of the list() method is:

public String[] list()

The list() method returns n array of strings naming the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. It Returns null if this abstract pathname does not denote a directory.

A program to demonstrate the list() of File class

Bin>edit file3.java

```
import java.io.File;
class file3
```

```
{
    public static void main(String argv[])
    {
            File f1=new File("c:\\program files\\java\\
                                    jdk1.5.0_05\\bin");
            String s[]=f1.list();
            for(int i=0;i<s.length;i++)
            {
                    File f=new File(f1,s[i]);
                    if(f.isDirectory())
                    System.out.print("\n Dir :" +s[i]);
                    else
                    System.out.print("\n File :" +s[i]);
            }
    }
}
```

Bin>javac file3.java
Bin>java file3

File: A.java
File: javac.exe
File: java.exe
Dir: mydir
Dir: pack1

Input/Output streams in Java

A program can get input from a data source by reading a sequence characters from a stream attached to the source(Ex: file, keyboard, etc.). A program can produce output by writing a sequence of characters to an output stream attached to a destination(Ex: file, monitor etc.). All I/O operations in Java are stream-based. All the data is treated as a stream of bytes or characters. The sources from where data are read from is called input stream. The source to which the data is written into is called output stream. Java has two types of streams, **1) Byte stream** and **2) Character stream**. A byte stream deals with bytes, character stream deals with unicodes characters. Each stream has its own input stream and output stream classes . The I/O operations of byte stream are handled by two abstract classes **InputStream** and **OutputStream** . The I/O operations of character stream are handled by two abstract classes **Reader** and **Writer**.

Byte Stream

The byte stream can be used to read/write from a data source by reading/writing a sequence characters from a stream attached to the source(Ex: file, keyboard, monitor etc.). The importance of the byte stream is it can be used read/write any type of data such including objects. The two topmost byte stream superclasses are **InputStream** and **OuputStream**.

InputStream class

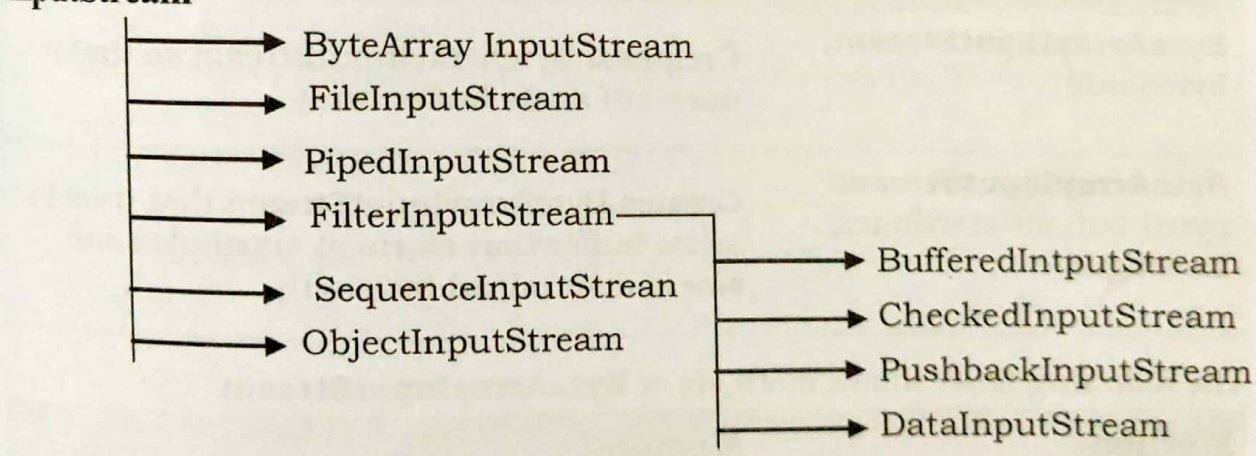
The **InputStream** abstract class is the superclass of all classes representing an input stream of bytes. This class contains various methods which can be used to perform byte input operation. Applications that need to define a subclass of **InputStream** must always provide a method that returns the next byte of input. This class in implemented from Closeable interface. Most of the methods of this class throws **IOException.** The following table shows the methods of **InputStream** Class

Methods	Meaning
int **available**()	Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
void **close**()	Closes this input stream and releases any system resources associated with the stream.
void **mark**(int readlimit)	Marks the current position in this input stream.
boolean **markSupported**()	Tests if this input stream supports the mark and reset methods.
bstract int **read**()	Reads the next byte of data from the input stream.
int **read**(byte[] b)	Reads some number of bytes from the input stream and stores them into the buffer array b.
int **read**(byte[] b, int off, int len)	Reads up to len bytes of data from the input stream into an array of bytes.
void **reset**()	Repositions this stream to the position at the time the mark method was last called on this input stream.
long **skip**(long n)	Skips over and discards n bytes of data from this input stream.

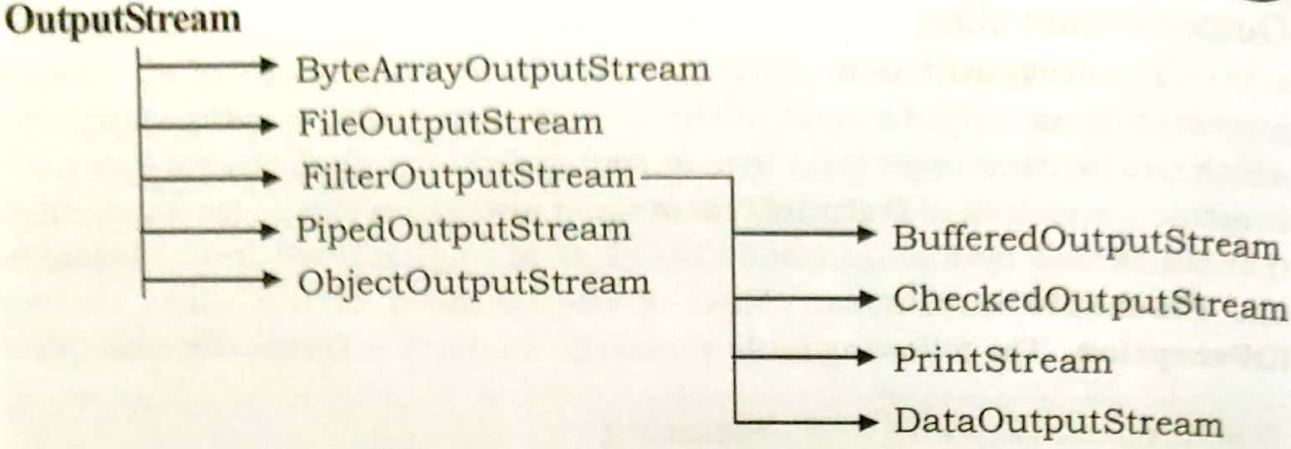
OutputStream class

The **OutputStream** abstract class is the superclass of all classes representing an output stream of bytes. This class contains various methods which can be used to perform byte output operation. Applications that need to define a subclass of **OutputStream** must always provide at least a method that writes one byte of output. This class in implemented from **Closeable** and **Flushable** interfaces. Most of the methods of this class throws **IOException.** The following table shows the methods of **OutputStream** Class

Methods	Meaning
void **close**()	Closes this output stream and releases any system resources associated with this stream.
void **flush**()	Flushes this output stream and forces any buffered output bytes to be written out.
void **write**(byte[] b)	Writes b.length bytes from the specified byte array to this output stream.
void **write**(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to this output stream.
abstract void **write**(int b)	Writes the specified byte to this output stream.



Class Hierarchy in InputStream



Class Hierarchy in OutputStream

ByteArrayInputStream class

The **ByteArrayInputStream** class is used to read bytes from a memory. The objects of this class are used to create input stream with memory buffers as data source. All the methods of the **InputStream** are either inherited or overridden in this class

Closing a **ByteArrayInputStream** has no effect. The methods in this class can be called after the stream has been closed without generating an **IOException**.

The following table shows constructors of **ByteArrayInputStream**.

Constructors	Meaning
ByteArrayInputStream(byte[] buf)	Creates a ByteArrayInputStream so that it uses buf as its buffer array.
ByteArrayInputStream(byte[] buf, int startindex, int length)	Creates ByteArrayInputStream that uses buf as its buffer that starts at startindex and size is determined by length

The following table shows methods of **ByteArrayInputStream**.

Methods	Meaning
int **available**()	Returns the number of remaining bytes that can be read (or skipped over) from this input stream.
void **close**()	Closing a ByteArrayInputStream has no effect.

void **mark**(int readAheadLimit)	Set the current marked position in the stream.
boolean **markSupported**()	Tests if this InputStream supports mark/reset.
int **read**()	Reads the next byte of data from this input stream.
int **read**(byte[] b, int off, int len)	Reads up to len bytes of data into an array of bytes from this input stream.
void **reset**()	Resets the buffer to the marked position.
long **skip**(long n)	Skips n bytes of input from this input stream.

A program to demonstrate ByteArrayInputStream to read data from byte stream

Bin>edit bytestream1.java

```
import java.io.*;
class bytestream1
{
 public static void main(String argv[]) throws IOException
 {
   String str="apex computers warangal";
   byte b[]=str.getBytes();
   ByteArrayInputStream obj=new ByteArrayInputStream(b);
   byte ch;
   while( (ch=(byte)obj.read())!=-1)
   {
      System.out.print((char)ch);
   }
 }
}
```

Bin>javac bytestream1.java
Bin>java bytestream1
apex computers warangal

Explanation:

In the above program, object **obj** is created to **ByteArrayInputStream** class which creates buffer to byte array **b**. In the while loop, one byte of the other is read by using **read()** method and is printed on the screen as character format. The loop terminates when **read()** reaches to end of stream. We get the output as shown above.

ByteArrayOutputStream class

The **ByteArrayOutStream** class is used to write bytes into memory. The **Object** of this class are used to create output stream with memory buffers as data sink. This has all the methods of the superclass **OutputStream**.

Closing a **ByteArrayOutputStream** has no effect. The methods in this class can be called after the stream has been closed without generating an **IOException**.

The following table shows constructors of **ByteArrayOutputStream**.

Constructors	Meaning
ByteArrayOutputStream()	Creates a new byte array output stream.
ByteArrayOutputStream(int size)	Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

The following table shows methods of **ByteArrayOutputStream**.

Methods	Meaning
void **close**()	Closing a ByteArrayOutputStream has no effect.
void **reset**()	Resets the count field of this byte array output stream to zero, so that all currently accumulated output in the output stream is discarded.
int **size**()	Returns the current size of the buffer.
byte[]**toByteArray**()	Creates a newly allocated byte array.
String **toString**()	Converts the buffer's contents into a string decoding bytes using the platform's default character set.

String **toString**(String charsetName)	Converts the buffer's contents into a string by decoding the bytes using the specified charsetName.
void **write**(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to this byte array output stream.
void **write**(int b)	Writes the specified byte to this byte array output stream.
void **writeTo**(OutputStream out)	Writes the complete contents of this byte array output stream to the specified output stream argument, as if by calling the output stream's write method using out.write(buf, 0, count).

A program to demonstrate ByteArrayOutputStream to write data into byte stream

Bin>edit bytestream2.java

```
import java.io.*;
class bytestream2
{
 public static void main(String argv[]) throws IOException
 {
   String str="apex computers warangal";
   byte b[]=str.getBytes();
   ByteArrayOutputStream obj=new ByteArrayOutputStream();
   obj.write(b,0,b.length);
   System.out.print("\n Buffer as string :" + obj.toString());
   byte barray[]=obj.toByteArray();
   System.out.print("\nbarray as string :");
   for(int i=0;i<obj.size();i++)
   System.out.print((char)barray[i]);
 }
}
```

Bin>javac bytestream2.java
Bin>java butestream2
Buffer as string : apex computers warangal
barray as string :apex computers warangal

Explanation:

In the above program, str string converts into byte array and is assigned to **b**. An object **obj** is instantiated to **ByteArrayOutputStream** and into this buffer **b** array bytes are written using the **write()**. The statement **obj.toString()** prints the bytes as string. The statement, **obj.toByteArray()** method retuns output stream buffer data in **obj** as byte array and assigns to **barray** byte array which is printed using the for loop as shown above output.

Filtered byte streams

The byte streams access the data in byte form. The raw bytes cannot be used for any useful purpose. For converting raw bytes into useful forms such as **char**, **string**, **int**, etc., Java has several streams. Java has several streams to convert between byte to useful form or vice versa. Such streams that can take other stream as argument are called filtered streams. The Java has **FilterInputStream** and **FilterOutputStream**.

There are several filtered streams which can work only on other streams. Some of the filtered streams are **BufferedInputStream**, **CheckedInputStream**, **PushbackInputStream**, **DataInputStream**

BufferedInputStream class

The **BufferedInputStream** class extends from FilterInputStream and adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods. When the **BufferedInputStream** is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time. The mark operation remembers a point in the input stream and the reset operation causes all the bytes read since the most recent mark operation to be reread before new bytes are taken from the contained input stream.

The following table shows the constructors of **BufferedInputStream** class

Constructors	Meaning
BufferedInputStream(InputStream in)	Creates a BufferedInputStream and saves its argument, the input stream in, for later use.
BufferedInputStream(InputStream in, int size)	Creates a BufferedInputStream with the specified buffer size, and saves its argument, the input stream in, for later use.

The following table shows the methods of **BufferedInputStream** class

Methods	Meaning
int **available**()	Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
void **close**()	Closes this input stream and releases any system resources associated with the stream.
void **mark**(int readlimit)	Marks the current position in this input stream.
boolean **markSupported**()	Tests if this input stream supports the mark and reset methods.
int **read**()	Reads the next byte of data from the input stream.
int **read**(byte[] b, int off, int len)	Reads bytes from this byte-input stream into the specified byte array, starting at the given offset.
void **reset**()	Repositions this stream to the position at the time the mark method was last called on this input stream.
long **skip**(long n)	Skips over and discards n bytes of data from this input stream.

A program to demonstrate BufferedInputStream class

Bin>edit bytestream3.java

```
import java.io.*;
class bytestream3
{
public static void main(String argv[]) throws IOException
{
   String str="apex computers warangal.
               It is one of the computer institute";
   byte b[]=str.getBytes();
   ByteArrayInputStream in=new ByteArrayInputStream(b);
```

```
    BufferedInputStream bis=new BufferedInputStream(in);
    byte ch;
    int i,len;
    len=bis.available();
    System.out.print("");
    for(i=0;i<10;i++)
    {
       if( ( ch=(byte)bis.read()) !=-1)
           System.out.print((char)ch);
    }
    System.out.print("\n");
    bis.mark(11);
    for(i=10;i<b.length;i++)
    {
       if( ( ch=(byte)bis.read()) !=-1)
           System.out.print((char)ch);
    }
    System.out.print("\n resetting to mark(11) : \n");
    bis.reset();
    while( ( ch=(byte)bis.read()) !=-1)
       System.out.print((char)ch);
    bis.close();
  }
}
```

Bin>javac bytestream3.java

Bin>java bytestream3

```
apex compu
ters warangal. It is one of the computer institute
 resetting to mark(11) :
ters warangal. It is one of the computer institute
```

Explanation:

In the above program, the **bis** object of **BufferedInputStream** reads data from the input stream object **in**.

The first for loop prints **10** bytes of data from the buffer and then the **mark(11)** marks the position in the buffer to **11th** byte. The second for loop prints all the remaining data from **11th** byte to last.

The **reset()** method makes the position of pointer in the buffer to the marked position **11th** byte. The while loop reads data from **11th** byte till end, and we get the output as shown above.

BufferedOutputStream class

This class is a subclass of **FilterOutputStream.** This class implements a buffered output stream, by setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.

The following table shows the constructors of **BufferedOutputStream**

Constructors	Meaning
BufferedOutputStream(OutputStream out)	Creates a new buffered output stream to write data to the specified underlying output stream.
BufferedOutputStream(OutputStream out, int size) size.	Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer

The following table shows the methods of **BufferedOutputStream**

Methods	Meaning
void **flush**()	Flushes this buffered output stream.
void **write**(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to this buffered output stream.
void **write**(int b)	Writes the specified byte to this buffered output stream.

A program to demonstrate BufferedOutputStream class

Bin>edit bytestream4.java

```
import java.io.*;
class bytestream4
{
   public static void main(String argv[]) throws IOException
   {
     String str="apex computers warangal";
     byte b[]=str.getBytes();
```

```
    ByteArrayOutputStream out=new ByteArrayOutputStream();
    BufferedOutputStream bos=new BufferedOutputStream(out);
    bos.write(b,0,b.length);
    System.out.print("\n Buffer as string :" +
                              out.toString());
    byte barray[]=out.toByteArray();
    System.out.print("\nbarray as string :");
    for(int i=0;i<out.size();i++)
      System.out.print((char)barray[i]);
  }
}
```

Bin> javac bytestream4.java

Bin> java bytestream4

Buffer as string :

barray as string :

DataInputStream class

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

DataInputStream is not necessarily safe for multithreaded access. Thread safety is optional and is the responsibility of users of methods in this class.

The following table shows constructors of **DataInputStream** class

Constructors	Meaning
DataInputStream(InputStream in)	reates a DataInputStream that uses the specified underlying InputStream

The following table shows methods of **DataInputStream** class

Methods	Meaning
int **read**(byte[] b)	Reads some number of bytes from the contained input stream and stores them into the buffer array b.
int **read**(byte[] b, int off, int len)	Reads up to len bytes of data from the contained input stream into an array of bytes.
boolean **readBoolean**()	See the general contract of the readBoolean method of DataInput.
byte **readByte**()	See the general contract of the readByte method of DataInput.
char **readChar**()	See the general contract of the readChar method of DataInput.
double **readDouble**()	See the general contract of the readDouble method of DataInput.
float **readFloat**()	See the general contract of the readFloat method of DataInput.
void **readFully**(byte[] b)	See the general contract of the readFully method of DataInput.
void **readFully**(byte[] b, int off, int len)	See the general contract of the readFully method of DataInput.
int **readInt**()	See the general contract of the readInt method of DataInput.
String **readLine**()	read a line of text from input stream such as keyboard or file etc.
long **readLong**()	See the general contract of the readLong method of DataInput.
short **readShort**()	See the general contract of the readShort method of DataInput.
int **readUnsignedByte**()	See the general contract of the readUnsignedByte method of DataInput.
int **readUnsignedShort**()	See the general contract of the readUnsignedShort method of DataInput.
String **readUTF**()	See the general contract of the readUTF method of DataInput.

static String **readUTF**(DataInput in)	Reads from the stream in a representation of a Unicode character string encoded in modified UTF-8 format; this string of characters is then returned as a String.
int **skipBytes**(int n)	See the general contract of the skipBytes method of DataInput.

A program to demonstrate DataInputStream class

Bin>edit bystream5.java

```
import java.io.*;
class bytestream5
{
   public static void main(String argv[]) throws IOException
   {
      int n;
      String str;
      DataInputStream dis=new DataInputStream(System.in);
      try
      {
         System.out.print("\n Enter no :");
         str=dis.readLine();
         n=Integer.parseInt(str);
         System.out.print("\n n=" +n);
      }
      catch(Exception e)
      {
         System.out.print("\n Error :" +e);
      }
   }
}
```

Bin>javac bytestream5.java

Bin>java bytestream5

```
Enter no :123

n=123
```

Explanation:

In the above program, the **DataInputStream** object **dis** is created to read values from input stream object **System.in** (keyboard). The **readLine()** method reads a line to string from keyboard and assigns to **str** which is then converts to integer and assigns to **n**. Finally, **n** is printed on screen as shown in the above output.

DataOutputStream class

This class is an abstract class of **FilterOutputStream** class. The methods of **DataOutputStream** class can be used to write primitive Java data types(**int**, **float**, **double** ect,.) to an output stream in a portable way. An application can then use a data input stream to read the data back in.

The following table shows the constructors of **DataOutputStream** class

Constructors	Meaning
DataOutputStream(OutputStream out)	Creates a new data output stream to write data to the specified underlying output stream.

The following table shows the methods of **DataOutputStream** class

Methods	Meaning
void **flush**()	Flushes this data output stream.
int **size**()	Returns the current value of the counter written, the number of bytes written to this data output stream so far.
void **write**(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to the underlying output stream.
void **write**(int b)	Writes the specified byte (the low eight bits of the argument b) to the underlying output stream.
void **writeBoolean**(boolean v)	Writes a boolean to the underlying output stream as a 1-byte value.
void **writeByte**(int v)	Writes out a byte to the underlying output stream as a 1-byte value.
void **writeBytes**(String s)	Writes out the string to the underlying output stream as a sequence of bytes.

void **writeChar**(int v)	Writes a char to the underlying output stream as a 2-byte value, high byte first.
void **writeChars**(String s)	Writes a string to the underlying output stream as a sequence of characters.
void **writeDouble**(double v)	Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.
void **writeFloat**(float v)	Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.
void **writeInt**(int v)	Writes an int to the underlying output stream as four bytes, high byte first.
void **writeLong**(long v)	Writes a long to the underlying output stream as eight bytes, high byte first.
void **writeShort**(int v)	Writes a short to the underlying output stream as two bytes, high byte first.
void **writeUTF**(String str)	Writes a string to the underlying output stream using modified UTF-8 encoding in a machine-independent manner.

Class: SequenceInputStream

This class is a subclass of **InputStream** class. A **SequenceInputStream** represents the logical concatenation of other input streams. It starts out with an ordered collection of input streams and reads from the first one until end of file is reached, whereupon it reads from the second one, and so on, until end of file is reached on the last of the contained input streams.

The following table shows constructors of **SequenceInputStream**

Constructors	Meaning
SequenceInputStream(Enumeration<? extends InputStream> e)	Initializes a newly created SequenceInputStream by remembering the argument, which must be an Enumeration that produces objects whose run-time type is InputStream.

SequenceInputStream(InputStream s1, InputStream s2) .	Initializes a newly created SequenceInputStream by remembering the two arguments, which will be read in order, first s1 and then s2, to provide the bytes to be read from this SequenceInputStream

The following table shows methods of **SequenceInputStream**

Methods	Meaning
int **available**()	Returns an estimate of the number of bytes that can be read (or skipped over) from the current underlying input stream without blocking by the next invocation of a method for the current underlying input stream.
void **close**()	Closes this input stream and releases any system resources associated with the stream.
int **read**()	Reads the next byte of data from this input stream.
int **read**(byte[] b, int off, int len)	Reads up to len bytes of data from this input stream into an array of bytes.

A program to demonstrate SequenceInputStream class
Bin>edit bytestream7.java

```
import java.io.*;
class bytestream7
{
  public static void main(String argv[]) throws IOException
  {
    try
    {
      FileInputStream f1=new FileInputStream("c:\\program
          files\\java\\jdk1.5.0_05\\bin\\a.txt");
      FileInputStream f2=new FileInputStream("c:\\
          program files\\java\\jdk1.5.0_05\\bin\\b.txt");
      SequenceInputStream sis=new SequenceInputStream(f1,f2);
```

```
            byte ch;
            while( (ch=(byte)sis.read()) !=-1)
            {
              System.out.print((char)ch);
            }
            sis.close();
            f1.close();
            f2.close();
        }
        catch(Exception e)
        {
            System.out.print("\n Error :" +e);
        }
    }
}
```

Bin>javac bytestream7.java
Bin>java bytestream7
This is a.txt file This is b.txt file

Explanation:

In the above program, **a.txt** file is opened in **f1** and **b.txt** file is opened in **f2**. The file objects are passed to **SequenceInputStream** object **sis**. This **sis** object reads data from two files and prints on screen as shown in above output.

File Handling

In java, there are several file stream classes are present which are used to read or write the data from or into files. Files can be byte stream files and character stream files. These file streams are of two types such as byte stream files and character file streams. Byte stream file classes can be used to write or read data in binary mode. Using byte stream file classes any type of data i.e. basic types or user-defined objects can be written or read from or to files. Where as character stream classes can be used to write or read only characters into or from files. Using character streams any language data can be stored into files. These file streams we discuss later in this same chapter.

Byte stream files

In byte stream files, two concrete classes such as **FileInputStream** and **FileOutputStream** are used to create or open files for writing or reading.

These classes are subclasses of **InputStream** and **OutputStream** abstract classes. The file classes overrides methods of these abstract classes to perform input and output operation with files.

FileOutputStream class

The **FileOutputStream** class can be used to create files for writing in bytes. Files are created using the constructors of this class. If file is unable to create then the constructors throws an exception **FileNotFoundException**. Therefore this exception should be caught using try and catch blocks.

The following table shows constructors of **FileOutputStream** class

Constructors	Meaning
FileOutputStream(File file) Creates	a file output stream to write to the file represented by the specified File object.
FileOutputStream(File file, boolean append)	Creates a file output stream to write to the file represented by the specified File object.
FileOutputStream(FileDescriptor fdObj)	Creates an output file stream to write to the specified file descriptor, which represents an existing connection to an actual file in the file system.
FileOutputStream(String name)	Creates an output file stream to write to the file with the specified name.
FileOutputStream(String name, boolean append)	Creates an output file stream to write to the file with the specified name.

The following table shows methods of **FileOutputStream** class

Methods	Meaning
void **close**()	Closes this file output stream and releases any system resources associated with this stream.
protected void **finalize**()	Cleans up the connection to the file, and ensures that the close method of this file output stream is called when there are no more references to this stream.

FileChannel **getChannel**()	Returns the unique FileChannel object associated with this file output stream.
FileDescriptor **getFD**()	Returns the file descriptor associated with this stream.
void **write**(byte[] b)	Writes b.length bytes from the specified byte array to this file output stream.
void **write**(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to this file output stream.
void **write**(int b)	Writes the specified byte to this file output stream.

A program to write data into file using FileOutputStream class in bytes mode.

Bin>edit file4.java

```
import java.io.*;
class  file4
{
  public static void main(String argv[]) throws Exception
  {
    try
    {
      FileOutputStream fos=new FileOutputStream("sample.txt");
      byte ch;
      System.out.print("\n type text…\n");
      while( (ch=(byte)System.in.read())!=-1)
      {
        fos.write(ch);
      }
      fos.close();
      System.out.print("\n file created.");
    }
    catch(Exception e)
    {
      System.out.print("\n error :"+e);
    }
  }
}
```

Bin>javac file4.java
Bin>java file4

```
C:\Program Files\Java\jdk1.5.0_05\bin>javac file4.java

C:\Program Files\Java\jdk1.5.0_05\bin>java file4

 type text:
this is sample file
apex computers, wgl.
^Z

 file created.
C:\Program Files\Java\jdk1.5.0_05\bin>
```

Explanation:

In the above program, **fos** object is created to **FileOuputStream** class that creates **sample.txt**. In the while loop, **System.in.read()** reads a character from keyboard and returns its ascii code which is them converts into **ch**. The **fos.write(ch);** writes one byte of data present in **ch** into file. The procedure continues until **F6**(End of file) key is typed.

If you open and see the sample.txt you see data in bytes. Open the file using type command.

Class : FileInputStream

The **FileInputStream** class can be used to open file for reading in byte mode. If the file is not found then the constructors throws an **FileNotFoundException**. Therefore the code should be placed in **try** and **catch** block.

The following table shows constructors of **FileInputStream** class

Constructors	Meaning
FileInputStream(File file)	Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.
FileInputStream(FileDescriptor fdObj)	Creates a FileInputStream by using the file descriptor fdObj, which represents an existing connection to an actual file in the file system.
FileInputStream(String na me)	Creates a FileInputStream by opening a connection to an actual file, the file named by the path name name in the file system.

The following table shows methods of **FileInputStream** class

Methods	Meaning
int **available**()	Returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
void **close**()	Closes this file input stream and releases any system resources associated with the stream.
protected void **finalize**()	Ensures that the close method of this file input stream is called when there are no more references to it.
FileChannel **getChannel**()	Returns the unique FileChannel object associated with this file input stream.
FileDescriptor **getFD**()	Returns the FileDescriptor object that represents the connection to the actual file in the file system being used by this FileInputStream.
int **read**()	Reads a byte of data from this input stream.
int **read**(byte[] b)	Reads up to b.length bytes of data from this input stream into an array of bytes.
int **read**(byte[] b, int off, int len)	Reads up to len bytes of data from this input stream into an array of bytes.
long **skip**(long n)	Skips over and discards n bytes of data from the input stream.

A program to read bytes from file using FileInputStream class and prints on monitor as characters.

Bin>edit file5.java

```
import java.io.*;
class file5
{
  public static void main(String argv[]) throws IOException
  {
    try
    {
      FileInputStream fin=new FileInputStream("sample.txt");
      byte ch;
      while((ch=(byte)fin.read())!=-1)
      {
        System.out.print((char)ch);
      }
      fin.close();
    }
    catch(Exception e)
    {
      System.out.print("\nError :"+e);
    }
  }
}
```

Bin>javac file5.java

Bin>java file5

this is sample file
apex computers, wgl.

Explanation:

In the above program, fin object is loaded with sample.txt for reading. The while loop reads one byte after the other from the file and prints on the screen until reaching end of file. We get the output as shown above.

Serialization

We can write/read basic type values into/from files. But in some situations we want to write objects into file. This is useful when we want to save the state of program into file. This can be achieved in java using Serialization concept.

Serialization is the concept of saving the current state of an object into a storage media such as file, so that when required the same object can be reconstructed and this is known as de-Serialization. The objects contents are stored into a file such as variable names, types and values.

The objects whose classes are implemented from **java.io.Serializable** or **java.io.Externalizable** interface can only be written into file using serialization concept otherwise the jvm throws an exception.

Serializable interface

Serializable interface is defined in **java.io** package. The classes that implements this interface whose objects can only be serialized and de-serialized. This interface is an empty interface which does not contain fields or methods. Therefore the sub-classes need not to override any methods. This interface is used for identification to the jvm that its sub-classes can be serialized or de-serialized.

If the object contains **static** or **transient** variables then they are not saved into file by the serialization.

Serialized objects can be written into file using **ObjectOutputStream** and can be read using **ObjectInputStream** classes.

ObjectOutputStream class

An **ObjectOutputStream** class is inherited from **OutputStream** and implements from **ObjectOutput** interface. This class can be used to write primitive data types and graphs of Java objects to a file. The objects can be read (reconstituted) using an **ObjectInputStream**. Only objects that support the **java.io.Serializable** or **java.io.Externalizable** interface can be written to files. The class of each serializable object is encoded including the class name and signature of the class, the values of the object's fields and arrays, and the closure of any other objects referenced from the initial objects.

The method **writeObject()** method is used to write an object to the file stream. Any object, including strings and arrays, is written with **writeObject()** method. Multiple objects or primitives can be written to the file. The objects must be read back from the corresponding **ObjectInputstream** with the same types and in the same order as they were written.

The following table shows constructors of **ObjectOuptutStream** class

Constructors	Meaning
protected **ObjectOutputStream**()	Provide a way for subclasses that are completely reimplementing ObjectOutputStream to not have to allocate private data just used by this implementation of ObjectOutputStream.
ObjectOutputStream(OutputStream out)	Creates an ObjectOutputStream that writes to the specified OutputStream.

The following table shows methods **of ObjectOuptutStream** class

Methods	Meaning
void **close**()	Closes the stream.
void **defaultWriteObject**()	Write the non-static and non-transient fields of the current class to this stream.
protected boolean **enableReplaceObject**(boolean enable)	Enable the stream to do replacement of objects in the stream.
void **flush**()	Flushes the stream.
protected Object **replaceObject**(Object obj)	This method will allow trusted subclasses of ObjectOutputStream to substitute one object for another during serialization.
void **reset**()	Reset will disregard the state of any objects already written to the stream.
void **write**(byte[] buf)	Writes an array of bytes.
void **write**(byte[] buf, int off, int len)	Writes a sub array of bytes.
void **write**(int val)	Writes a byte.
void **writeBoolean**(boolean val)	Writes a boolean.
void **writeByte**(int val)	Writes an 8 bit byte.
void **writeBytes**(String str)	Writes a String as a sequence of bytes.
void **writeChar**(int val)	Writes a 16 bit char.
void **writeChars**(String str)	Writes a String as a sequence of chars.

protected void **writeClassDescriptor**(ObjectStreamClass desc)	Write the specified class descriptor to the ObjectOutputStream.
void **writeDouble**(double val)	Writes a 64 bit double.
void **writeFields**()	Write the buffered fields to the stream.
void **writeFloat**(float val)	Writes a 32 bit float.
void **writeInt**(int val)	Writes a 32 bit int.
void **writeLong**(long val)	Writes a 64 bit long.
void **writeObject**(Object obj)	Write the specified object to the ObjectOutputStream.
void **writeShort**(int val)	Writes a 16 bit short.
void **writeUnshared**(Object obj)	Writes an "unshared" object to the ObjectOutputStream.
void **writeUTF**(String str)	Primitive data write of this String in modified UTF-8 format.

ObjectInputStream class

An **ObjectInputStream** deserializes primitive data and objects previously written using an **ObjectOutputStream**.

ObjectInputStream is used to recover those objects previously serialized. Other uses include passing objects between hosts using a socket stream or for marshaling and unmarshaling arguments and parameters in a remote communication system.

ObjectInputStream ensures that the types of all objects in the graph created from the stream match the classes present in the Java Virtual Machine. Classes are loaded as required using the standard mechanisms. Only objects that support the **java.io.Serializable** or **java.io.Externalizable** interface can be read from streams.

The following table shows constructors of **ObjectInputStream** class

Constructors	Meaning
protected **ObjectInputStream**()	Provide a way for subclasses that are completely reimplementing ObjectInputStream to not have to allocate private data just used by this implementation of ObjectInputStream.

ObjectInputStream(InputStream in)	Creates an ObjectInputStream that reads from the specified InputStream.

The following table shows methods of **ObjectInputStream** class

Methods	Meaning
int **available**()	Returns the number of bytes that can be read without blocking.
void **close**()	Closes the input stream.
protected boolean **enableResolveObject**(boolean enable)	Enable the stream to allow objects read from the stream to be replaced.
int **read**()	Reads a byte of data.
int **read**(byte[] buf, int off, int len)	Reads into an array of bytes.
boolean **readBoolean**()	Reads in a boolean.
byte **readByte**()	Reads an 8 bit byte.
char **readChar**()	Reads a 16 bit char.
double **readDouble**()	Reads a 64 bit double.
float **readFloat**()	Reads a 32 bit float.
void **readFully**(byte[] buf)	Reads bytes, blocking until all bytes are read.
void **readFully**(byte[] buf, int off, int len)	Reads bytes, blocking until all bytes are read.
int **readInt**()	Reads a 32 bit int.
long **readLong**()	Reads a 64 bit long.
Object **readObject**()	Read an object from the ObjectInputStream.
short **readShort**()	Reads a 16 bit short.
int **skipBytes**(int len)	Skips bytes.

A program to write objects into file using ObjectOutputStream class(Serialization).

Bin>edit serial.java

```
import java.io.*;
class bank implements Serializable
{
  private String name;
  private int accno;
  public  void setbank(String n,int a)
  {
    name=n;
    accno=a;
  }
  public void showbank()
  {
    System.out.print("\nName="+name+"\t Accno="+accno);
  }
}
class serial
{
  public static void main(String argv[])
  {
    try
    {
      bank b1,b2;
      b1=new bank();
      b2=new bank();
      b1.setbank("kumar",105);
      b2.setbank("sai",106);
      FileOutputStream fout=new FileOutputStream("bank.dbf");
      ObjectOutputStream out=new ObjectOutputStream(fout);
      out.writeObject(b1);
      out.writeObject(b2);
      out.close();
      fout.close();
      System.out.print("\nobjects written into file.");
    }
    catch(Exception e)
    {
      System.out.print("\nError :"+e);
    }
  }
}
```

Bin>javac serial.java
Bin>java serial
objects written into file.

Explanation:

In the above program, **b1** and **b2** objects are created to **bank** class. **b1** object stores with **"kumar"** and **105**, **b2** object stores with **"sai"** and **106**. The **FileOutputStream** object **fout** is opened with **"bank.dbf"** file. This **fout** object is passed to **ObjectOutputStream** object **out**. The **write()** methods writes two objects **b1** and **b2** to file. Finally, the stream objects are closed.

To see what is present in **stud.dbf** file, use the type command as shown below and data will be displayed in binary language.

Bin>type stud.dbf

A program to read objects from file (de-serialization)
Bin>edit serial2.java

```
import java.io.*;
class bank implements Serializable
{
  private String name;
  private int accno;
  public  void setbank(String n,int a)
  {
    name=n;
    accno=a;
  }
  public void showbank()
  {
    System.out.print("\nName="+name+"\t Accno="+accno);
  }
}
```

```
class serial2
{
   public static void main(String argv[])
   {
      try
      {
         FileInputStream fin=new FileInputStream("bank.dbf");
         ObjectInputStream in=new ObjectInputStream(fin);
         bank ob1,ob2;
         ob1=(bank)in.readObject();
         ob2=(bank)in.readObject();
         System.out.print("\nob1...");ob1.showbank();
         System.out.print("\nob2...");ob2.showbank();
      }
      catch(Exception e)
      {
         System.out.print("\nError :"+e);
      }
   }
}
```

Bin>javac serial2.java
Bin>java serial2

```
ob1...
Name=kumar      Accno=105
ob2...
Name=sai        Accno=106
```

Explanation:

In the above program, the **FileInputStream** object **fin** is opened with **stud.dbf** file and this **fin** object is passed to **ObjectOutputStream** object **in**. The statement **in.readObject()** methods reads objects from the file and is type cased to **bank** type and assigns to **ob1** and **ob2**. The **ob1.showbank()** and **ob2.showbank()** displays the details of objects as shown above.

Class RandomAccessFile

The **RandamAccessFile** class is implemented from **DataOutput**, **DataInput** and Closeable interfaces. This class can be used to read or write to a random access file. A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the ***file pointer***; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read. If the random access file is created in read/write mode, then output operations are also available; output operations write bytes starting at the file pointer and advance the file pointer past the bytes written. Output operations that write past the current end of the implied array cause the array to be extended. The file pointer can be read by the **getFilePointer()** method and set by the seek method.

If the reading methods successfully reads the data from file then the methods returns true, and false if fail to read desired number of bytes from file. An **EOFException** (which is a kind of **IOException)** is thrown on reaching end of file. If any byte cannot be read for any reason other than end-of-file, an **IOException** other than **EOFException** is thrown. In particular, an **IOException** may be thrown if the stream has been closed.

The following table shows the constructors of **RandomAccessFile** class.

Constructors	Meaning
RandomAccessFile(File file, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode) Creates a random access file stream to read from, and optionally to write to, a file with the specified name.	

The following table shows the methods of **RandomAccessFile** class.

Methods	Meaning
void **close**()	Closes this random access file stream and releases any system resources associated with the stream.
FileDescriptor **getFD**()	Returns the opaque file descriptor object associated with this stream.

long **getFilePointer**()	Returns the current offset in this file.
long **length**()	Returns the length of this file.
int **read**()	Reads a byte of data from this file.
int **read**(byte[] b)	Reads up to b.length bytes of data from this file into an array of bytes.
int **read**(byte[] b, int off, int len)	Reads up to len bytes of data from this file into an array of bytes.
boolean **readBoolean**()	Reads a boolean from this file.
byte **readByte**()	Reads a signed eight-bit value from this file.
char **readChar**()	Reads a character from this file.
double **readDouble**()	Reads a double from this file.
float **readFloat**()	Reads a float from this file.
void **readFully**(byte[] b)	Reads b.length bytes from this file into the byte array, starting at the current file pointer.
void **readFully**(byte[] b, int off, int len)	Reads exactly len bytes from this file into the byte array, starting at the current file pointer.
int **readInt**()	Reads a signed 32-bit integer from this file.
String **readLine**()	Reads the next line of text from this file.
long **readLong**()	Reads a signed 64-bit integer from this file.
short **readShort**()	Reads a signed 16-bit number from this file.
int **readUnsignedByte**()	Reads an unsigned eight-bit number from this file.
int **readUnsignedShort**()	Reads an unsigned 16-bit number from this file.
void **seek**(long pos)	Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void **setLength**(long newLength)	Sets the length of this file.
int **skipBytes**(int n)	Attempts to skip over n bytes of input discarding the skipped bytes.

void **write**(byte[] b)	Writes b.length bytes from the specified byte array to this file, starting at the current file pointer.
void **write**(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to this file.
void **write**(int b)	Writes the specified byte to this file.
void **writeBoolean**(boolean v)	Writes a boolean to the file as a one-byte value.
void **writeByte**(int v)	Writes a byte to the file as a one-byte value.
void **writeBytes**(String s)	Writes the string to the file as a sequence of bytes.
void **writeChar**(int v)	Writes a char to the file as a two-byte value, high byte first.
void **writeChars**(String s)	Writes a string to the file as a sequence of characters.
void **writeDouble**(double v)	Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
void **writeFloat**(float v)	Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
void **writeInt**(int v)	Writes an int to the file as four bytes, high byte first.
void **writeLong**(long v)	Writes a long to the file as eight bytes, high byte first.
void **writeShort**(int v)	Writes a short to the file as two bytes, high byte first.

Constructors Description

RandomAccessFile(String name,String mode)throws FileNotFoundException

Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

The mode argument specifies the access mode with which the file is to be opened. The permitted values and their meanings are:

Value	Meaning
"r"	Open for reading only. Invoking any of the write methods of the resulting object will cause an IOException to be thrown.
"rw"	Open for reading and writing. If the file does not already exist then an attempt will be made to create it.
"rws"	Open for reading and writing, as with "rw", and also require that every update to the file's content or metadata be written synchronously to the underlying storage device.
"rwd"	Open for reading and writing, as with "rw", and also require that every update to the file's content be written synchronously to the underlying storage device.

RandomAccessFile(File file,String mode)throws FileNotFoundException

Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.

The mode argument specifies the access mode in which the file is to be opened. The permitted values and their meanings are as specified for the above constructor.

A Program to read data randomly from file
Bin>edit randfile.java

```
import java.io.*;
class randfile
{
  public static void main(String argv[])throws IOException
  {
    try
    {
      RandomAccessFile fin=new RandomAccessFile
          ("sample.txt","r");
      byte ch;
      fin.seek(10);
```

```
                while((ch=(byte)fin.read())!=-1)
                {
                        System.out.print((char)ch);
                }
                fin.close();
        }
        catch(Exception e)
        {
        System.out.print("\nError :"+e);
        }
    }
}
```

Bin>javac randfile.java
Bin>java randfile

mple file
apex computers, wgl.

Explanation:

In the above program, **fin** object is opened with **sample.txt** file for reading. The **seek()** moves the file pointer to10th byte in the file. The while reads the data from the **10th** byte till end of file and we get the output as shown above.

Character Streams

In the previous section we have seen about Byte streams which handle the data in binary form, which is efficient for data processing. The character stream deals with the text. The characters are stored and retrieved in a human readable form. Using character streams any language data that can be stored or retrieved(uni-code).

Java has two abstract classes **Reader** and **Writer**. The **Reader** class deals with reading of streaming character and **Writer** class deals with writing of streaming character.

Reader class

Abstract class for reading character streams. The only methods that a subclass must implement are read(**char[]**, **int**, **int**) and **close()**. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

The following table shows constructors of **Reader** class

Constructors	Meaning
protected **Reader**()	Creates a new character-stream reader whose critical sections will synchronize on the reader itself.
protected **Reader**(Object lock)	Creates a new character-stream reader whose critical sections will synchronize on the given object.

Methods	Meaning
abstract void **close**()	Closes the stream and releases any system resources associated with it.
void **mark**(int readAheadLimit)	Marks the present position in the stream.
boolean **markSupported**()	Tells whether this stream supports the mark() operation.
int **read**()	Reads a single character.
int **read**(char[] cbuf)	Reads characters into an array.
abstract int **read**(char[] cbuf, int off, int len)	Reads characters into a portion of an array.
int **read**(CharBuffer target)	Attempts to read characters into the specified character buffer.
boolean **ready**()	Tells whether this stream is ready to be read.
void **reset**()	Resets the stream.
long **skip**(long n)	Skips characters.

Writer class

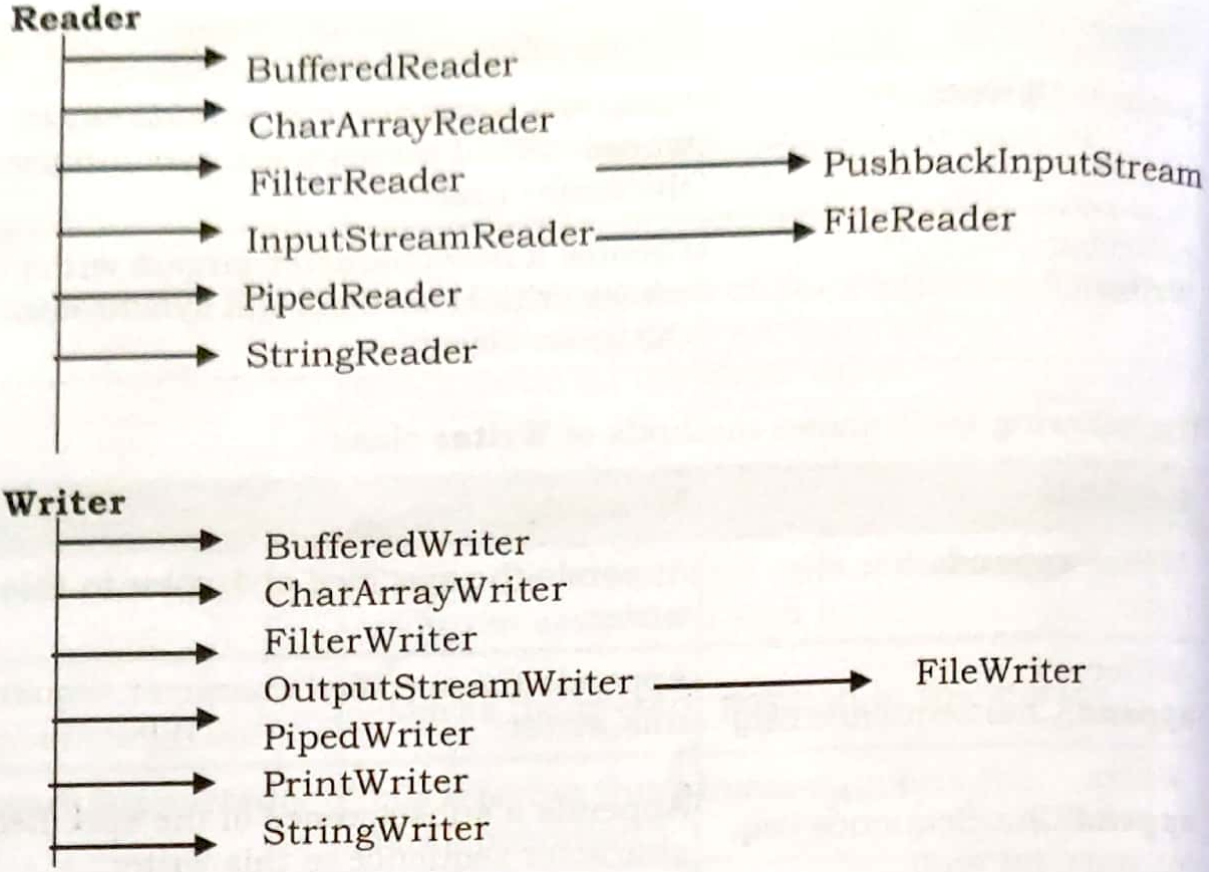
Abstract class for writing to character streams. The only methods that a subclass must implement are write(char[], int, int), flush(), and close(). Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

The following table shows constructors of **Writer** class

Constructors	Meaning
protected**Writer**()	Creates a new character-stream writer whose critical sections will synchronize on the writer itself.
protected **Writer**(Object lock)	Creates a new character-stream writer whose critical sections will synchronize on the given object.

The following table shows methods of **Writer** class

Methods	Meaning
Writer **append**(char c)	Appends the specified character to this writer.
Writer **append**(CharSequence csq)	Appends the specified character sequence to this writer.
Writer **append**(CharSequence csq, int start, int end)	Appends a subsequence of the specified character sequence to this writer.
abstract void **close**()	Closes the stream, flushing it first.
abstract void **flush**()	Flushes the stream.
void **write**(char[] cbuf)	Writes an array of characters.
abstract void **write**(char[] cbuf, int off, int len)	Writes a portion of an array of characters.
void **write**(int c)	Writes a single character.
void **write**(String str)	Writes a string.
void **write**(String str, int off, int len)	Writes a portion of a string.



Class BufferedReader

The **BufferedReader** class extends from Reader class. This class methods can be used read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a **BufferedReader** around any Reader whose read() operations may be costly, such as **FileReaders** and **InputStreamReaders**. For example,

BufferedReader in = new BufferedReader(new FileReader("a.java"));

will buffer the input from the specified file. Without buffering, each invocation of **read()** or **readLine()** could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

The following table shows constructors of **BufferedReader** class

Constructors	Meaning
BufferedReader(Reader in) Creates	a buffering character-input stream that uses a default-sized input buffer.
BufferedReader(Reader in, int sz)	Creates a buffering character-input stream that uses an input buffer of the specified size.

The following table shows methods of **BufferedReader** class

Methods	Meaning
void **close**()	Closes the stream and releases any system resources associated with it.
void **mark**(int readAheadLimit)	Marks the present position in the stream.
boolean **markSupported**()	Tells whether this stream supports the mark() operation, which it does.
int **read**()	Reads a single character.
int **read**(char[] cbuf, int off, int len)	Reads characters into a portion of an array.
String **readLine**()	Reads a line of text.
boolean **ready**()	Tells whether this stream is ready to be read.
void **reset**()	Resets the stream to the most recent mark.
long **skip**(long n)	Skips characters.

Class BufferedWriter

The **BufferedWriter** class extends from Writer abstract class. This class writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.

The buffer size may be specified, or the default size may be accepted. The default is large enough for most purposes.

The following table shows constructors of **BufferedWriter** class.

Constructors	Meaning
BufferedWriter(Writer out)	Creates a buffered character-output stream that uses a default-sized output buffer.
BufferedWriter(Writer out, int sz)	Creates a new buffered character-output stream that uses an output buffer of the given size.

The following table shows methods of **BufferedWriter** class.

Methods	Meaning
void **close**()	Closes the stream, flushing it first.
void **flush**()	Flushes the stream.
void **newLine**()	Writes a line separator.
void **write**(char[] cbuf, int off, int len)	Writes a portion of an array of characters.
void **write**(int c)	Writes a single character.
void **write**(String s, int off, int len)	Writes a portion of a String.

Class InputStreamReader

The **InputStreamReader** class is extended from Reader abstract class. This class is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

Each invocation of one of an **InputStreamReader's read()** methods may cause one or more bytes to be read from the underlying byte-input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation.

For top efficiency, consider wrapping an **InputStreamReader** within a BufferedReader. For example:

```
BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
```

The following table shows the constructors of **InputStreamReader** class.

Constructors	Meaning
InputStreamReader(InputStream in)	Creates an InputStreamReader that uses the default charset.
InputStreamReader(InputStream in, Charset cs)	Creates an InputStreamReader that uses the given charset.
InputStreamReader(InputStream in, CharsetDecoder dec)	Creates an InputStreamReader that uses the given charset decoder.
InputStreamReader(InputStream in, String charsetName)	Creates an InputStreamReader that uses the named charset.

The following table shows the methods of **InputStreamReader** class.

Method	Meaning
void **close**()	Closes the stream and releases any system resources associated with it.
String **getEncoding**()	Returns the name of the character encoding being used by this stream.
int **read**()	Reads a single character.
int **read**(char[] cbuf, int offset, int length)	Reads characters into a portion of an array.
boolean **ready**()	Tells whether this stream is ready to be read.

A program that reads two numbers from keyboard and prints its addition.

Bin>edit readkeyboard.java

```
import java.io.*;
class readkeyboard
{
  public static void main(String argv[])throws IOException
  {
```

```
        int a=0,b=0,c=0;
        InputStreamReader isr=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(isr);
        try
        {
           System.out.print("\n Enter 2 numbers :");
           a=Integer.parseInt(br.readLine());
           b=Integer.parseInt(br.readLine());
           c=a+b;
           System.out.print("\nc="+c);
        }
        catch(NumberFormatException e)
        {
           System.out.print("\nError :"+e);
        }
    }
}
```

Bin>javac readkeyboard.java
Bin>java readkeyboard

```
 Enter 2 numbers :100
200

c=300
```

Explanation:

In the above program, the **InputStreamReader** class object **isr** is instantiated to **System.in** which is inputstream object of keyboard. The **isr** object reads values from keyboard. The **br** object of **BufferedReader** class reads data from the input stream reader object **isr**. The **readLine()** method of **BufferedReader** reads data from buffer which exist in string is converted to integer by **Integer.parse()** method. Values reads from keyboard stores into **a** and **b** variables and stores their addition into **c**. Finally, prints **c** on the monitor.

Class OutputStreamWriter

The **OutputStreamWriter** class extends from Writer abstract class. The **OutputStreamWriter** is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

Each invocation of a **write()** method causes the encoding converter to be invoked on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream. The size of this buffer may be specified, but by default it is large enough for most purposes. Note that the characters passed to the **write()** methods are not buffered.

For top efficiency, consider wrapping an **OutputStreamWriter** within a **BufferedWriter** so as to avoid frequent converter invocations. For example:

```
Writer out = new BufferedWriter(new OutputStreamWriter(System.out));
```

Constructors	Meaning
OutputStreamWriter(OutputStream out)	Creates an OutputStreamWriter that uses the default character encoding.
OutputStreamWriter(OutputStream out, Charset cs)	Creates an OutputStreamWriter that uses the given charset.
OutputStreamWriter(OutputStream out, CharsetEncoder enc)	Creates an OutputStreamWriter that uses the given charset encoder.
OutputStreamWriter(OutputStream out, String charsetName)	Creates an OutputStreamWriter that uses the named charset.

Methods	Meaning
void **close**()	Closes the stream, flushing it first.
void **flush**()	Flushes the stream.
String **getEncoding**()	Returns the name of the character encoding being used by this stream.
void **write**(char[] cbuf, int off, int len)	Writes a portion of an array of characters.
void **write**(int c)	Writes a single character.
void **write**(String str, int off, int len)	Writes a portion of a string.

Class FileWriter

The **FileWriter** class extends from **OutputStreamWriter** class which inturn extends from Writer abstract class. Therefore the methods of super class are available to **FileWriter** class. This class can be used for writing the text into file in character stream.

The following table shows the constructors of **FileWriter** class

Constructors	Meaning
FileWriter(File file)	Constructs a FileWriter object given a File object.
FileWriter(File file, boolean append)	Constructs a FileWriter object given a File object.
FileWriter(FileDescriptor fd)	Constructs a FileWriter object associated with a file descriptor.
FileWriter(String fileName)	Constructs a FileWriter object given a file name.
FileWriter(String fileName, boolean append)	Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

Note: Methods of **FileWriter** are same as **OutputStreamWriter** and **Writer** classes.

A program to write text into file using FileWriter class.

Bin>edit file6.java

```
import java.io.*;
class  file6
{
   public static void main(String argv[]) throws Exception

   {
         try
         {
               FileWriter fout=new FileWriter("myfile.txt");
               int ch;
               System.out.print("\n type text:\n");
               while( (ch=System.in.read())!=-1)
               {
                     fout.write(ch);
```

```
                }
                fout.close();
                System.out.print("\n file created.");
            }
            catch(Exception e)
            {
                System.out.print("\n error :"+e);
            }
        }
    }
```

Bin>javac file6.java
Bin>java file6

```
type text:
Apex computers, warangal
It is one of the computer institutes.
^Z

file created.
```

To see the text in the file use the type command as:
Bin> type myfile.txt

```
Apex computers, warangal
It is one of the computer institutes.
```

Class FileReader

The **FileReader** class extends from **InputStreamReader** class which intrun inherited form Reader. Therefore methods of super classes are available to **FileReader** class. This class can be used to read characters from files.

The following table shows the constructors of **FileWriter** class

Constructors	Meaning
FileReader(File file)	Creates a new FileReader, given the File to read from.
FileReader(FileDescriptor fd)	Creates a new FileReader, given the FileDescriptor to read from.
FileReader(String filename)	Creates a new FileReader, given the name of the file to read from.

Note: Methods of **FileReader** class are same as **InputStreamReader** and **Reader** classes.

A program to read text from file using FileReader class.

Bin>edit file7.java

```
import java.io.*;
class  file7
{
      public static void main(String argv[]) throws Exception
      {
            try
            {
                  FileReader fin=new FileReader("A.txt");
                  int ch;
                  while( (ch=fin.read())!=-1)
                  {
                        System.out.print((char)ch);
                  }
                  fin.close();
            }
            catch(Exception e)
            {
                  System.out.print("\n error :"+e);
            }
      }
}
```

Bin>javac file7.java

Bin>java file7

```
this is a.txt file
```

Class CharArrayWriter

The **CharArrayWriter** class extends from **Writer** abstract class. This class implements a character buffer that can be used as an Writer. The buffer automatically grows when data is written to the stream. The data can be retrieved using **toCharArray()** and **toString()**.

Note: Invoking **close()** on this class has no effect, and methods of this class can be called after the stream has closed without generating an **IOException**.

The following table shows the constructors of **CharArrayWriter** class.

Constructors	Meaning
CharArrayWriter()	Creates a new CharArrayWriter.
CharArrayWriter(int initialSize)	Creates a new CharArrayWriter with the specified initial size.

The following table shows the methods of **CharArrayWriter** class.

Methods Meaning	
CharArrayWriter **append**(char c)	Appends the specified character to this writer.
CharArrayWriter **append**(CharSequence csq)	Appends the specified character sequence to this writer.
CharArrayWriter **append**(CharSequence csq, int start, int end)	Appends a subsequence of the specified character sequence to this writer.
void **close**()	Close the stream.
void **flush**()	Flush the stream.
void **reset**()	Resets the buffer so that you can use it again without throwing away the already allocated buffer.
int **size**()	Returns the current size of the buffer.
char[]**toCharArray**()	Returns a copy of the input data.
String **toString**()	Converts input data to a string.
void **write**(char[] c, int off, int len)	Writes characters to the buffer.
void **write**(int c)	Writes a character to the buffer.
void **write**(String str, int off, int len)	Write a portion of a string to the buffer.
void **writeTo**(Writer out)	Writes the contents of the buffer to another character stream.

Bin>edit charstream1.java

```
import java.io.*;
class charstream1
{
  public static void main(String argv[])throws IOException
  {
    String str="apex computers warangal";
    CharArrayWriter obj=new CharArrayWriter();
    obj.write(str,0,str.length());
    System.out.print("\n Buffer as string :" +
                                    obj.toString());
    char c[]=obj.toCharArray();
    System.out.print("\nc as string :");
    for(int i=0;i<obj.size();i++)
      System.out.print(c[i]);
  }
}
```

Bin>javac charstream1.java
Bin>java charstream1

```
 Buffer as string :apex computers warangal
c as string :apex computers warangal
```

Class CharArrayReader

The **CharArrayReader** class extends from **Reader** abstract class. This class implements a character buffer that can be used as a character-input stream.

The following table shows constructors of **CharArrayReader** class.

Constructors	Meaning
CharArrayReader(char[] buf)	Creates a CharArrayReader from the specified array of chars.
CharArrayReader(char[] buf, int offset, int length)	Creates a CharArrayReader from the specified array of chars.

The following table shows methods of **CharArrayReader** class.

Methods	Meaning
void **close**()	Closes the stream and releases any system resources associated with it.
void **mark**(int readAheadLimit)	Marks the present position in the stream.
boolean **markSupported**()	Tells whether this stream supports the mark() operation, which it does.
int **read**()	Reads a single character.
int **read**(char[] b, int off, int len)	Reads characters into a portion of an array.
boolean **ready**()	Tells whether this stream is ready to be read.
void **reset**()	Resets the stream to the most recent mark, or to the beginning if it has never been marked.
long **skip**(long n)	Skips characters.

A program to demonstrate CharArrayReader class

Bin>edit charstream2.java

```
import java.io.*;
class charstream2
{
  public static void main(String argv[ ])throws IOException
  {
    String str="Apex computers, warangal.";
    char ch[]=str.toCharArray();
    CharArrayReader car=new CharArrayReader(ch);
    int c;
    while( (c=car.read())!=-1 )
    {
      System.out.print((char)c);
    }
  }
}
```

```
Bin>javac charstream2.java
Bin>java charstream2
```
Apex computers, warangal.

Class PrintWriter

The PrintWriter class extends from Writer abstract class. This class prints formatted representations of objects to a text-output stream. This class implements all of the print methods found in **PrintStream**. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the PrintStream class, if automatic flushing is enabled it will be done only when one of the **println**, **printf**, or format methods is invoked, rather than whenever a newline character happens to be output.

Methods in this class never throw I/O exceptions, although some of its constructors may.

The following table shows constructors of **PrintWriter** class

Constructors	Meaning
PrintWriter(File file)	Creates a new PrintWriter, without automatic line flushing, with the specified file.
PrintWriter(File file, String csn)	Creates a new PrintWriter, without automatic line flushing, with the specified file and charset.
PrintWriter(OutputStream out)	Creates a new PrintWriter, without automatic line flushing, from an existing OutputStream.
PrintWriter(OutputStream out, boolean autoFlush)	Creates a new PrintWriter from an existing OutputStream.
PrintWriter(String fileName)	Creates a new PrintWriter, without automatic line flushing, with the specified file name.
PrintWriter(String fileName, String csn)	Creates a new PrintWriter, without automatic line flushing, with the specified file name and charset.
PrintWriter(Writer out)	Creates a new PrintWriter, without automatic line flushing.

PrintWriter(Writer out, boolean autoFlush)	Creates a new PrintWriter.

The following table shows methods of **PrintWriter** class

Methods	Meaning
PrintWriter **append**(char c)	Appends the specified character to this writer.
PrintWriter **append**(CharSequence csq)	Appends the specified character sequence to this writer.
PrintWriter **append**(CharSequence csq, int start, int end)	Appends a subsequence of the specified character sequence to this writer.
boolean **checkError**()	Flushes the stream if it's not closed and checks its error state.
protected void **clearError**()	Clears the error state of this stream.
void **close**()	Closes the stream and releases any system resources associated with it.
void **flush**()	Flushes the stream.
PrintWriter **format**(Locale l, String format, Object... args)	Writes a formatted string to this writer using the specified format string and arguments.
PrintWriter **format**(String format, Object... args)	Writes a formatted string to this writer using the specified format string and arguments.
void **print**(boolean b)	Prints a boolean value.
void **print**(char c)	Prints a character.
void **print**(char[] s)	Prints an array of characters.
void **print**(double d)	Prints a double-precision floating-point number.
void **print**(float f)	Prints a floating-point number.
void **print**(int i)	Prints an integer.
void **print**(long l)	Prints a long integer.
void **print**(Object obj)	Prints an object.
void **print**(String s)	Prints a string.

PrintWriter **printf**(String format, Object... args)	A convenience method to write a formatted string to this writer using the specified format string and arguments.
void **println**()	Terminates the current line by writing the line separator string.
void **println**(boolean x)	Prints a boolean value and then terminates the line.
void **println**(char x)	Prints a character and then terminates the line.
void **println**(char[] x)	Prints an array of characters and then terminates the line.
void **println**(double x)	Prints a double-precision floating-point number and then terminates the line.
void **println**(float x)	Prints a floating-point number and then terminates the line.
void **println**(int x)	Prints an integer and then terminates the line.
void **println**(long x)	Prints a long integer and then terminates the line.
void **println**(Object x)	Prints an Object and then terminates the line.
void **println**(String x)	Prints a String and then terminates the line.
protected void **setError**()	Indicates that an error has occurred.
void **write**(char[] buf)	Writes an array of characters.
void **write**(char[] buf, int off, int len)	Writes A Portion of an array of characters.
void **write**(int c)	Writes a single character.
void **write**(String s)	Writes a string.
void **write**(String s, int off, int len)	Writes a portion of a string.

A program to demonstrate PrintWriter class to print values on output stream (monitor)

Bin>edit print1.java

```
import java.io.*;
class print1
{
   public static void main(String argv[])
   {

         String name="kumar";
         int accno=105;
         double bal=10000.00;

         PrintWriter out=new PrintWriter(System.out);
         out.println("name="+name);
         out.println("accno="+accno);
         out.println("bal="+bal);
         out.close();

   }
}
```

Bin>javac print1.java
Bin>java print1

```
name=kumar
accno=105
bal=10000.00
```