ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

Object-Oriented Programming

File Handling in Java

Objectives

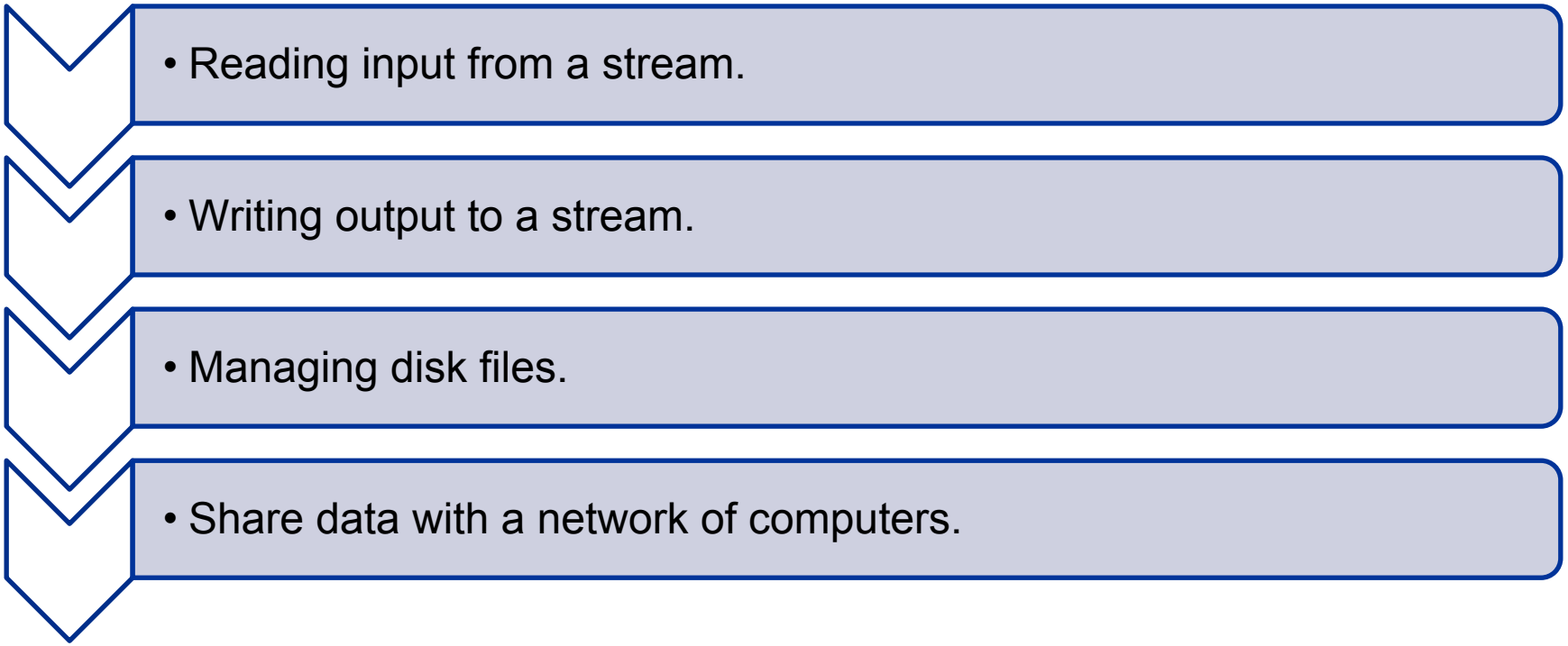
- Define data streams
- Identify the need for streams
- Identify the purpose of the File class, its constructors and methods
- Describe the DataInput and DataOutput interfaces
- Describe the byte stream and character stream in the java.io package
- Explain the InputStream and OutputStream classes
- Describe the BufferedInputStream and BufferedOutputStream classes
- Describe Character stream classes
- Describe the chaining of I/O systems
- Define Serialization and describe the need and purpose of Serialization

Stream Classes

- Java works with streams of data.
- A stream is a sequence of data or logical entity that produces or consumes information.
- A data stream is a channel through which data travels from a source to a destination.
- This source or destination can be an `input` or `output` device, storage media, or network computers.
- A physical file can be read using different types of streams, for example, `FileInputStream` or `FileReader`.
- Java uses such streams to perform various `input` and `output` operations.
- The standard `input/output` stream in Java is represented by three fields of the `System` class:
  - `in`: The standard input stream is used for reading characters of data.
  - `out`: The standard output stream is used to typically display the output on the screen or any other output medium.
  - `err`: This is the standard error stream.

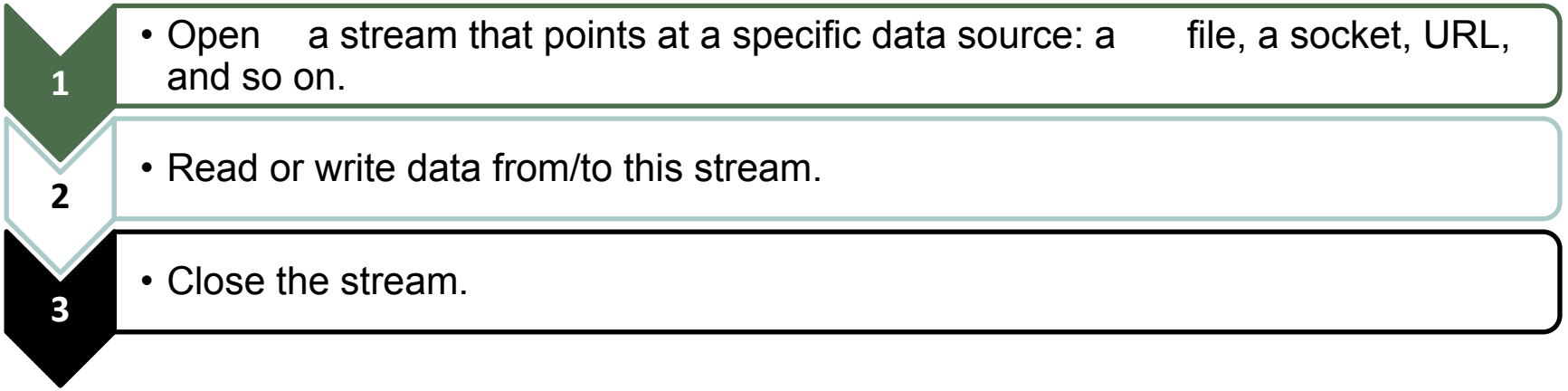
Need for Stream Classes

- In Java, streams are required to perform all the `input/output` (I/O) operations.
- Thus, `Stream` classes help in:



Steps for Using Stream Classes

- To read or write data using `Input/Output` streams, the following steps need to be performed. They are:



- `Input` and `Output` streams are abstract classes and are used for reading and writing of unstructured sequence of bytes.
- The other input and output streams are subclasses of the basic `Input` and `Output` stream class and are used for reading and writing to a file.
- The different types of byte streams can be used interchangeably as they inherit the structure of `Input/Output` stream class.
- For reading or writing bytes, a subclass of the `InputStream` or `OutputStream` class has to be used respectively.

File Class [1-2]

- `File` class directly works with files and the file system.
- The files are named using the file-naming conventions of the host operating system.
- These conventions are encapsulated using the `File` class constants.
- A pathname can be absolute or relative.
- In an absolute pathname, no other information is required in order to locate the required file as the pathname is complete.
- In a relative pathname, information is gathered from some other pathname.
- The classes in the `java.io` package resolve relative pathnames against the current user directory, which is named by the system property `user.dir`.

File Class [2-2]

- The directory methods in the `File` class allow creating, deleting, renaming, and listing of directories.
- The interfaces and classes defined by the `java.nio.file` package helps the Java virtual machine to access files, file systems, and file attributes.
- The `toPath()` method helps to obtain a Path that uses the abstract path. A `File` object uses this path to locate a file.
- The constructors of the `File` class are as follows:
  - `File(String dirpath)`
  - `File(String parent, String child)`
  - `File(File fileobj, String filename)`
  - `File(URL urlobj)`

Methods of File Class [1-4]

- The methods in `File` class help to manipulate the file on the file system.
- Some of the methods in the `File` class are:
  - `renameTo(File newname)`: Names the existing `File` object with the new name specified by the variable `newname`.
  - `delete()`: Deletes the file represented by the abstract path name.
  - `exists()`: Tests the existence of file or directory denoted by this abstract pathname.
  - `getPath()`: Converts the abstract pathname into a pathname string.
  - `isFile()`: Checks whether the file denoted by this abstract pathname is a normal file.
  - `createNewFile()`: Creates a new empty file whose name is the pathname for this file. It is only created when the file of similar name does not exist.
  - `mkdir()`: Creates the directory named by this abstract pathname.
  - `toPath()`: Returns a `java.nio.file.Path` object constructed from the abstract path.
  - `toURI()`: Constructs a file, URI. This file represents this abstract pathname.

Methods of File Class [2-4]

- The following Code Snippet displays the use of methods of the `File` class:

Code Snippet

```
. . .
File fileObj = new File("C:/Java/Hello.txt");
System.out.println("Path is: " +fileObj.getPath());
System.out.println("Name is: " +fileObj.getName());
System.out.println("File exists is: " +fileObj.exists());
System.out.println("File is: " +fileObj.isFile());
...
```

- Displays the full path and the filename of the invoking `File` object.
- Checks for the existence of the file and returns true if the file exists, false if it does not.
- `isFile()` method returns true if called on a file and returns false if called on a directory.

Methods of File Class [3-4]

The following Code Snippet displays the use of `FilenameFilter` class to filter files with a specific extension:

Code Snippet

```
import java.io.*;
class FileFilter implements FilenameFilter {
    String ext;
    public FileFilter(String ext) {
    this.ext = "." + ext;
    }
    public boolean accept (File dir, String fName) {
    return fName.endsWith(ext);
    }
}
public class DirList {
    public static void main (String [] args) {
    String dirName = "d:/resources";
```

Methods of File Class [4-4]

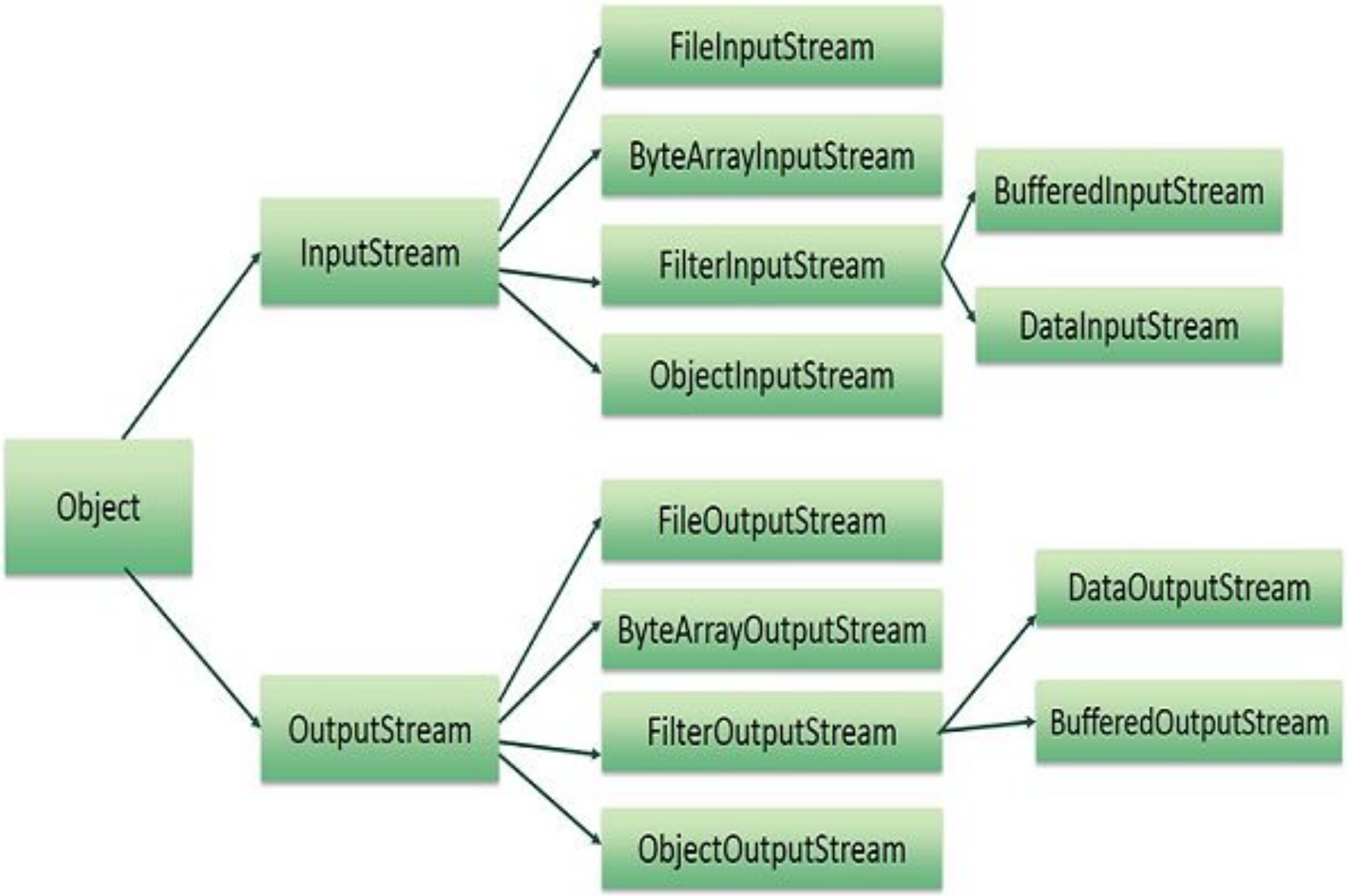
```
    File fileObj = new File ("d:/resources");
    FilenameFilter filterObj = new FileFilter("java");
    String[] fileName = fileObj.list(filterObj);
    System.out.println("Number of files found : " +
fileName.length);
    System.out.println("" );
    System.out.println("Names of the files are : " );
    System.out.println("------------------------- " );
    for(int ctr=0; ctr < fileName.length; ctr++) {
    System.out.println(fileName[ctr]);
    }
    }
}
```

`FileDescriptor` Class

- `FileDescriptor` class provides access to the file descriptors that are maintained by the OS when files and directories are being accessed.
- In practical use, a file descriptor is used to create a `FileInputStream` or `FileOutputStream` to contain it.
- File descriptors should not be created on their own by applications as they are tied to the operating system.
- The `FileDescriptor` class has the following public fields:
  - `static final FileDescriptor err`
  - `static final FileDescriptor in`
  - `static final FileDescriptor out`
- Following are the constructor and methods of `FileDescriptor`:
  - `FileDescriptor()`
  - `sync()`
  - `valid()`

DataInput Interface and DataOutput Interface

- Data stream supports input/output of primitive data types and string values. The data streams implement `DataInput` or `DataOutput` interface.
- The `DataInput` interface has methods for:
  - Reading bytes from a binary stream and convert the data to any of the Java primitive types.
  - Converting data from Java modified Unicode Transmission Format (UTF)-8 format into string form.
- The `DataOutput` interface has methods for:
  - Converting data present in Java primitive type into a series of bytes and write them onto a binary stream.
  - Converting string data into Java-modified UTF-8 format and write it into a stream.


Object
InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
OutputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream

Methods of DataInput Interface

- The methods in this interface are:
  - `readBoolean()`
  - `readByte()`
  - `readInt()`
  - `readDouble()`
  - `readChar()`
  - `readLine()`
  - `readUTF()`

Code Snippet

Code Snippet displays the use of `DataInput` interface:

```
try
{
DataInputStream dis = new DataInputStream(System.in);
double d = dis.readDouble();
int num = dis.readInt();
}
catch(IOException e) {}
. . .
```

Methods of DataOutput Interface

- The important methods in this interface are:
  - `writeBoolean(boolean b)`
  - `writeByte(int value)`
  - `writeInt(int value)`
  - `writeDouble(double value)`
  - `writeChar(int value)`
  - `writeChars(String value)`
  - `writeUTF(String value)`
- The following Code Snippet displays the use of `DataOutput` interface:

Code Snippet

```
try
{
outStream.writeBoolean(true);
outStream.writeDouble(9.95);
. . .
}
catch (IOException e) {}
. . .
```

java.io Package [1-7]

- A stream represents many sources and destinations, such as disk files and memory arrays.
- It is a sequence of data.
- An I/O Stream represents an input source or an output destination.
- Streams support many forms of data, such as simple bytes, primitive date type, localized characters and so on.
- Certain streams allow data to pass and certain streams transform the data in an useful way.
- However, all streams provide a simple model to programs to use them.
- A program uses an input stream to read data from a source. It reads one item at a time.

java.io Package [2-7]

The following figure illustrates the input stream model:

The following figure illustrates that a program uses an output stream to write data to a destination:

java.io Package [3-7]

The following Code Snippet displays the working of byte streams using the `FileInputStream` class and `FileOutputStream` class: **(đọc/ghi 1 byte)**

Code Snippet

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class ByteStreamApp {
public static void main(String[] args) throws IOException {
   FileInputStream inObj = null;
   FileOutputStream outObj = null;
   try {
    inObj = new FileInputStream("c:/java/hello.txt");
    outObj = new FileOutputStream("outagain.txt");
    int ch;
    while ((ch = inObj.read()) != -1) {
     outObj.write(ch);
```

java.io Package [4-7]

```
    }
   } finally {
     if (inObj != null) {
      inObj.close();
      }
      if (outObj != null) {
       outObj.close();
      }
     }
   }
  }
```

- In the Code Snippet, `read()` method:
  - Reads a character and returns an `int` value which indicates that the end of the stream is reached by returning a value of `-1`.

java.io Package [5-7]

- A program that uses character streams adapts to the local character set and is ready for internationalization.
- All character stream classes are derived from the `Reader` and `Writer` class.
- There are character stream classes that specialize in file I/O operations such as `FileReader` and `FileWriter`. **(đọc ghi 2 byte unicode)**
- The following Code Snippet displays the reading and writing of character streams using the `FileReader` and `FileWriter` class:

Code Snippet

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class CharStreamApp {
 public static void main(String[] args) throws IOException {
  FileReader inObjStream = null;
  FileWriter outObjStream = null;
```

java.io Package [6-7]

```
 try {
  inObjStream = new FileReader("c:/java/hello.txt");
  outObjStream = new FileWriter("charoutputagain.txt");
  int ch;
  while ((ch = inObjStream.read()) != -1) {
   outObjStream.write(ch);
  }
 } finally {
   if (inObjStream != null) {
    inObjStream.close();
   }
  }
}
```

- Character streams act as wrappers for byte streams.
- The character stream manages translation between characters and bytes and uses the byte stream to perform the physical I/O operations.

java.io Package [7-7]

- Character I/O typically occurs in bigger units than single characters, such as a line that includes a string of characters with a line terminator at the end.
- A line terminator can be any one of the following:
  - Carriage-return/line-feed sequence (“\r\n”)
  - A single carriage-return (“\r”)
  - A single line-feed (“\n”).
- **`BufferedReader.readLine()` and `PrintWriter.println()` methods: hoặc BufferedWriter (đọc từng dòng)**
  - The `readLine()` method returns a line of text with the line.
  - The `println()` method outputs each line on a new line as it appends the line terminator for the current operating system.

Methods of `InputStream` Class [1-2]

`read()`:

- The `read()` method reads the next bytes of data from the input stream and returns an `int` value in the range of 0 to 255.
- The method returns -1 when end of file is reached.

```
public abstract int read() throws IOException
```

`available()`:

- The `available()` method returns the number of bytes that can be read without blocking.

```
public int available() throws IOException
```

`close()`:

- The `close()` method closes the input stream.
- It releases the system resources associated with the stream.

```
public void close() throws IOException
```

Methods of `InputStream` Class [2-2]

`mark(int n)`:

- The `mark(int n)` method marks the current position in the stream and will remain valid until the number of bytes specified in the variable, `n`, is read.
- A call to the `reset()` method will position the pointer to the last marked position.

```
public void mark(int readlimit)
```

`skip(long n)`:

- The `skip(long n)` method skips `n` bytes of data while reading from an input stream.

```
public long skip(long n) throws IOException
```

`reset()`:

- The `reset()` method rests the reading pointer to the previously set mark in the stream.

```
public void reset() throws IOException
```

FileInputStream Class [1-3]

- File stream objects can be created by either passing the name of the file or a `File` object or a `FileDescriptor` object respectively.
- `FileInputStream` class is used to read bytes from a file.
- When an object of `FileInputStream` class is created, it is also opened for reading.
- `FileInputStream` class overrides all the methods of the `InputStream` class except `mark()` and `reset()` methods.
- The `reset()` method will generate an `IOException`.
- Commonly used constructors of this class are as follows:
  - `FileInputStream(String sObj)`
  - `FileInputStream(File fObj)`
  - `FileInputStream(FileDescriptor fdObj)`

FileInputStream Class [2-3]

The following Code Snippet displays the creation of `FileInputStream` object:

Code Snippet

```
. . .
FileInputStream fileName = new FileInputStream("Helloworld.txt");
File fName = new File("/command.doc");
FileInputStream fileObj = new FileInputStream(fName);
```

The following Code Snippet demonstrates how to create a `FileInputStream` object using different constructors:

Code Snippet

```
import java.io.FileInputStream;
import java.io.IOException;
public class FIStream {
public static void main(String argv[]){
try {
```

FileInputStream Class [3-3]

```
    FileInputStream intest;
intest = new FileInputStream("D:/resources/Client.java");
    int ch;
    while ((ch = intest.read()) > -1) {
      StringBuffer buf = new StringBuffer();
       buf.append((char) ch);
     System.out.print(buf.toString());
    }
   } catch (IOException e) {
     System.out.println(e.getMessage());
   }
}
```

ByteArrayInputStream Class [1-2]

- `ByteArrayInputStream` contains a buffer that stores the bytes that are read from the stream.
- `ByteArrayInputStream` class uses a byte array as the source.
- `ByteArrayInputStream` class has an internal counter, which keeps track of the next byte to be read.
- This class does not support any new methods.
- It only overrides the methods of the `InputStream` class such as `read()`, `skip()`, `available()`, and `reset()`.

`protected byte[] buf`:

This refers to an array of bytes that is provided by the creator of the stream.

`protected int count`:

This refers to the index greater than the last valid character in the input stream buffer.

`protected int mark`:

This refers to the currently marked position in the stream.

ByteArrayInputStream Class [2-2]

protected int pos:

This refers to the index of the next character to be read from the input stream buffer.

The constructors of this class are as follows:

- `ByteArrayInputStream(byte[] b)`
- `ByteArrayInputStream(byte[] b, int start, int num)`

The following Code Snippet displays the use of the `ByteArrayInputStream` class:

Code Snippet

```
. . .
String content = "Hello World";
Byte [] bObj = content.getBytes();
ByteArrayInputStream inputByte = new ByteArrayInputStream(bObj);
. . .
```

`OutputStream` Class and its Subclasses

- The `OutputStream` class is an abstract class that defines the method in which bytes or arrays of bytes are written to streams.
- `ByteArrayOutputStream` and `FileOutputStream` are the subclasses of `OutputStream` class.

Methods in `OutputStream` Class

`write(int b)`

`write(byte[] b)`

`write(byte[] b, int off, int len)`

`flush()`

`close()`

FileOutputStream Class [1-2]

- FileOutputStream class creates an OutputStream that is used to write bytes to a file.
- FileOutputStream may or may not create the file before opening it for output and it depends on the underlying platform.
- Certain platforms allow only one file-writing object to open a file for writing.
- Therefore, if the file is already open, the constructors in the class fail.
- An IOException will be thrown only when a read-only file is opened.
- Some of the commonly used constructors of this class are as follows:
  - `FileOutputStream(String filename)`
  - `FileOutputStream(File name)`
  - `FileOutputStream(String filename, boolean flag)`
  - `FileOutputStream(File name, boolean flag)`

FileOutputStream Class [2-2]

The following Code Snippet displays the use of FileOutputStream class:

Code Snippet

```
...
String temp = "One way to get the most out of life is to look upon
it as an adventure."
byte [] bufObj = temp.getBytes();
OutputStream fileObj = new FileOutputStream("Thought.txt");
fileObj.write(bufObj);
fileObj.close();
...
```

Code Snippet first stores the content of the Stringvariable in the byte array, bufObj, using the getBytes() method.

Then, the entire content of the byte array is written to the file, Thought.txt.

`ByteArrayOutputStream` Class

- `ByteArrayOutputStream` class creates an output stream in which the data is written using a byte array.
- It allows the output array to grow in size so as to accommodate the new data that is written.
- `ByteArrayOutputStream` class defines two constructors which are as follows:
  - `ByteArrayOutputStream()`
  - `ByteArrayOutputStream(int size)`

Methods in `ByteArrayOutputStream` Class [1-2]

`reset()`

`size()`

`toByteArray()`

`writeTo(OutputStream out)`

`toString()`

Methods in ByteArrayOutputStream Class [2-2]

- The following Code Snippet displays the use of the `ByteArrayOutputStream` class:

Code Snippet

```
. . .
String strObj = "Hello World";
byte[] buf = strObj.getBytes();
ByteArrayOutputStream byObj = new ByteArrayOutputStream();
byObj.write(buf);
System.out.println("The string is:" + byObj.toString());
. . .
```

- In the Code Snippet, a `ByteArrayOutputStream` object is created and then the content from the byte array is written to the `ByteArrayOutputStream` object.
- Finally, the content from the output stream is converted to a string using the `toString()` method and displayed.

Filter Streams [1-8]

- The `FilterInputStream` class provides additional functionality by using an input stream as its basic source of data.
- The `FilterOutputStream` class streams are over existing output streams.
- They either transform the data along the way or provide additional functionality.

`FilterInputStream`:

- The `FilterInputStream` class overrides all the methods of the `InputStream` class that pass all requests to the contained input stream.
- The subclasses can also override certain methods and can provide additional methods and fields.
- Following are the fields and constructors for `java.io.FilterInputStream` class:
  - `protected InputStream in`
  - `protected FilterInputStream(InputStream in)`

Filter Streams [2-8]

- Following are the methods of this class:
  - `mark(int readlimit)`
  - `markSupported()`
  - `read()`
  - `available()`
  - `close()`
  - `read(byte[] b)`
  - `reset()`
  - `skip(long n)`
  - `read(byte[] b, int off, int len)`
- The following Code Snippet demonstrates the use of `FilterInputStream` class:

Code Snippet

```
package javaioapplication;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FilterInputStream;
```

Filter Streams [3-8]

```
import java.io.IOException;
import java.io.InputStream;
public class FilterInputApplication {
 public static void main(String[] args) throws Exception {
  InputStream inputObj = null;
  FilterInputStream filterInputObj = null;
try {
     // creates input stream objects
    inputObj = new FileInputStream("C:/Java/Hello.txt");
    filterInputObj = new BufferedInputStream(inputObj);
    // reads and prints from filter input stream
    System.out.println((char) filterInputObj.read());
    System.out.println((char) filterInputObj.read());
    // invokes mark at this position
    filterInputObj.mark(0);
    System.out.println("mark() invoked");
    System.out.println((char) filterInputObj.read());
    System.out.println((char) filterInputObj.read());
} catch (IOException e) {
```

Filter Streams [4-8]

```
   // prints if any I/O error occurs
   e.printStackTrace();
  } finally {
    // releases system resources associated with the stream
    if (inputObj != null) {
     inputObj.close();
    }
  if (filterInputObj != null) {
      filterInputObj.close();
      }
   }
  }
}
```

Filter Streams [5-8]

`FilterOutputStream` Class:

- The `FilterOutputStream` class overrides all methods of `OutputStreamclass` that pass all requests to the underlying output stream.
- Subclasses of `FilterOutputStream` can also override certain methods and give additional methods and fields.
- The `java.io.FilterOutputStream` class includes the protected `OutputStreamout` field, which is the output stream to be filtered.
- `FilterOutputStream(OutputStream out)` is the constructor of this class.
- This creates an output stream filter that exist class over the defined output stream.

Filter Streams [6-8]

The following Code Snippet demonstrates the use of `FilterOutputStream` class:

Code Snippet

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FilterOutputStream;
import java.io.IOException;
import java.io.OutputStream;
public class FilterOutputApplication {
  public static void main(String[] args) throws Exception {
  OutputStream OutputStreamObj = null;
  FilterOutputStream filterOutputStreamObj = null;
  FileInputStream filterInputStreamObj = null;
  byte[] bufObj = {81, 82, 83, 84, 85};
  int i=0;
  char c;
```

Filter Streams [7-8]

```
 //encloses the creation of stream objects within try-catch block
 try{
 // creates output stream objects
 OutputStreamObj = new FileOutputStream("C:/Java/test.txt");
 filterOutputStreamObj = new FilterOutputStream(OutputStreamObj);
// writes to the output stream from bufObj
   filterOutputStreamObj.write(bufObj);
   // forces the byte contents to be written to the stream
   filterOutputStreamObj.flush();
   // creates an input stream object
   filterInputStreamObj = new FileInputStream("C:/Java/test.txt");
   while((i=filterInputStreamObj.read())!=-1)
   { // converts integer to character
    c = (char)i;
    // prints the character read
    System.out.println("Character read after conversion is: "+ c);
    }
```

Filter Streams [8-8]

```
}catch(IOException e){
 // checks for any I/O errors
 System.out.print("Close() is invoked prior to write()");
 }finally{
  // releases system resources associated with the stream
  if(OutputStreamObj!=null)
   OutputStreamObj.close();
 if(filterOutputStreamObj!=null)
```

Buffered Streams

- A buffer is a temporary storage area for data.
- By storing the data in a buffer, time is saved as data is immediately received from the buffer instead of going back to the original source of the data.
- Java uses buffered input and output to temporarily cache data read from or written to a stream.
- This helps programs to read or write small amounts of data without adversely affecting the performance of the system.
- Buffer allows skipping, marking, and resetting of the stream.
- Filters operate on the buffer, which is located between the program and the destination of the buffered stream.

BufferedInputStream Class

- `BufferedInputStream` class allows the programmer to wrap any `InputStream` class into a buffered stream.
- The `BufferedInputStream` act as a cache for inputs.
- It does so by creating the array of bytes which are utilized for future reading.
- The simplest way to read data from an instance of `BufferedInputStream` class is to invoke its `read()` method.
- `BufferedInputStream` class also supports the `mark()` and `reset()` methods.
- The function `markSupported()` will return true if it is supported.
- `BufferedInputStream` class defines two constructors:
  - `BufferedInputStream(InputStream in)`
  - `BufferedInputStream(InputStream in, int size)`

BufferedOutputStream Class

- `BufferedOutputStream` creates a buffer which is used for an output stream.
- It provides the same performance gain that is provided by the `BufferedInputStream` class.
- The main concept remains the same, that is, instead of going every time to the operating system to write a byte, it is cached in a buffer.
- It is the same as `OutputStream` except that the `flush()` method ensures that the data in the buffer is written to the actual physical output device.
- The constructors of this class are as follows:
  - `BufferedOutputStream(OutputStream os)`
  - `BufferedOutputStream(OutputStream os, int size)`

Character Streams [1-4]

- Byte stream classes provide methods to handle any type of I/O operations except Unicode characters.
- Character streams provide functionalities to handle character oriented input/output operations.
- They support Unicode characters and can be internationalized.
- Reader and Writer are abstract classes at the top of the class hierarchy that supports reading and writing of Unicode character streams.
- All character stream class are derived from the `Reader` and `Writer` class.

`Reader` Class:

- `Reader` class is an abstract class used for reading character streams.
- The subclasses of this class override some of the methods present in this class to increase the efficiency and functionality of the methods.
- All the methods of this class throw an `IOException`.
- The `read()` method returns -1 when end of the file is encountered.
- The following are the two constructors for the `Reader` class:
  - `Reader()`
  - `Reader(Object lock)`

Character Streams [2-4]

`Writer` Class:

- `Writer` class is an abstract class and supports writing characters into streams through methods that can be overridden by its subclasses.
- The methods of the `java.io.Writer` class are same as the methods of the `java.io.OutputStream` class.
- All the methods of this class throw an `IOException` in case of errors.
- The constructors for the `Writer` class are as follows:
  - `Writer()`
  - `Writer(Object lock)`

`PrintWriter` Class:

- The `PrintWriter` class is a character-based class that is useful for console output.
- It implements all the print methods of the `PrintStream` class.
- It does not have methods for writing raw bytes.
- In such a case, a program uses unencoded byte streams.

Character Streams [3-4]

- The `PrintWriter` class differs from the `PrintStream` class as it can handle multiple bytes and other character sets properly.
- This class provides support for Unicode characters.
- The class overrides the `write()` method of the `Writer` class with the difference that none of them raise any `IOException`.
- The printed output is tested for errors using the `checkError()` method.
- The `PrintWriter` class also provides support for printing primitive data types, character arrays, strings and objects.
- It provides formatted output through its `print()` and `println()` methods.
- The `toString()` methods will enable the printing of values of objects.
- The constructors for `PrintWriter` class are as follows:
  - `PrintWriter(OutputStream out)`
  - `PrintWriter(OutputStream out, boolean autoFlush)`
  - `PrintWriter(Writer out)`
  - `PrintWriter(Writer out, boolean autoFlush)`

Character Streams [4-4]

- The following Code Snippet displays the use of the `PrintWriter` class:

Code Snippet

```
. . .
InputStreamReader reader = new InputStreamReader (System.in);
OutputStreamWriter writer = new OutputStreamWriter (System.out);
PrintWriter pwObj = new PrintWriter (writer,true);
. . .
try
{
while (tmp != -1)
{
tmp = reader.read ();
ch = (char) tmp;
pw.println ("echo " + ch);
}
}
catch (IOException e)
{
System.out.println ("IO error:" + e );
}
. . .
```

CharArrayReader Class

- CharArrayReader class is a subclass of Reader class.
- The class uses character array as the source of text to be read.
- CharArrayReader class has two constructors and reads stream of characters from an array of characters.
- The constructors of this class are as follows:
  - CharArrayReader(char arr[])
  - CharArrayReader(char arr[], int start, int num)
- The following Code Snippet displays the use of the CharArrayReader class:

Code Snippet

```
. . .
String temp = "Hello World";
int size = temp.length();
char [] ch = new char[size];
temp.getChars(0, size, ch, 0);
CharArrayReader readObj = new CharArrayReader(ch, 0, 5);
```

CharArrayWriter Class [1-2]

- CharArrayWriter class is a subclass of Writer class.
- CharArrayWriter uses a character array into which characters are written.
- The size of the array expands as required.
- The methods to CharArray(), toString(), and writeTo() method can be used to retrieve the data.
- CharArrayWriter class inherits the methods provided by the Writer class.
- The constructors of this class are as follows:
  - CharArrayWriter()
  - CharArrayWriter(int num)

CharArrayWriter Class [2-2]

The following Code Snippet displays the use of the CharArrayWriter class:

Code Snippet

```
. . .
CharArrayWriter fObj = new CharArrayWriter();
. . .
String temp = "Hello World";
int size = temp.length();
char [] ch = new char[size];
temp.getChars(0, temp.length(), ch, 0);
fObj.write(ch);
char[] buffer = fObj.toCharArray();
System.out.println(buffer);
System.out.println(fObj.toString());
. . .
```

Chaining I/O Systems

- A program, typically, uses a series of streams to process the data.
- The following figure illustrates this:

- The following figure displays the chaining of an output stream:

Serialization

- Serialization is the process of reading and writing objects to a byte stream.
- An object that implements the `Serializable` interface will have its state saved and restored using serialization and deserialization facilities.
- When a Java object's class or superclass implements the `java.io.Serializable` interface or its subinterface, `java.io.Externalizable`, the Java object becomes serializable.
- The `java.io.Serializable` interface defines no methods.
- It indicates that the class should be considered for serialization.
- If a superclass is serializable, then its subclasses are also serializable.
- The only exception is if a variable is transient and static, its state cannot be saved by serialization facilities.
- When the serialized form of an object is converted back into a copy of the object, this process is called deserialization.

ObjectOutputStream Class [1-2]

- `ObjectOutputStream` class extends the `OutputStream` class and implements the `ObjectOutput` interface.
- It writes primitive data types and object to the output stream.
- The constructors of this class are as follows:
  - `ObjectOutputStream()`
  - `ObjectOutputStream(OutputStreamout)`

Methods in `ObjectOutputStream` Class:

`writeFloat(float f)`

`writeObject(Object obj)`

`defaultWriteObject()`

ObjectOutputStream Class [2-2]

The following Code Snippet displays the use of methods of `ObjectOutputStream` class:

Code Snippet

```
. . .
Point pointObj = new Point(50,75);
FileOutputStream fObj = new FileOutputStream("point");
ObjectOutputStream oos = new ObjectOutputStream(fObj);
oos.writeObject(pointObj);
oos.writeObject(new Date());
oos.close();
. . .
```

ObjectInputStream Class [1-5]

- `ObjectInputStream` class extends the `InputStream` class and implements the `ObjectInput` interface.
- `ObjectInput` interface extends the `DataInput` interface and has methods that support object serialization.
- `ObjectInputStream` is responsible for reading object instances and primitive types from an underlying input stream.
- It has `readObject()` method to restore an object containing non-static and non-transient fields.
- The constructors of this class are as follows:
  - `ObjectInputStream()`
  - `ObjectInputStream(InputStream in)`

Methods in `ObjectInputStream` Class:

`readFloat()` `readBoolean()` `readByte()`

`readChar()` `readObject()`

ObjectInputStream Class [2-5]

The following Code Snippet displays the creation of an instance of `ObjectInputStream` class:

Code Snippet

```
. . .
FileInputStream fObj = new FileInputStream("point");
ObjectInputStream ois = new ObjectInputStream(fObj);
Point obj = (Point) ois.readObject();
ois.close();
```

- In the Code Snippet, an instance of `FileInputStream` is created that refers to the file named point.
- An `ObjectInputStream` is created from that file stream.
- The `readObject()` method returns an object which deserialize the object.
- Finally, the object input stream is closed.

ObjectInputStream Class [3-5]

- The `ObjectInputStream` class deserializes an object.
- The object to be deserialized must had already been created using the `ObjectOutputStream` class.
- The following Code Snippet demonstrates the `Serializable` interface.

Code Snippet

```
import java.io.Serializable;
public class Employee implements Serializable{
    String lastName;
    String firstName;
    double sal;
}
public class BranchEmpProcessor {
public static void main(String[] args) {
        FileInputStream fIn = null;
        FileOutputStream fOut = null;
ObjectInputStream oIn = null;
```

ObjectInputStream Class [4-5]

```
        ObjectOutputStream oOut = null;
               try {
            fOut = new FileOutputStream("E:\\NewEmplyee.Ser");
            oOut = new ObjectOutputStream(fOut);
            Employee e = new Employee();
            e.lastName = "Smith";
            e.firstName = "John";
            e.sal = 5000.00;
            oOut.writeObject(e);
            oOut.close();
            fOut.close();
            fIn = new FileInputStream("E:\\NewEmplyee.Ser");
            oIn = new ObjectInputStream(fIn);
            //de-serializing employee
            Employee emp = (Employee) oIn.readObject();
          System.out.println("Deserialized  -  "  +  emp.firstName
+   "   "   +
          emp.lastName + " from NewEmployee.ser");
```

ObjectInputStream Class [5-5]

```
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
             System.out.println("finally");
        }
    }
}
```

Summary

- A stream is a logical entity that produces or consumes information.
- Data stream supports input/output of primitive data types and String values.
- InputStream is an abstract class that defines how data is received.
- The OutputStream class defines the way in which output is written to streams.
- File class directly works with files on the file system.
- A buffer is a temporary storage area for data.
- Serialization is the process of reading and writing objects to a byte stream.