Client Server Application Programming

Week 3: Data Streams(Overview, How Streams Work, Filter Streams, Readers and Writers, Object Persistence and Object Serialization)

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Agenda

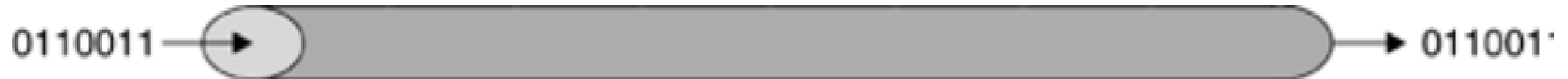
1. Data Streams(Overview How Streams Work,
2. Filter Streams,
3. Readers and Writers,
4. Object Persistence and
5. Object Serialization)

Data Streams(Overview How Streams Work)

Data Streams-Overview

are conduits through which information—bytes of data—is sent and received. Byte-level communication is represented in Java by data streams.

A simple analogy is a pipe through which material such as water may be moved from one location to another. Provided that the pipe is installed correctly, what goes in one end comes out from the other end. As seen below.



Communication over networks, with files, and even between applications, is represented in Java by streams. Stream-based communication is central to almost any type of Java application. The concept of streams is especially important when dealing with networking applications. Almost all network communication (except UDP communication) is conducted over streams, so it is essential that programmers be familiar with this concept.

Data Streams-Overview +

When designing a system, the correct stream must be selected; when building a system, the type of stream used is important, as a consistent interface is provided. Streams may be chained together, to provide an easier and more manageable interface.

If, **for example**, the data needed to be processed in a particular way, a second stream could connect to an existing stream, to provide for processing of the data. To mention, bytes might be converted from one form into another (such as a number), or a stream of bytes may be interpreted as sequences of characters, as shown in Figure below.

Data Streams-Overview ++

In Java, streams take a flexible, one-size-fits-all approach—they are fairly interchangeable, and can be applied on top of another stream, or even several other streams.

Interchangeability is an extremely important attribute of streams, as it simplifies programming considerably. Streams are divided into two categories— **input streams** that may be **read from** and **output streams** that may be **written to**.

Provided you don't try to read from an output stream, or write to an input stream, you can safely attach any **"filter" stream** (a stream that filters data in some fashion, such as processing it or converting it from bytes into a different form) to any lowlevel stream (such as a file or network stream).

Note! although streams are usually one-way, multiple streams can be used together (e.g., an input stream and an output stream) for two-way communication

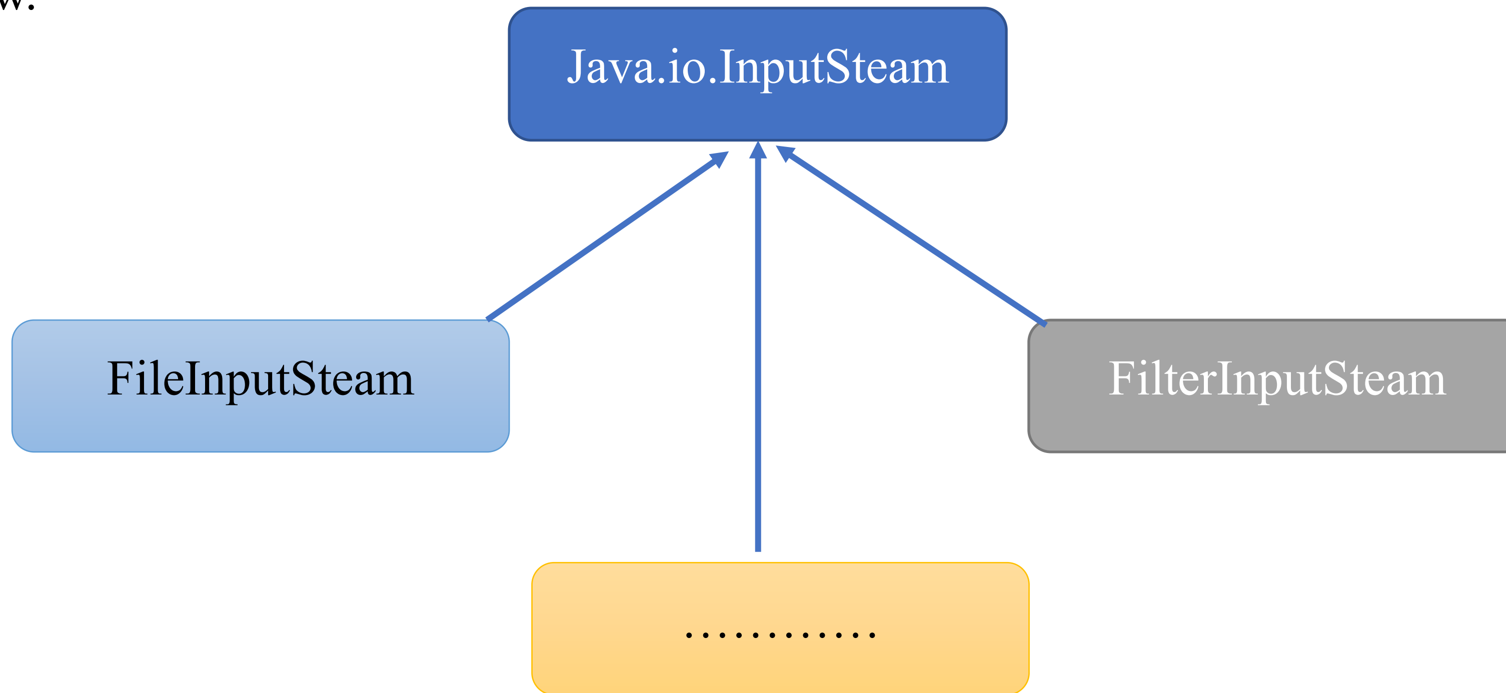
How Streams Relate to Networking

Transmission of data and the sending of sequences of bytes is the heart of network programming. A stream-based approach to communication simplifies programming, as a standard interface for data transmission is offered regardless of the type of data being sent and received, or of the mechanism by which data is sent across the network.

Of course, streams are not limited to networking—they can read from and be written to data structures, files, and other applications. However, as will be seen in later chapters, there is more than one way to send data over a network, and I/O streams provide a consistent interface for working with network communication is being used.

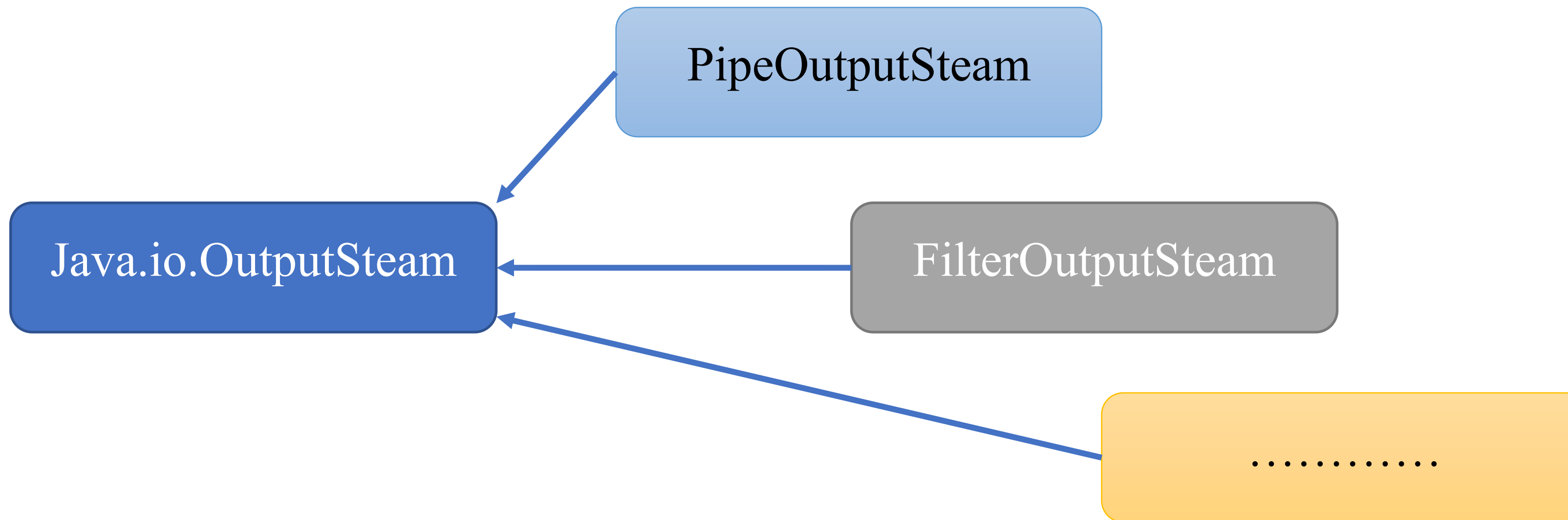
How Streams Work+

As mentioned earlier, streams provide communication of data at the byte level, and are **used** for either **reading** or **writing**. Streams for reading inherit from a common superclass, the java.io.InputStream class, as shown in the figure below.



How Streams Work

On the other hand, streams used for writing data inherit from the superclass java.io.OutputStream, as hown in the figure below.



These are abstract classes; they cannot be instantiated. Instead, an appropriate subclass for the task at hand is created. Several streams inherit directly from either InputStream or OutputStream, but most inherit from a filter stream.

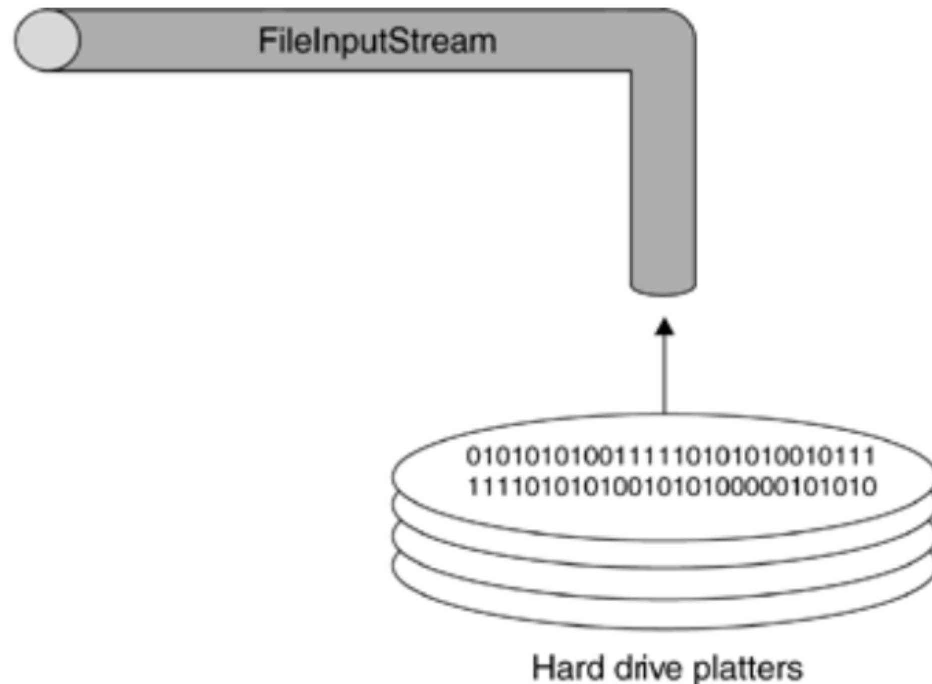
Reading from an InputStream

Many input streams are provided by the java.io package, and choosing the right low-level stream is a fairly straightforward task, since the name of the stream matches the data source it will read from. As populated on the table below, there are **six low-level streams** to choose from, each of which performs an entirely different task

Low-Level Input Stream	Purpose of Stream
`ByteArrayInputStream`	Reads bytes of data from an in-memory array
`FileInputStream`	Reads bytes of data from a file on the local file system
`PipedInputStream`	Reads bytes of data from a thread pipe
`StringBufferInputStream`	Reads bytes of data from a string
`SequenceInputStream`	Reads bytes of data from two or more low-level streams, switching from one stream to the next
`System.in`	Reads bytes of data from the user console

Reading from an InputStream+

When a low-level input stream is created, it reads from a source of information that supplies it with data, see Figure below.



Input streams act as consumers of information—they devour bytes of information as they read them. i.e, bytes are read from a file sequentially—subject to a few exceptions, once they have been read, one can't go back and read them again.

They haven't been erased; the stream has simply moved on to the next byte of information.

Reading from an InputStream++

However, some high-level filter streams support a limited push-back ability, allowing you to jump back to a specific point in the stream, on the condition that the point is marked. This functionality is useful at times for looking ahead to see the contents of a stream, but is not supported by many filter streams

The java.io.InputStream Class Methods

The abstract InputStream class defines public methods, listed below, which are common to all input streams.

1. **int available()** throws java.io.IOException— returns the number of bytes currently available for reading. More bytes may be available in the future, but reading more than the number of available bytes will result in a read that will block indefinitely.

2. **void close()** throws java.io.IOException— closes the input stream and frees any resources (such as file handles or file locks) associated with the input stream.

3. **void mark(int readLimit)**— records the current position in the input stream, to allow an input stream to revisit the same sequence of bytes at a later point in the future, by invoking the InputStream.reset() method. Not every input stream will support this functionality.

The java.io.InputStream Class Methods +

4. **boolean markSupported()**— returns "true" if an input stream supports the mark()and reset() methods, "false" if it does not. Unless over ridden by a subclass of InputStream, the default value returned is false.

5. **int read()** throws java.io.IOException— returns the next byte of data from the stream. Subclasses of InputStream usually override this method to provide custom functionality (such as reading from a file or a string). As mentioned earlier, input streams use blocking I/O, and will block indefinitely if no further bytes are yet available. When the end of the stream is reached, a value of –1 is returned.

6. **int read(byte[] byteArray)** throws java.io.IOException— reads a sequence of bytes and places them in the specified byte array, by calling the read()method repeatedly until the array is filled or no more data can be obtained. This method returns the number of bytes successfully read, or –1 if the end of the stream has been reached.

The java.io.InputStream Class Methods ++

7. **int read(byte[] byteArray, int offset, int length)** throws java.io.IOException, java.lang.IndexOutOfBoundsException— reads asequence of bytes, placing them in the specified array. Unlike the previous method,read(byte[] byteArray), this method begins stuffing bytes into the array at the **specified offset**, and for the **specified length**, if possible. This allows developers to fill up only part of an array. Developers should be mindful that at runtime, out-of-bounds exceptions may be thrown if the array size, offset, and length exceed array capacity.

8. **void reset()** throws java.io.IOException— moves the position of the input stream back to a preset mark, determined by the point in time when the mark() method was invoked. Few input streams support this functionality, and may cause an IOException to be thrown if called.

9. **long skip(long amount)** throws java.io.IOException— reads, but ignores, the specified amount of bytes. These bytes are discarded, and the position of the input stream is updated. Though unlikely, it is entirely possible that the specified number of bytes could not be skipped (for example, as stated in the Java API, if the end of the stream is reached). The skip method returns the number of bytes skipped over, which may be less than the requested amount.

Using a Low-Level Input Stream Example code - FISandFOS_Demo class

Below, we examine a practical application using a low-level InputStream to display the contents of a file(**myFile.txt**) that has been written by OutputStream . A byte at a time is read from the file and displayed to the screen.

This code is in three stages.

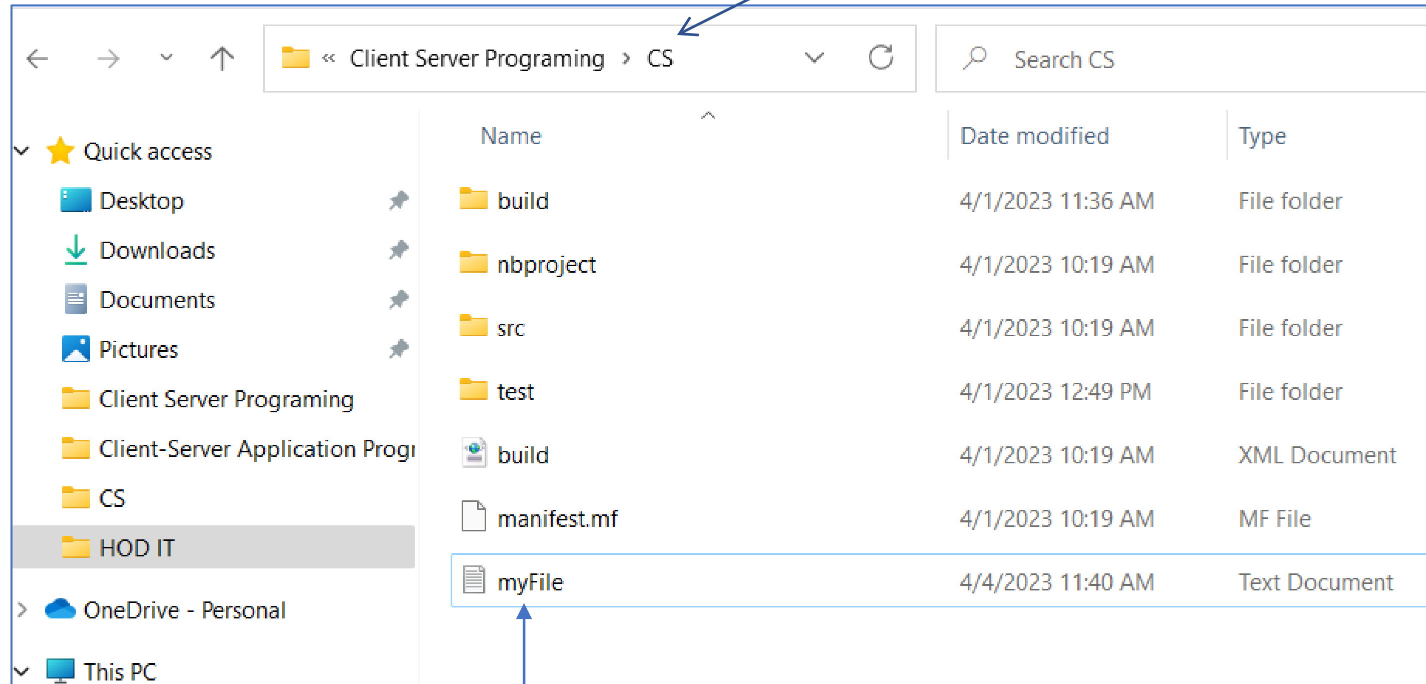
1. Creation of a txt file called **myFile.txt**.
2. Writing of the content into the file using the **FileOutputStream**
3. Reading and displaying the content of the file called myFile.txt using the **FileInputStream**

Using a Low-Level InputStream Example code - FISandFOS_Demo class

1. Creation a txt file called **myFile.txt**.

```
package cs; import java.io.*;
public class FISandFOS_Demo{
public static void main(String[] args) throws
IOException {
        // Representing the file
            File file = new File("myFile.txt");
        // Creation of the file
          if(!file.exists()){
              file.createNewFile();
          }
```

myFile Location and File content on creation



The file is usually created in the current project folder by default. However, one could change the directory where the file can be created.

Note the presence of myFile in CS project folder.

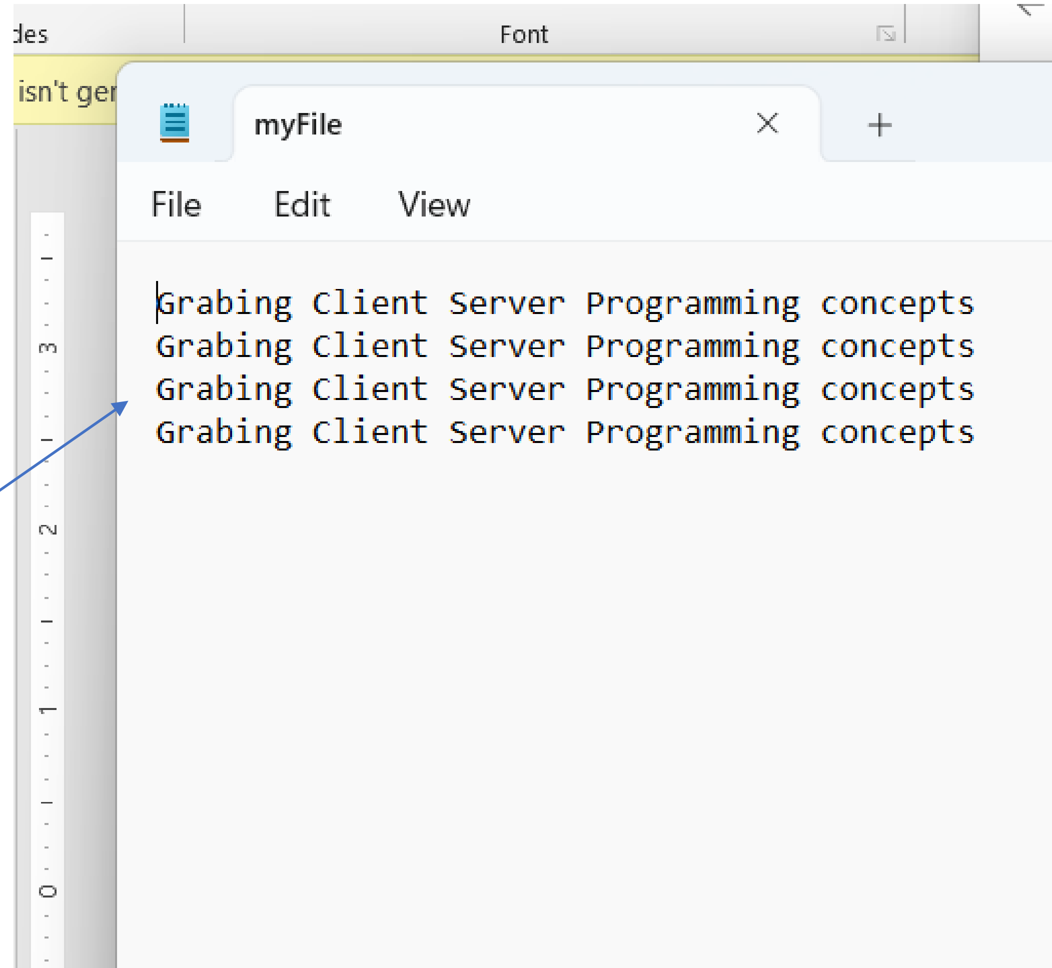
Using a Low-Level Input Stream Example code - FISandFOS_Demo class+

2. Writing of the content into the file using the **FileOutputStream**

```
/*      //Writing text into the file above using FileOutputStream
        FileOutputStream fos = new FileOutputStream(file);
        String textIn = "Grabing Client Server Programming concepts\n";
        int i = 0;
        while(i<4){
         fos.write(textIn.getBytes());
        System.out.println("Text inserted");
        i++;
        }
        fos.flush();
*/
```

myFile content after inserting bytes of data

Content can now be seen in myFile.txt file



Using a Low-Level Input Stream Example code - FISandFOS_Demo class++

3. Reading and displaying the content of the file called myFile.txt using the **FileInputStream**

```
//Reading the text from the file using FileInputstream
        FileInputStream fis = new FileInputStream(file);
        int i = fis.read();
        //loop through the bytes in the file
        while(!(i==-1)){
            char c = (char)i;
            System.out.print(c);
            i = fis.read();
        }
        fis.close();
    }
}
```

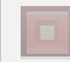
Code in Netbeans

```
1   package cs; import java.io.*;
2   public class FISandFOS_Demo {//FIS-FileinputStream
3       public static void main(String[] args) throws IOException {
4         // Representing the file
5           File file = new File("myFile.txt");
6         // Creation of the file
7           if(!file.exists()){
8               file.createNewFile();
9           }
10    /*      //Writing text into the file above using FileOutputStream
11          FileOutputStream fos = new FileOutputStream(file);
12          String textIn = "Grabing Client Server Programming concepts\n";
13          int i =0;
14          while(i<4){
15           fos.write(textIn.getBytes());
16          System.out.println("Text inserted");
```

Code in Netbeans and output.

```
17          i++;
18          }
19          fos.flush();
20      */
21          //Reading the text from the file using FileInputstream
            FileInputStream fis = new FileInputStream(file);
23          int i = fis.read();
24          //loop through the bytes in the file
25          while(!(i==-1)){
26              char c = (char)i;
27              System.out.print(c);
28              i = fis.read();
29          }
30          fis.close();
31      }}
```

Output - CS (run)



```
run:
Grabing Client Server Programming concepts
Grabing Client Server Programming concepts
Grabing Client Server Programming concepts
Grabing Client Server Programming concepts
BUILD SUCCESSFUL (total time: 0 seconds)
```

Writing to an OutputStream

A number of output streams are available in the java.io package for a variety of tasks, such as writing to data structures including strings and arrays, or to files or communication pipes. As seen in Table below, **six important** low-level output streams that may be written to (in addition to filter streams that may be connected to these low-level streams)

Low-Level Output Stream	Purpose of Stream
`ByteArrayOutputStream`	Writes bytes of data to an array of bytes.
`FileOutputStream`	Writes bytes of data to a local file.
`PipedOutputStream`	Writes bytes of data to a communications pipe, which will be connected to a `java.io.PipedInputStream`.
`StringBufferOutputStream`	Writes bytes to a string buffer (a substitute data struc-ture for the fixed-length string).
`System.err`	Writes bytes of data to the error stream of the user console, also known as standard error. In addition, this stream is cast to a `PrintStream`.
`System.out`	Writes bytes of data to the user console, also known as standard output. In addition, this stream is cast to a `PrintStream`.

Writing to an OutputStream+

As mentioned earlier, there are other streams which may be written to that developers cannot create and instantiate directly, but that nonetheless will be encountered. For example, a networking stream such as a connection to a TCP service is not created directly, but is provided by invoking the appropriate method on a socket. These types of streams will be covered in later chapters, as appropriate

The java.io.OutputStream Class methods

The java.io.OutputStream class is an abstract class, from which all output streams (be they low-level streams or filter streams) are inherited. The following methods are defined by the OutputStream class (all are public).

1. **void close()** throws java.io.IOException— closes the output stream, notifying the other side that the stream has ended. Pending data that has not yet been sent will be sent, but no more data will be delivered.
2. **void flush()** throws java.io.IOException— performs a "flush" of any unsent data and sends it to the recipient of the output stream. To improve performance, streams will often be buffered, so data remains unsent. This is useful at times, but obstructive at others. The method is particularly important for OutputStream subclasses that represent network operations, as flushing should always occur after a request or response is sent so that the remote side isn't left waiting for data.
3. **void write(int byte)** throws java.io.IOException— writes the specified byte. This is an abstract method, overridden by OutputStream subclasses.

The java.io.OutputStream Class methods+

The java.io.OutputStream class is an abstract class, from which all output streams (be they low-level streams or filter streams) are inherited. The following methods are defined by the OutputStream class (all are public).

4. **void write(byte[] byteArray)** throws java.io.IOException— writes the contents of the byte array to the output stream. The entire contents of the array (barring any error) will be written.
5. **void write(byte[] byteArray, int offset, int length)** throws java.io.IOException— writes the contents of a subset of the byte array to the output stream. This method allows developers to specify just how much of an array is sent, and which part, as opposed to the OutputStream.write(byte[] byteArray) method, which sends the entire contents of an array.

Filter Streams

Filter Streams

are kind of streams that add additional functionality to an existing stream i.e low-level streans, by processing data in some form (such as buffering for performance) or offering additional methods that allow data to be accessed in a different manner (for example, reading a line of text rather than a sequence of bytes).

Filters make life easier for programmers, as they can work with familiar constructs such as strings, lines of text, and numbers, rather than individual bytes. Instead of the programmer writing a string one character at a time and converting each character to an int value for the

OutputStream.write(int) method, the filter stream does this for them

Connecting a Filter Stream to an Existing Stream

Filter streams can be connected to any other stream, to a low-level stream or even another filter stream. Filter streams are extended from the java.io package.

FilterInputStream and java.io.FilterOutputStream classes. Each filter stream supports one or more constructors that accept either an InputStream, in the case of an input filter, or an OutputStream, in the case of an output filter. Connecting a filter stream is as simple as creating a new instance of a filter, passing an instance of an existing stream, and using the filter from then on to read or write.

For example, suppose you wanted to connect a PrintStream (used to print text to an OutputStream subclass) to a stream that wrote to a file. The following code may be used to connect the filter stream and write a message using the new filter.

Connecting a Filter Stream to an Existing Stream+

```
FileOutputStream fout = new FileOutputStream ( somefile );
PrintStream pout = new PrintStream (fout);
pout.println ("hello world");
```

This process is fairly simple as long as the programmer remembers two things:

1. Read and write operations must take place on the new filter stream.
2. Read and write operations on the underlying stream can still take place, but not at the same time as an operation on the filter stream.

Useful Filter Input Streams of the java.io package

There are many useful filter streams, some of which are present in all versions of Java; others (such as streams for data compression) are available only in JDK1.1 or Java 2. The most common, and most useful, are shown in

Filter Input Stream	Purpose of Stream
`BufferedInputStream`	Buffers access to data, to improve efficiency.
`DataInputStream`	Reads primitive data types, such as an int, a float, a double, or even a line of text, from an input stream.
`LineNumberInputStream`	Maintains a count of which line is being read, based on interpretation of end-of-line characters. Handles both Unix and Windows end-of-line sequences.
`PushBackInputStream`	Allows a byte of data to be pushed into the head of the stream.

Useful FilterOutputStreams of the java.io package

While there are plenty of filter input streams, a few useful output streams are available as well. The most useful are presented below.

Filter Output Stream	Purpose of Stream
`BufferedOutputStream`	Provides buffering of data writes, to improve efficiency
`DataOutputStream`	Writes primitive datatypes, such as bytes and numbers
`PrintStream`	Offers additional methods for writing lines of text, and other datatypes as text

BufferedOutputStream Class

The BufferedOutputStream provides data buffering similar to the BufferedInputStream. As suggested by the name of the class, however, it buffers writes, not reads. An internal buffer is maintained, and when the buffer is complete (or earlier, if a request to flush the buffer is made), the buffer contents are dumped to the output stream to which the buffered stream is connected.

Constructors

1. BufferedOutputStream (OutputStream output)— creates a buffer for writing to the specified output stream. The default size of this buffer is 512 bytes in length.
2. BufferedOutputStream (OutputStream output int bufferSize) throws java.lang.IllegalArgumentException— creates a buffer for writing to the specified output stream, overriding the default buffer sizing. The buffer is set to the specified buffer size, which must be greater than zero or an exception is thrown

DataOutputStream Class

Like its sister class, DataInputStream, the DataOutputStream class is designed to deal with primitive datatypes, such as numbers or bytes. Most of the read methods of DataInputStream have a corresponding write method mirrored in DataOutputStream.

This allows developers to write datatypes to a file or other type of stream, and to have them read back by another Java application without any compatibility issues over how primitive datatypes are represented by different hardware and software platforms. It implements the java.io.DataOutput interface, which provides additional methods for writing primitive datatypes.

Constructor

DataOutputStream (OutputStream output)— creates a data output stream,which will write to the specified stream.

DataOutputStream Class-Methods

This class adds the following new methods plus other not displayed, all of which are public unless other wise noted.

1. int size()— returns the number of bytes written to the data output stream at any given moment
2. void writeBoolean (boolean value) throws java.io.IOException—writes the specified boolean value, represented as a one-byte value. If the boolean value is "true," the value 1 is sent, and if "false," the value 0 is sent.
3. void writeByte (int byte) throws java.io.IOException— writes the specified byte to the output stream.
4. void writeBytes (String string) throws java.io.IOException— writes the entire contents of a string to the output stream a byte at a time.
5. void writeChar (int char) throws java.io.IOException— writes the character (represented by an int value) to the output stream as a two-byte value.
6. void writeChars (String string) throws java.io.IOException— writes the entire contents of a string to the output stream, represented as two-byte values.

PrintStream Class

The PrintStream is the most unusual of all filter output streams. Aside from the change in naming convention (PrintStream versus the expected PrintOutputStream), it is typical in that it overrides methods inherited from FilterOutputStream without throwing the expected java.io.IOException class.

The PrintStream adds additional methods as well, none of which may throw an IOException. No errors are overtly reported, and instead the presence of an error is determined by invoking the checkError() method—although no further details may be obtained as to the cause of the error. Despite its idiosyncrasies, the PrintStream is an extremely useful class, as it provides a convenient way to print primitive datatypes as text using the print(..) method, and to print these with line separators using the println(..) method.

PrintStream Class

Constructors

1. PrintStream (OutputStream output)— creates a print stream for printing of datatypes as text.
2. PrintStream (OutputStream output, boolean flush)— creates a print stream for printing of datatypes as text. If the specified boolean flag is set to "true," whenever a byte array, println method, or newline character is sent, the underlying buffer will be automatically flushed.

PrintStream Class methods

This class added numerous methods not limited to the: -

1. boolean checkError()— automatically flushes the output stream and checks to see if an error has occurred. Instead of throwing an IOException, an internal flag is maintained that checks for errors.
2. void print(boolean value)— prints a boolean value.
3. void print(char character)— prints a character value.
4. void print(char[] charArray)— prints an array of characters.
5. void print(double doubleValue)— prints a double value.
6. void print(float floatValue)— prints a float value.
7. void print(int intValue)— prints an int value.
8. void print(long longValue)— prints a long value.
9. void print(Object obj)— prints the value of the specified object's toString()method. etc.

Readers and Writers

Readers and Writers

While input streams and output streams may be used to read and write text as well as bytes of information and primitive data types, a better alternative is to use readers and writers. Readers and writers were introduced in JDK1.1 to better support Unicode character streams.

What Are Unicode Characters?

Unicode is an extended character set. Most people think of characters as being composed of 8 bits of data, offering a range of 256 possible characters. Low ASCII (0–127) characters are followed by high ASCII characters (128–255). The high ASCII characters represent characters and symbols such as those used in foreign languages or punctuation. However, people quickly realized that even 256 characters were not enough to handle the many characters used in languages around the world. This is where Unicode came in. For more details visit: http://www.unicode.org/

The Importance of Readers and Writers

Readers and writers **are a better alternative** than input streams and output streams when used on text data. For those dealing solely with primitive data types, use of input streams and output streams may by all means be continued. However, if applications are processing text information only, use of a reader and/or a writer, to better support Unicode characters, should be considered.

Readers and writers support **Unicode character sequences**. **Internationalization** may not seem like a significant concern for those of us with an English-speaking background. However, in a growing global economy, internationalization may become more important, and the Y2K bug (which, while overrated, required significant cost and effort to repair) certainly taught us that software should be written with an eye toward the future.

From InputStreams to Readers

The java.io.InputStream class has a character-based equivalent in the form of the java.io.Reader class. The reader class has similar method signatures to that of the InputStream class, and existing code may be quickly converted to use it.

However, some slight changes are made to the method signatures, to support character, and not byte, reading. Additionally, the available() method has been removed, and replaced by the ready() method.

The java.io.Reader Class Methods

No public constructors are available for this class. Instead, a reader subclass should be instantiated.

Methods

The class includes the following methods, all of which are public:

1. void close() throws java.io.IOException— closes the reader.
2. void mark(int amount) throws java.io.IOException— marks the current position within the reader, and uses the specified amount of characters as a buffer. Not every reader will support the mark(int) and reset() methods.
3. boolean markSupported()— returns "true" if the reader supports mark and reset operations.
4. int read() throws java.io.IOException— reads and returns a character, blocking if no character is yet available. If the end of the reader's stream has been reached, a value of –1 is returned.
5. int read(char[] characterArray) throws java.io.IOException—populates an array of characters with data. This method returns an int value, representing the number of bytes that were read. If the end of the reader's stream is reached, a value of –1 is returned and the array is not modified. Etc.

Low-Level Reader Streams of the `java.io` Package

Low-Level Reader	Purpose of Reader
`CharArrayReader`	Reads from a character array
`FileReader`	Reads from a file on the local file system, just like a `FileInputStream`
`PipedReader`	Reads a sequence of characters from a thread communications pipe, exactly like a `PipedInputStream`
`StringReader`	Reads a sequence of characters from a String, as if it were a `StringBufferInputStream`
`InputStreamReader`	Bridges the divide between an input stream and a reader, by reading from the input stream

Combining Streams and Readers

To illustrate the use of input streams and readers together, we'll examine how an input stream may be converted to a reader for easy character reading without using a DataInputStream.

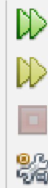
```
package cs;  import java.io.*;
public class InputStreamToReaderDemo {
    public static void main(String[] args) {
        try{
        System.out.print ("Please enter your name : ");
        // Get the input stream representing standard input
        InputStream input = System.in;
        // Create an InputStreamReader
        InputStreamReader reader = new InputStreamReader ( input );
```

Combining Streams and Readers +

```
// Connect to a buffered reader, to use the readLine() method
        BufferedReader bufReader = new BufferedReader( reader );
        String name = bufReader.readLine();
        System.out.println ("Welcome, " + name);
    }
    catch (IOException ioe){
        System.err.println ("I/O error : " + ioe);
    }}}
```

Combining Streams and Readers ++

```java
package cs;  import java.io.*;
public class InputStreamToReaderDemo {
    public static void main(String[] args) {
        try{
        System.out.print ("Please enter your name : ");
        // Get the input stream representing standard input
        InputStream input = System.in;
        // Create an InputStreamReader
        InputStreamReader reader = new InputStreamReader ( input );
        // Connect to a buffered reader, to use the
        // readLine() method
        BufferedReader bufReader = new BufferedReader( reader );
        String name = bufReader.readLine();
        System.out.println ("Welcome, " + name);
    }
     catch (IOException ioe){
        System.err.println ("I/O error : " + ioe);
    }}}
```



```
run:
Please enter your name : Elubu Joseph
Welcome, Elubu Joseph
BUILD SUCCESSFUL (total time: 20 seconds)
```

Types of Filter Readers

Filter readers, just like filter input streams, provide additional functionality in the form of new methods, or process data in a different way (such as buffering). Generally, but not always, they are extended from the java.io. FilterReader class, and always connect to another reader.

Filters may even be chained together (for example, connecting a BufferedReader to a custom reader, or vice versa). Some useful filter readers are listed below.

Filter Reader	Purpose of Stream
BufferedReader	Buffers access to data, to improve efficiency
FilterReader	Provides a class to extend when creating filters
PushBackReader	Allows text data to be pushed back into the reader's stream
LineNumberReader	Buffered reader subclass, which maintains a count of which line it is on

From Output Streams to Writers

Going from an output stream to a writer is not a difficult task. An equivalent class to java.io.OutputStream, the java.io.Writer class has similar method signatures and supports Unicode characters

The java.io.Writer Class

Has no public constructors. Instead, a writer subclass should be instantiated.

Low-Level Writers of the `java.io` Package

Low-Level Writer	Purpose of Writer
`CharArrayWriter`	Writes to a variable length character array (and resizes as characters are added)
`FileWriter`	Writes to a file on the local file system, just like a `FileOutputStream`
`PipedWriter`	Writes characters to a thread communications pipe, just like a `PipedOutputStream`
`StringWriter`	Writes characters to a string buffer (not a string, as the name might suggest), like a `StringBufferOutputStream`
`OutputStreamWriter`	Writes to a legacy output stream

The java.io.Writer Class Methods+

Among others, the following methods are included in this class; all are public .

1. **void close()** throws java.io.IOException— invokes the flush() method to send any buffered data, and then closes the writer.

2. **void flush()** throws java.io.IOException— flushes any unsent data, sending it immediately. A buffered writer might not yet have enough data to send, and may be storing it for later. Flushing sends this data immediately, and is particularly useful when working with networking streams, as a client might want to send data immediately when a command request is made.

3. **void write(int character)** throws java.io.IOException— writes the specified character.

4. **void write(char[] charArray)** throws java.io.IOException— reads the entire contents of the specified character array and writes it.

The java.io.Writer Class Methods

Among others, the following methods are included in this class; all are public .

5. **void write(char[] charArray, int offset, int length)** throws java.io.IOException— reads a subset of the character array, starting at the specified offset and lasting for the specified length, and writes it.
6. **void write(String string)** throws java.io.IOException— writes the specified string.
7. **void write(String string, int offset, int length)** throws java.io.IOException— writes a subset of the string, starting from the specified offset and lasting for the specified length.

OutputStreamToWriter Example code

The example below demonstrates how a writer may be connected to an output stream by using the OutputStreamToWriterPro class. To convert from an output stream to a writer is extremely compact, and doesn't require much effort.

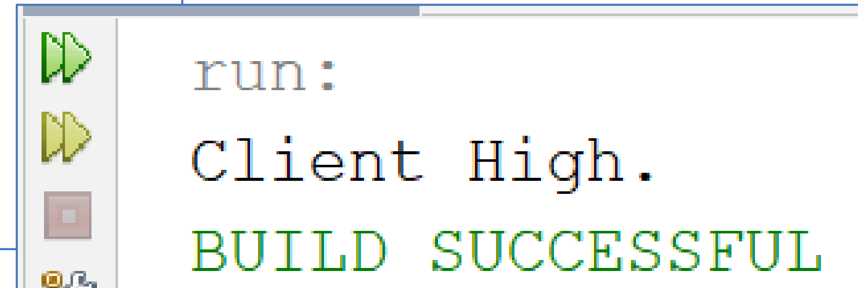
```
package cs; import java.io.*;
public class OutputStreamToWriterPro {
    public static void main(String[] args) {
        try{
        // Get the output stream representing standard output
        OutputStream output = System.out;
       // Create an OutputStreamWriter
        OutputStreamWriter writer = new OutputStreamWriter (output);
```

OutputStreamToWriter Example code+

```
// Write to standard output using a writer
writer.write ( "Client High.\n“ );
// Flush and close the writer, to ensure it is written
 writer.flush(); writer.close();
}
 catch (IOException ioe)
{
System.err.println ("I/O error : " + ioe);
}
}}
```

OutputStreamToWriter Example code on Netbeans

```
1  package cs;import java.io.*;
2  public class OutputStreamToWriterPro {
3      public static void main(String[] args) {
4          try{
5          // Get the output stream representing standard output
6          OutputStream output = System.out;
7        // Create an OutputStreamWriter
8          OutputStreamWriter writer = new OutputStreamWriter (output);
9          // Write to standard output using a writer
10         writer.write ("Client High.\n");
11         // Flush and close the writer, to ensure it is
12         // written
13         writer.flush(); writer.close();
14         }
15         catch (IOException ioe)
16         {
17         System.err.println ("I/O error : " + ioe);
18         }
19     }}
```



Object Persistence

is the ability of an object to maintain its state beyond the execution of the program that created it. In other words, it refers to the ability of an object to exist beyond the lifetime of the program that created it.

Object persistence is important in software development when data needs to be stored or transferred between different applications or systems. By persisting objects, we can ensure that data remains consistent and available even after the application or system that created it has been shut down.

Object persistence allows an object to outlive the JVM that hosts it. A custom class representing a key component of a system, a java.util.Vector list containing state data, or even a java.util.Properties object containing system settings, are all good examples of objects that might be important enough to be needed every time a program is run.

While the memory address space that is allocated to an object cannot be locked away for later use, the data stored there comprising the essence of an object may be stored elsewhere to be read and reconstructed at another date. The technique in Java is known as object serialization.

Object Serialization

Object serialization is the process of converting an object in a programming language into a format that can be stored or transmitted, such as a file, stream, or network message. This allows the object's state to be saved and restored later, or sent across a network to another application or system.

Serialization involves converting the object's data and state into a series of bytes that can be stored or transmitted. This process can be done using built-in serialization methods in programming languages or using third-party libraries.

During serialization, the object's data is typically written to a file or network stream in a standard format, such as XML, JSON, or binary. When the object needs to be restored, it can be deserialized by reading the serialized data from the file or stream and recreating the object's state.

Object Serialization+

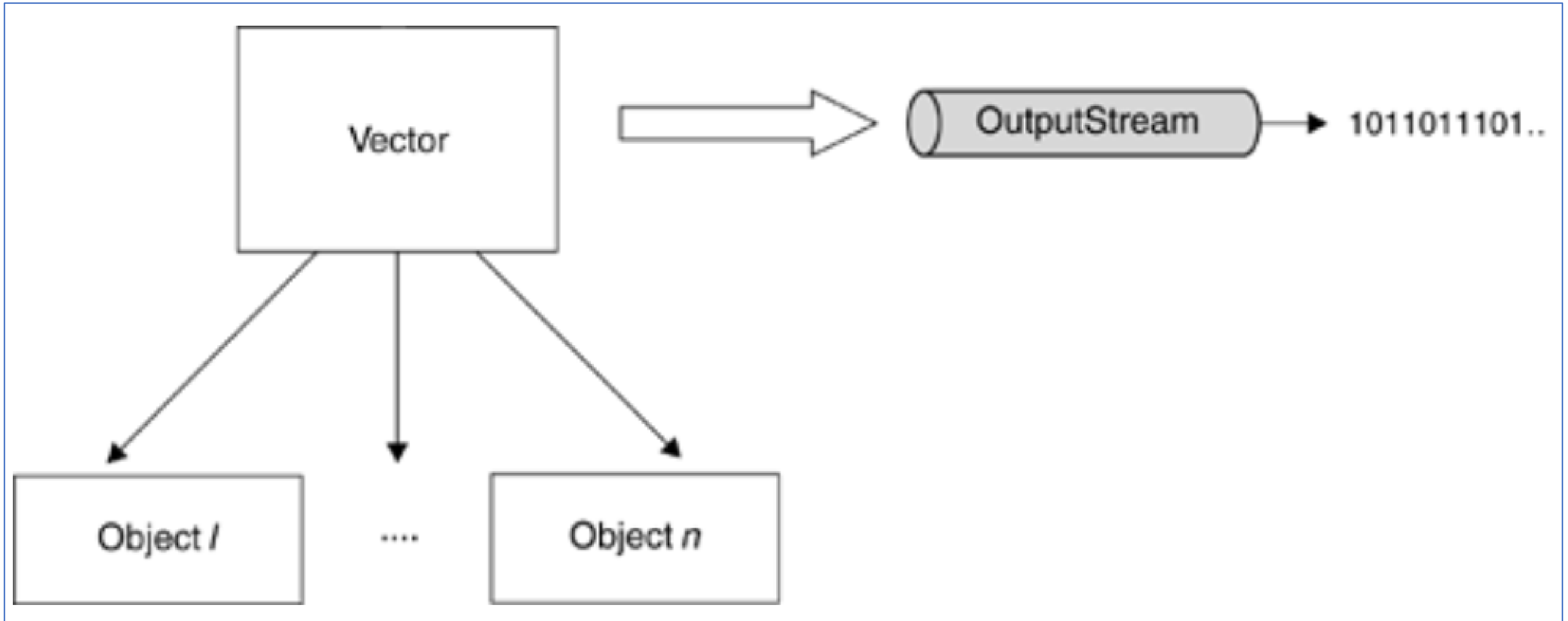
Serialization works by examining the variables of an object and writing primitive datatypes like numbers and characters to a byte stream. But if an object contains an object as a member variable (as opposed to a primitive datatype), the story is somewhat different.

The object member variable would cease to function correctly if the object was left out, so the variable must be serialized as well. If it contains an object or a collection of objects (such as an array or a vector), then they too must be serialized.

This must be done recursively, so that if an object has a reference to an object, which has a reference to another object (and so on), they are all saved together. This happens transparently—developers do not need to manually specify which objects are to be written.

The set of all objects referenced is called a graph of objects, and **object serialization converts entire graphs to byte form** (as shown below).

Entire graphs of objects may be serialized to an output stream



Object Serialization++

Serialization is commonly used in distributed computing, where objects need to be transmitted across a network to other systems. It is also used for caching and persisting objects in databases or file systems. However, it's important to note that not all objects can be serialized, as some objects contain references to other objects or resources that may not be serializable.

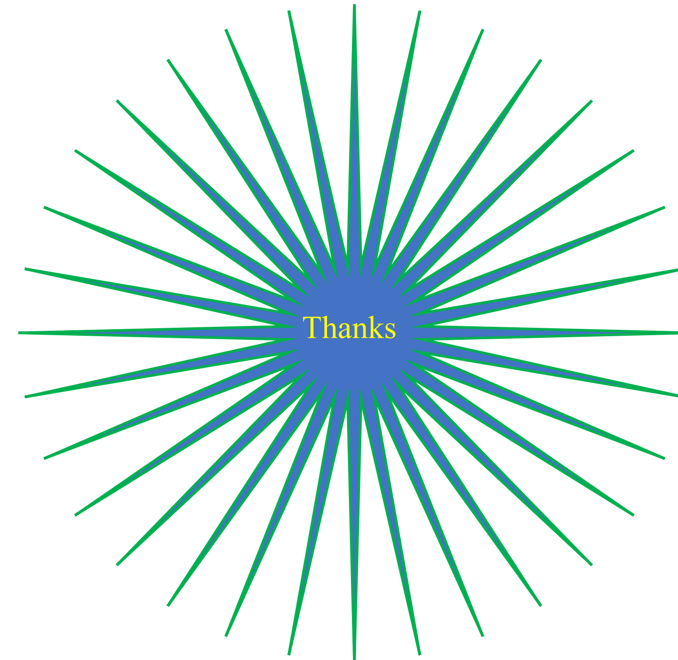
Some popular serialization frameworks and libraries include Java Serialization, XML Serialization in .NET, JSON Serialization in JavaScript, and Protocol Buffers from Google. Much on this section seen in the code later.

Summary

Summary

1. What streams are, and how they work
2. What readers and writers are, and how they work
3. How to connect filter streams to low-level streams
4. How to connect readers and writers to streams
5. About object serialization

Thank you for Listening



References

Java Network Programming and Distributed Computing, David R, Michael R (2002), Publisher: Addison Wesley, ISBN: 0-201-71037-4

PramatarovHi, M. (2022, November 18). *What is domain name resolution?* ClouDNS Blog. Retrieved March 30, 2023, from https://www.cloudns.net/blog/domain-name-resolution/

What is domain name resolution. BleepingComputer. (2004, April 9). Retrieved March 30, 2023, from https://www.bleepingcomputer.com/tutorials/what-is-domain-name-resolution/