CMP5332 Object-Oriented Programming

I/O Streams

Introduction to Streams

Often a program needs to bring in information from an external source or to send information to an external destination.

The information can be anywhere: in a file on disk, standard system input, somewhere on the network, in memory, or in another program.

Also, the information can be of any type: bytecode, objects, characters, images or sound.

In Java, information can be stored and retrieved using a communication system called streams, which are implemented in the java.io package. To use these classes, a program needs to import the java.io package.

In this lecture we learn how to create **input streams** to read information and **output streams** to store information. We will work with:

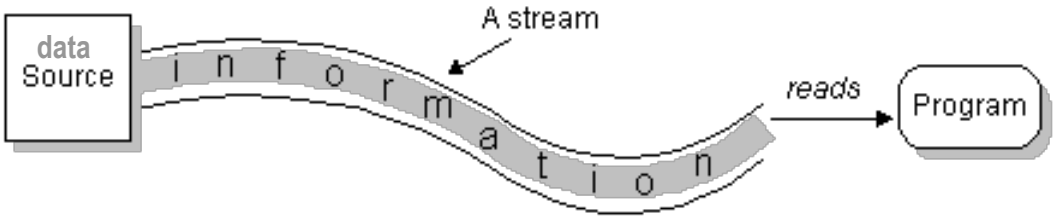
- **Byte streams**, which are used to handle bytes, integers, and other primitive data types
- **Character streams**, which handle text files and other text sources.

Whether you are using a byte stream or a character stream the procedure for using either in Java is largely the same.

A stream is a path travelled by data in a program:

- **An input stream** send data from a source into a program.

  For an input stream, the first step is to create a stream object that is associated with the data source



To read data from an input Stream:

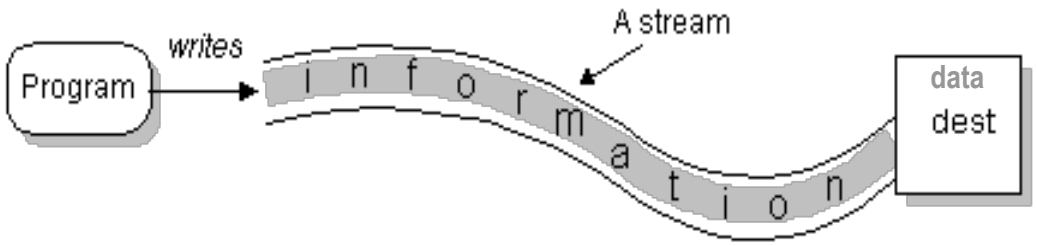
Open/Create a stream
while more information
read information
close the stream

If the source is a file on your hard drive, a **FileInputStream** object could be associated with this file. After you have a stream object, you can read information from that stream by using one of the object methods. FileInputStream includes a **read()** method that returns a byte read from a file.

When you're done reading information from the stream, you call the **close()** method to indicate that you're done using the stream.

- An **output stream** sends data out of a program to a destination.

For an output stream, we begin by creating an object that is associated with the data's destination.



To write data from to an input Stream:

Open\Create a stream
while more information
write information
close the stream

If the source is a file on your hard drive, a **FileOutputStream** object could be associated with this file.

The FileOutputStream **write()** method is the simplest way to send information to the output stream's destination. As with input streams, the **close()** method is called on an output stream when you have no more information to send.

An I/O class that has not been closed will generate an IOException if you try to use it again.

Filtering a Stream

The simplest way to use a stream is to create it and then call its methods to send or receive data, depending whether it's an output stream or an input stream. It is possible to achieve more sophisticated results by associating a filter with a stream before read or writing any data.

A filter *is a type of stream that modifies the way an existing stream is handled.*

The procedure for using a filter on a stream is basically as follows:

- **Create a stream** associated with a data source or a data destination.
- **Associate a filter** with that stream.
- **Read or write data** from the filter rather than the original stream.

The methods you call on a filter are the same as the methods you call on a stream: There are **read()** and **write()** methods, just as there would be on an unfiltered stream.

Byte Streams

All byte streams are either a subclass of InputStream or OutputStream.

File Streams

The byte streams we will work with the most are likely to be file streams, which are used to exchange data with files on your disk drive.

You can send bytes to a file output stream and receive bytes from a file input stream.

➢ File Input Streams

A file input stream can be created with the **FileInputStream(String)** constructor (a subclass of InputStream). The String argument should be the name of the file.

The following statement creates a file input stream for the file ***students.dat***

```
FileInputStream fis = new FileInputStream("students.dat");
```

After you create a file input stream, you can read bytes from the stream by calling its read() method.

The ReadBytes application reads a file input stream; Prints and counts the number of bytes read:

```
import java.io.*;

public class ReadBytes {
   public static void main(String[] arguments) {
      try {
            FileInputStream fis = new FileInputStream("students.dat");
            boolean eof = false;
            int count = 0;

            while (!eof) {
              int input = fis.read();
              System.out.print((char)input);
              if (input == -1)
                eof = true;
              else
                 count++;
              }

            fis.close();
            System.out.println("Number Bytes read: " + count);
      } catch (IOException e) {
            System.out.println("Error -- " + e.toString());
      }
   }
}
```

if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading. FileInputStreams throws a FileNotFoundException (an IOException).

(Error -- java.io.FileNotFoundException: classes.dat (The system cannot find the file specified)

➢ File Output Streams

A file output stream can be created with the **FileOutputStream(String)** constructor. The String argument should be the name of the file.

The following statement creates a file output stream for the file (fileNames.dat):

```
FileOuputStream fos = new FileOuputStream("fileName.dat");
```

You have to be careful when specifying the file to which to write an output stream. If it's the same as an existing file, the original file will be wiped out when you start writing data to the stream.

You can create a file output stream that appends data after the end of an existing file with the **FileOutputStream(String, boolean)** constructor.

The file output stream's write(int) method is used to write bytes to the stream. After the last byte has been written to the file, the stream's close() method closes the stream.

The WriteBytes application writes an integer array to a file output stream:


```
import java.io.*;
Import java.lang.Exception;

public class WriteBytes {
   public static void main(String[] arguments) {

      int[] data = {119, 114, 105, 116, 101, 66, 121, 116, 101, 115, 46, 106, 97, 118, 97, 10};

      try {
            FileOutputStream fos = new FileOutputStream("FileName.dat");

            for (int i = 0; i < data.length; i++)
                  fos.write(data[i]);

            fos.close();
      } catch (IOException e) {
         System.out.println("Error -- " + e.toString());
      }
   }
}
```

writeBytes.java

If the file exists but is a directory rather than a regular file, does not exist, cannot be created, or cannot be opened for any other reason then a **FileNotFoundException** is thrown.

Filtering a Stream

Filtered streams are streams that modify the information sent through an existing stream. They are created using one of the subclasses of **FilterInputStream** or **FilterOutputStream**.

Buffered Streams

A buffer is a storage place where data can be kept before it is needed by a program that reads or writes that data. By using a buffer, you can get data without always going back to the original source of data.

Buffered byte streams use the **BufferedInputStream(InputStream)** and **BufferedOutputStream(OutputStream)** classes (subclasses of the FilterInputStream and FilterOutputStreams respectively).

The simplest way to read data from a buffered input stream is to call its read() method. The simplest way to write data to a buffered output stream is to call its write() method.

When data is directed to a buffered stream, it will not be output to its destination until the buffer stream fills up, closed or the buffered stream's flush() method is called.

WriteBuffer.java

```
import java.io.*;

public class  WriteBuffer {
    WriteBuffer(){
            int[] data = {82, 101, 99, 116, 97, 110, 103, 108, 101, 46, 106, 97, 118, 97, 10};
            try {
                FileOutputStream fos = new FileOutputStream("student.dat", true);
                BufferedOutputStream buff = new BufferedOutputStream(fos);
                for (int i = 0; i < data.length ; i++) {
                        buff.write(data[i]);
                }
                buff.flush();
                buff.close();
            } catch (IOException e) {
                System.out.println("Exception: " + e.getMessage());
            }
    }

    public static void main(String[] arguments) {
            WriteBuffer write = new WriteBuffer();
    }
}
```

ReadBuffer.java

```
import java.io.*;
 public class  ReadBuffer {
    ReadBuffer(){
                try {
                    FileInputStream fis = new FileInputStream("student.dat");
                    BufferedInputStream buff = new BufferedInputStream(fis);

                    int in = 0;
                    do {
                                in = buff.read();
                                if (in != -1)
                                    System.out.print((char)in);
                    } while (in != -1);

                     buff.flush();
                    buff.close();
                } catch (IOException e) {
                    System.out.println("Exception: " + e.getMessage());
                }
    }
    public static void main(String[] arguments) {
                ReadBuffer write = new ReadBuffer(); }
 }
```

DataStreams - Filters

If you need to work with data that isn't represented as bytes or characters, you can use **data input** and **data output** streams.

These streams filter an existing byte stream so that each of the following primitive types can be read or written directly from the stream:

boolean, byte, double, float, int, long and short.

A data input stream is created using a **DataInputStream(InputStream)** constructor. Conversely, a data output stream requires the **DataOutputStream(OutputStream)** constructor.

The argument should be an existing input/output stream such as a buffered input stream or a file output stream.

readBoolean()	writeBoolean(boolean)	readLong()	writeLong(long)
readDouble()	writeDouble(double)	readInt()	writeInt(int)
readFloat()	writeFloat(float)	readShort()	writeShort(short)

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

   try{
          FileOutputStream file = new FileOutputStream("prices.dat");
          BufferedOutputStream buff = new BufferedOutputStream(file);
          DataOutputStream dos = new DataOutputStream(buff);

          dos.writeDouble(39.95);
          dos.writeDouble(3.22);
          dos.writeDouble(1.08);

          dos.close();
          buff.close();
          file.close();

        }catch (IOException e) {
          System.out.println("Error -- " + e.toString());
       }

    }
}
```

```
import java.io.*;

class ReadPrices {
    public static void main(String[] arguments) {
        try {
            FileInputStream file = new FileInputStream("prices.dat");
            BufferedInputStream buff = new  BufferedInputStream(file);
            DataInputStream data = new DataInputStream(buff);

            try {
                                while (true) {
                                    double in = data.readDouble();
                                    System.out.println(“Price = “ + in);
                }
            } catch (EOFException eof) {
                buff.close();
            }
        } catch (IOException e) {
            System.out.println("Error -- " + e.toString());
        }
    }
}
```

```
Price = 39.95

Price = 3.22

Price = 1.08
```

Character Streams

Character streams are used to work with any text that is represented by an ASCII character set or Unicode, an international character set that includes ASCII.

Reading Text Files

FileReader is the main class used when reading character streams from a file. This class inherits from InputStreamReader, which reads a byte stream and converts the bytes into integer values that represent Unicode characters.

A character input stream is associated with a file using the FileReader(String) constructor. The following statement creates a new FileReader called look and associates it with a text file called index.html:

```
FileReader look = new FileReader("index.html");
```

The read() method of FileReader returns the next character in the stream as integer.

If you want to read a line of text at a time instead of reading a file character by character, you can use the **BufferedReader** class in conjunction with a **FileReader**.

The readline() method of the BufferedReader class returns a String object containing the next line of text on the stream, not including the character or characters that represent the end of file.

Reading files

```
import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class NameSorter {
    public static final String NAMES_FILE = "names.txt";

    public static void main(String[] args) {
        try {
            BufferedReader file = new BufferedReader(
                new FileReader(NAMES_FILE)
            );

            List<String> names = new ArrayList<>();

            while(true) {
                String name = file.readLine();
                if(name == null) {
                    // reached end of file
                    file.close();
                    break;
                }
                names.add(name);
            }

            Collections.sort(names);
            for(String name : names) {
                System.out.println(name);
            }
        } catch(FileNotFoundException ex) {
            System.out.println("File not found: " + NAMES_FILE);
        } catch(IOException ex) {
            System.out.println("Error while reading the file:");
            System.out.println(ex.getMessage());
        }
    }
}
```

The file names.txt contains the names, one on each line, in a random order. The program reads the names from that file, collects them into a list, sorts them, and prints them:

```
Alice
Bob
Charles
Diane
Elaine
```

Don’t use ‘Scanner’ with System.in

Many authors recommend using the Scanner class to read from the keyboard, as a simpler alternative to the BufferedReader class. This is a bad idea!

Scanner is not suitable for keyboard input, because users often enter invalid input and need to try again. Unfortunately, Scanner is not good at recovering from errors.

Consider the following method, which is supposed to ask a user to enter a number, and keep asking until they do:

```
public int readNumber() {
    Scanner keyboard = new Scanner(System.in);

    System.out.println("Please enter a number.");
    while(true) {
        try {
            System.out.print("> ");
            return keyboard.nextInt();
        } catch(InputMismatchException ex) {
            System.out.println("Please try again.");
        }
    }
}
```

Reading from the keyboard

The keyboard is represented by the object System.in. This object is an "input stream", so to read from it you need an InputStreamReader and a BufferedReader.

For example:

```
import java.io.*;

public class Parrot {
    public static void main(String[] args) throws IOException {
        BufferedReader keyboard = new BufferedReader(
            new InputStreamReader(System.in)
        );

        System.out.println("Enter some text.");
        System.out.print("> ");
        String line = keyboard.readLine();
        System.out.println("The parrot says: " + line);
    }
}
```

```
Enter some text.
> Polly wants a cracker
The parrot says: Polly wants a cracker
```

Writing Text Files

The FileWriter class is used to write a character stream to a file.

There are two FileWriter constructors: **FileWriter(String)** and **FileWrite(String, boolean).** The string indicates the name of the file that the character stream will be directed into, which can include a folder path. The optional boolean argument should equal true if the stream is to be appended to an existing text file.

The following example writes a character stream to a file using the FileWriter class and its write() method:

```
FileWriter  letters = new FileWriter("alphabet.txt");
for (int  i= 65;  i < 91;  i++)
    letters.write( (char)i  );
letters.close();
```

```
ABCDEFGHIJKLMNOPKRSTUVWXYZ
```

The close() method is used to close the stream after all characters have been sent to the destination file.

The **BufferedWriter** class can be used to write a buffered character stream.

Writing Text Files

The **PrintWriter** class can also be used to write a buffered character stream.

```
FileWriter fileOut = new  FileWriter("FileName");
PrintWriter print = new  PrintWriter(fileOut);

print.println(String);
```

Writing files

A program can write to a file using a FileWriter object. This must be wrapped in a PrintWriter object to write a whole line at a time. For example:

```
public class WorstTextEditor {
    public static final String OUTPUT_FILE = "out.txt";

    public static void main(String[] args) throws IOException {
        BufferedReader keyboard = new BufferedReader(
            new InputStreamReader(System.in)
        );
        PrintWriter file = new PrintWriter(
            new FileWriter(OUTPUT_FILE)
        );

        System.out.println("Writing to " + OUTPUT_FILE);
        System.out.println("Enter 'end' to stop writing.");
        while(true) {
            System.out.print("> ");
            String line = keyboard.readLine();
            if("end".equals(line)) {
                break;
            }
            file.println(line);
        }

        file.close();
    }
}
```

This program reads lines of text from the keyboard, and writes them to a file. The writing is done by the file.println(line); statement.

When the user enters the end command, the loop stops and the file is closed. This program doesn't catch any exceptions, so if something goes wrong it will simply crash.

Files and Filename Filters

If you want to copy files, rename files, or handle other tasks involving a file, **a file object must be created**.

The File class, which is also part of the *java.io* package, represents a file or a folder reference. The following File constructors can be used:

File(String) ----Creates a File object with the specified name.

File(String, String) ----Creates a File object within the specified folder path and the specified name.

You can call several useful methods on a File object.

- The exist() method returns a Boolean value indicating whether the file exists under the name and folder path established when the File object was created.
- The length() method returns a long integer indicating the size of the file in bytes.
- The renameTo(File) method renames the file to the name specified by the File argument.
- The delete() method should be called to delete a file or a folder, returns a boolean.
- The getName() Returns the name of the file or directory denoted by this abstract pathname.

Converts all characters in a file to capital letter

```
public void convert(){
try {
        File source = new File("Story.dat"); // Create file objects
        File temp = new File("cap" + source.getName() + ".tmp");

       Create a BufferedReader to read from the source file

       Create a BufferedWriter to write to the temp file


        boolean eof = false;
         int inChar = 0;
        do {
              inChar = in.read();
              if (inChar != -1) {
                 char outChar = Character.toUpperCase( (char)inChar );
                 out.write(outChar);
              } else
                 eof = true;
            } while (!eof);
         in.close();
        out.close();
         boolean deleted = source.delete();
         if (deleted)
              temp.renameTo(source);
     } catch (IOException e) {
              System.out.println("Error -- " + e.toString());
        }
   }
```