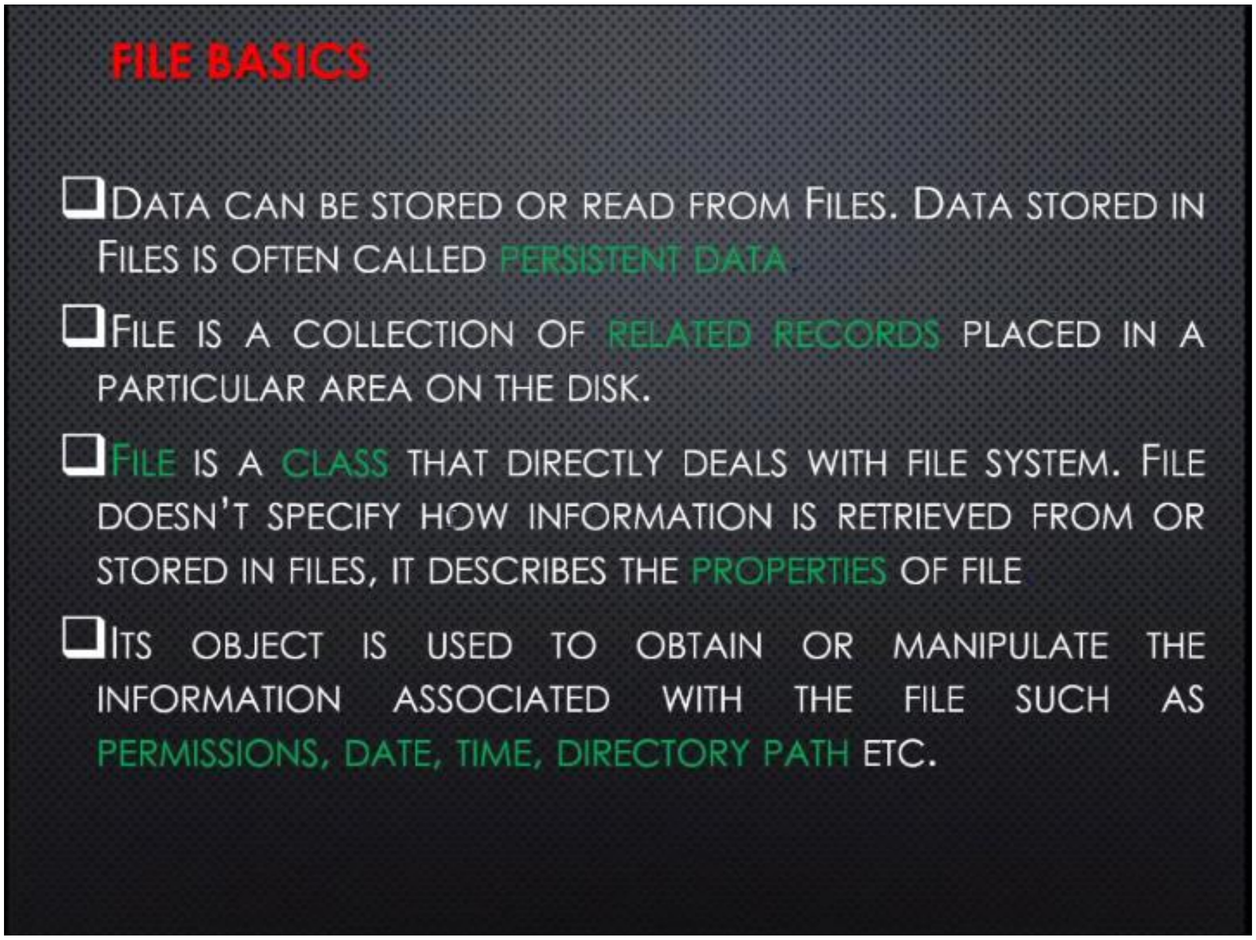
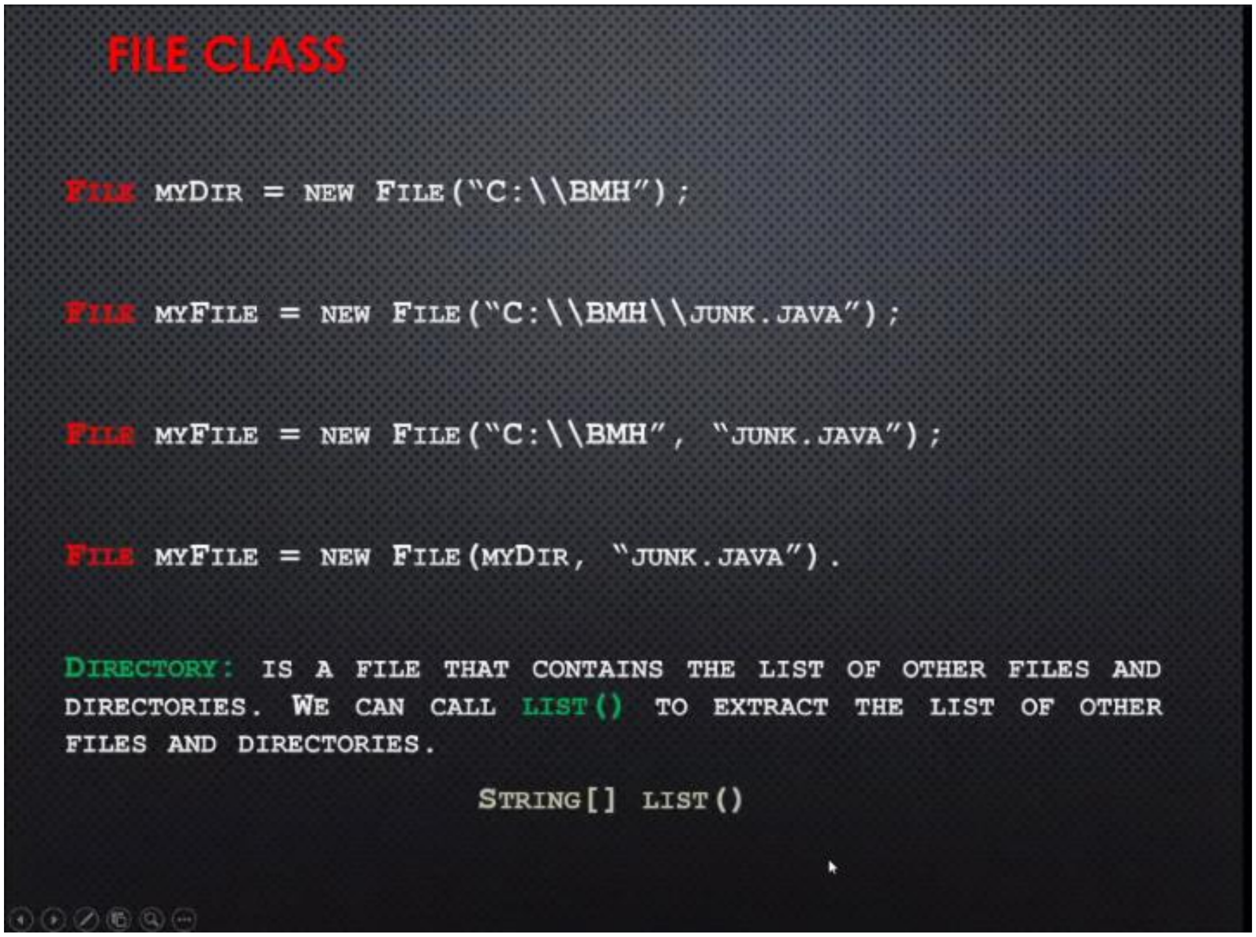
File I/O



FILE BASICS

- Data can be stored or read from Files. Data stored in Files is often called persistent data.
- File is a collection of related records placed in a particular area on the disk.
- File is a class that directly deals with file system. File doesn't specify how information is retrieved from or stored in files, it describes the properties of file.
- Its object is used to obtain or manipulate the information associated with the file such as permissions, date, time, directory path etc.

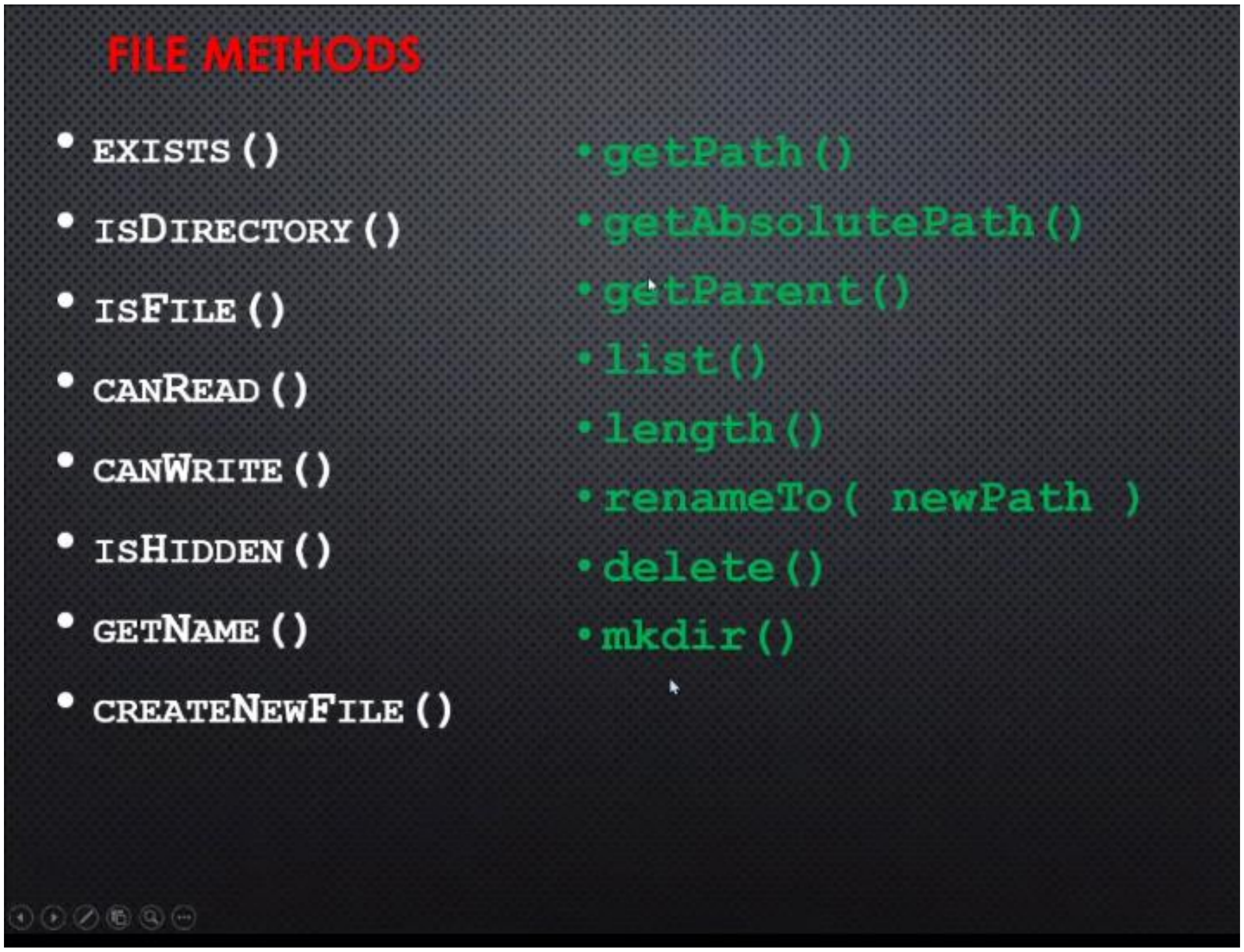


FILE CLASS

```
File myDir = new File("C:\\BMH");

File myFile = new File("C:\\BMH\\junk.java");

File myFile = new File("C:\\BMH", "junk.java");

File myFile = new File(myDir, "junk.java").
```

Directory: is a file that contains the list of other files and directories. We can call list() to extract the list of other files and directories.

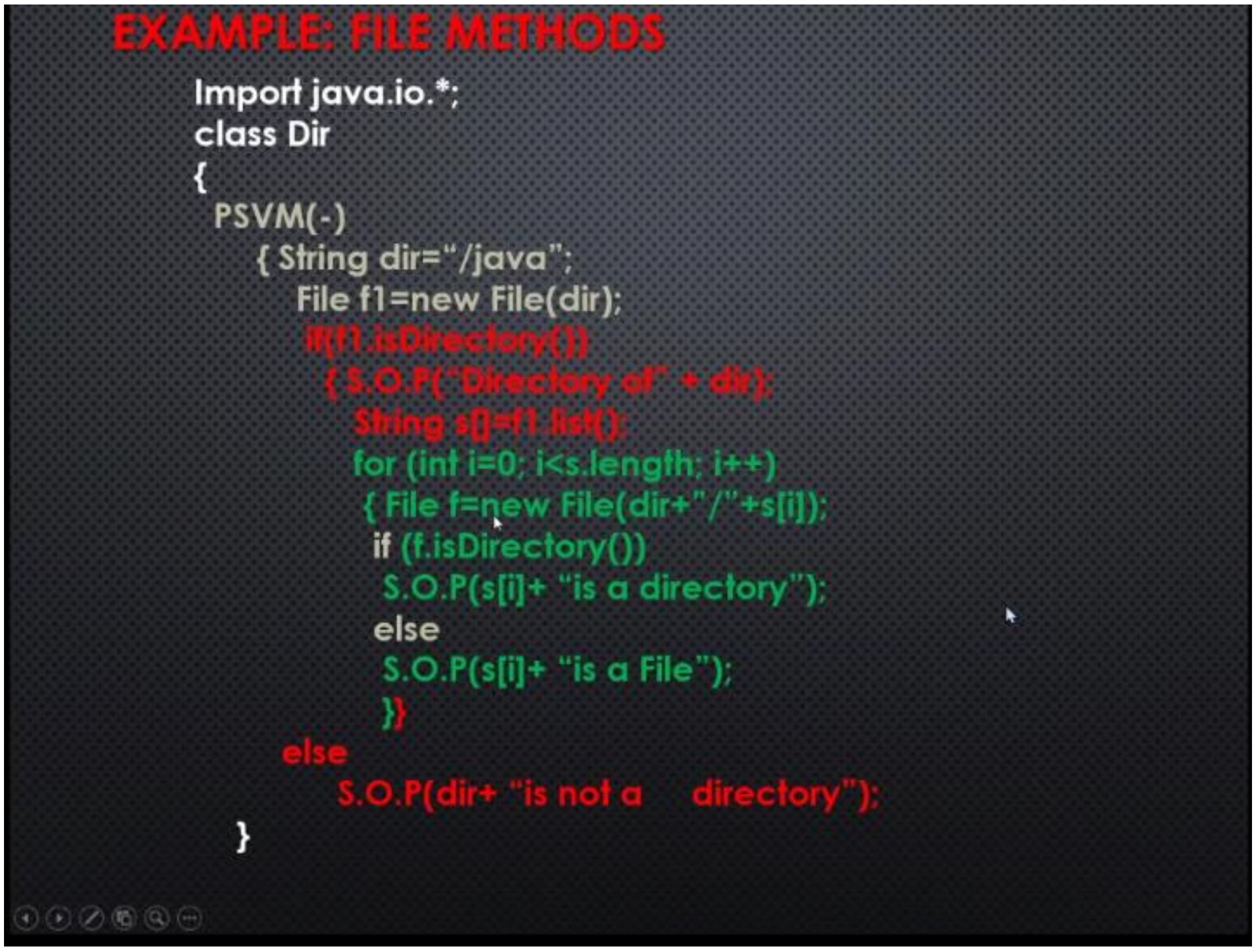
```
String[] list()
```



FILE METHODS

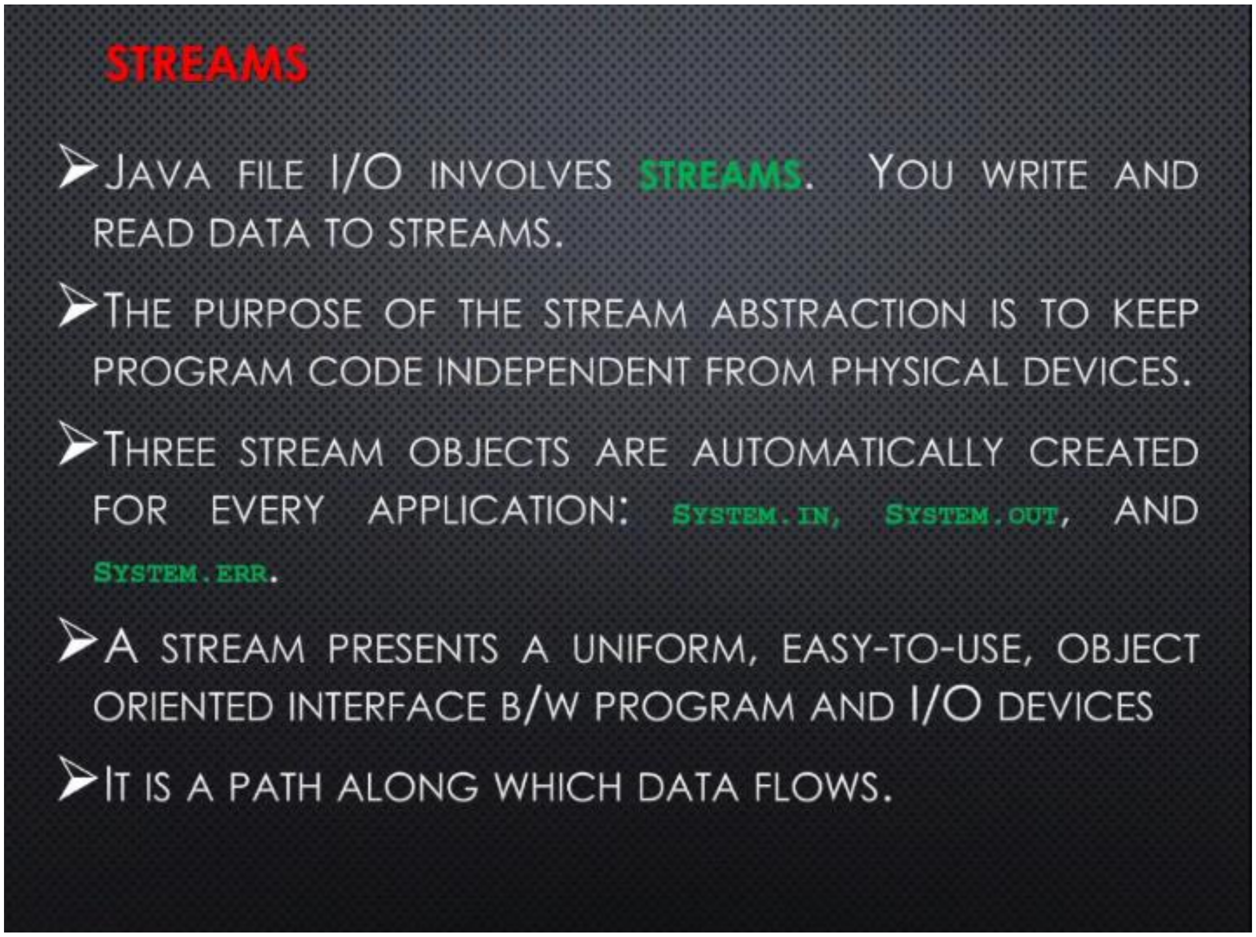
- EXISTS()
- ISDIRECTORY()
- ISFILE()
- CANREAD()
- CANWRITE()
- ISHIDDEN()
- GETNAME()
- CREATENEWFILE()

- getPath()
- getAbsolutePath()
- getParent()
- list()
- length()
- renameTo(newPath)
- delete()
- mkdir()



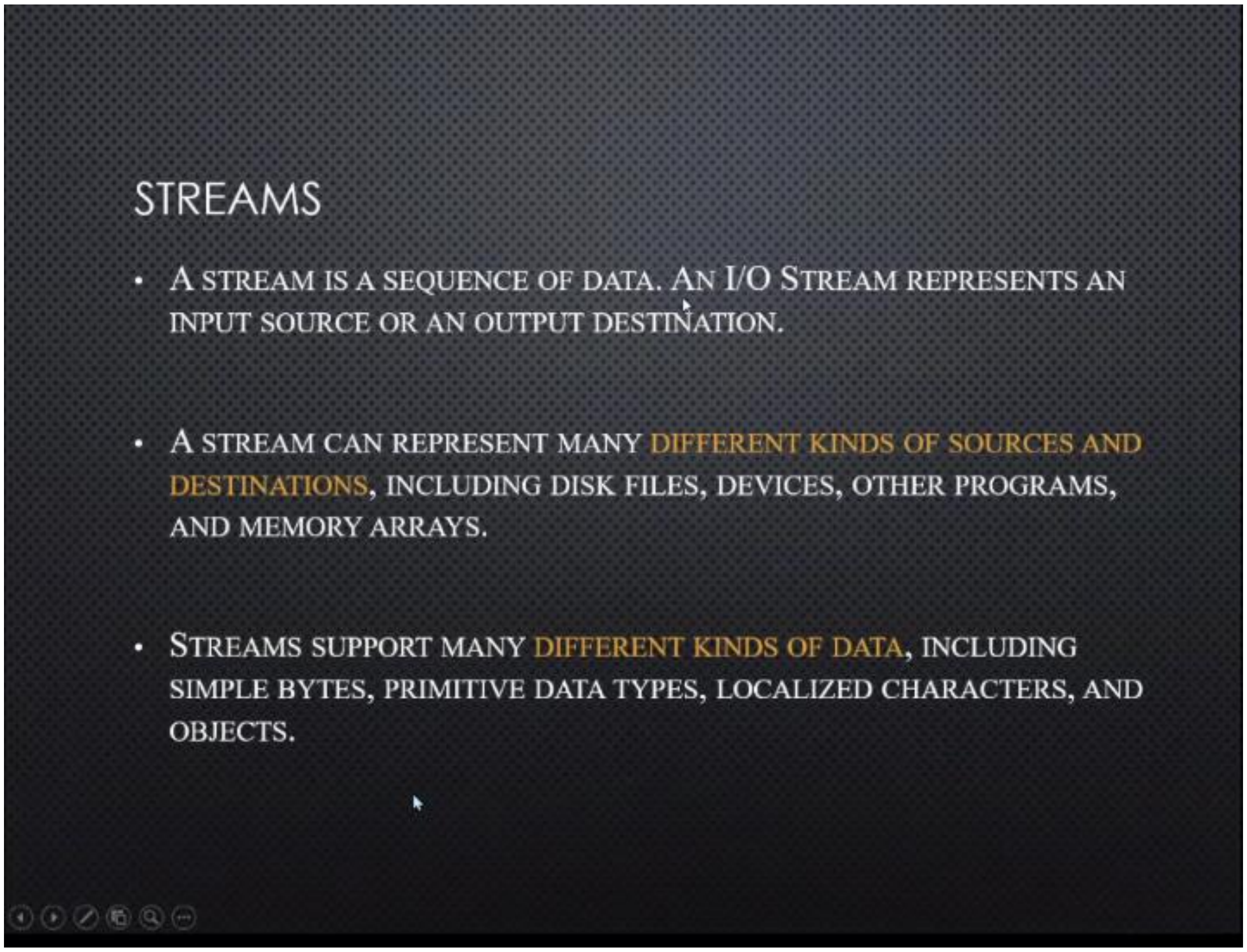
EXAMPLE: FILE METHODS

```
Import java.io.*;
class Dir
{
 PSVM(-)
    { String dir="/java";
        File f1=new File(dir);
         if(f1.isDirectory())
          { S.O.P("Directory of" + dir);
             String s[]=f1.list();
             for (int i=0; i<s.length; i++)
              { File f=new File(dir+"/"+s[i]);
               if (f.isDirectory())
                S.O.P(s[i]+ "is a directory");
               else
                S.O.P(s[i]+ "is a File");
                }}
        else
            S.O.P(dir+ "is not a    directory");
    }
```



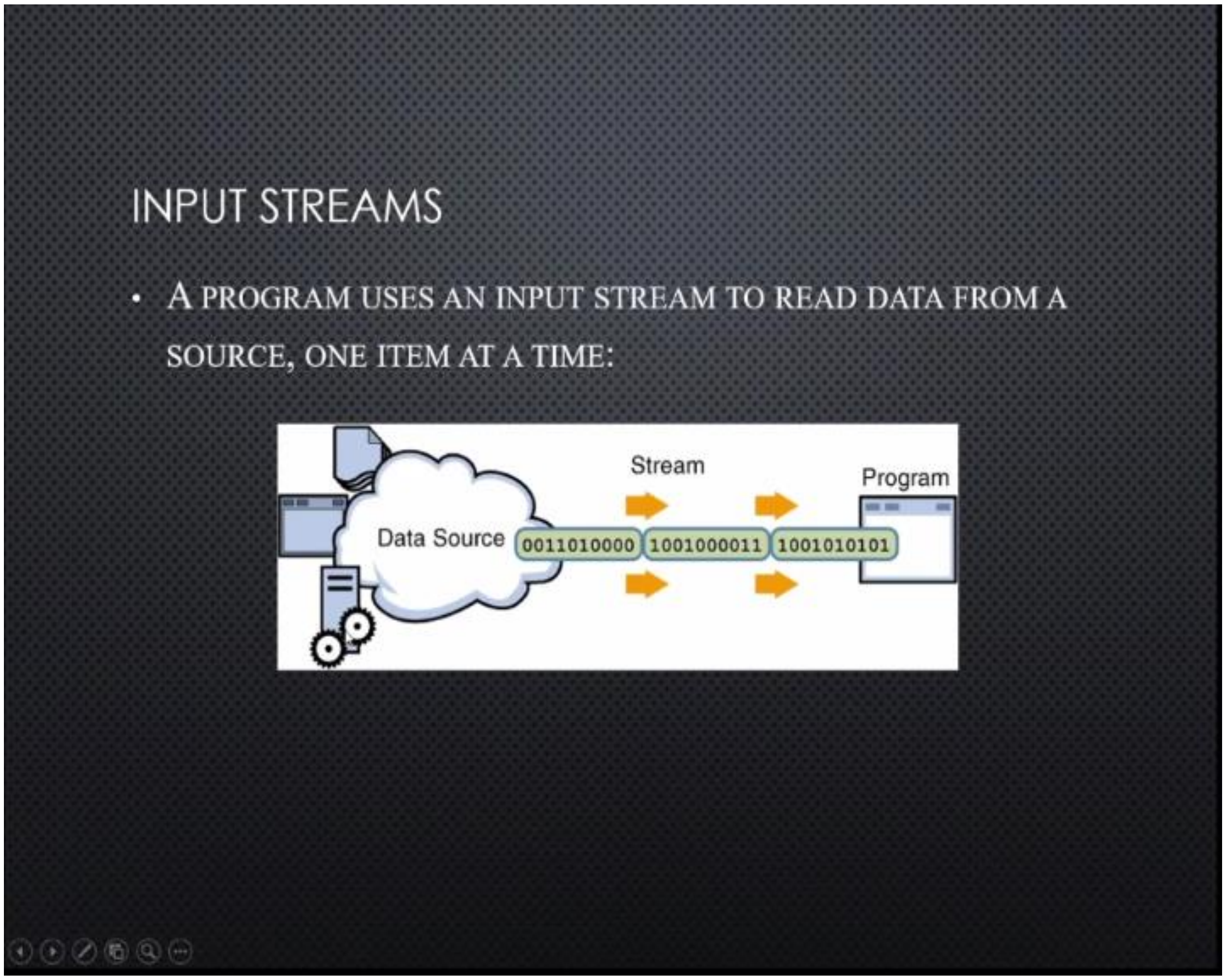
STREAMS

- ➢Java file I/O involves **streams**. You write and read data to streams.
- ➢The purpose of the stream abstraction is to keep program code independent from physical devices.
- ➢Three stream objects are automatically created for every application: `System.in`, `System.out`, and `System.err`.
- ➢A stream presents a uniform, easy-to-use, object oriented interface b/w program and I/O devices
- ➢It is a path along which data flows.



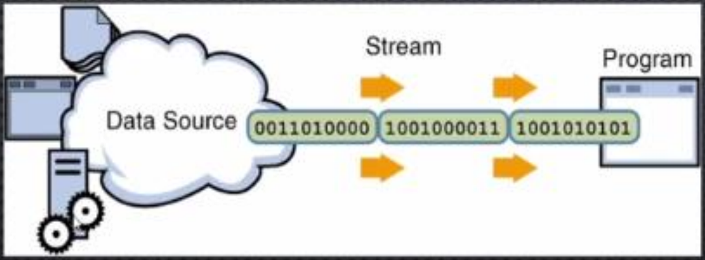
STREAMS

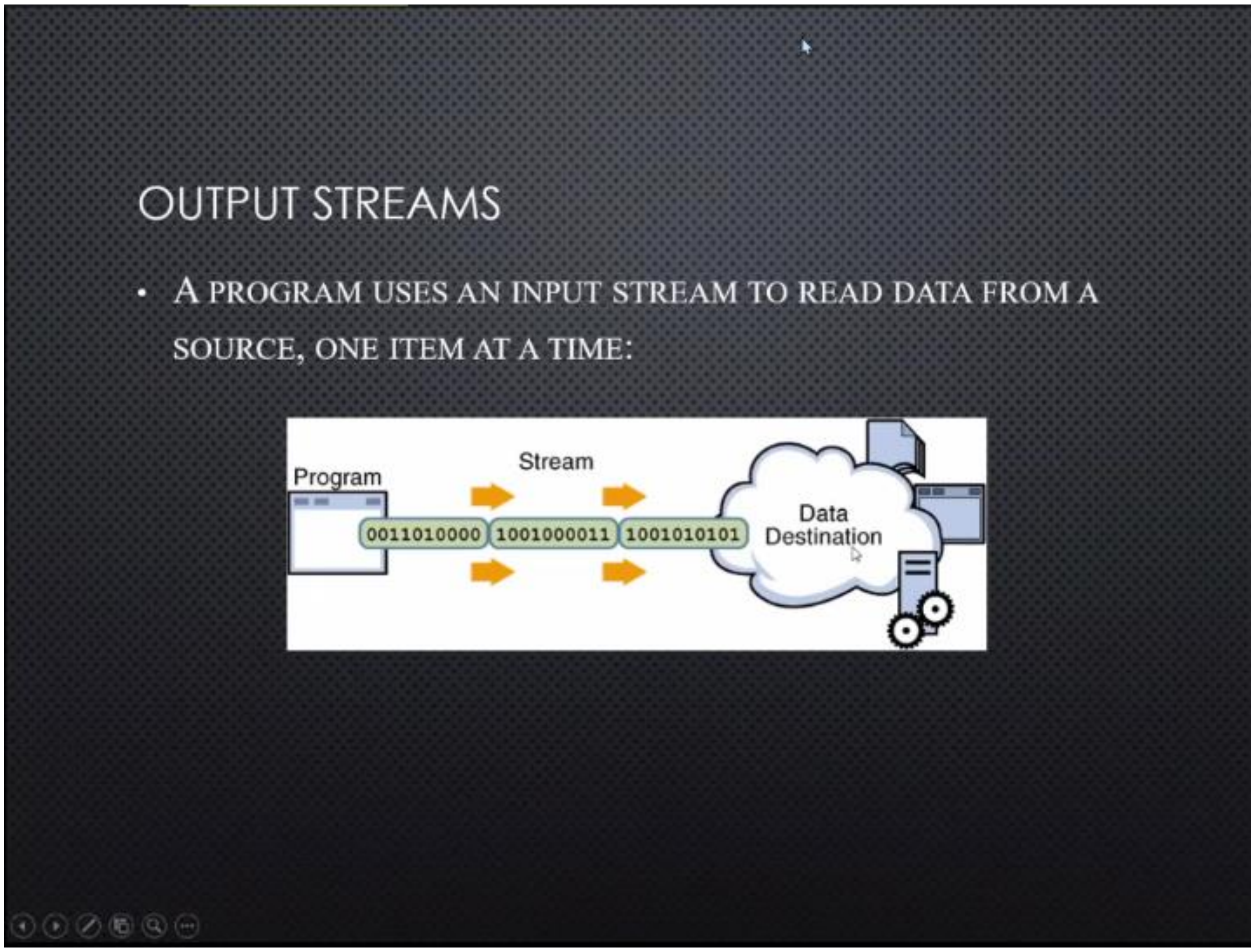
- A stream is a sequence of data. An I/O Stream represents an input source or an output destination.
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.



INPUT STREAMS

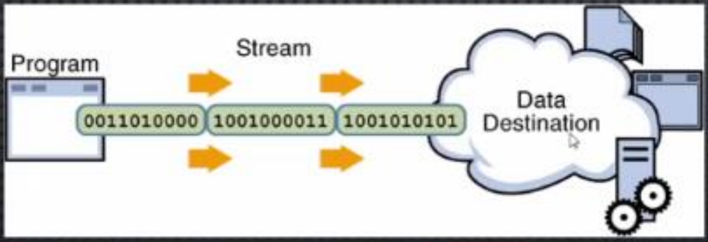
- A PROGRAM USES AN INPUT STREAM TO READ DATA FROM A SOURCE, ONE ITEM AT A TIME:

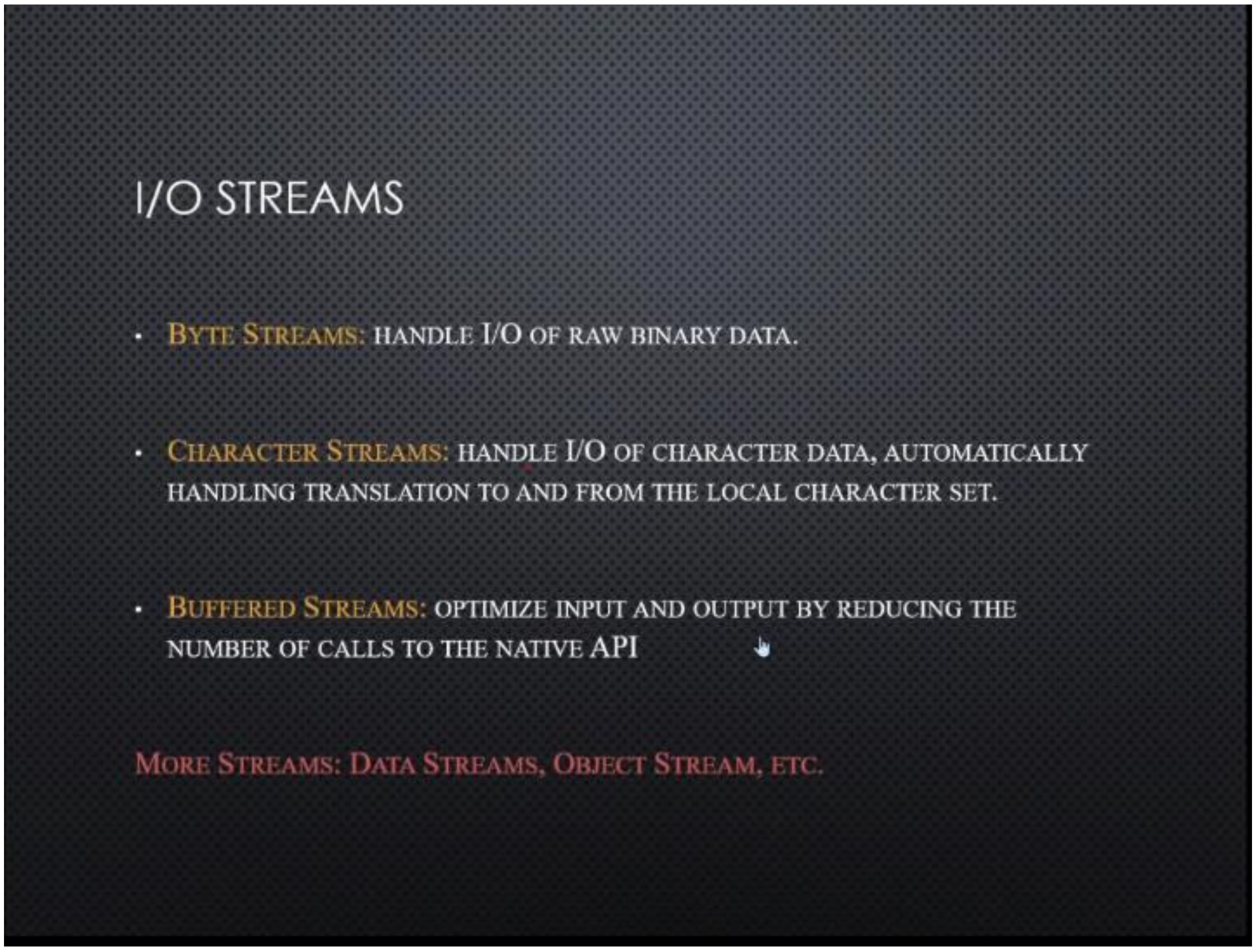




OUTPUT STREAMS

- A PROGRAM USES AN INPUT STREAM TO READ DATA FROM A SOURCE, ONE ITEM AT A TIME:

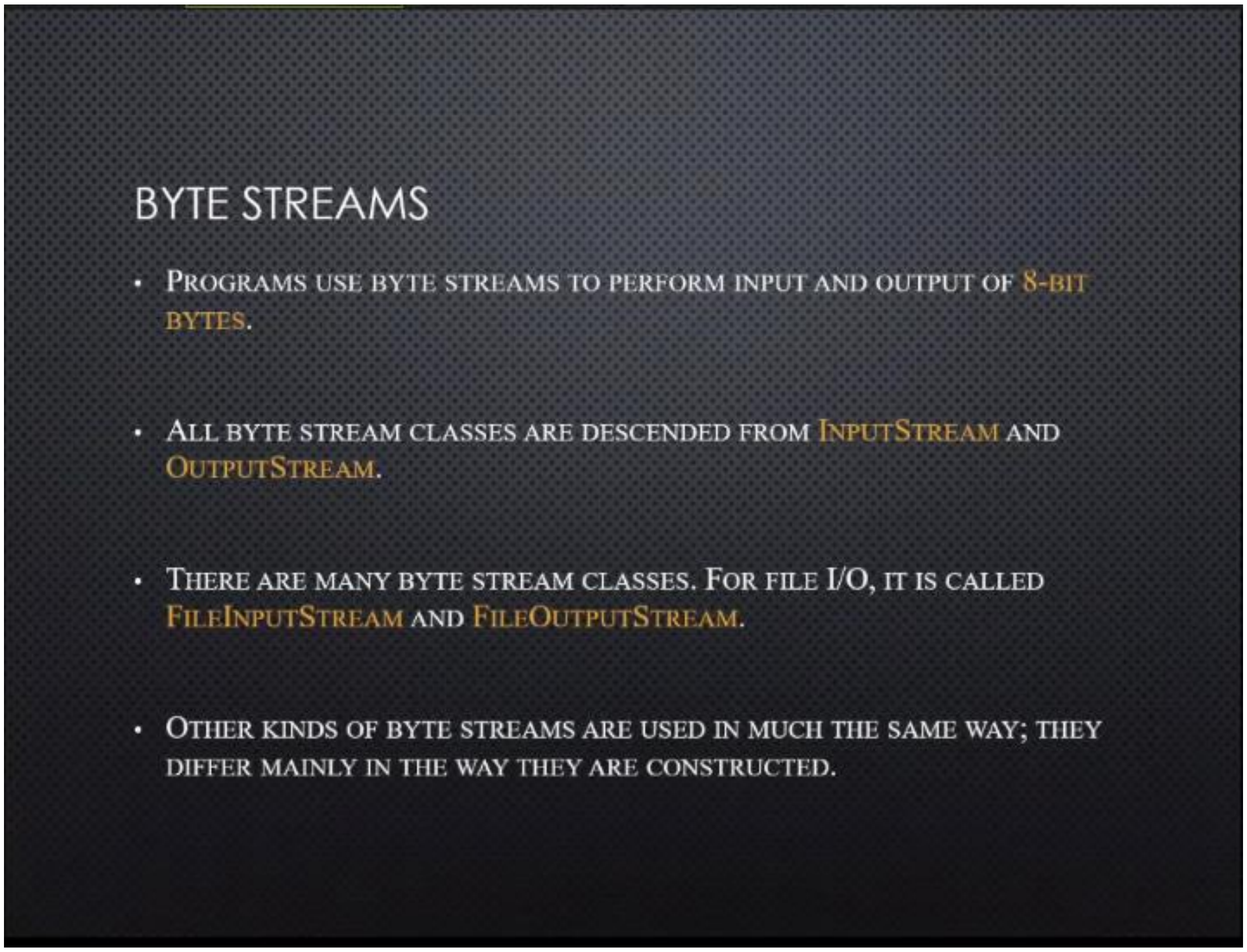




I/O STREAMS

- Byte Streams: handle I/O of raw binary data.
- Character Streams: handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams: optimize input and output by reducing the number of calls to the native API

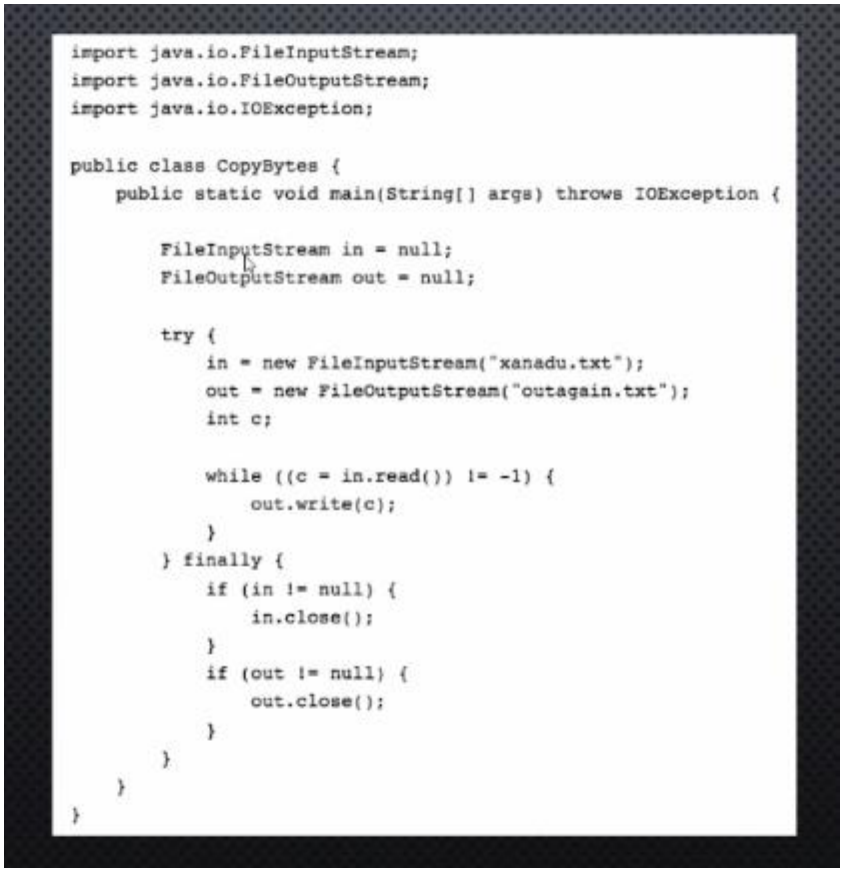
More Streams: Data Streams, Object Stream, etc.



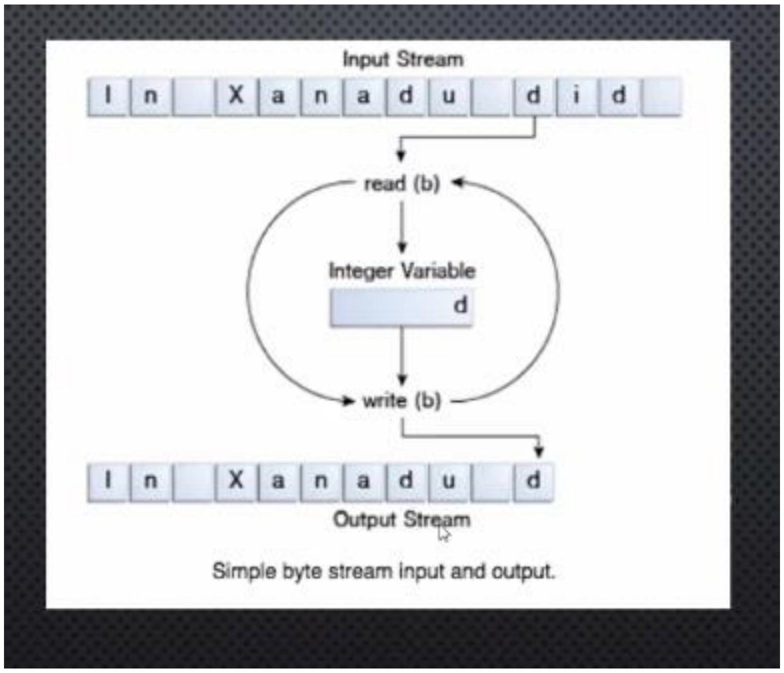
BYTE STREAMS

- Programs use byte streams to perform input and output of 8-bit bytes.
- All byte stream classes are descended from InputStream and OutputStream.
- There are many byte stream classes. For file I/O, it is called FileInputStream and FileOutputStream.
- Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

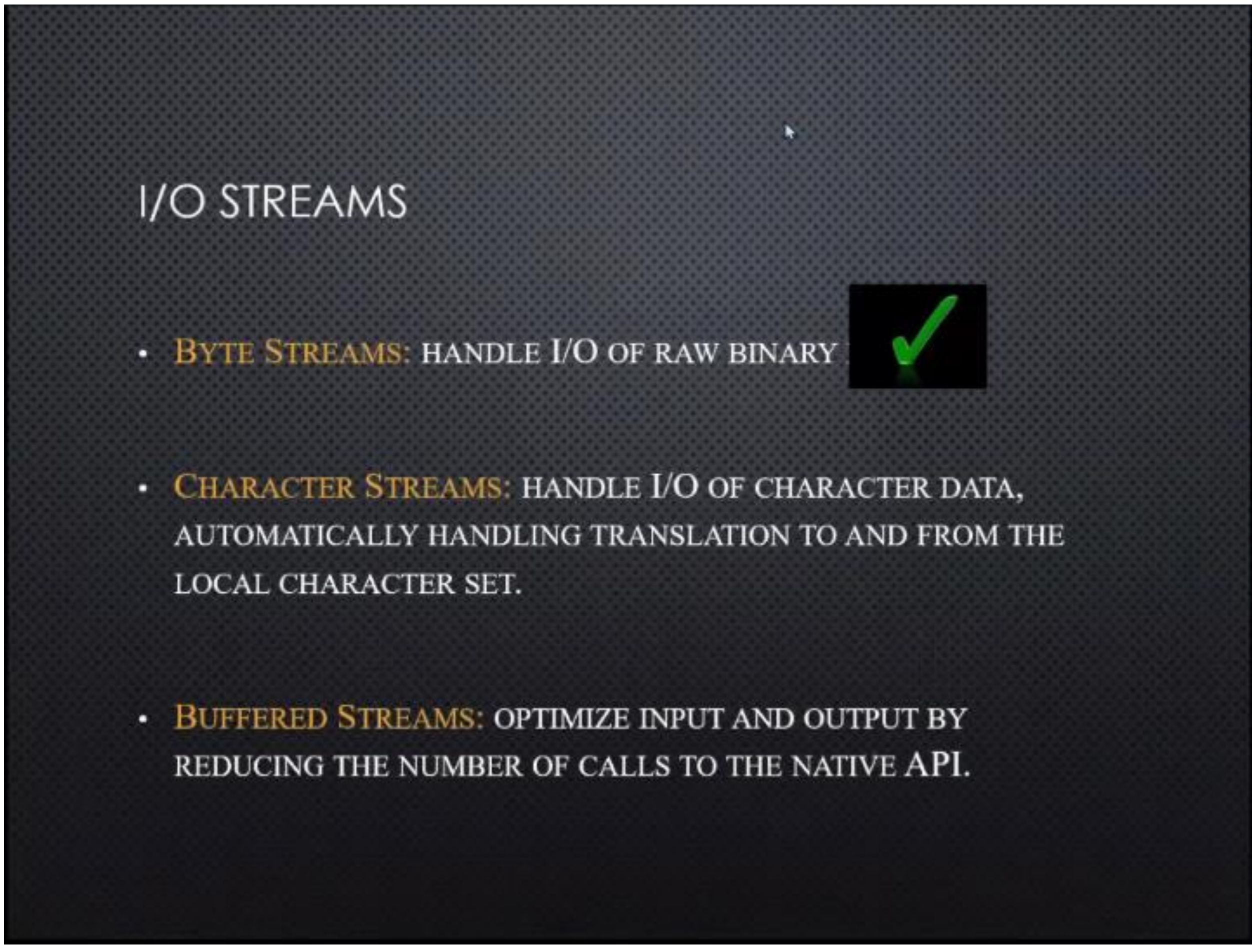
EXAMPLE PROGRAM: COPYBYTES



```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

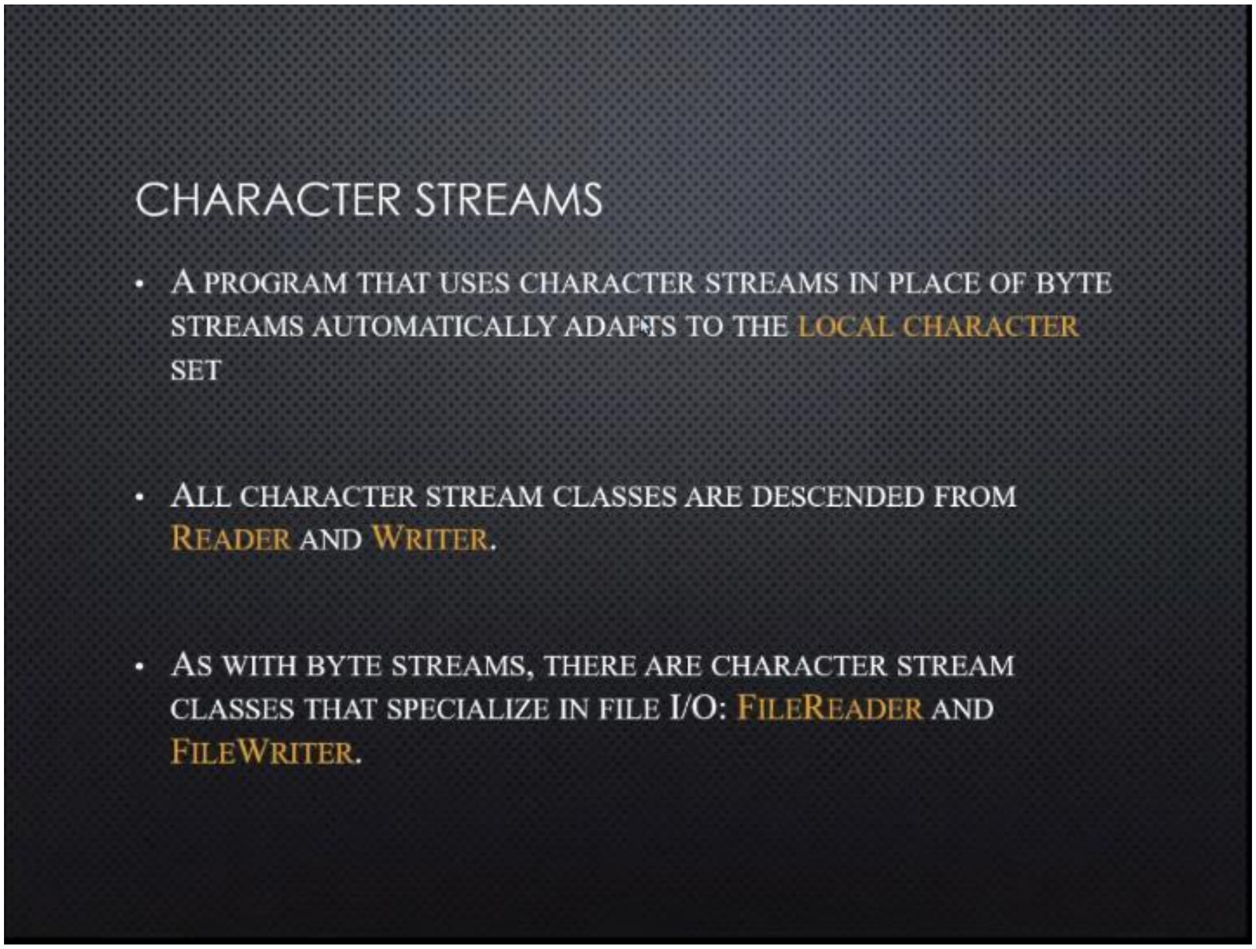


Simple byte stream input and output.



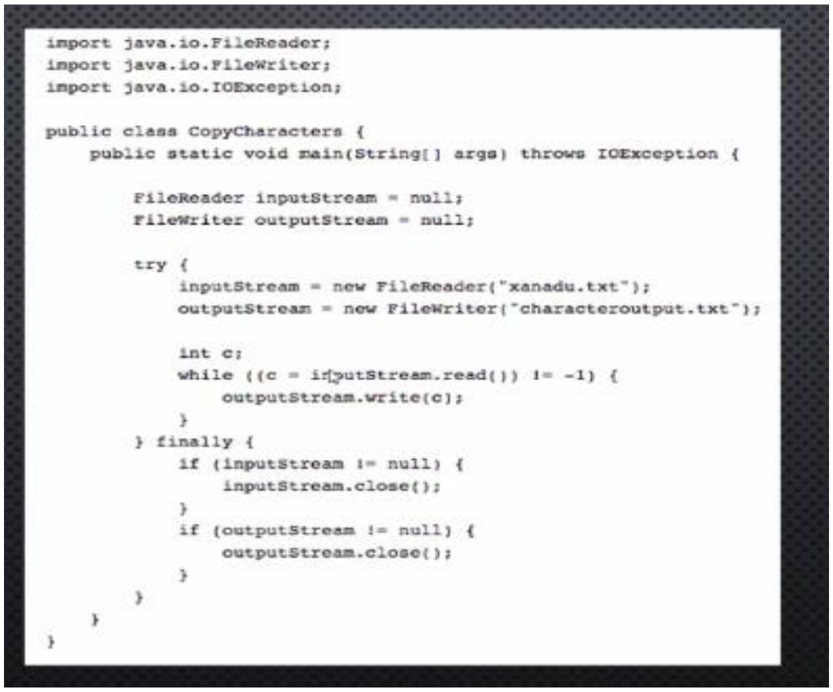
I/O STREAMS

- BYTE STREAMS: HANDLE I/O OF RAW BINARY ✓
- CHARACTER STREAMS: HANDLE I/O OF CHARACTER DATA, AUTOMATICALLY HANDLING TRANSLATION TO AND FROM THE LOCAL CHARACTER SET.
- BUFFERED STREAMS: OPTIMIZE INPUT AND OUTPUT BY REDUCING THE NUMBER OF CALLS TO THE NATIVE API.

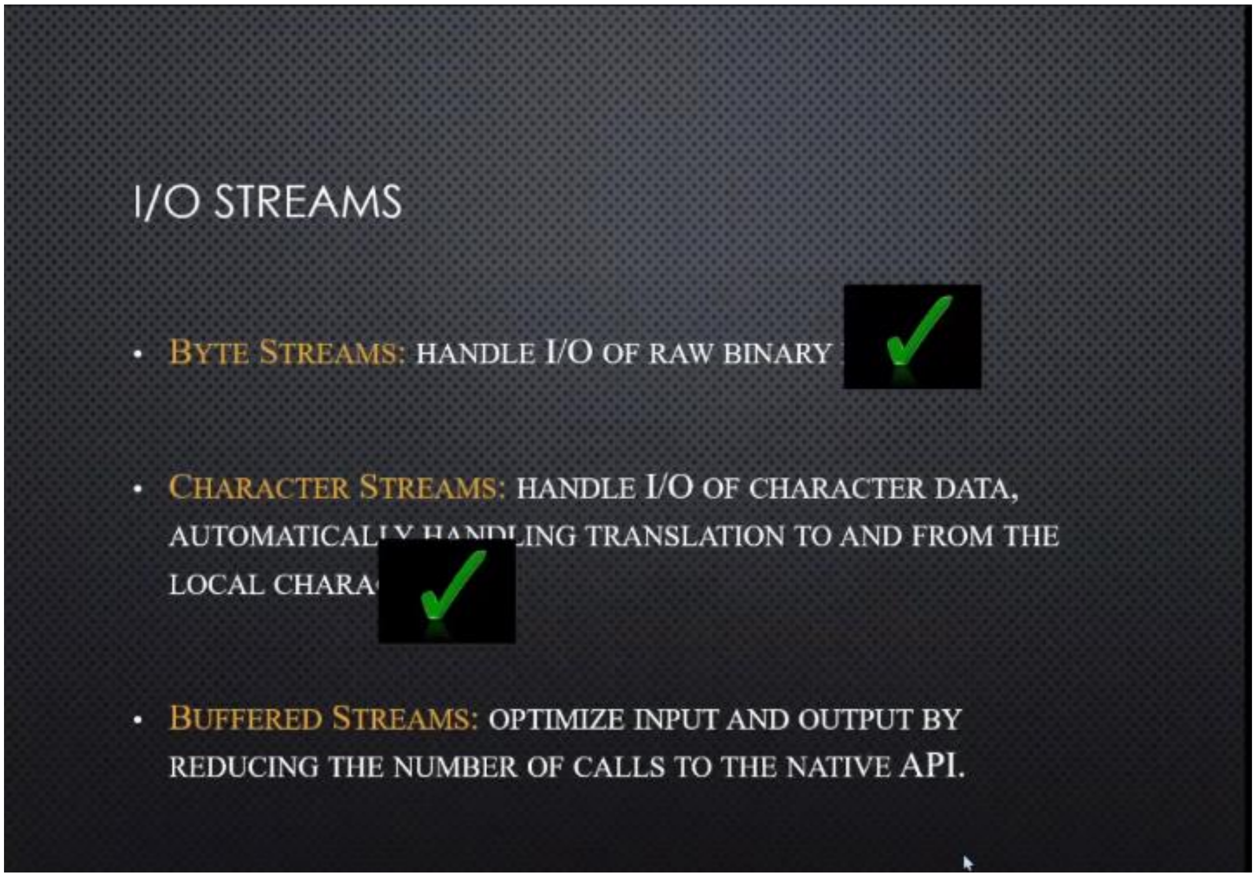


CHARACTER STREAMS

- A PROGRAM THAT USES CHARACTER STREAMS IN PLACE OF BYTE STREAMS AUTOMATICALLY ADAPTS TO THE LOCAL CHARACTER SET
- ALL CHARACTER STREAM CLASSES ARE DESCENDED FROM READER AND WRITER.
- AS WITH BYTE STREAMS, THERE ARE CHARACTER STREAM CLASSES THAT SPECIALIZE IN FILE I/O: FILEREADER AND FILEWRITER.

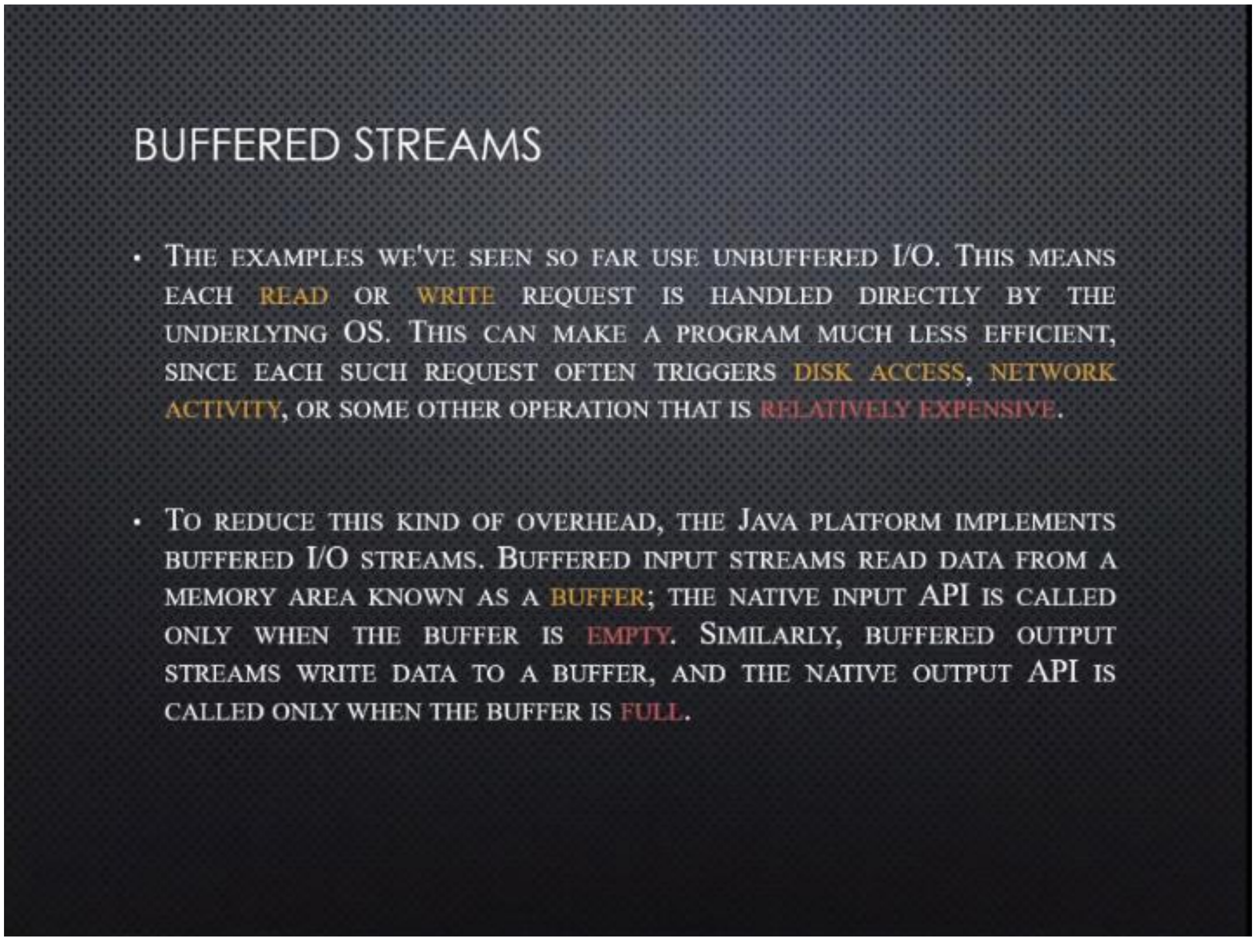


```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```



I/O STREAMS

- BYTE STREAMS: HANDLE I/O OF RAW BINARY ✓
- CHARACTER STREAMS: HANDLE I/O OF CHARACTER DATA, AUTOMATICALLY HANDLING TRANSLATION TO AND FROM THE LOCAL CHARA ✓
- BUFFERED STREAMS: OPTIMIZE INPUT AND OUTPUT BY REDUCING THE NUMBER OF CALLS TO THE NATIVE API.



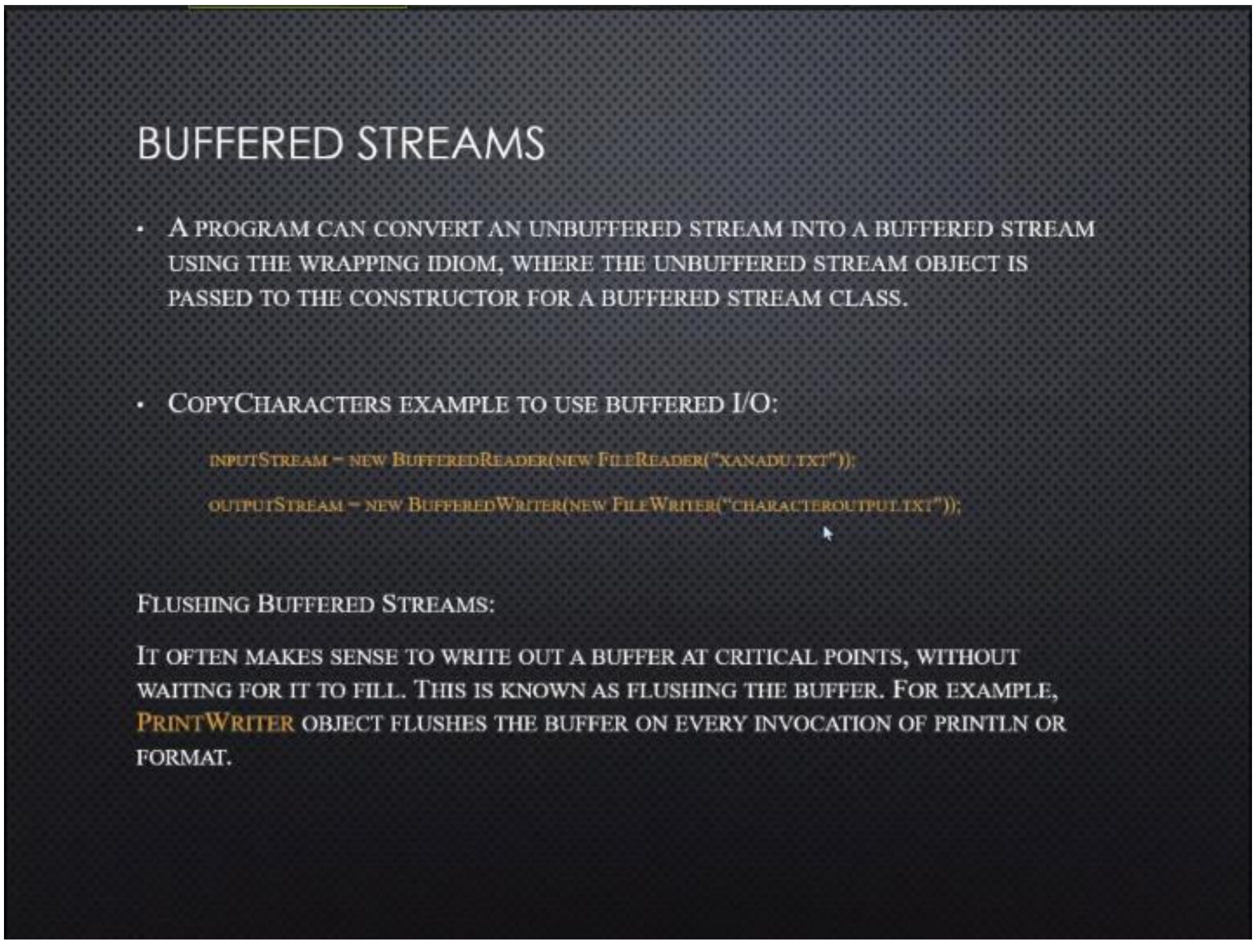
BUFFERED STREAMS

- The examples we've seen so far use unbuffered I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.
- To reduce this kind of overhead, the Java platform implements buffered I/O streams. Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

Buffer er maddhome block akare byte/char read write kora hoy

>>>A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods. When the BufferedInputStream is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time.

>>>Java.io.BufferedOutputStream class implements a buffered output stream. By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.



Buffered Streams

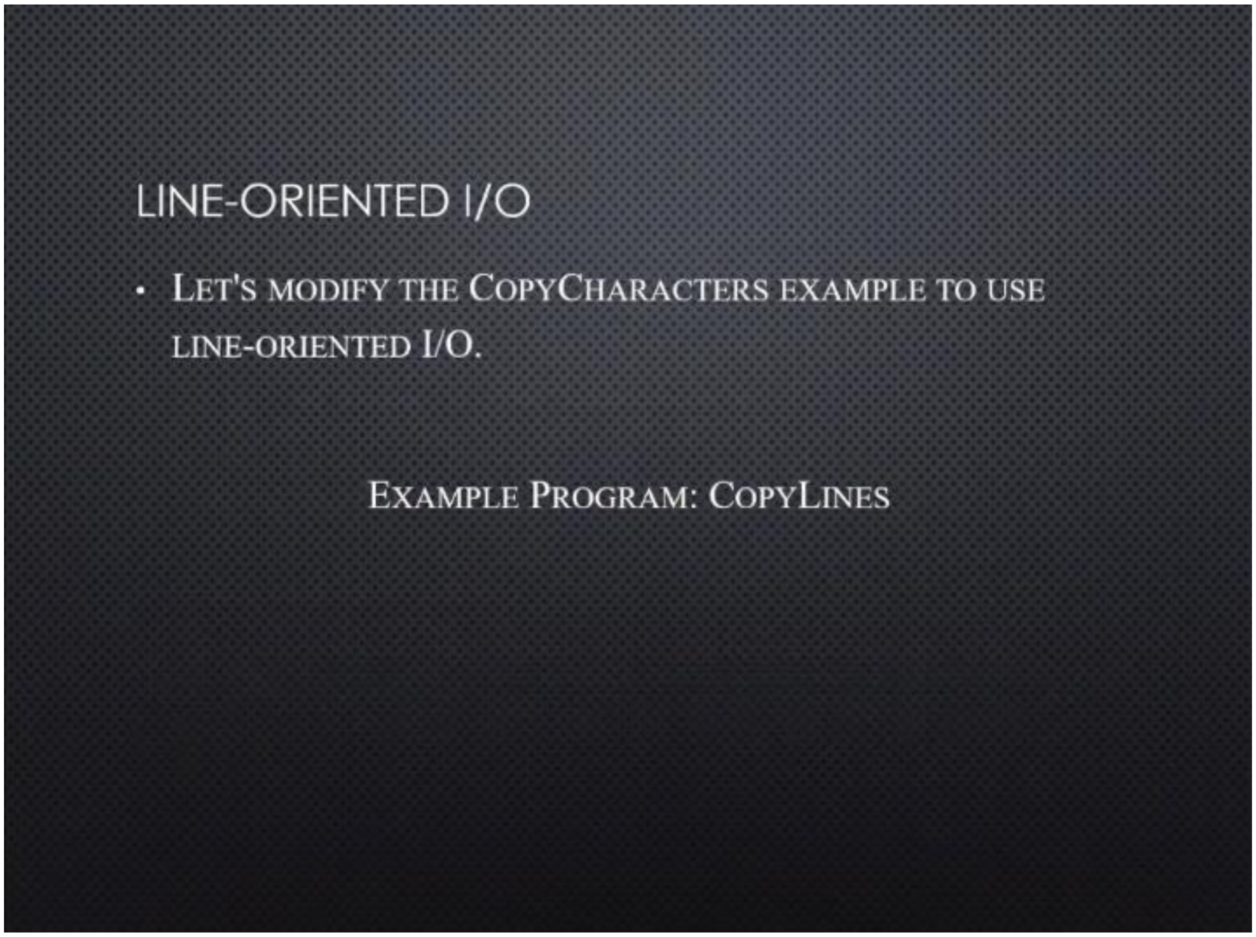
- A program can convert an unbuffered stream into a buffered stream using the wrapping idiom, where the unbuffered stream object is passed to the constructor for a buffered stream class.

- CopyCharacters example to use buffered I/O:

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

Flushing Buffered Streams:

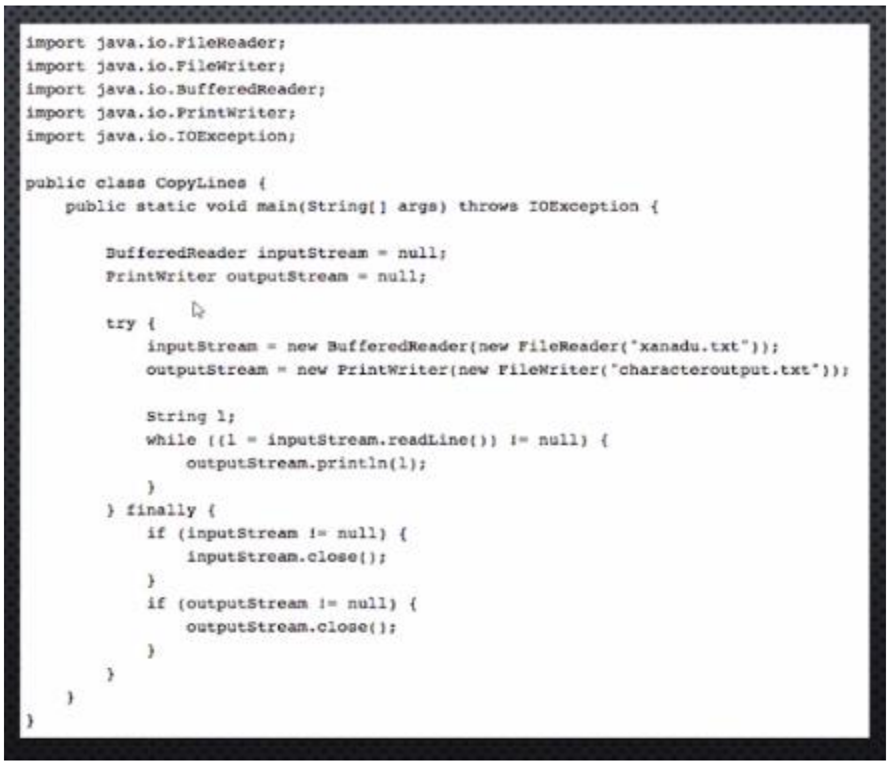
It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer. For example, PrintWriter object flushes the buffer on every invocation of println or format.



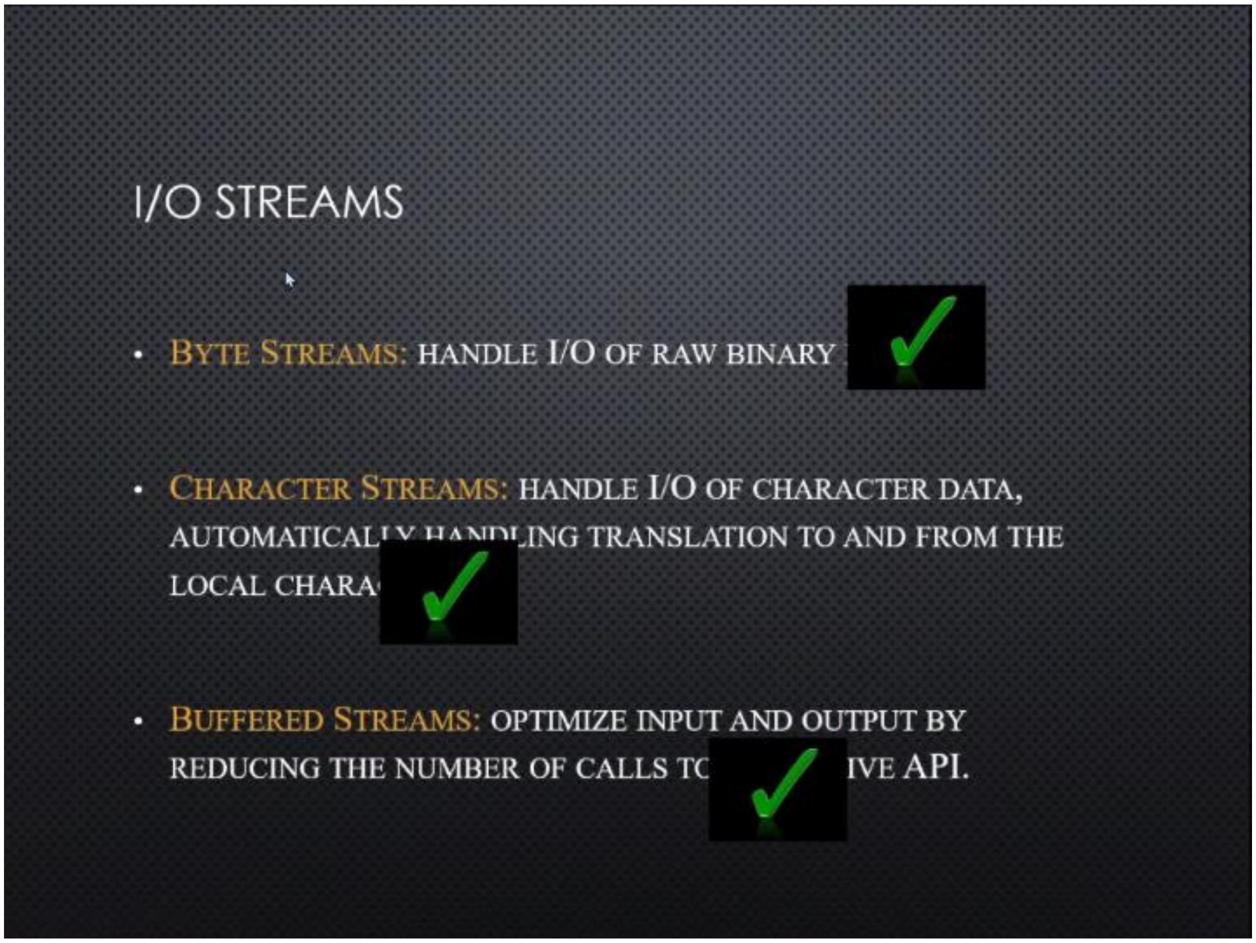
LINE-ORIENTED I/O

- LET'S MODIFY THE COPYCHARACTERS EXAMPLE TO USE LINE-ORIENTED I/O.

EXAMPLE PROGRAM: COPYLINES

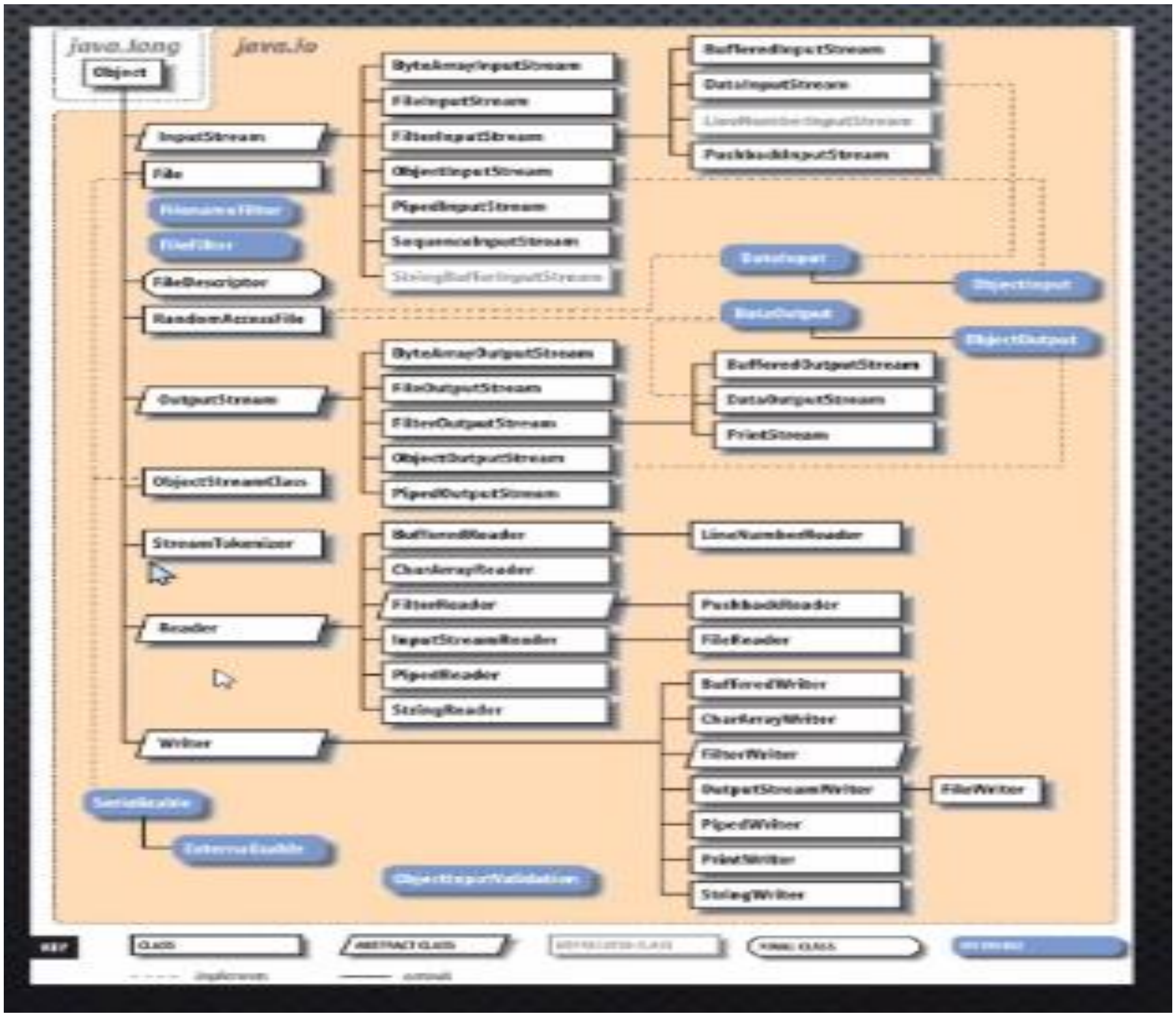


```java
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```



I/O STREAMS

- BYTE STREAMS: HANDLE I/O OF RAW BINARY
- CHARACTER STREAMS: HANDLE I/O OF CHARACTER DATA, AUTOMATICALLY HANDLING TRANSLATION TO AND FROM THE LOCAL CHARA
- BUFFERED STREAMS: OPTIMIZE INPUT AND OUTPUT BY REDUCING THE NUMBER OF CALLS TC IVE API.



Byte Stream:

FileInputStream

FileOutputStream

Char Stream:

FileReader

FileWriter

Buffered Stream:

BufferedReader (new FileReader())

BufferedWriter   (new FileWriter()) / PrintWriter (new FileWriter())