COSC 121: Computer Programming II

Today's Key Concepts

- Binary data is not human-readable
  - Binary I/O works with raw bytes (byte, int, etc.)
  - Files are sequences of bytes
- Main uses for handling binary files:
  - Program state persistence
  - File splitting / recombination
  - Working with non-text files (e.g., media, PDFs, executables)
- Key classes: FileInputStream and FileOutputStream
  DataInputStream and DataOutputStream
- Advanced concept: buffering reduces system calls and improves I/O performance
  - BufferedInputStream and BufferedOutputStream

Introduction

- Computers don't differentiate between text and binary files
- All files are stored in binary format
- What does it mean for files to be in binary format?
  - Recall: number conversion
    - 199 (decimal) = 11000111 (binary) = C7 (hexadecimal)
    - Unicode and ASCII representations
- Text I/O is built upon binary I/O in order to provide a level of abstraction for character encoding and decoding

	Text I/O	Binary I/O
Can handle..	Text files (readable by human) Example: .java, .txt	Binary files (not readable by human) Example: .class, .dat
Efficiency	Less efficient Involves encoding or decoding	**More efficient.** Doesn't involve encoding or decoding
Classes	Descendent of • **`Reader`** • **`Writer`**	Descendent of • **`InputStream`** • **`OutputStream`**

How Text and Binary I/O Differs

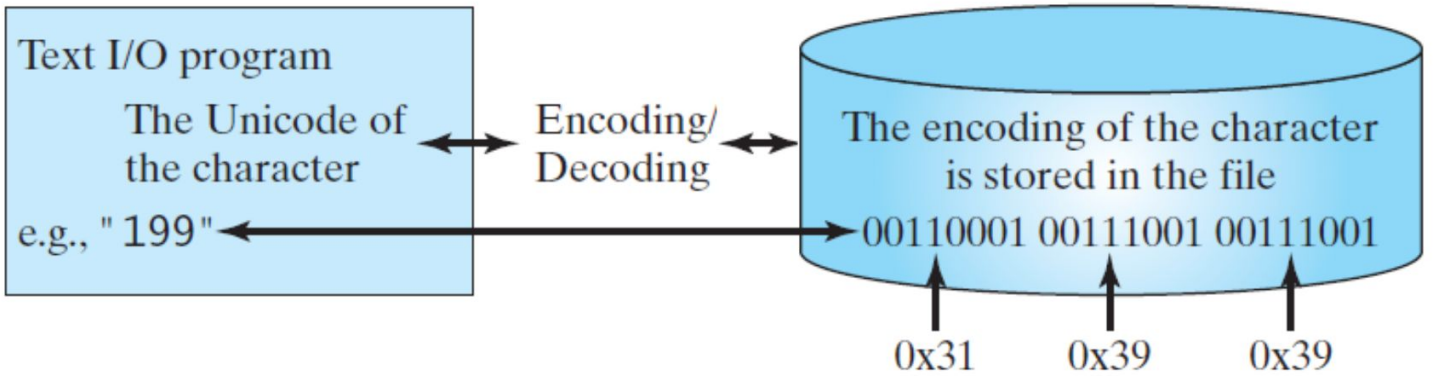
- Suppose you want to write the number 199 to a file
- Using text I/O:
  - You write each character '1', '9', '9' separately
  - Data conversion:

Character	Unicode	Decimal	Hex	Binary
'1'	U+0031	49	31	00110001
'9'	U+0039	57	39	00111001
'9'	U+0039	57	39	00111001

  - What we end up storing: 00110001 00111001 00111001 (3 bytes)
    - Not: 11000111 (which would be binary of 199)

How Text and Binary I/O Differs

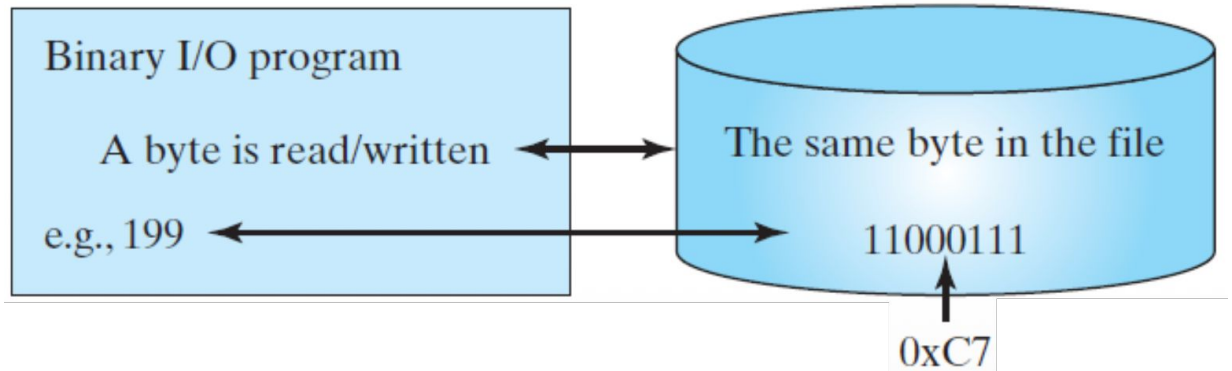
- Suppose you want to write the number 199 to a file
- Using text I/O:
  - You write each character '1', '9', '9' separately
  -



  - What we end up storing: 00110001 00111001 00111001 (3 bytes)
    - Not: 11000111 (which would be binary of 199)

How Text and Binary I/O Differs

- Suppose you want to write the number 199 to a file
- Using binary I/O:
  - You write numeric binary equivalent of 199
  - No data conversion needed



  - What we end up storing: 11000111 (representing binary of 199)

number of bytes stored will depend on method used for writing

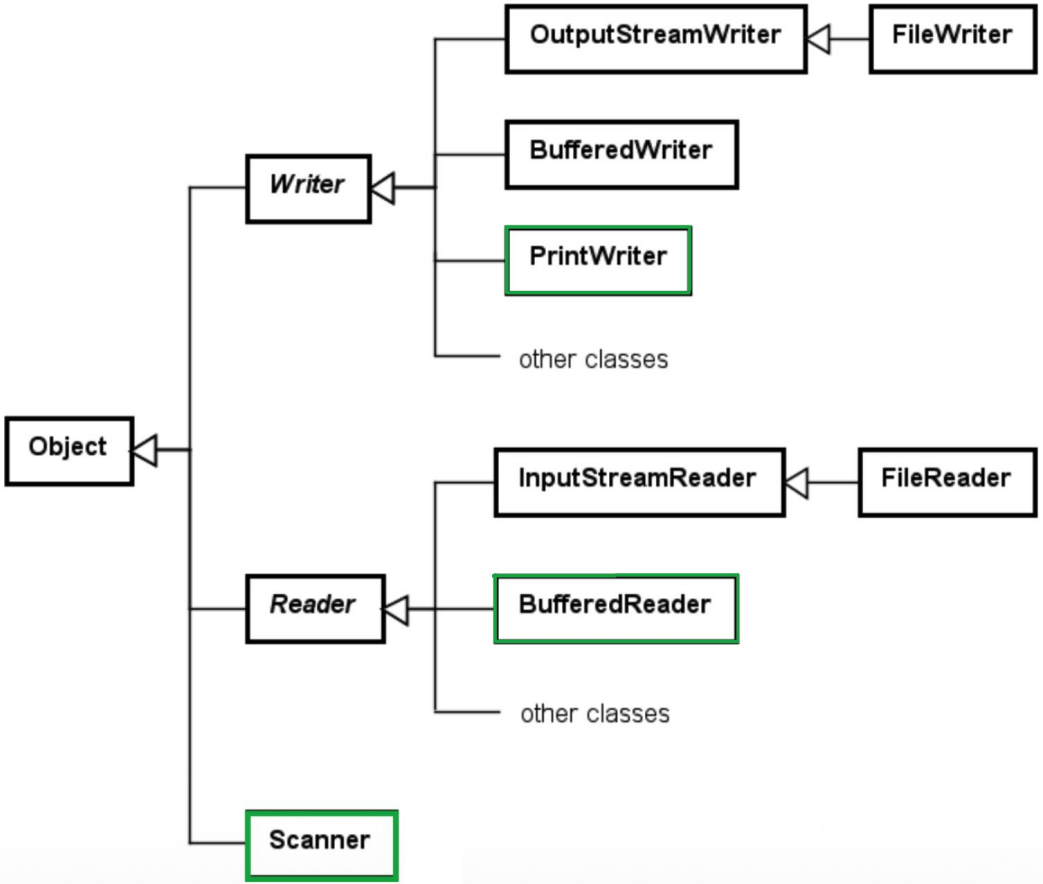
iClicker Question

Using a text I/O method such as the PrintWriter class and the default ASCII encoding, how many bytes will be written to t.txt by the following code?

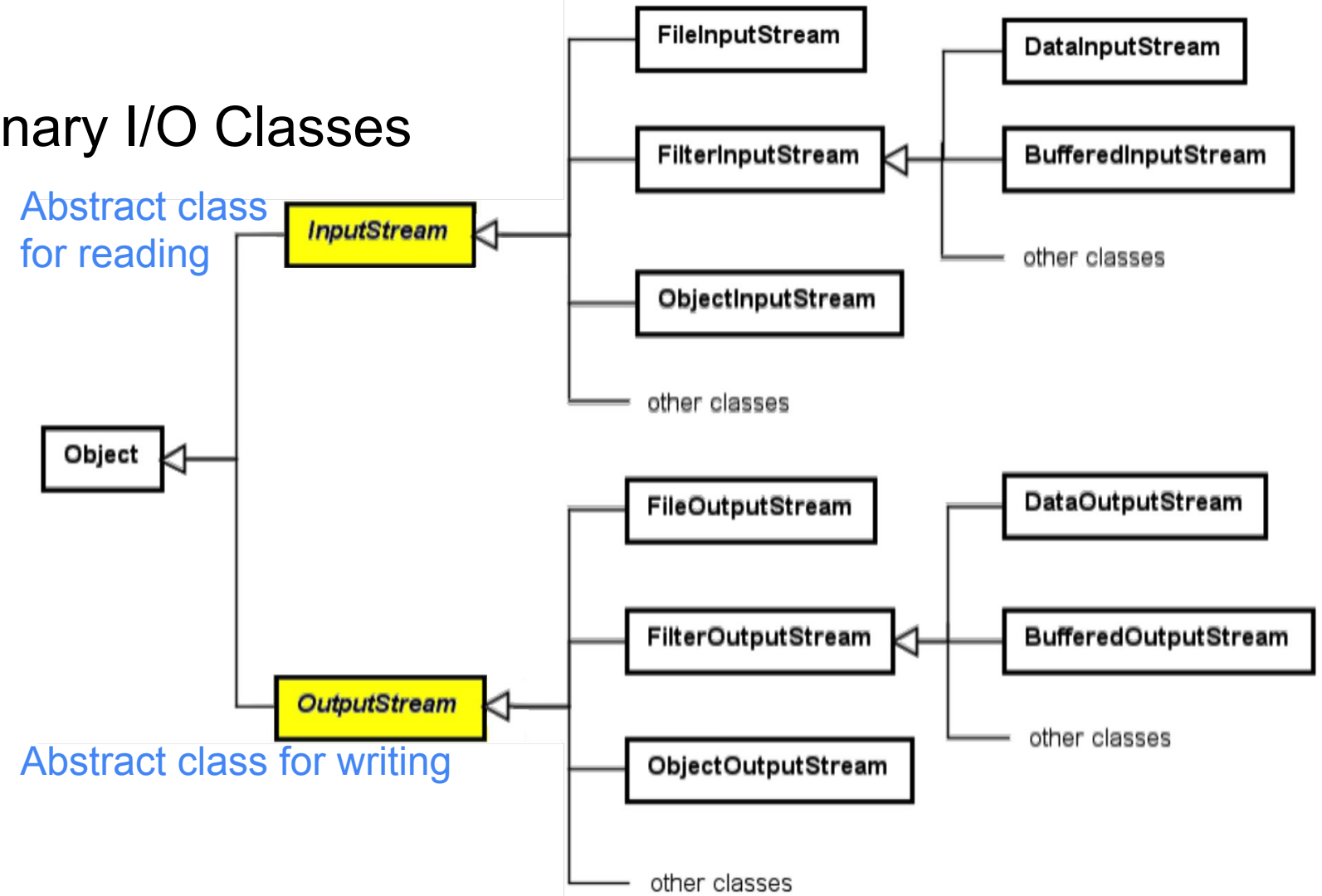
```
int x = 1234567890;
try(PrintWriter out = new PrintWriter("t.txt")){
      out.print(x);
}
```

A. 4
B. 8
C. 10
D. 16
E. Nothing will be written because we did not have out.close() at the end

Recall: Text I/O Classes



Binary I/O Classes



Useful InputStream Methods

+read(): int

- reads **next byte** of data **as int** (0 to 255)
- **–1 is returned at end of stream**

+read(b: byte[]): int

- Reads up to `b.length` bytes into **array b**

+read(b: byte[], off: int, len: int): int

- stores read bytes into `b[off], b[off+1],..., b[off+len-1]`

+skip(n: long): long

- Skips **n bytes** of data from this stream. actual # of bytes skipped returned

+available(): int

- Returns the **number of bytes remaining in the input stream**
- available()==0 indicates the **end of file (EOF)**

+close(): void

Useful OutputStream Methods

+write(int b): void

- writes **(byte) b** to this output stream

+write(b: byte[]): void

- Writes all the **bytes** in array `b` to the output stream

+write(b: byte[], off: int, len: int): void

- Writes `b[off], b[off+1],..., b[off+len-1]` into the output stream

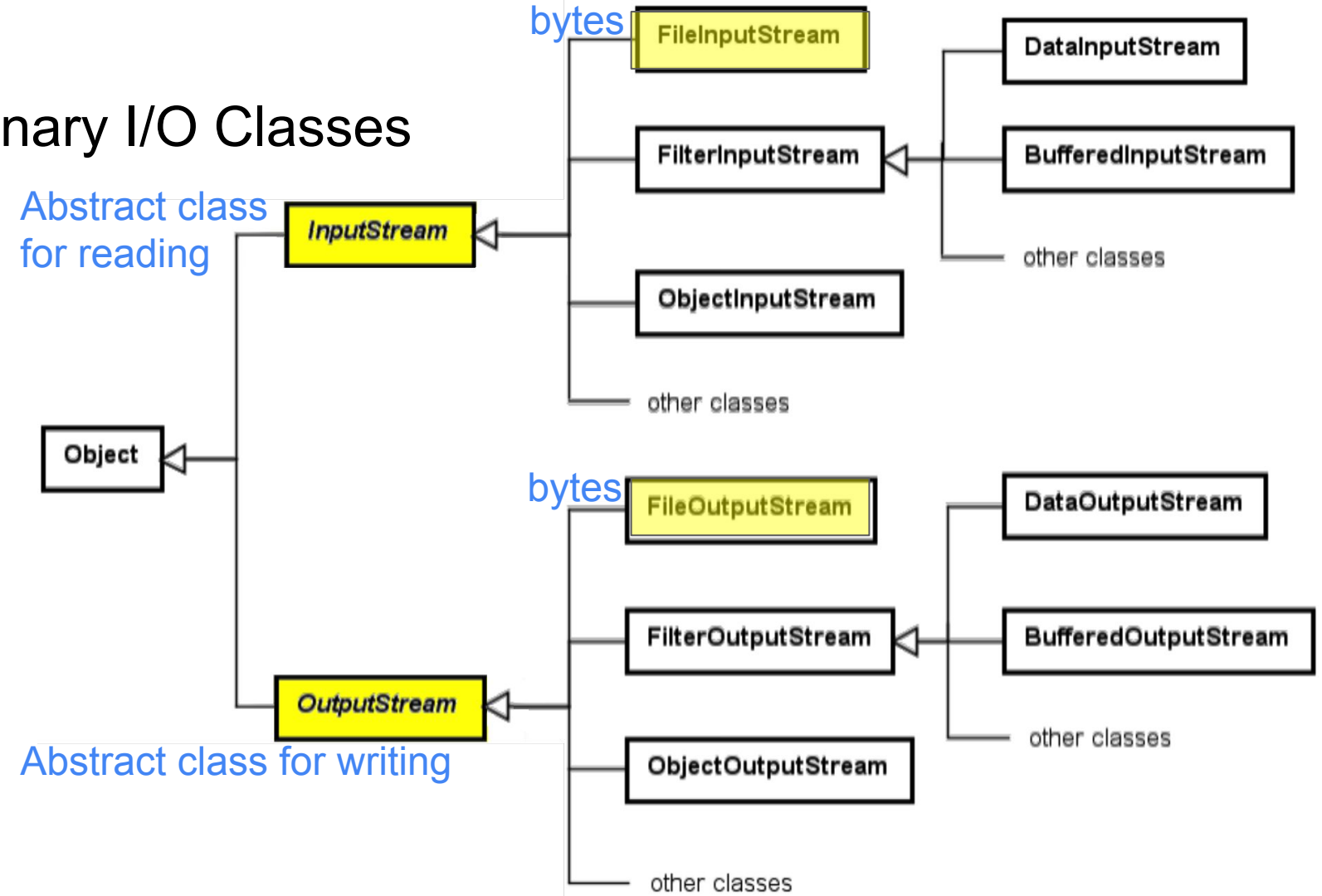
+flush(): void

- Flushes this output stream and forces any buffered output **bytes** to be written out

+close(): void

- Closes this output stream

Binary I/O Classes



FileInputStream Class

Methods:

- All methods from *InputStream*

Constructors

- public FileInputStream(String filename)
- public FileInputStream(File file)

Exceptions

- Throws IOException

FileOutputStream Class

Methods:

- All methods from *OutputStream*

Constructors

- public FileOutputStream(String filename)
- public FileOutputStream(File file)
  - If the file doesn't exist, a new file would be created
  - If the file already exists, the current contents in the file are deleted
- public FileOutputStream(String filename, **boolean append**)
- public FileOutputStream(File file, **boolean append**)
  - Used to retain the current content and append new data into the file

Exceptions

- Throws IOException

Example for Writing Byte Data

- Java program to write 1 to 10 as bytes into "temp.dat"
- Use write() from OutputStream

```
FileOutputStream fos = new FileOutputStream("temp.dat");
for (int i = 1; i <= 10; i++)
    fos.write(i);
```

- Full program needs to have:
  - Necessary library imports
  - Exception handling

Example Full Program

```
import java.io.FileOutputStream;
import java.io.IOException;
public class WriteBytesToFile {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("temp.dat")) {
            for (int i = 1; i <= 10; i++)
                fos.write(i);
            System.out.println("Bytes written successfully.");
        } catch (IOException e) {
            System.out.println("Error writing to file: " + e.getMessage())
        }
    }
}
```

Read and Write Bytes

- Modify previous program to read from a .png file and write first few bytes into "temp.dat"
- Find the appropriate read/write methods needed:
  - read(b: byte[]) : int
    stores read contents into b, returns length of data read
  - write(b: bytes[], off: int, len: int) : void
    takes contents from b, starting at index off, writes for len bytes
- Do some exception handling

```
byte[] buffer = new byte[10];          // temp bytes array
int bytesRead = fin.read(buffer);      // read data into buffer
fout.write(buffer, 0, bytesRead);      // write data from buffer
```

```
// try creating file input and output objects
try (FileInputStream fin = new FileInputStream(inputFile);
     FileOutputStream fout = new FileOutputStream("temp.dat")) {
    byte[] buffer = new byte[10];
    int bytesRead = fin.read(buffer);
    fout.write(buffer, 0, bytesRead);
```

```
// try creating file input and output objects
try (FileInputStream fin = new FileInputStream(inputFile);
     FileOutputStream fout = new FileOutputStream("temp.dat")) {
    byte[] buffer = new byte[10];
    int bytesRead = fin.read(buffer);
    fout.write(buffer, 0, bytesRead);
    System.out.println("Bytes read: " + bytesRead);
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
}
                                        // do appropriate exception handling
```

// embed within a class

```
public class CopyFirstTenBytes {
    public static void main(String[] args) {
```

```
        try (FileInputStream fin = new FileInputStream(inputFile);
             FileOutputStream fout = new FileOutputStream("temp.dat")) {
            byte[] buffer = new byte[10];
            int bytesRead = fin.read(buffer);
            fout.write(buffer, 0, bytesRead);
            System.out.println("Bytes read: " + bytesRead);
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

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

public class CopyFirstTenBytes {
    public static void main(String[] args) {
```

// import necessary libraries

```java
        try (FileInputStream fin = new FileInputStream(inputFile);
             FileOutputStream fout = new FileOutputStream("temp.dat")) {
            byte[] buffer = new byte[10];
            int bytesRead = fin.read(buffer);
            fout.write(buffer, 0, bytesRead);
            System.out.println("Bytes read: " + bytesRead);
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

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

public class CopyFirstTenBytes {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Usage: java CopyFirstTenBytes <input_image_file>");
            return;
        }
        String inputFile = args[0];
        try (FileInputStream fin = new FileInputStream(inputFile);
             FileOutputStream fout = new FileOutputStream("temp.dat")) {
            byte[] buffer = new byte[10];
            int bytesRead = fin.read(buffer);
            fout.write(buffer, 0, bytesRead);
            System.out.println("Bytes read: " + bytesRead);
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

what is args? try running this!

// defensively handle case if input file is not given

// grab name if input file is given

iClicker Question

What does the variable bytesRead represent from the following code in the CopyFirstTenBytes program:

```
int bytesRead = fin.read( buffer );
```

A. The total size of the file in bytes
B. The number of bytes actually read into the buffer, or -1 if end of file is reached
C. The number of bytes written to the output file
D. The length of the buffer array

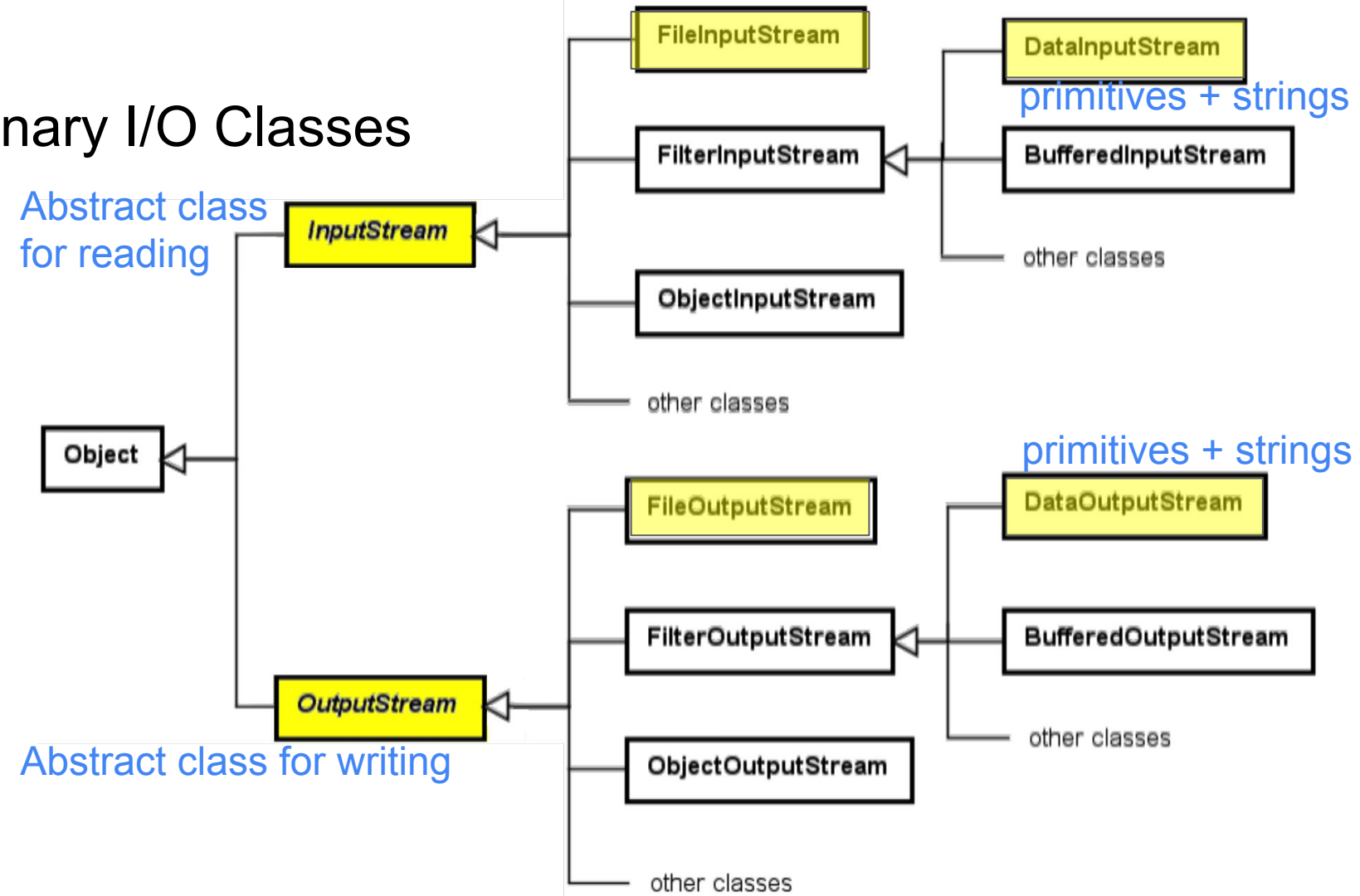
iClicker Question

What is wrong with the following code?

```
import java.io.*;
public class Ex1 {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("test.dat");) {
        } catch (IOException ex) {
            ex.printStackTrace();
        }
         catch (FileNotFoundException ex) {
             ex.printStackTrace();
         }
    }
}
```

A. We cannot create a FileInputStream object inside try(…)
B. We can't have more than one catch statement
C. We need one more catch statement
D. We need a finally statement
E. FileNotFoundException is unreachable

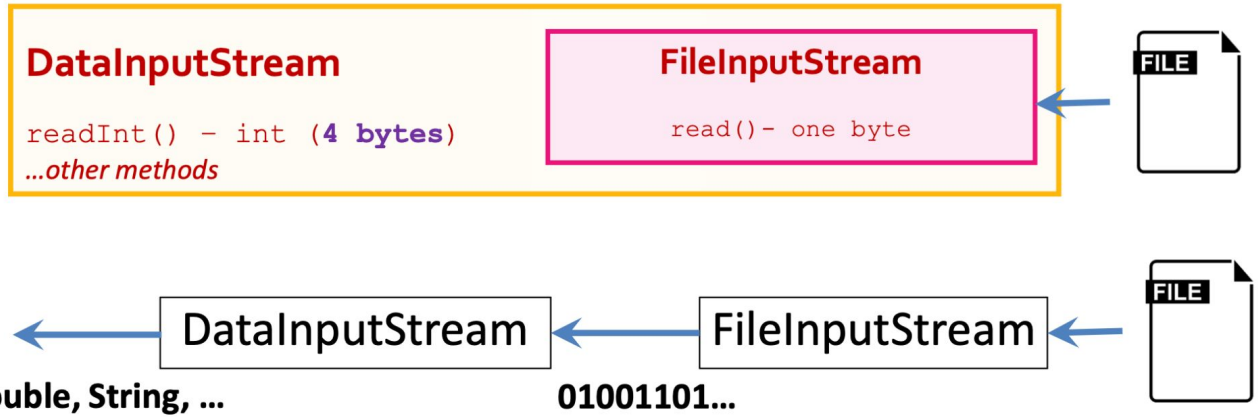
Binary I/O Classes



Stream Chaining

- Stream chaining connects multiple stream classes together to get the data in the form required
- DataInputStream wraps around FileInputStream

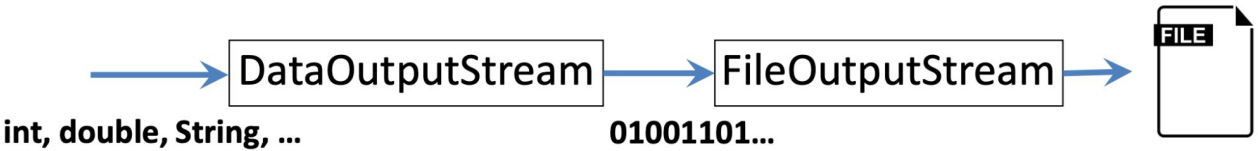
```
DataInputStream in =
        new DataInputStream(new FileInputStream("file.dat"));
```



Stream Chaining

- Stream chaining connects multiple stream classes together to get the data in the form required
- DataOutputStream wraps around FileOutputStream

```
DataOutputStream out =
        new DataOutputStream(new FileOutputStream("file.dat"));
```



DataInputStream Class

Methods: All methods of `InputStrem` in addition to:

- `+readBoolean():` Reads 1 byte, returns true if that byte is nonzero, false if it is zero
- `+readByte(): byte` reads and returns 1 bytes
- `+readShort(): short` reads 2 bytes, returns a short value
- `+readInt(): int` reads 4 bytes, returns an int value
- `+readLong(): long` reads 8 bytes, returns a long value
- `+readFloat(): float` reads 4 bytes, returns a float
- `+readDouble():double` reads 8 bytes, returns a double
- `+readChar(): char` reads 2 bytes, returns a char in Unicode
- `+readUTF(): String` reads a string in modified-UTF format (UTF uses different lengths to represent different characters)

DataOutputStream Class

Methods: All methods of `OutputStrem` in addition to:

- **`+writeBoolean(b:boolean): void`**
- **`+writeByte(v:int): void`**
- **`+writeShort(v:short): void`**
- **`+writeInt(v:int): void`**
- **`+writeLong(v:long): void`**
- **`+writeFloat(v:float): void`**
- **`+writeDouble(v:double): void`**
- **`+writeChar(c:char): void`**
  - Writes c character composed of 2 bytes (Unicode)
- **`+writeUTF(s:String): void`**
  - Writes the length of `s`, then `s` in modified-UTF format
    - Must use readUTF to read

size of data written is same as corresponding read methods

iClicker Question

Which of the following statements is correct in completing the following code to create a DataOutputStream to write to a file named a.dat?

```
DataOutputStream out = ??
```

A. new DataOutputStream(new File("a.dat"));
B. new DataOutputStream(FileOutputStream("a.dat"));
C. new DataOutputStream(new FileOutputStream("a.dat"));
D. new DataOutputStream("a.dat");
E. Something else

Example: Writing/Reading Numbers to/from Binary File

- Write 100 double values created randomly into a file called "numbers.dat"
- How to start?

Example: Writing/Reading Numbers to/from Binary File

- Write 100 double values created randomly into a file called "numbers.dat"
- How to start?
  - Create output file object
  - Generate 100 random doubles
  - Write them out to the file object
  - Close file

Example: Writing/Reading Numbers to/from Binary File

- Write 100 double values created randomly into a file called "numbers.dat"

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("numbers.dat", true));
for (int i = 0; i < 100; i++)
    out.writeDouble(Math.random() * 100000);
out.close();
```

Example: Writing/Reading Numbers to/from Binary File

- Write 100 double values created randomly into a file called "numbers.dat"

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("numbers.dat", true));
for (int i = 0; i < 100; i++)
    out.writeDouble(Math.random() * 100000);
out.close();
```

- Read 100 double values from binary file
- How to start?

Example: Writing/Reading Numbers to/from Binary File

- Write 100 double values created randomly into a file called "numbers.dat"

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("numbers.dat", true));
for (int i = 0; i < 100; i++)
    out.writeDouble(Math.random() * 100000);
out.close();
```

- Read 100 double values from binary file
- How to start?
  - Create file input object
  - Read in 100 doubles
  - Close file

Example: Writing/Reading Numbers to/from Binary File

- Write 100 double values created randomly into a file called "numbers.dat"

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("numbers.dat", true));
for (int i = 0; i < 100; i++)
    out.writeDouble(Math.random() * 100000);
out.close();
```

- Read 100 double values from binary file

```
DataInputStream in = new DataInputStream(new FileInputStream("numbers.dat"));
for (int i = 0; i < 100; i++)
    System.out.println(in.readDouble());
in.close();
```

\+ embed into java programs

iClicker Question

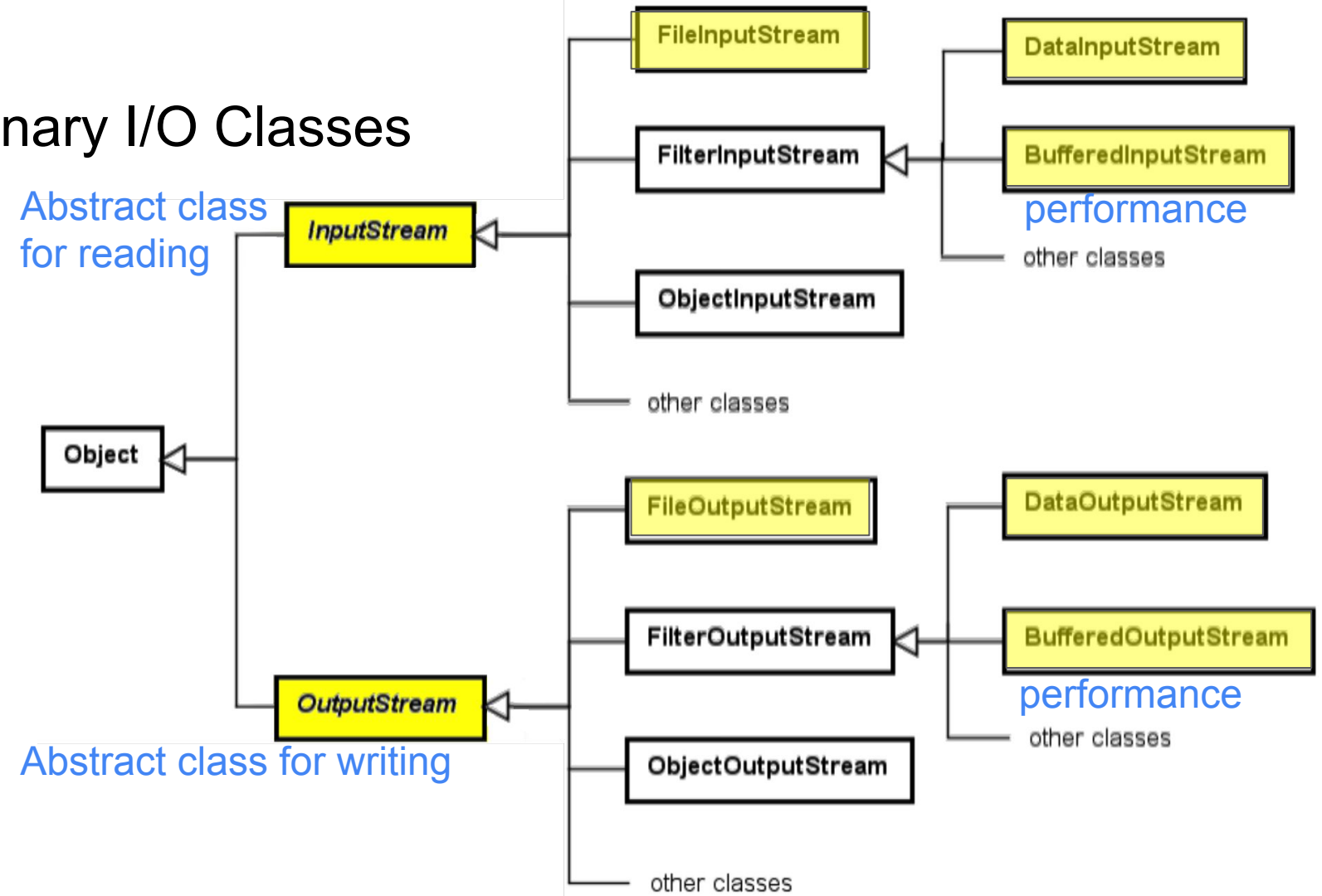
Using a binary I/O method and writeInt() to write an integer in Java, how many bytes will be written to t.txt by the following code?

```
int x = 1234567890;

try(DataOutputStream out = new DataOutputStream(
                           new FileOutputStream("t.dat"))){
   out.writeInt(x);
}
```

A. 4
B. 8
C. 10
D. 16
E. Nothing will be written because we did not have out.close() at the end

Binary I/O Classes



BufferedInputStream / BufferedOutputStream

- Used to speed up I/O by reducing number of disk reads and writes

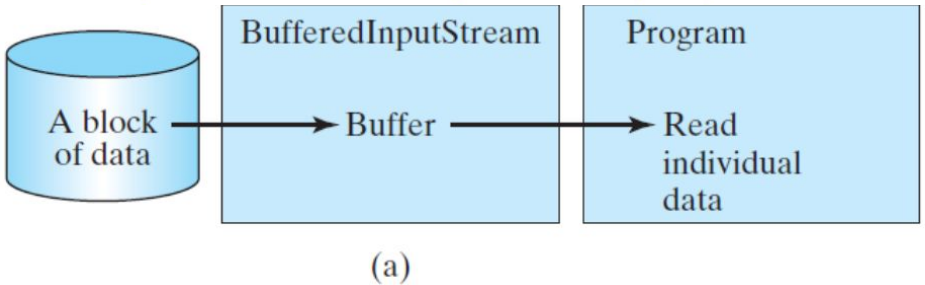
Methods:

- All methods from *InputStream* / *OutputStream*

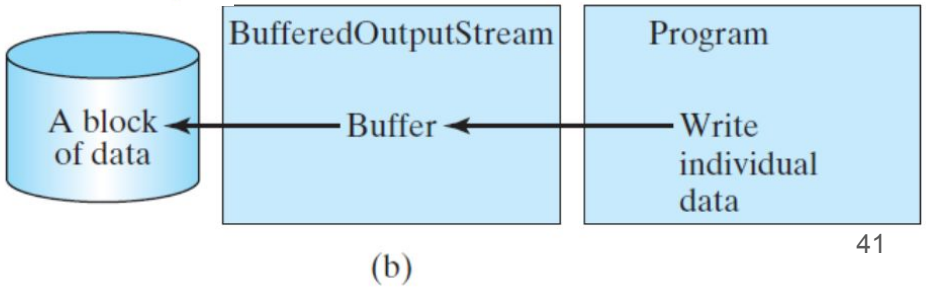
Constructors

- public BufferedInputStream(InputStream in) // bufferSize=512
- public BufferedInputStream(InputStream in, int bufferSize)

- public BufferedOutputStream(OutputStream in) //bufferSize=512
- public BufferedOutputStream(OutputStream in, int bufferSize)



(a)



(b)

Example: Using BufferedInputStream

```
BufferedInputStream input =                                    /*read byte*/
        new BufferedInputStream(new FileInputStream("temp.dat"));
```


```
DataInputStream input = new DataInputStream(          /*read primitive and string*/
        new BufferedInputStream(new FileInputStream("temp.dat")));
```

Example

- Suppose you want to back up a huge file (e.g., a 10GB AVI file) by splitting the file into smaller pieces and backing up these pieces separately. Write a utility program that splits a large file into smaller ones sourceFile.1, sourceFile.2, . . . , sourceFile.n, where n is numberOfPieces and the output files are about the same size.
- Use this header: void splitFile(String sourceFile, int splitFileSize)
- How to start?
  - Set up input file object and create output file object number 1
  - As long as there is data left to read:
    - Repeatedly write K bytes to output file
    - If reach splitFileSize, create next output file object
  - Close everything when done

```
while ((bytesRead = fin.read(buffer)) != -1) {
    if (bytesWritten + bytesRead > splitFileSize) {
        // handles when there's too much data to write
    } else {
        fout.write(buffer, 0, bytesRead);
        bytesWritten += bytesRead;
    }
    if (bytesWritten == splitFileSize) {
        fout.close();
        fileNum++;
        fout = new BufferedOutputStream(new FileOutputStream(sourceFile + "." + fileNum));
        bytesWritten = 0;     // prep next output file
    }
}
```

read data into bytesRead

there's enough space left to hold bytesRead

handles boundary case

```
while ((bytesRead = fin.read(buffer)) != -1) {
    if (bytesWritten + bytesRead > splitFileSize) {
        int bytesToWrite = splitFileSize - bytesWritten;
        fout.write(buffer, 0, bytesToWrite);
        fout.close();
        fileNum++;
        fout = new BufferedOutputStream(new FileOutputStream(sourceFile + "." + fileNum));
        fout.write(buffer, bytesToWrite, bytesRead - bytesToWrite);
        bytesWritten = bytesRead - bytesToWrite;
    } else {
        fout.write(buffer, 0, bytesRead);
        bytesWritten += bytesRead;
    }
    if (bytesWritten == splitFileSize) {
        fout.close();
        fileNum++;
        fout = new BufferedOutputStream(new FileOutputStream(sourceFile + "." + fileNum));
        bytesWritten = 0;
    }
}
```

// 1. write what you can

// 2. prep next output file

// 3. write the rest

initial setup

files and exceptions

```
public class FileSplitter {
    public static void splitFile(String sourceFile, int splitFileSize) {
        BufferedInputStream fin = null;
        BufferedOutputStream fout = null;
        try {
            fin = new BufferedInputStream(new FileInputStream(sourceFile));
            int fileNum = 1;
            int bytesWritten = 0;
            fout = new BufferedOutputStream(new FileOutputStream(sourceFile + "." + fileNum));
            byte[] buffer = new byte[4096];
            int bytesRead;
            while ((bytesRead = fin.read(buffer)) != -1) {
                // core logic from last slide
            }
        } catch (IOException e) {
            System.out.println("I/O error occurred while splitting the file.");
        } finally {
            try {
                if (fout != null) fout.close();
                if (fin != null) fin.close();
            } catch (IOException e) {
                System.out.println("Error closing file streams.");
            }
        }
    }
}
```

would this program work if we don't use the Buffered… Stream classes?

iClicker Question

What is the main reason for using a byte array buffer (shown in the line below) when reading and writing the file?

```
byte[] buffer = new byte[4096];
```

A. To satisfy a necessary requirement for using the BufferedInputStream and BufferedOutputStream classes
B. To limit the size of each output file to exactly 4096 bytes
C. To guarantee that the output files are all the same size
D. To reduce the number of input/output operations and improve performance by reading and writing data in chunks
E. To ensure that the entire file is loaded into memory before writing