Binary I/O

CSE 114, Computer Science 1

Stony Brook University

http://www.cs.stonybrook.edu/~cse114

Motivation

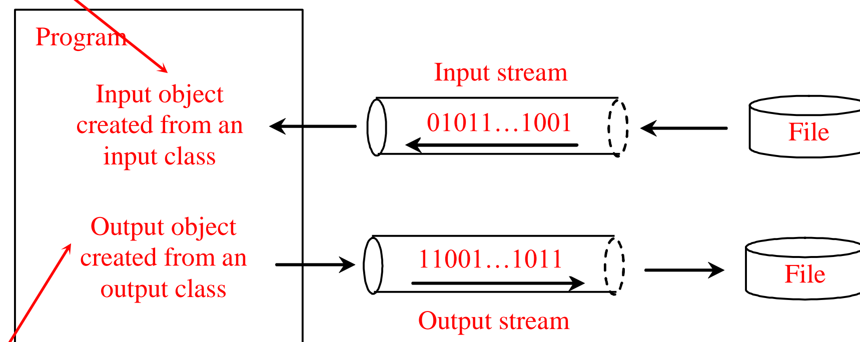
- Data stored in a text files is represented in human-readable form
- Data stored in a ***binary files*** is represented in binary form
  - The advantage of binary files is that they are more efficient to process than text files (e.g., the number 123 is smaller than the text "123")
  - But, people cannot read binary files
    - Binary files are designed to be read by programs
    - Java bytecode classes are stored in binary files and are read by the JVM

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

How is I/O Handled in Java?

- A **`File`** object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file
  - In order to perform I/O, you need to create objects using appropriate Java I/O classes: **`Scanner`** and **`PrintWriter`**:

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```




```
PrintWriter output = new PrintWriter("temp.txt");
output.println("Java 101");
output.close();
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

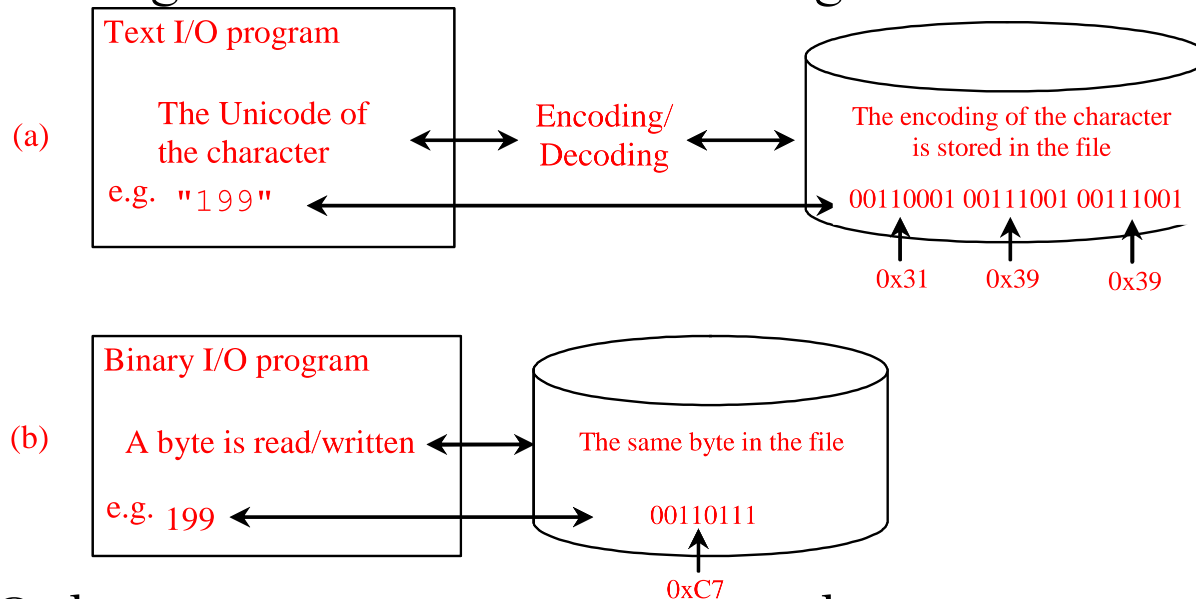
Text Files vs. Binary Files

- A text file consists of a sequence of characters
  - For example, the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file
- A binary file consists of a sequence of bits
  - For example, the decimal integer 199 is stored as a binary value for the hexadecimal number C7 in a binary file, because decimal 199 equals to hex C7

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Text Files vs. Binary Files

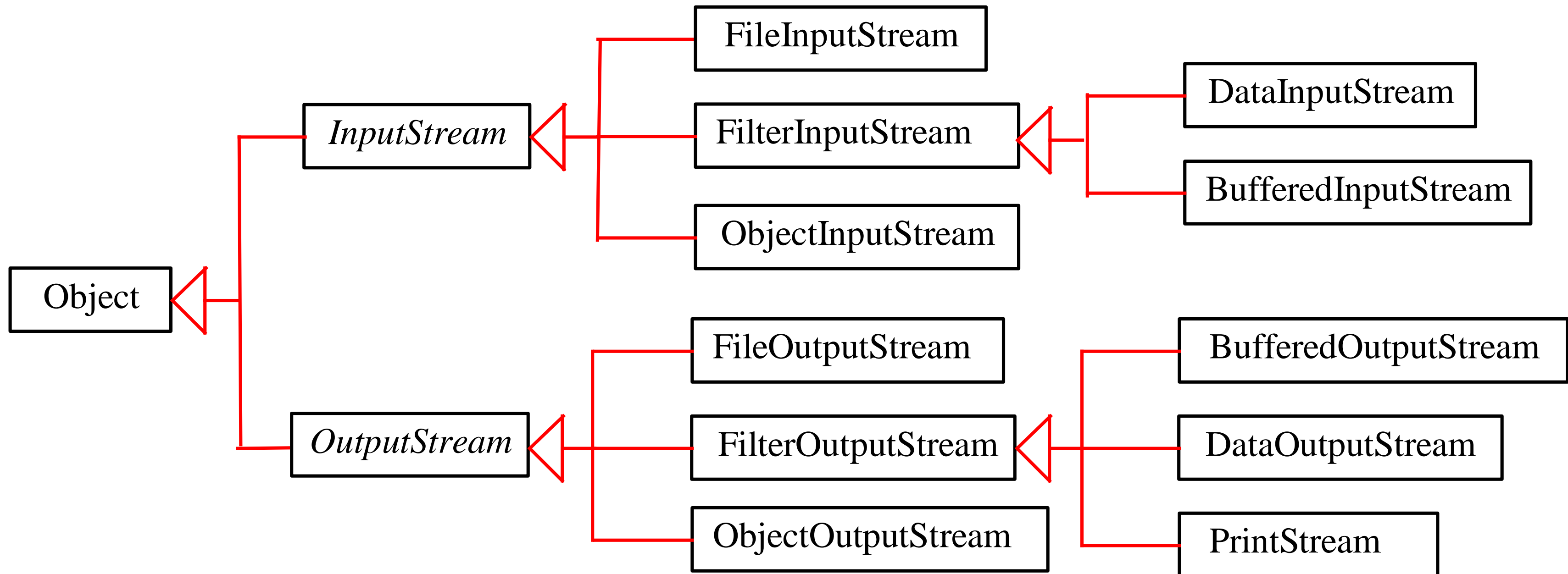
- Text I/O requires encoding and decoding: the JVM converts a Unicode to a file specific encoding when writing a character and coverts a file specific encoding to a Unicode when reading a character



- Binary I/O does not require conversions: when you write a byte to a file, the original byte is copied into the file, and when you read a byte from a file, the exact byte in the file is returned

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Binary I/O Classes



- The abstract **`InputStream`** is the root class for reading binary data
- The abstract **`OutputStream`** is the root class for writing binary data
- The design of the Java I/O classes is a good example of applying inheritance, where common operations are generalized in superclasses, and subclasses provide specialized operations.

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

InputStream

java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores into b[off], b[off+1], …, b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+available(): int	Returns the number of bytes that can be read from the input stream.
+close(): void	Closes this input stream and releases any system resources associated with the stream.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.
+markSupported(): boolean	Tests if this input stream supports the mark and reset methods.
+mark(readlimit: int): void	Marks the current position in this input stream.
+reset(): void	Repositions this stream to the position at the time the mark method was last called on this input stream.

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

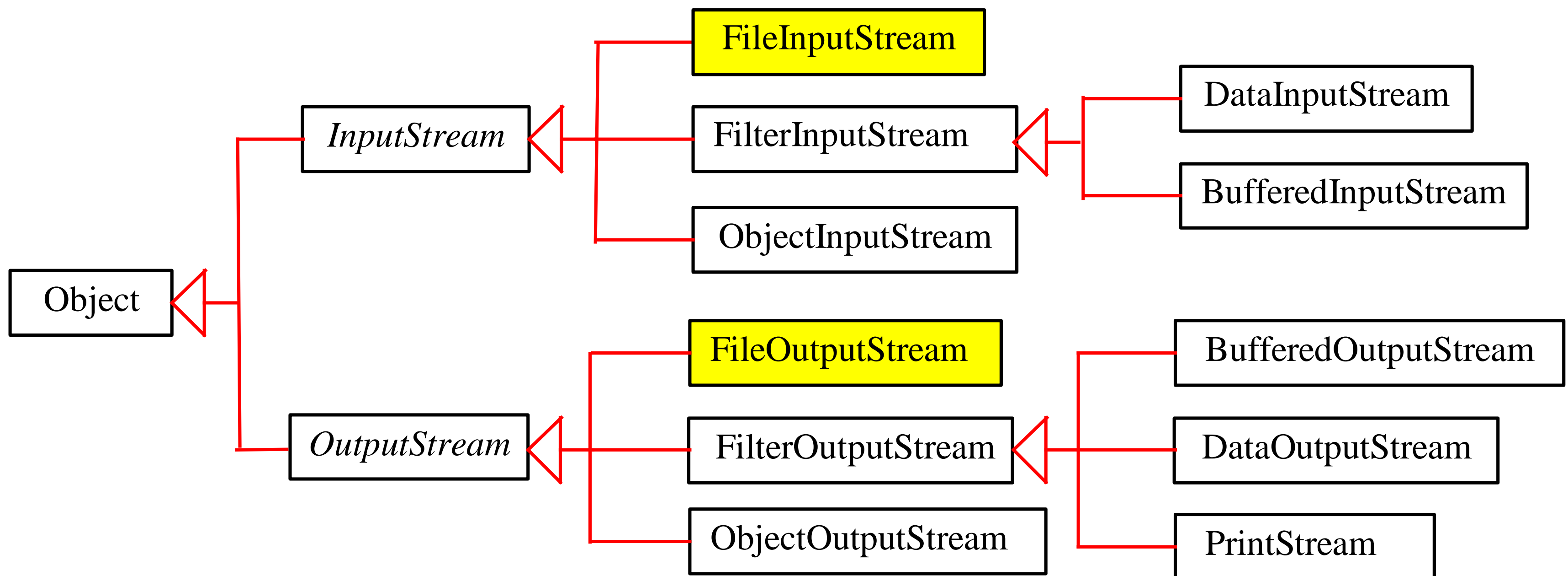
OutputStream

java.io.OutputStream	
+write(int b): void	Writes the specified byte to this output stream. The parameter b is an int value. (byte)b is written to the 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.
+close(): void	Closes this input stream and releases any system resources associated with the stream.
+flush(): void	Flushes this output stream and forces any buffered output bytes to be written out.

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

FileInputStream/FileOutputStream

- **`FileInputStream/FileOutputStream`** are for reading/writing bytes from/to files.
  - All the methods in **`FileInputStream/FileOuptputStream`** are inherited from its superclasses



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

FileInputStream

- To construct a **`FileInputStream`**, use the following constructors:

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

  - A **`java.io.FileNotFoundException`** would occur if you attempt to create a **`FileInputStream`** with a nonexistent file

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

FileOutputStream

- To construct a **FileOutputStream**, use the following constructors:

```
public FileOutputStream(String filename)
public FileOutputStream(File file)
public FileOutputStream(String filename,
    boolean append)
public FileOutputStream(File file,
    boolean append)
```

  - If the file does not exist, a new file would be created
  - If the file already exists, the first two constructors would delete the current contents in the file
  - To retain the current content and append new data into the file, use the last two constructors by passing **true** to the **append** parameter

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

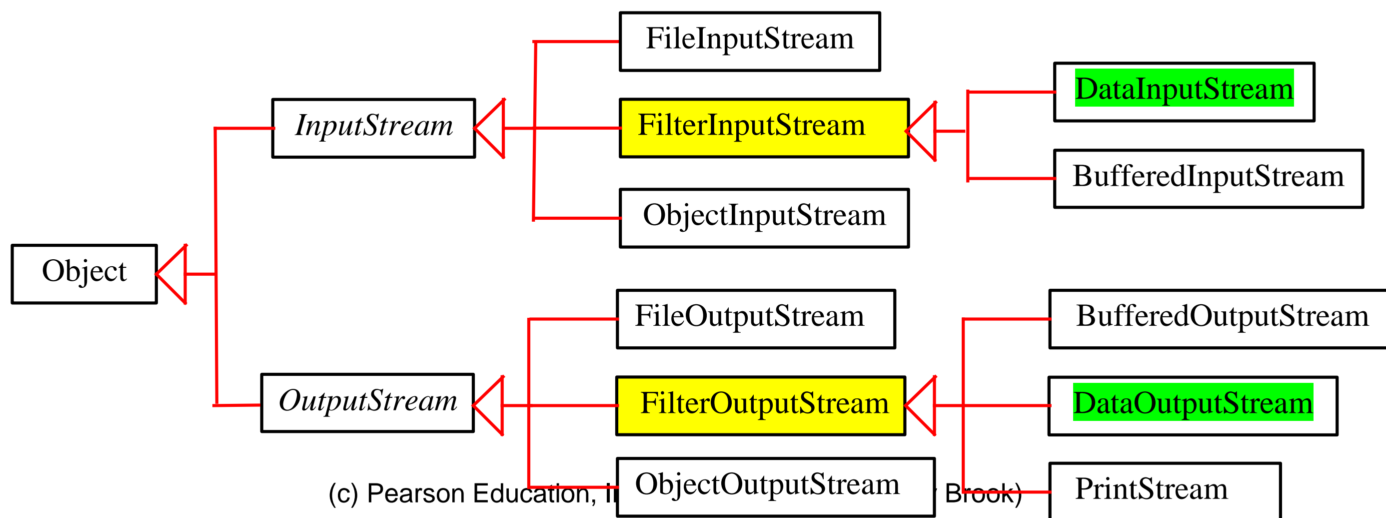
```java
import java.io.*;
public class TestFileStream {
    public static void main(String[] args) throws IOException {
        // Create an output stream for the file
        try (FileOutputStream output =
             new FileOutputStream("temp.dat");) {
            // Output values to the file
            for (int i = 1; i <= 10; i++) {
                output.write(i);
            }
        }
        // Create an input stream for the file
        try (FileInputStream input =
             new FileInputStream("temp.dat");){
            // Read values from the file
            int value;
            while ((value = input.read()) != -1) {
                System.out.print(value + " ");
            }
        }
    }
}
```

Run: 1 2 3 4 5 6 7 8 9 10

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

FilterInputStream/FilterOutputStream

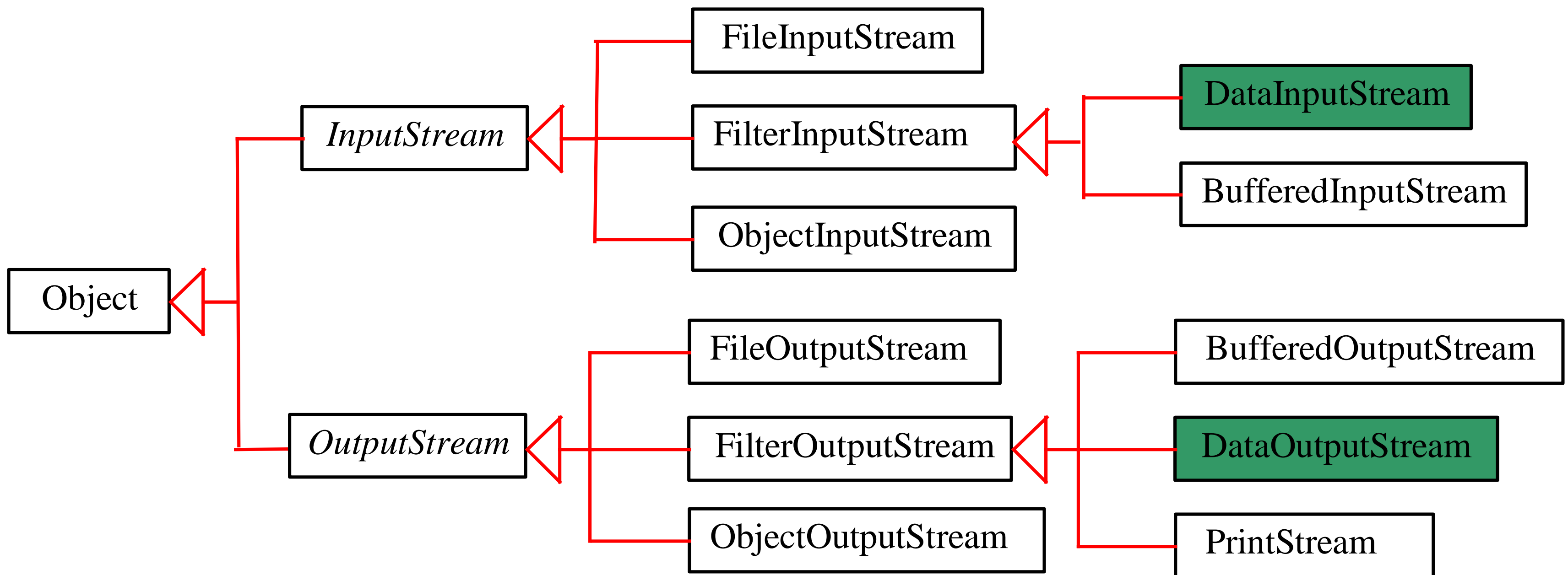
- ***Filter streams*** are streams that filter bytes for some purpose
  - The basic byte input stream provides a **`read`** method that can only be used for reading bytes
  - If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream
- Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters
- **`FilterInputStream`** and **`FilterOutputStream`** are the base classes for filtering data
  - When you need to process primitive numeric types, use **`DataInputStream`** and **`DataOutputStream`** to filter bytes



(c) Pearson Education, I… Brook)

DataInputStream/DataOutputStream

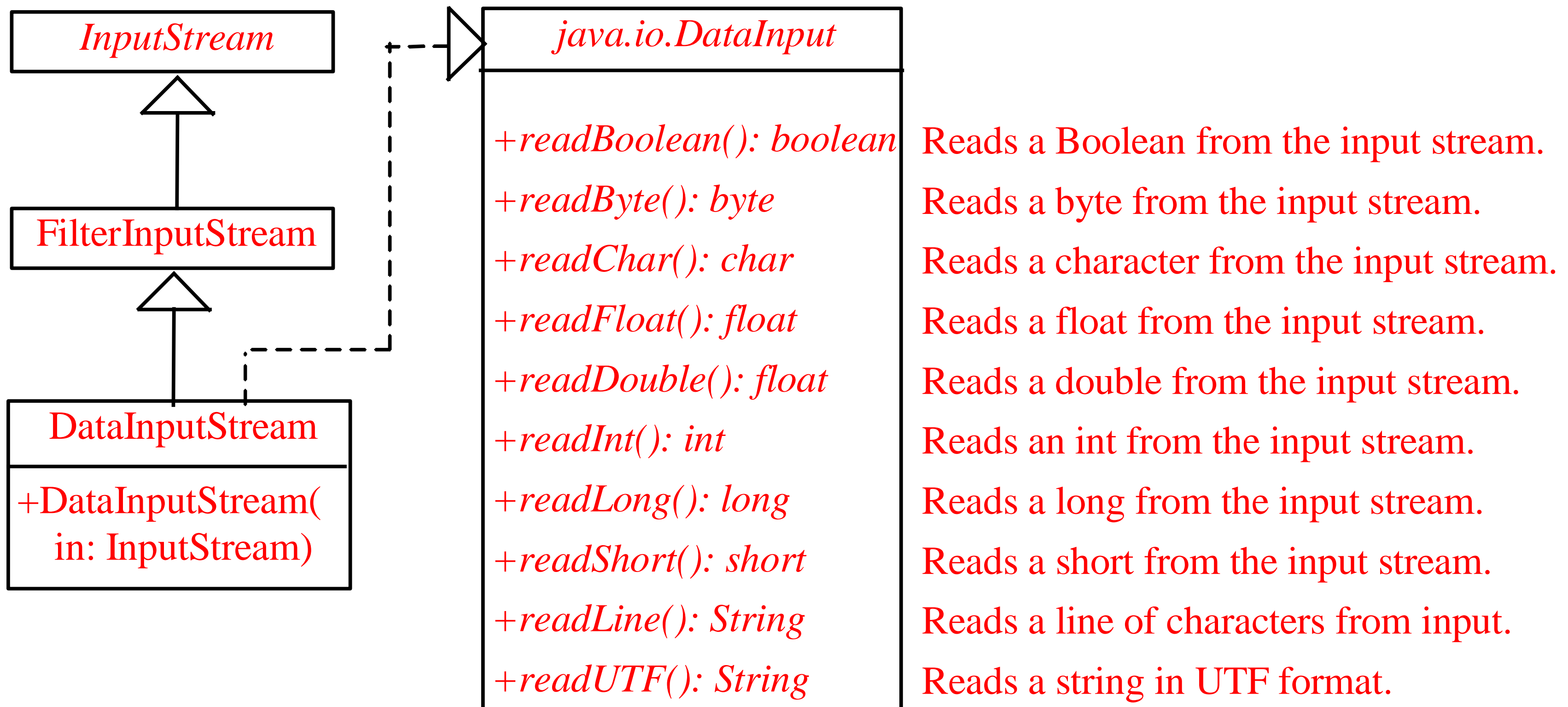
- **DataInputStream** reads bytes from the stream and converts them into appropriate primitive type values or strings
- **DataOutputStream** converts primitive type values or strings into bytes and output the bytes to the stream



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

DataInputStream

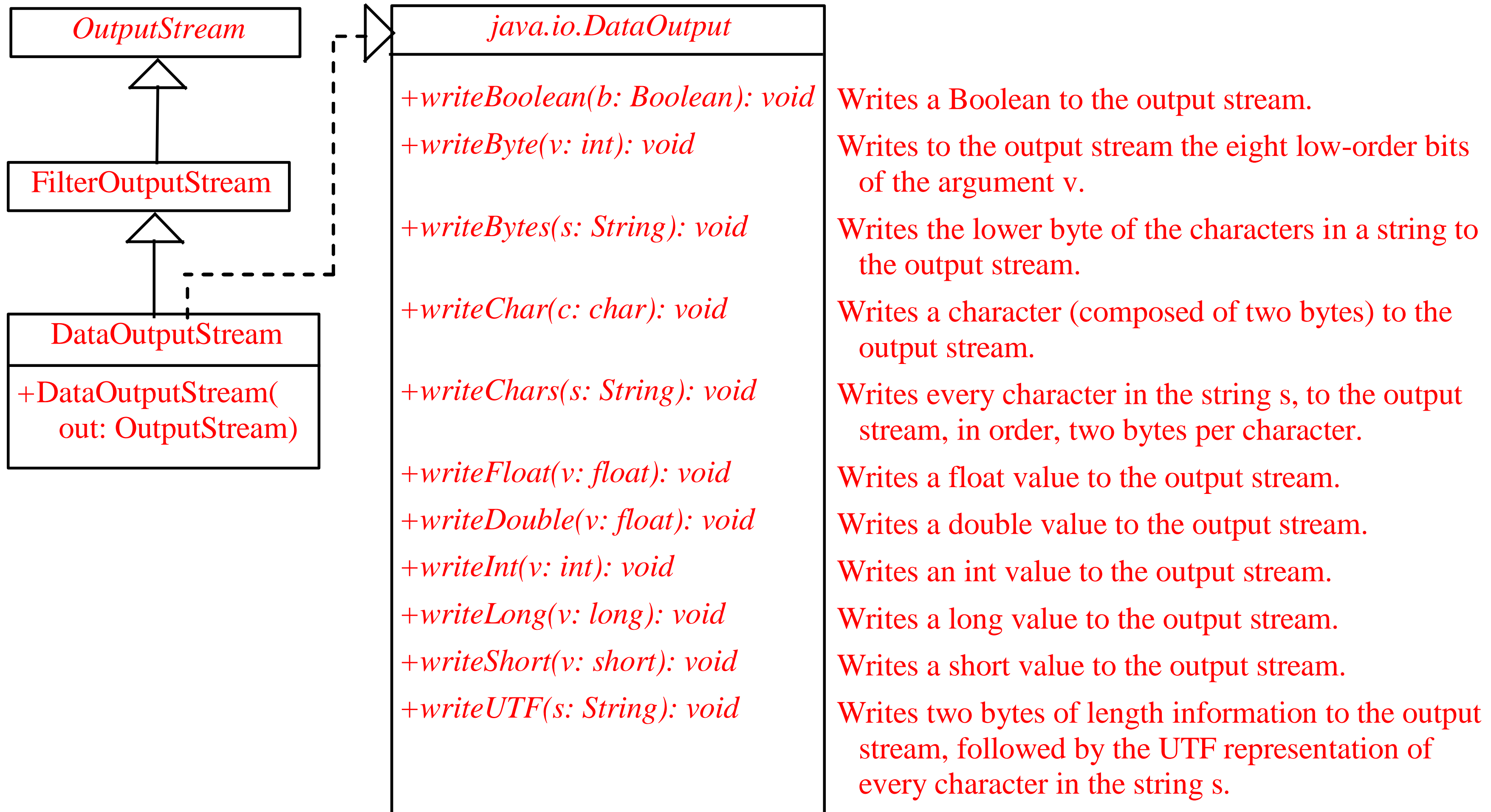
`DataInputStream` extends **`FilterInputStream`** and implements the **`DataInput`** interface



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

DataOutputStream

`DataOutputStream` extends **`FilterOutputStream`** and implements the **`DataOutput`** interface



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Characters and Strings in Binary I/O

- Java uses Unicode (UTF-16) and consists of two bytes
  - The **`writeChar(char c)`** method writes the Unicode of character **`c`** to the output
  - The **`writeChars(String s)`** method writes the Unicode for each character in the string s to the output
- Most operating systems use ASCII since most applications need only the ASCII character set and it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character
  - The UTF-8 is an alternative scheme that stores a character using 1, 2, or 3 bytes
    - ASCII values (less than 0x7F) are coded in one byte
    - Unicode values less than 0x7FF are coded in two bytes
    - Other Unicode values are coded in three bytes

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Using DataInputStream/DataOutputStream

- Data streams are used as wrappers on existing input and output streams to filter data in the original stream
  - They are created using the following constructors:

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream
  outstream)
```

- The statements given below create data streams: the first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**.

```
DataInputStream infile =
   new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream outfile =
  new DataOutputStream(new FileOutputStream("out.dat"));
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

```java
import java.io.*;
public class TestDataStream {
    public static void main(String[] args) throws IOException {
        // Create an output stream for file temp.dat
        try (DataOutputStream output =
                new DataOutputStream(new FileOutputStream("temp.dat"));) {
            // Write student test scores to the file
            output.writeUTF("John");
            output.writeDouble(85.5);
            output.writeUTF("Jim");
            output.writeDouble(185.5);
            output.writeUTF("George");
            output.writeDouble(105.25);
        }
        // Create an input stream for file temp.dat
        try (DataInputStream input =
                new DataInputStream(new FileInputStream("temp.dat"));) {
            // Read student test scores from the file
            System.out.println(input.readUTF() + " " + input.readDouble());
            System.out.println(input.readUTF() + " " + input.readDouble());
            System.out.println(input.readUTF() + " " + input.readDouble());
        }
    }
}
```

Run:

John 85.5

Jim 185.5

George 105.25

CAUTION: You have to read the data in the same order and same format in which they are stored. For example, since names are written in UTF-8 using writeUTF, you must read names using readUTF.

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Checking End of File

- If you keep reading data at the end of a stream, an **`EOFException`** would occur.
  - You can use **`input.available()`** to check it
  - **`input.available() == 0`** indicates that it is the end of a file

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

```java
import java.io.*;
public class DetectEndOfFile {
  public static void main(String[] args) {
    try {
      try (DataOutputStream output =
          new DataOutputStream(new FileOutputStream("test.dat"))) {
        output.writeDouble(4.5);
        output.writeDouble(43.25);
        output.writeDouble(3.2);
      }
      try (DataInputStream input =
          new DataInputStream(new FileInputStream("test.dat"))) {
        while (true) // or input.available() == 0
          System.out.println(input.readDouble());
      }
    } catch (EOFException ex) {
      System.out.println("All data were read");
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Run:

4.5

43.25

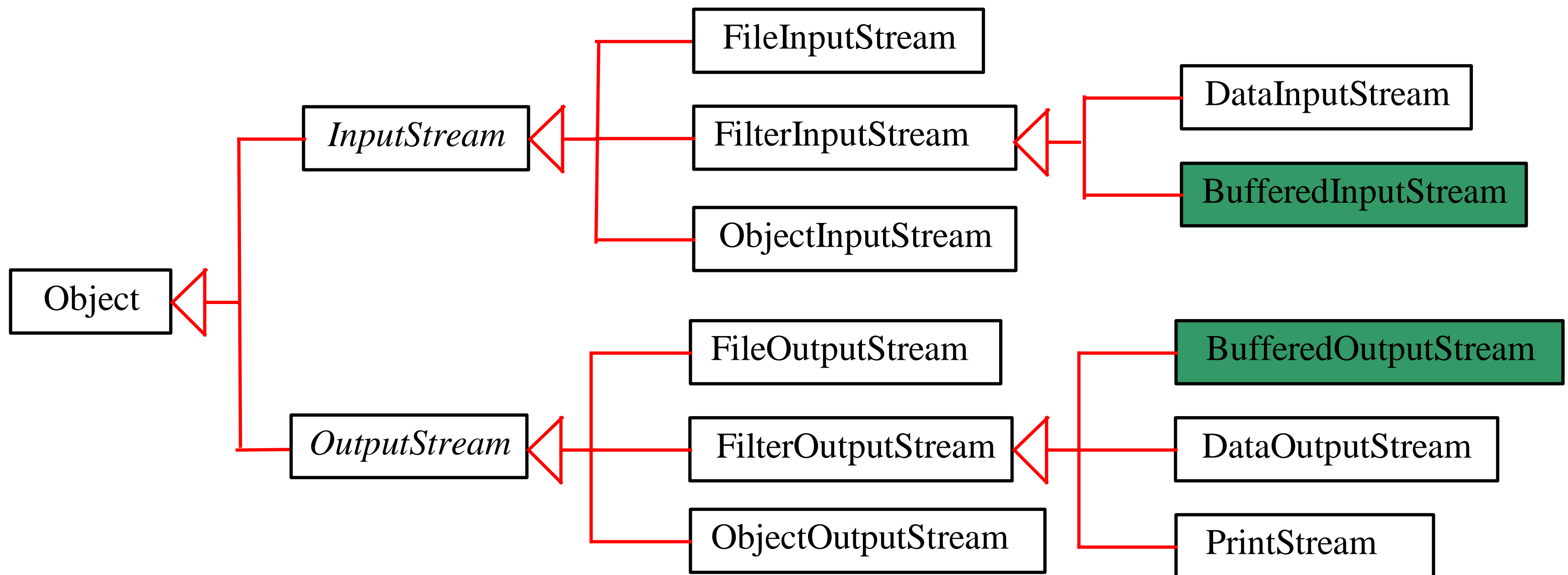
3.2

All data were read

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

BufferedInputStream/BufferedOutputStream

- Using buffers to speed up I/O:



BufferedInputStream/BufferedOutputStream does not contain new methods: all the methods BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Constructing BufferedInputStream/BufferedOutputStream

- Create a **BufferedInputStream**:

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int
  bufferSize)
```

- Create a **BufferedOutputStream**:

```
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStreamr out,
  int bufferSize)
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Copy File

```
import java.io.*;
public class CopyFile {
    /**
     * Main method
     *
     * @param args[0] for sourcefile
     * @param args[1] for target file
     */
    public static void main(String[] args) throws IOException {
        // Check command-line parameter usage
        if (args.length != 2) {
            System.out.println(
                    "Usage: java Copy sourceFile targetfile");
            System.exit(1);
        }
        // Check if source file exists
        File sourceFile = new File(args[0]);
        if (!sourceFile.exists()) {
            System.out.println("Source file " + args[0]
                    + " does not exist");
            System.exit(2);
        }
        // Check if target file exists
        File targetFile = new File(args[1]);
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

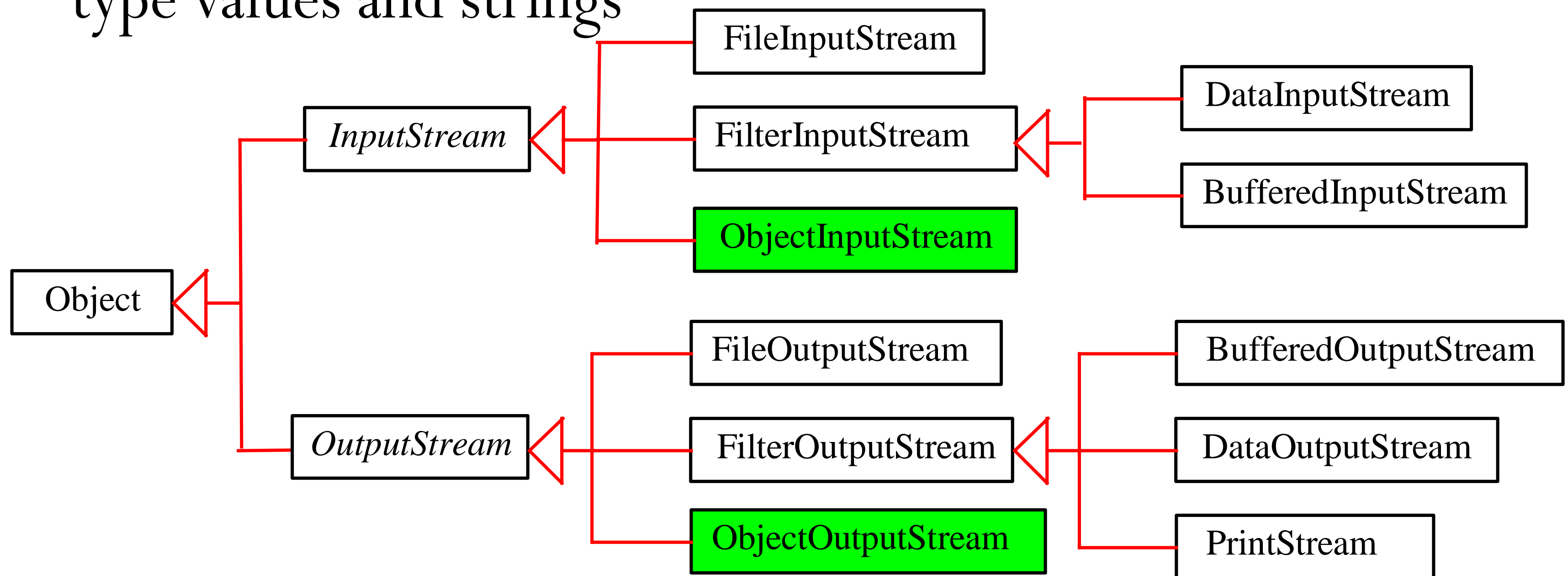
Copy File

```
        if (targetFile.exists()) {
            System.out.println("Target file " + args[1]
                    + " already exists");
            System.exit(3);
        }

        try (
                // Create an input stream
                BufferedInputStream input
                = new BufferedInputStream(new FileInputStream(sourceFile));
                // Create an output stream
                BufferedOutputStream output
                = new BufferedOutputStream(new FileOutputStream(targetFile));) {
            // Continuously read a byte from input and write it to output
            int r, numberOfBytesCopied = 0;
            while ((r = input.read()) != -1) {
                output.write((byte) r);
                numberOfBytesCopied++;
            }

            // Display the file size
            System.out.println(numberOfBytesCopied + " bytes copied");
        }
    }
}
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Object I/O

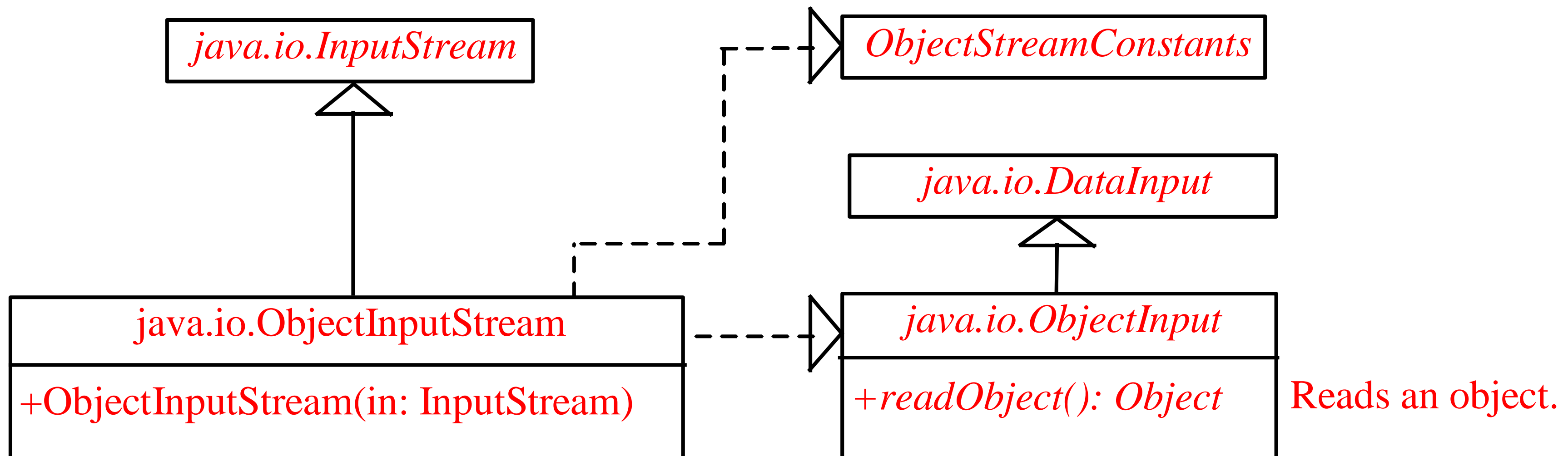
- **`DataInputStream/DataOutputStream`** enables you to perform I/O for primitive type values and strings.
- Finaly, **`ObjectInputStream/ObjectOutputStream`** enables you to perform I/O for objects in addition for primitive type values and strings



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

ObjectInputStream

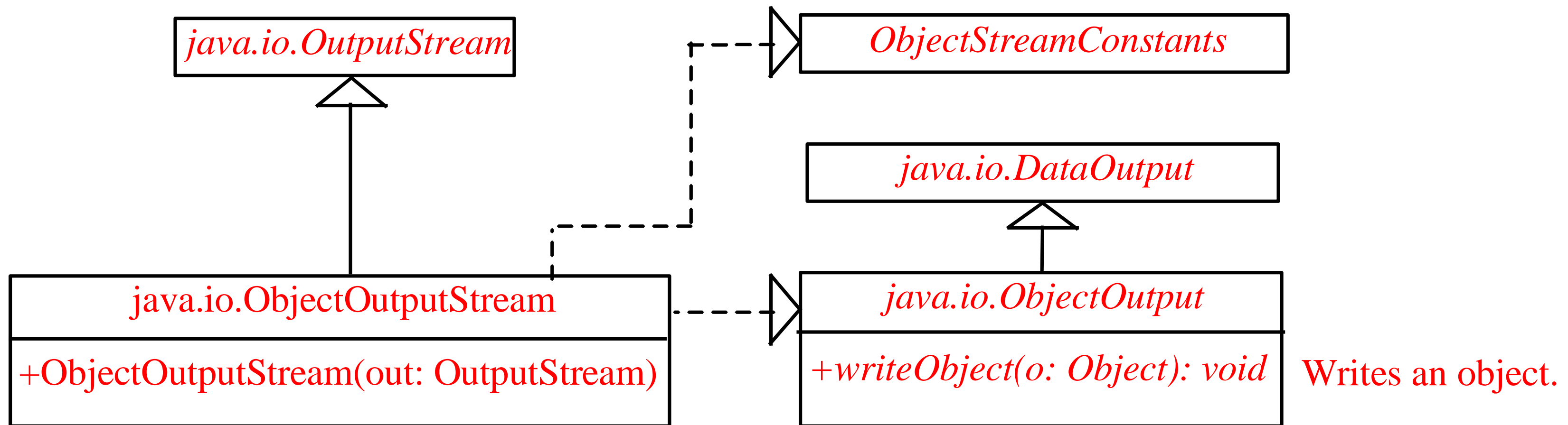
- **`ObjectInputStream`** extends **`InputStream`** and implements **`ObjectInput`** and **`ObjectStreamConstants`**



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

ObjectOutputStream

- **`ObjectOutputStream`** extends **`OutputStream`** and implements **`ObjectOutput`** and **`ObjectStreamConstants`**:



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Object Streams

- You may wrap an **ObjectInputStream/ObjectOutputStream** on any **InputStream/OutputStream** using the following constructors:

```
// Create an ObjectInputStream
public ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

```
import java.io.*;
public class TestObjectInputOutputStream {
  public static void main(String[] args) throws
       ClassNotFoundException, IOException {
    try ( // Create an output stream for file object.dat
      ObjectOutputStream output
       = new ObjectOutputStream(new FileOutputStream("object.dat"));) {
      // Write a string, double value, and object to the file
      output.writeUTF("John");
      output.writeDouble(85.5);
      output.writeObject(new java.util.Date());
    }
    try ( // Create an input stream for file object.dat
       ObjectInputStream input
        = new ObjectInputStream(new FileInputStream("object.dat"));) {
      // Read a string, double value, and object from the file
      String name = input.readUTF();
      double score = input.readDouble();
      java.util.Date date = (java.util.Date) (input.readObject());
      System.out.println(name + " " + score + " " + date);
    }
  }
}
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

The **`Serializable`** Interface

- Not all objects can be written to an output stream!
  - Objects that can be written to an object stream is said to be ***serializable***: a serializable object is an instance of the **`java.io.Serializable`** interface (so the class of a serializable object must implement **`Serializable`**)
    - The **`Serializable`** interface is a marker interface: it has no methods, so you don't need to add additional code in your class that implements **`Serializable`**
- Implementing this interface enables the Java serialization mechanism to automate the process of storing the objects and arrays

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

The `transient` Keyword

- An object instance of **`Serializable`** may contains **non-serializable instance data fields**
  - To enable the object to be serialized, you can use the transient keyword to mark these data fields to tell the JVM to ignore these fields when writing the object to an object stream:

```
public class Foo implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}
class A { } // A is not serializable
```

  - Note: When an object of the **`Foo`** class is serialized, only variable **`v1`** is serialized.
  - Variable **`v2`** is not serialized because it is a **`static`** variable, and variable **`v3`** is not serialized because it is marked **`transient`**
  - If **`v3`** were not marked transient, a **`java.io.NotSerializableException`** would occur.

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

Serializing Arrays

- An array is serializable if all its elements are serializable:
  - The entire array can be saved using writeObject into a file and later restored using readObject:

```
int[] numbers = {1, 2, 3, 4, 5};
String[] strings = {"John", "Susan", "Kim"};
try ( // Create an output stream for file array.dat
  ObjectOutputStream output = new ObjectOutputStream(new
    FileOutputStream("array.dat", true));) {
  // Write arrays to the object output stream
  output.writeObject(numbers);
  output.writeObject(strings);
}
try ( // Create an input stream for file array.dat
  ObjectInputStream input =
    new ObjectInputStream(new FileInputStream("array.dat"));){
  int[] newNumbers = (int[])(input.readObject());
  String[] newStrings = (String[])(input.readObject());
}
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

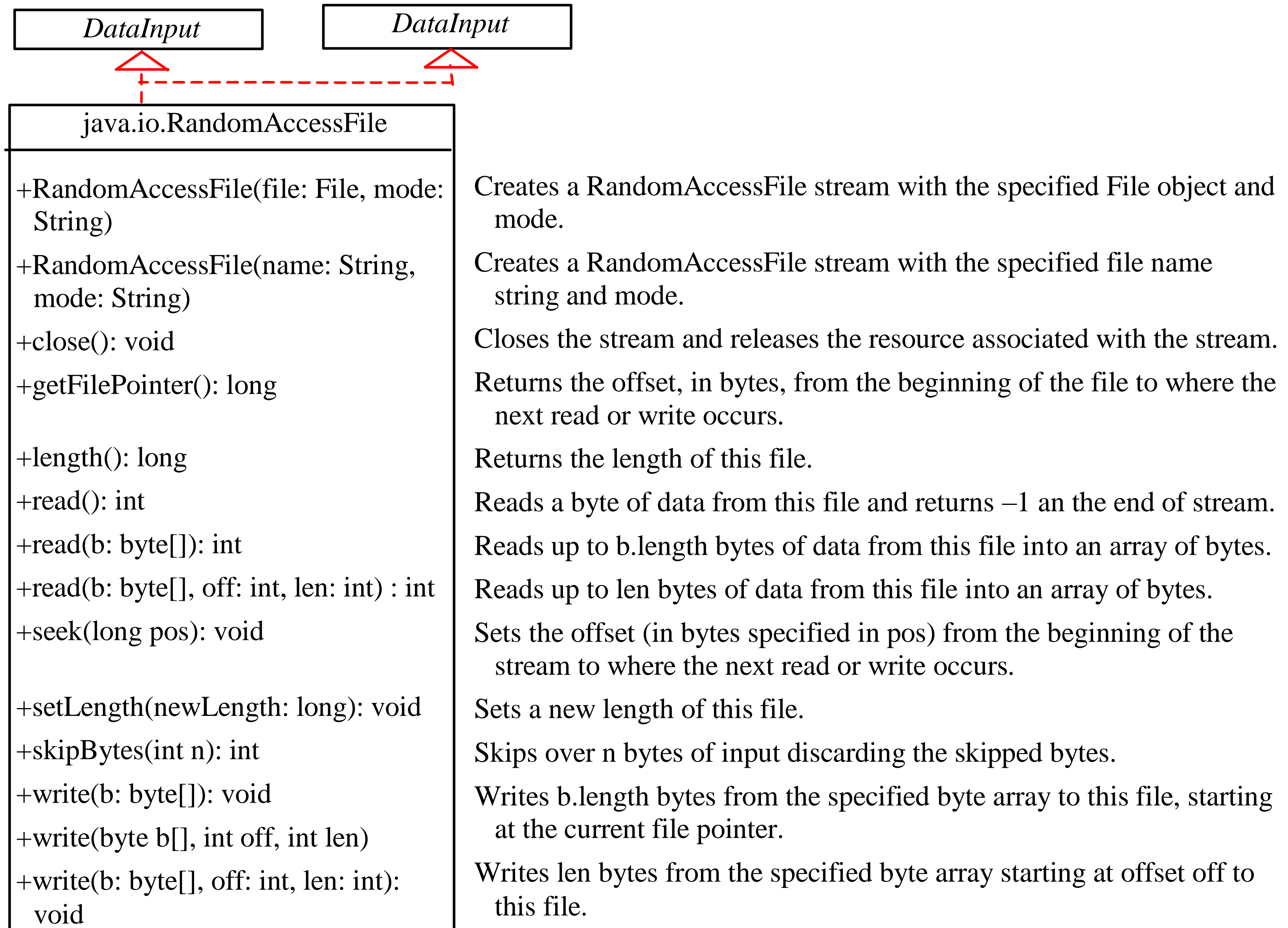
Random Access Files

- All of the previous are *read-only* or *write-only* streams
  - The external files of these streams are *sequential* files that cannot be updated without creating a new file
- **RandomAccessFile** class allows a file to be read from and write to at random locations.

```
RandomAccessFile raf =
  new RandomAccessFile("test.dat", "rw");
  //allows read and write

RandomAccessFile raf =
  new RandomAccessFile("test.dat", "r");
  //read only
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

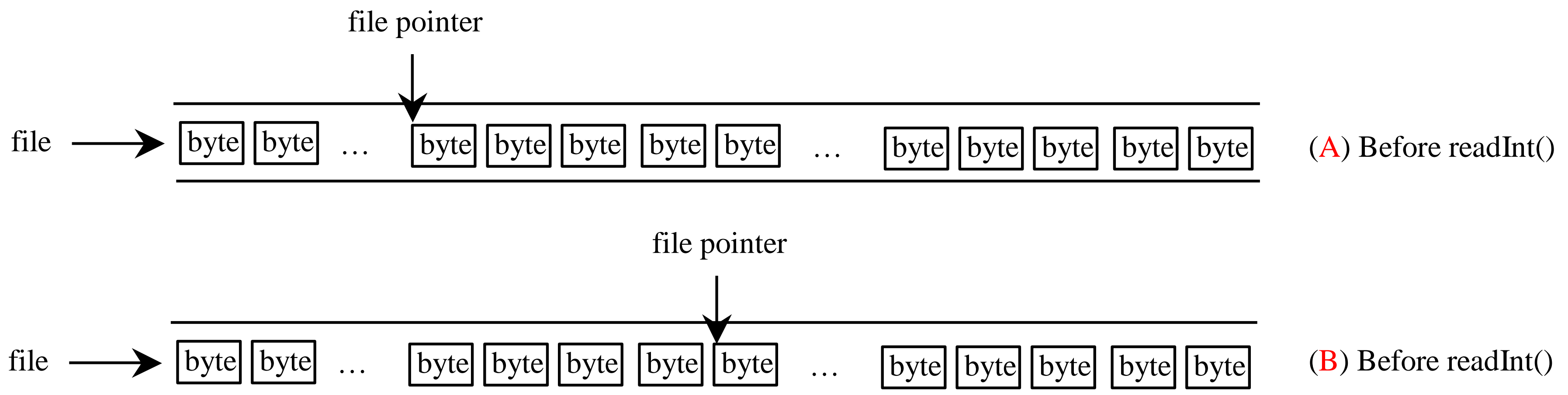
Random Access Files



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

File Pointer

- A random access file consists of a sequence of bytes and there is a special marker called ***file pointer*** that is positioned at one of these bytes
  - A read or write operation takes place at the location of the file pointer
  - When a file is opened, the file pointer sets at the beginning of the file
  - When you read or write data to the file, the file pointer moves forward to the next data
    - For example, if you read an **`int`** value using **`readInt()`**, the JVM reads four bytes from the file pointer and now the file pointer is four bytes ahead of the previous location.



(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

RandomAccessFile Methods

- Many methods in **RandomAccessFile** are the same as those in **DataInputStream** and **DataOutputStream**: **readInt()**, **readLong()**, **writeDouble()**, **readLine()**, **writeInt()**, **writeLong()** …
- **void seek(long pos)** sets the offset from the beginning of the **RandomAccessFile** stream to where the next read or write occurs.
- **long getFilePointer()** returns the current offset, in bytes, from the beginning of the file to where the next read or write occurs.
- **long length()** returns the length of the file.
- **final void writeChar(int v)** writes a character to the file as a two-byte Unicode, with the high byte written first.
- **final void writeChars(String s)** writes a string to the file as a sequence of characters.

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)

```java
import java.io.*;
public class TestRandomAccessFile {
  public static void main(String[] args) throws IOException {
    try ( // Create a random access file
      RandomAccessFile inout = new RandomAccessFile("inout.dat", "rw");
    ) {
      // Clear the file to destroy the old contents if exists
      inout.setLength(0);
      // Write new integers to the file
      for (int i = 0; i < 200; i++)
        inout.writeInt(i);
      // Display the current length of the file
      System.out.println("Current file length is " + inout.length());
      // Retrieve the first number
      inout.seek(0); // Move the file pointer to the beginning
      System.out.println("The first number is " + inout.readInt());
      // Retrieve the second number
      inout.seek(1 * 4); // Move the file pointer to the second number
      System.out.println("The second number is " + inout.readInt());
      // Retrieve the tenth number
      inout.seek(9 * 4); // Move the file pointer to the tenth number
      System.out.println("The tenth number is " + inout.readInt());
      // Modify the eleventh number
      inout.writeInt(555);
      // Append a new number
      inout.seek(inout.length()); // Move the file pointer to the end
      inout.writeInt(999);
      // Display the new length
      System.out.println("The new length is " + inout.length());
      // Retrieve the new eleventh number
      inout.seek(10 * 4); // Move the file pointer to the eleventh number
      System.out.println("The eleventh number is " + inout.readInt());
    }
  }
}
```

```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
The new length is 804
The eleventh number is 555
```


(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)