COP2805C
Advanced Java Programming

Module 2

Files and Streams

Java I/O: Files and Streams

- A stream is an abstraction which either consumes or produces information
- All streams behave in a similar manner, regardless of the endpoints they are associated with (file, keyboard, network socket, etc)
- Streams are implemented within the class hierarchies provided in the java.io package
- Java defines 2 types of streams: character streams and byte streams

Character Streams

- Character streams read and write text data
- Two abstract classes are the top of the character stream class hierarchy:

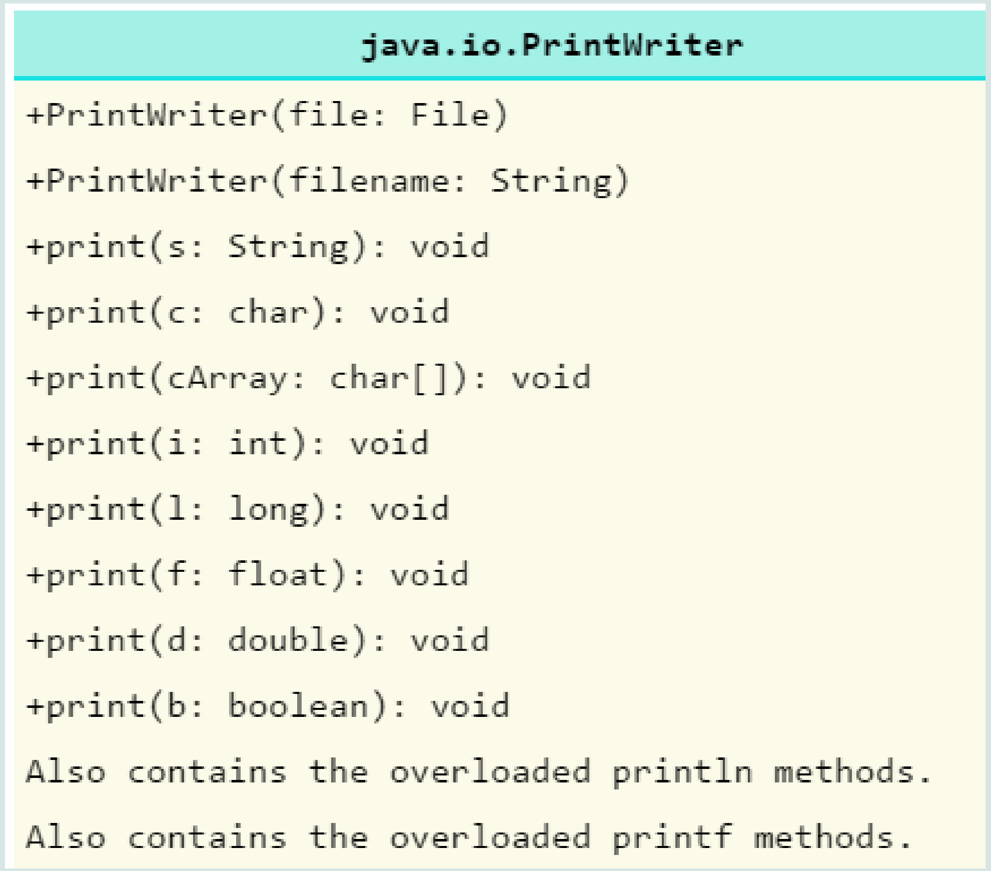
```
public abstract class Reader
    extends Object
    implements Readable, Closeable
```

```
public abstract class Writer
    extends Object
    implements Appendable, Closeable, Flushable
```

Text File I/O: PrintWriter

public class PrintWriter

extends Writer



```
java.io.PrintWriter
+PrintWriter(file: File)
+PrintWriter(filename: String)
+print(s: String): void
+print(c: char): void
+print(cArray: char[]): void
+print(i: int): void
+print(l: long): void
+print(f: float): void
+print(d: double): void
+print(b: boolean): void
Also contains the overloaded println methods.
Also contains the overloaded printf methods.
```

Using PrintWriter

```
import java.io.IOException;
import java.io.File;
import java.io.PrintWriter;

public class PrintWriterTest {
   public static void main(String[] args) throws IOException
   {
      final String FILENAME = "demofile.txt";

      File file = new File(FILENAME);
      if (file.exists())
         System.out.println("File " + FILENAME + " exists.");

      // create and/or open the file for writing
      PrintWriter output = new PrintWriter(file);

      // write formatted output to the file
      output.print("John Smith ");
      output.println(90);
      output.print("Eric Jones ");
      output.println(85);
      System.out.println("Wrote output file");

      // close the file
      output.close();
   }
}
```

The IOException Class

- Operations performed by some I/O classes throw an `IOException`
  - General I/O errors: data cannot be read or written (IOException)
  - A file does not exist or can't be found (FileNotFoundException)
  - The end of a file has been reached (EOFException)
- `IOException` is a <u>checked</u> exception

Catching PrintWriter Exceptions

```
import java.io.IOException;
import java.io.File;
import java.io.PrintWriter;

public class PrintWriterTest {
    public static void main(String[] args)
    {
        final String FILENAME = "demofile.txt";

        File file = new File(FILENAME);
        if (file.exists())
            System.out.println("File " + FILENAME + " exists.");

        // create and/or open the file for writing
        try (PrintWriter output = new PrintWriter(file);) {
            // write formatted output to the file
            output.print("John Smith ");
            output.println(90);
            output.print("Eric Jones ");
            output.println(85);
            System.out.println("Wrote output file");
        } catch (IOException e) {
            System.err.println("Exception! " + e);
        }

        // close the file
        //output.close(); // not needed!
    }
}
```

Text File I/O: BufferedReader

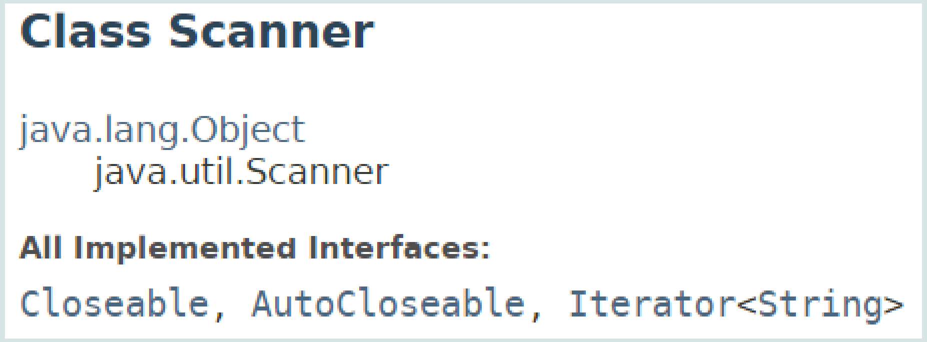
- FileReader extends Reader for character-by-character file input
- BufferedReader also extends Reader to read line-by-line
  - We want to combine the line-by-line abilities of BufferedReader with the file reading abilities of FileReader
  - We do this by "wrapping" the FileReader with a BufferedReader:

```
BufferedReader in =
    new BufferedReader(new FileReader(FILENAME))
```

Using BufferedReader

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

public class BufferedReaderDemo
{
   public static void main(String[] args)
   {
      final String FILENAME = "demofile.txt";

      // open the file for reading
      try (BufferedReader in = new BufferedReader(new FileReader(FILENAME)))
      {
         String inLine = null;

         // read data output to the file
         while ((inLine = in.readLine()) != null)
            System.out.println(inLine);
      } catch (IOException ex) {
         System.out.println("Exception! " + ex);
      }
   }
}
```

BufferedReader vs. Scanner

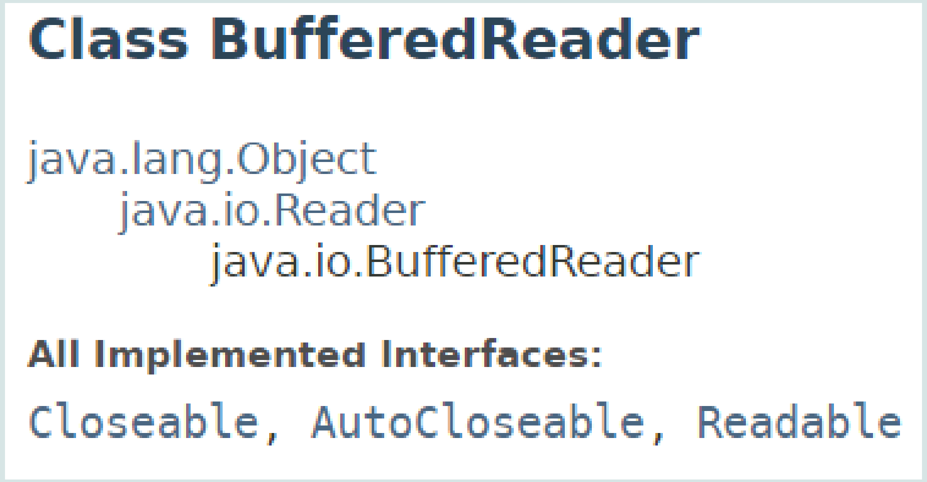


Class Scanner

java.lang.Object
    java.util.Scanner

All Implemented Interfaces:

`Closeable, AutoCloseable, Iterator<String>`



Class BufferedReader

java.lang.Object
    java.io.Reader
        java.io.BufferedReader

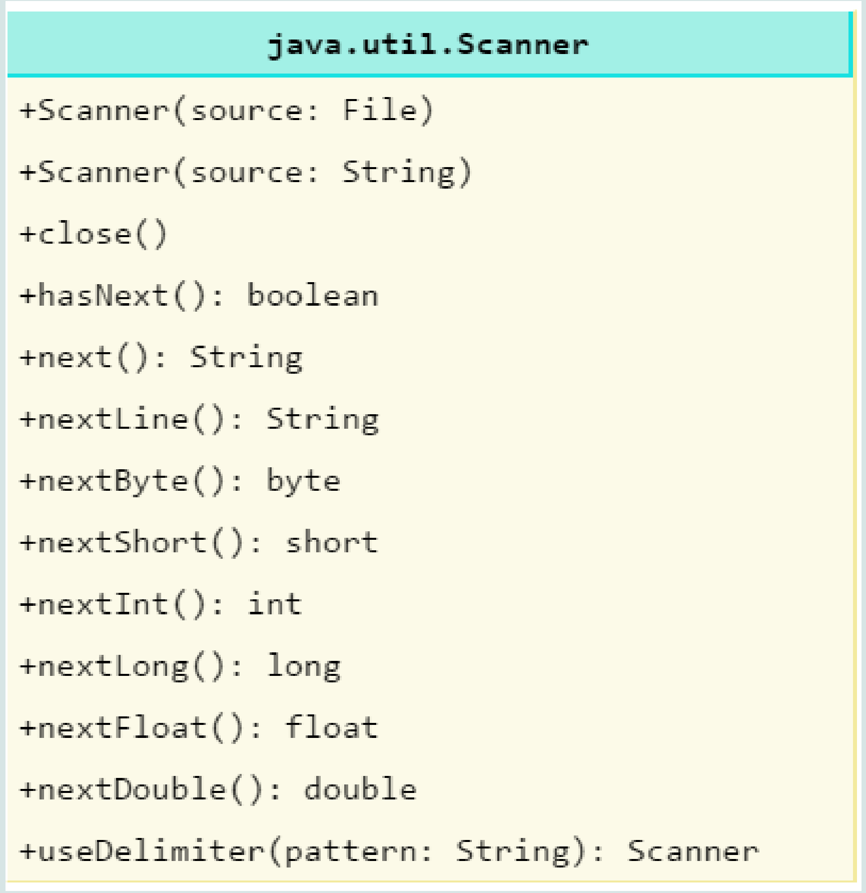
All Implemented Interfaces:

`Closeable, AutoCloseable, Readable`

- Scanner is token-oriented, easier to read individual data items by type
- BufferedReader is synchronized (thread-safe)

Reading Data with **Scanner**

java.util.Scanner input = new Scanner(System.in);



```
java.util.Scanner
+Scanner(source: File)
+Scanner(source: String)
+close()
+hasNext(): boolean
+next(): String
+nextLine(): String
+nextByte(): byte
+nextShort(): short
+nextInt(): int
+nextLong(): long
+nextFloat(): float
+nextDouble(): double
+useDelimiter(pattern: String): Scanner
```

Using the Scanner

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import java.util.InputMismatchException;

public class ScannerDemo {
    public static void main(String[] args) {
        final String FILENAME = "demofile.txt";

        // create a file instance
        File file = new File(FILENAME);
        try (Scanner input = new Scanner(file);) {
            // read data from file
            while (input.hasNext()) {
                String nameFirst = input.next();
                String nameLast = input.next();
                int score = input.nextInt();
                System.out.println(nameFirst + " " + nameLast + " " + score);
            }
          // catch with suppressed
        } catch (IOException | InputMismatchException ex) {
            System.err.println("Exception! " + ex);
        }
    }
}
```

Byte Streams and Binary I/O

- Byte streams read and write binary data
- Two abstract classes are the top of the class hierarchy:

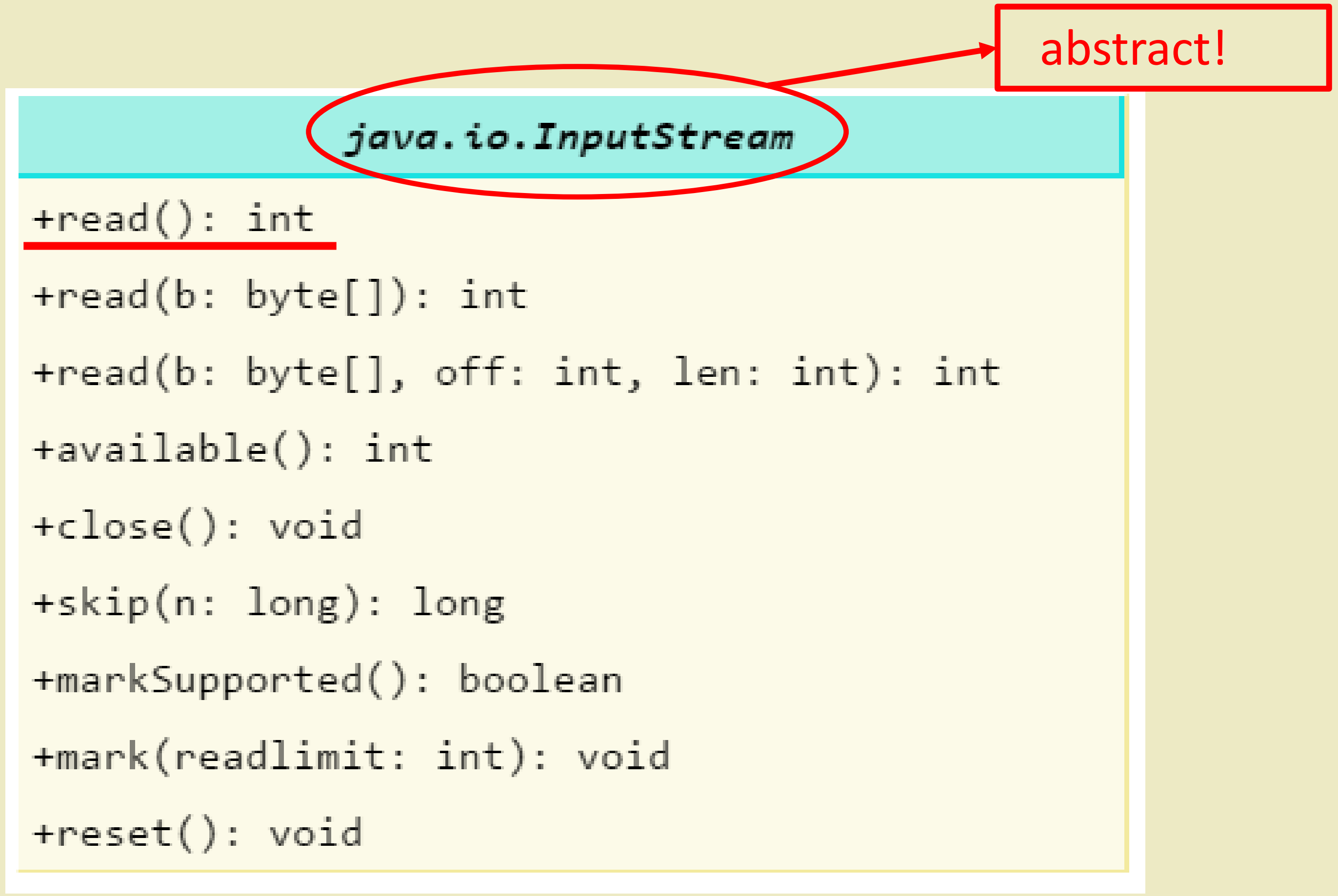
public abstract class InputStream
 extends Object implements Closeable

This abstract class is the superclass of all classes representing an input stream of bytes.

public abstract class OutputStream
 extends Object implements Closeable, Flushable

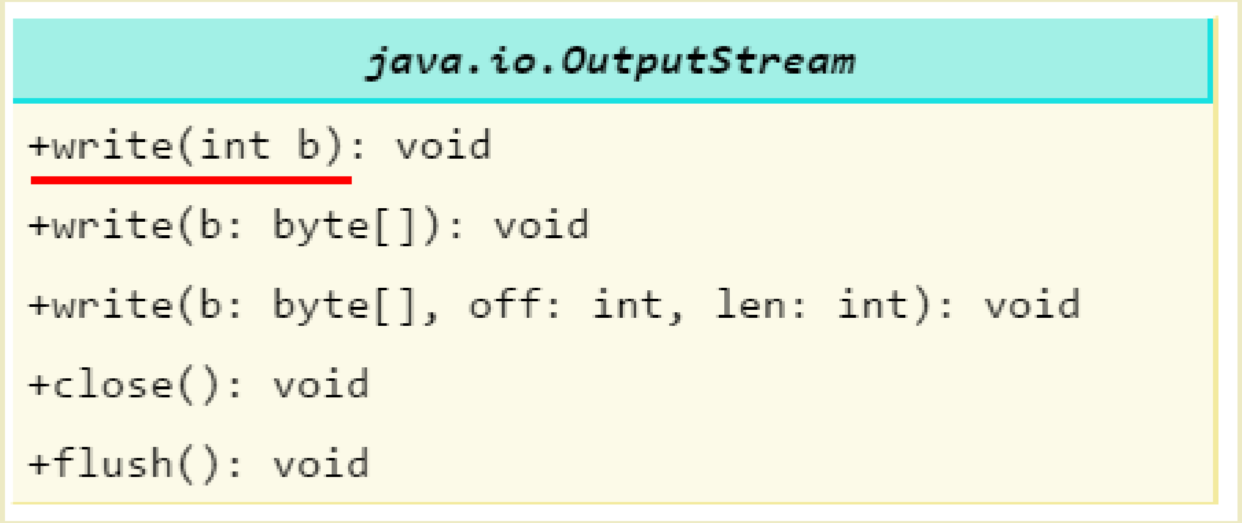
This abstract class is the superclass of all classes representing an output stream of bytes.

java.io.InputStream



```
java.io.InputStream
+read(): int
+read(b: byte[]): int
+read(b: byte[], off: int, len: int): int
+available(): int
+close(): void
+skip(n: long): long
+markSupported(): boolean
+mark(readlimit: int): void
+reset(): void
```

java.io.OutputStream

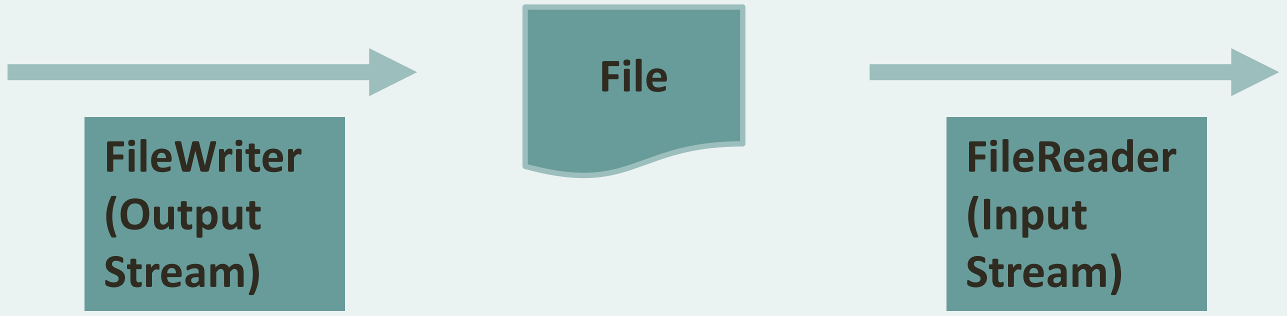


```
java.io.OutputStream
+write(int b): void
+write(b: byte[]): void
+write(b: byte[], off: int, len: int): void
+close(): void
+flush(): void
```

The Standard System Streams

- java.lang.System
  - docs.oracle.com/javase/8/docs/api/java/lang/System.html
- public static final InputStream in
  - The "standard" input stream. This stream is already open and ready to supply input data. Typically this stream corresponds to keyboard input or another input source specified by the host environment or user.
- public static final PrintStream out
  - The "standard" output stream. This stream is already open and ready to accept output data. Typically this stream corresponds to display output or another output destination specified by the host environment or user.
- public static final PrintStream err
  - The "standard" error output stream. This stream is already open and ready to accept output data.

- A program can manage multiple streams simultaneously



- There are three standard I/O streams:
  - *standard output* – defined by `System.out`
  - *standard input* – defined by `System.in`
  - *standard error* – defined by `System.err`
- We use `System.out` when we execute `println` statements
- `System.out` and `System.err` typically represent the console window
- `System.in` typically represents keyboard input, which we use with `Scanner`

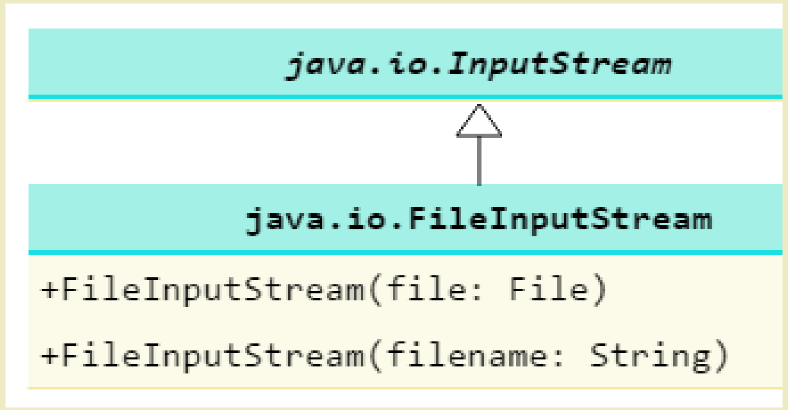
Binary File I/O

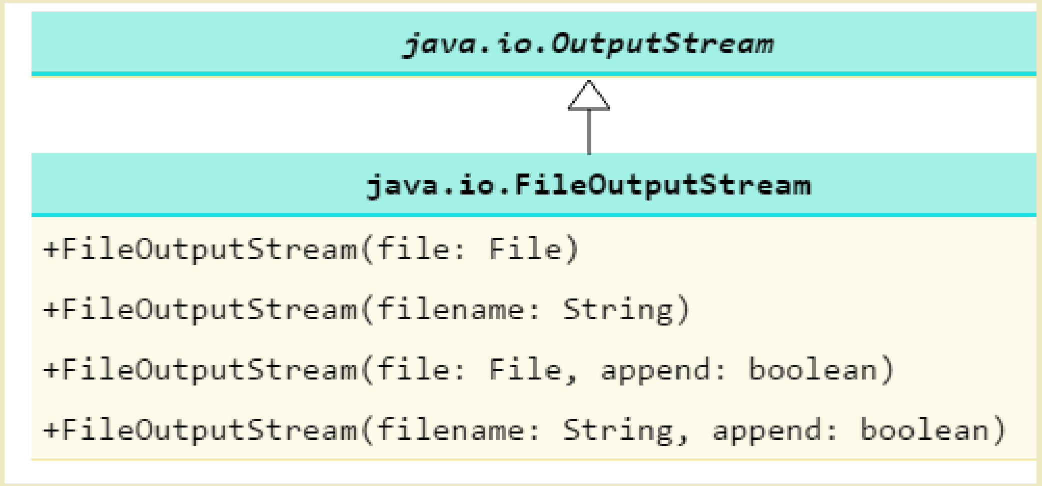
- File I/O is provided by binary file streams which extend byte streams

- **Class FileInputStream**
- java.lang.Object
  - java.io.InputStream
    - java.io.FileInputStream

- **Class FileOutputStream**
- java.lang.Object
  - java.io.OutputStream
    - java.io.FileOutputStream

FileInputStream & FileOutputStream


java.io.InputStream
java.io.FileInputStream
+FileInputStream(file: File)
+FileInputStream(filename: String)


java.io.OutputStream
java.io.FileOutputStream
+FileOutputStream(file: File)
+FileOutputStream(filename: String)
+FileOutputStream(file: File, append: boolean)
+FileOutputStream(filename: String, append: boolean)

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

public class SimpleFileInput {
    public static void main(String[] args) {
        final String FILENAME = "demofile.txt";

        // open file, read byte-wise content and display
        try (FileInputStream fInput = new FileInputStream(FILENAME);) {
            int i = -1;
            do {
                i = fInput.read();
                if (i != -1)
                    System.out.print((char)i);
            } while (i != -1);
        } catch (IOException ex) {
            System.err.println("Exception! " + ex);
        }
    }
}
```

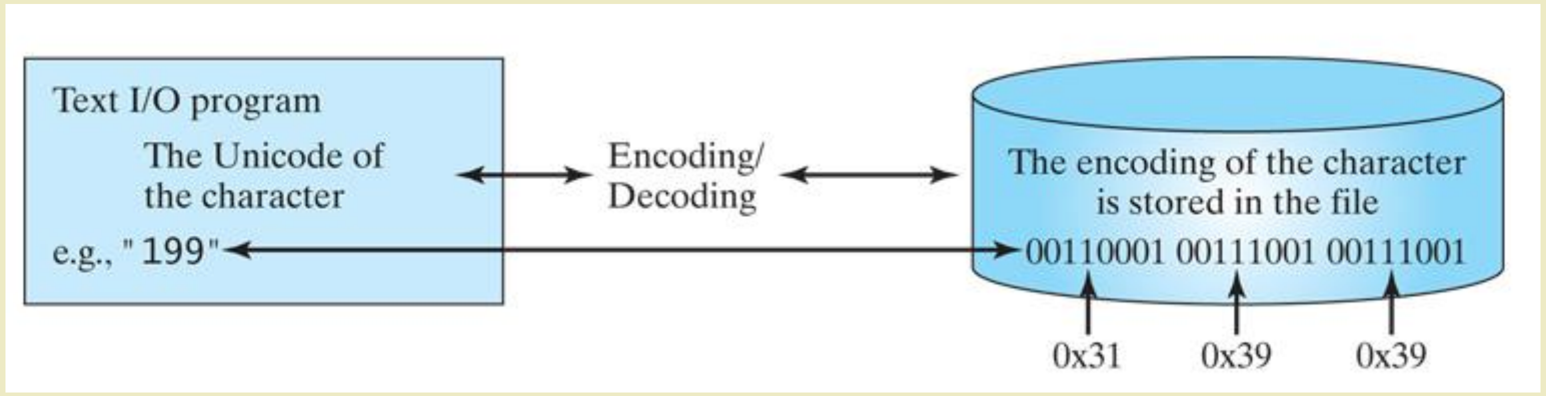
- FileOutputStream write(int b) writes a single unsigned byte to a file (even though the parameter is an int)
- FileInputStream read() reads an unsigned byte as an int, returns -1 for end of file
- This means that only values between 0 and 255 can be written/read using these methods

- Instantiating a FileOutputStream will create a new file if it doesn't already exist
  - Set the <u>append</u> parameter to **true** in the applicable constructors to prevent truncation of an existing file

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

public class SimpleFileOutput {
    public static void main(String[] args) {
        final String FILENAME = "demofile.txt";
        final String DATA = "Hello, World!\nThis is a test.";
        byte[] data = DATA.getBytes();

        // open the file and write the byte array
        try (FileOutputStream fOutput = new FileOutputStream(FILENAME, true);) {
            fOutput.write(data);
        } catch (IOException ex) {
            System.err.println("Exception!" + ex);
        }
    }
}
```

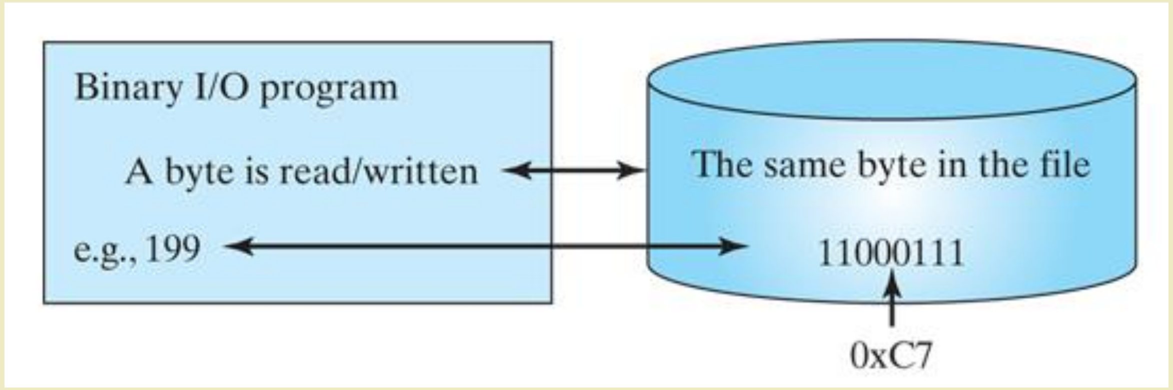
Binary I/O

- Files are text or binary
  - e.g. Java source code (.java) is stored as text data
  - e.g. Java byte code (.class) is stored as binary data



Text I/O vs. Binary I/O

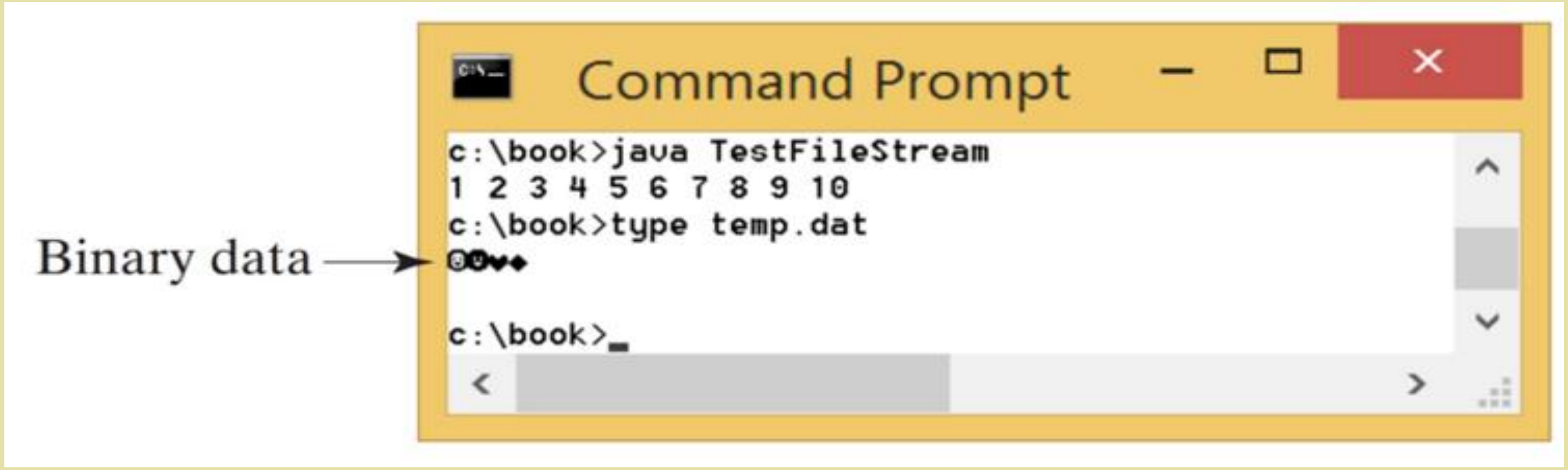
- In general, binary I/O is more efficient in both storage space and performance since no encoding is necessary



- If you need to read data with a text editor, use text files
  - Otherwise use binary files
- Binary is also platform-independent (portable); different operating systems can use different text encodings
  - This is why Java uses binary .class files

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

public class TestFileStream {
   public static void main(String[] args) {
      try (
            // Create an output stream for the file
            FileOutputStream output = new FileOutputStream("temp.dat"); ) {
         // output values to the file
         for (int i = 1; i <= 10; i++)
            output.write(i);
      } catch (IOException ex) {
         System.err.println("Output Exception: " + ex);
      }

      try (
            // Create an input stream for the file
            FileInputStream input = new FileInputStream("temp.dat"); ) {
         // read values from the file
         int value;
         while ((value = input.read()) != -1)
            System.out.print(value + " " );
      } catch (IOException ex) {
         System.err.println("Input Exception: " + ex);
      }
   }
}
```

The file "temp.dat" in the previous example is a binary file that can be read from a Java program, but not using a text editor:



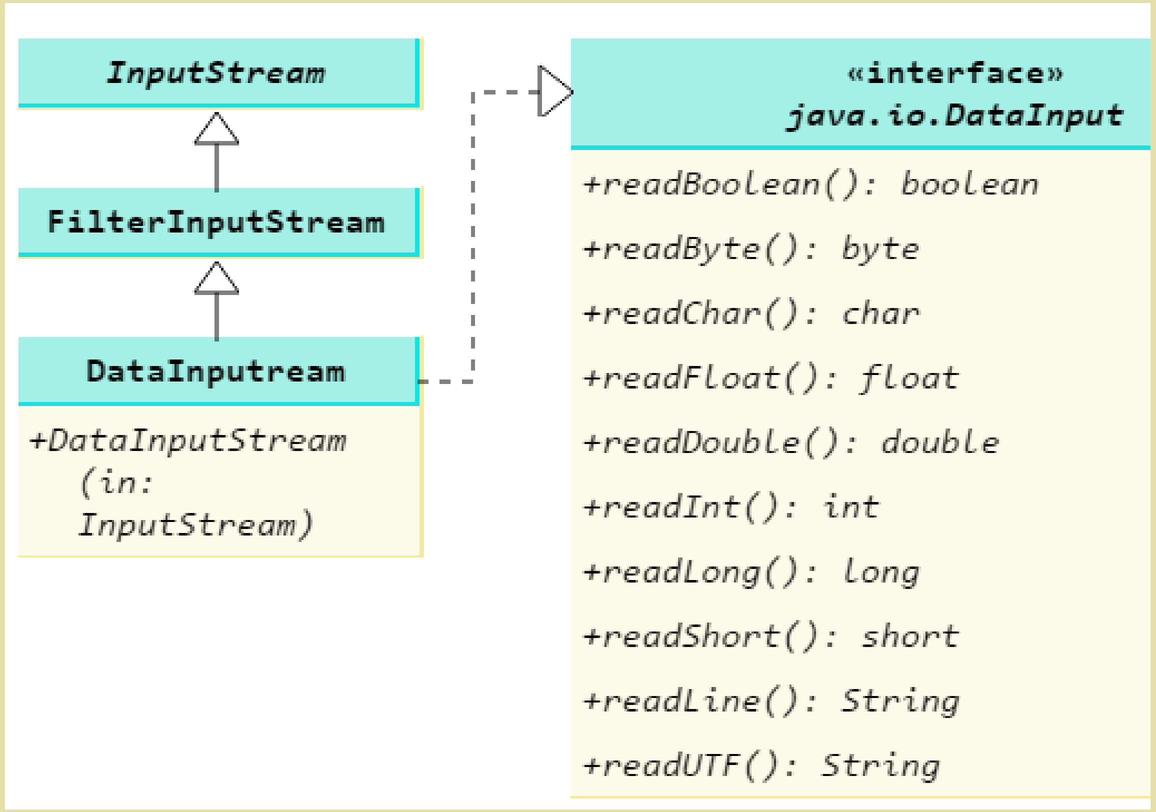
```
c:\book>java TestFileStream
1 2 3 4 5 6 7 8 9 10
c:\book>type temp.dat
☺☻♥♦

c:\book>_
```

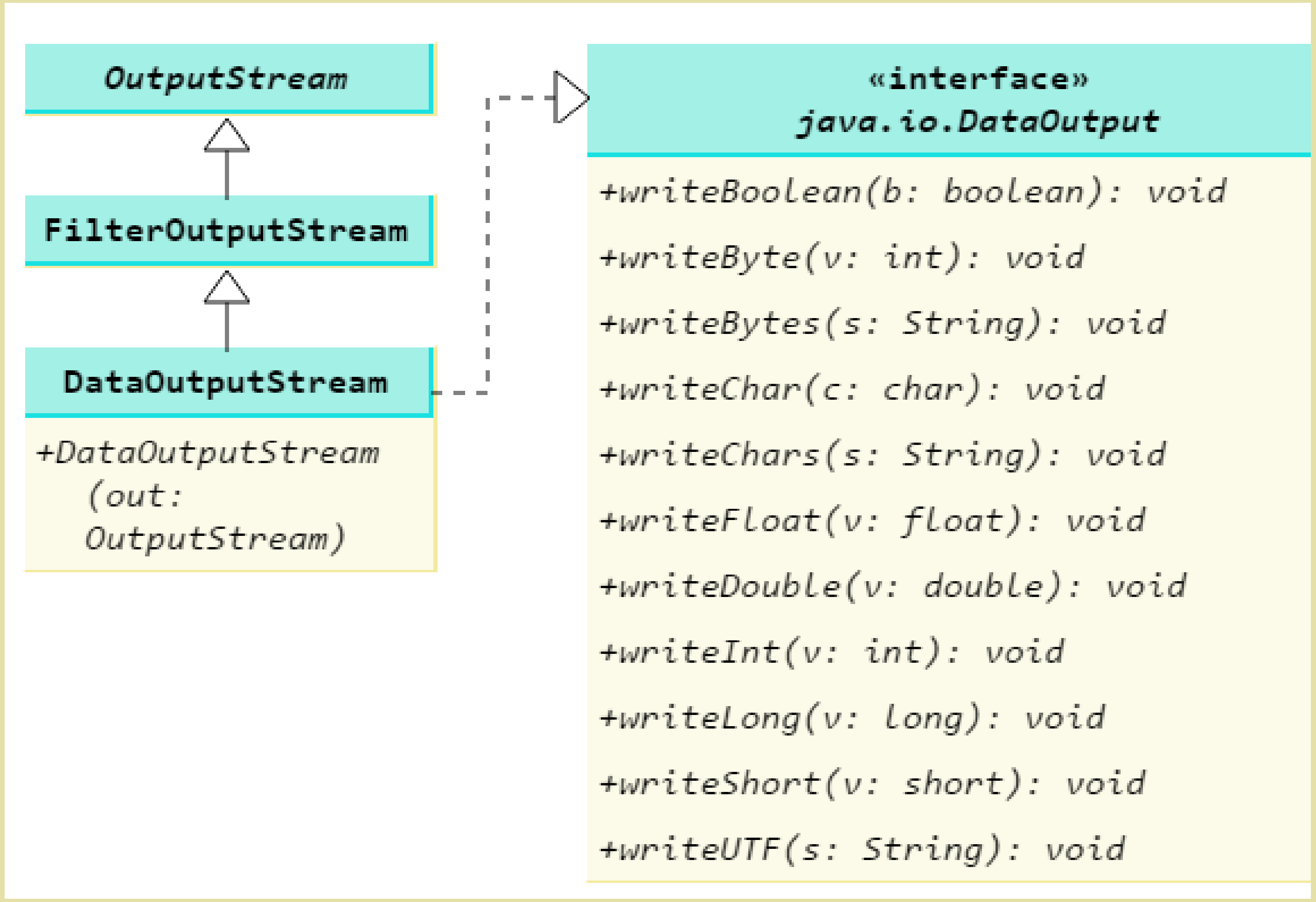
FilterInputStream & FilterOutputStream

- Basic streams are used for byte-based operations
- **Filter streams** wrap the byte stream with a filter to read/write other types (integer, double, etc)
- **FilterInputStream** and **FilterOutputStream** are base classes for the more useful **DataInputStream** and **DataOutputStream** classes (next slide)

DataInputStream & DataOutputStream

- DataInputStream and DataOutputStream do conversions between bytes and primitive types (or Strings)





OutputStream
FilterOutputStream
DataOutputStream
+DataOutputStream (out: OutputStream)
«interface» java.io.DataOutput
+writeBoolean(b: boolean): void
+writeByte(v: int): void
+writeBytes(s: String): void
+writeChar(c: char): void
+writeChars(s: String): void
+writeFloat(v: float): void
+writeDouble(v: double): void
+writeInt(v: int): void
+writeLong(v: long): void
+writeShort(v: short): void
+writeUTF(s: String): void

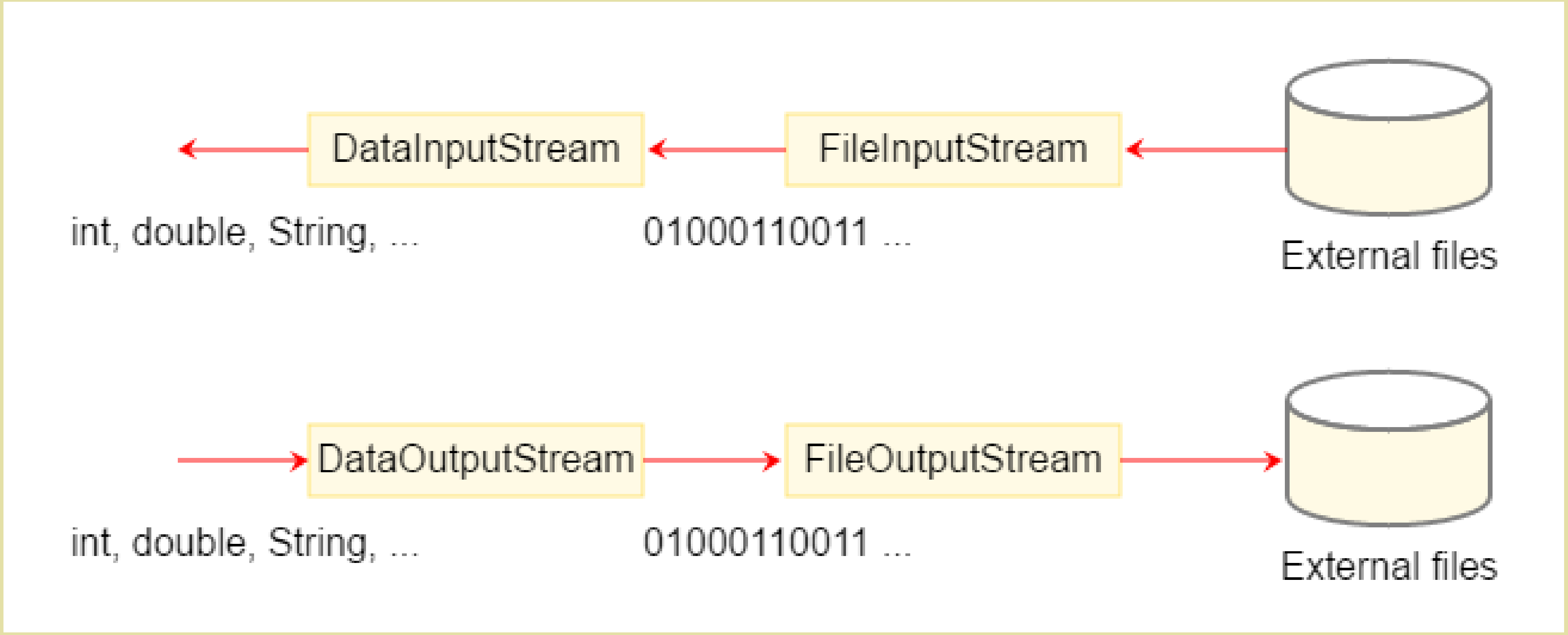
Characters and Strings in Binary I/O

- DataOutput.writeChar(char c) writes a Unicode character
- DataOutput.writeChars(String s) writes all characters of the String as Unicode
- DataOutput.writeBytes(String s) only writes the lower byte of the Unicode characters (useful for ASCII-only data)
- DataOutput.writeUTF(String s) writes a UTF-8 string (next slide)

UTF-8 Format

- UTF-8 is a character encoding format which writes variable length characters depending on whether a character is Unicode (2-3 bytes) or ASCII (one byte)
- If most of the characters in a long string are ASCII, using UTF-8 can save space

```
import java.io.*;

public class TestDataStream {
   public static void main(String[] args) {
      final String FILENAME = "temp.dat";
      try (   // create an output stream for file
              DataOutputStream output = new DataOutputStream(new FileOutputStream(FILENAME)); ) {
           // write student test scores to the file
           output.writeUTF("John");
           output.writeDouble(85.5);
           output.writeUTF("Jim");
           output.writeDouble(95.0);
           output.writeUTF("George");
           output.writeDouble(89.9);
      }  catch (IOException ex) {
           System.err.println("Output Exception: " + ex);
      }

      try (   // create an input stream for the file
              DataInputStream input = new DataInputStream(new FileInputStream(FILENAME)); ) {
           // 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());
      } catch (IOException ex) {
           System.err.println("Input Exception: " + ex);
      }
   }
}
```



Caution

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

Detecting the End of a File

```
 java.io.*;

public class DetectEndOfFile {
   public static void main(String[] args) {
      final String FILENAME = "temp.dat";
      try (   // create an output stream for file
            DataOutputStream output = new DataOutputStream(new FileOutputStream(FILENAME)); ) {
         // write student test scores to the file
         output.writeUTF("John");
         output.writeDouble(85.5);
         output.writeUTF("Jim");
         output.writeDouble(95.0);
         output.writeUTF("George");
         output.writeDouble(89.9);
      }  catch (IOException ex) {
         System.err.println("Output Exception: " + ex);
      }

      try (   // create an input stream for the file
            DataInputStream input = new DataInputStream(new FileInputStream(FILENAME)); ) {
         // read student test scores from the file
         while (true)
            System.out.println(input.readUTF() + " " + input.readDouble());
      } catch (EOFException ex) {
         System.out.println("All data have been read");
      } catch (IOException ex) {
         System.err.println("Input Exception: " + ex);
      }
   }
}
```

BufferedInputStream & BufferedOutputStream

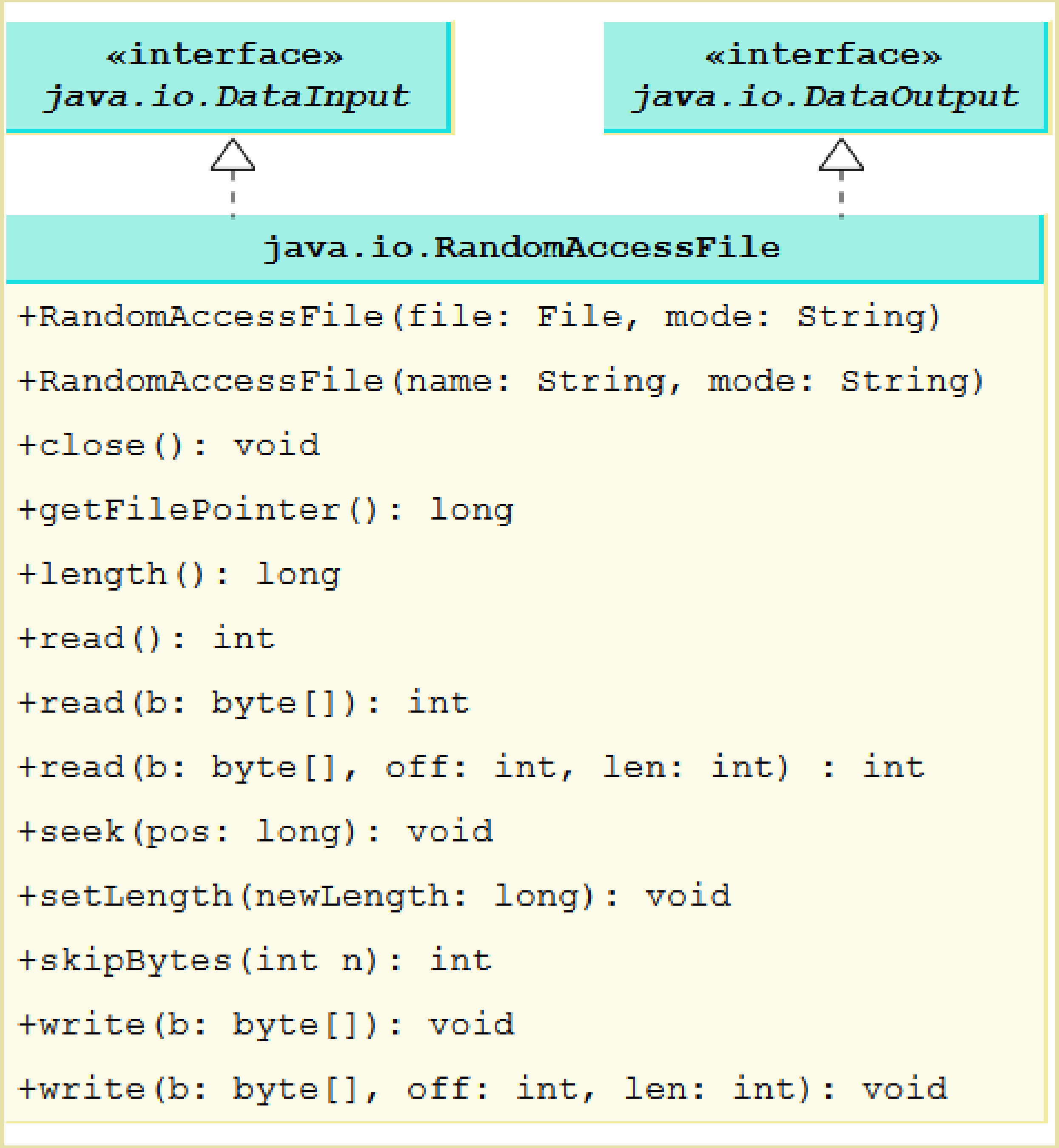
- Buffered I/O reads from and writes to memory until the allocated space (the buffer) is full. This reduces interaction with the storage device, resulting in faster processing.
- The default buffer size is 8192, but this is undocumented since there is rarely a need to modify it.

Using Buffered I/O to Copy Files

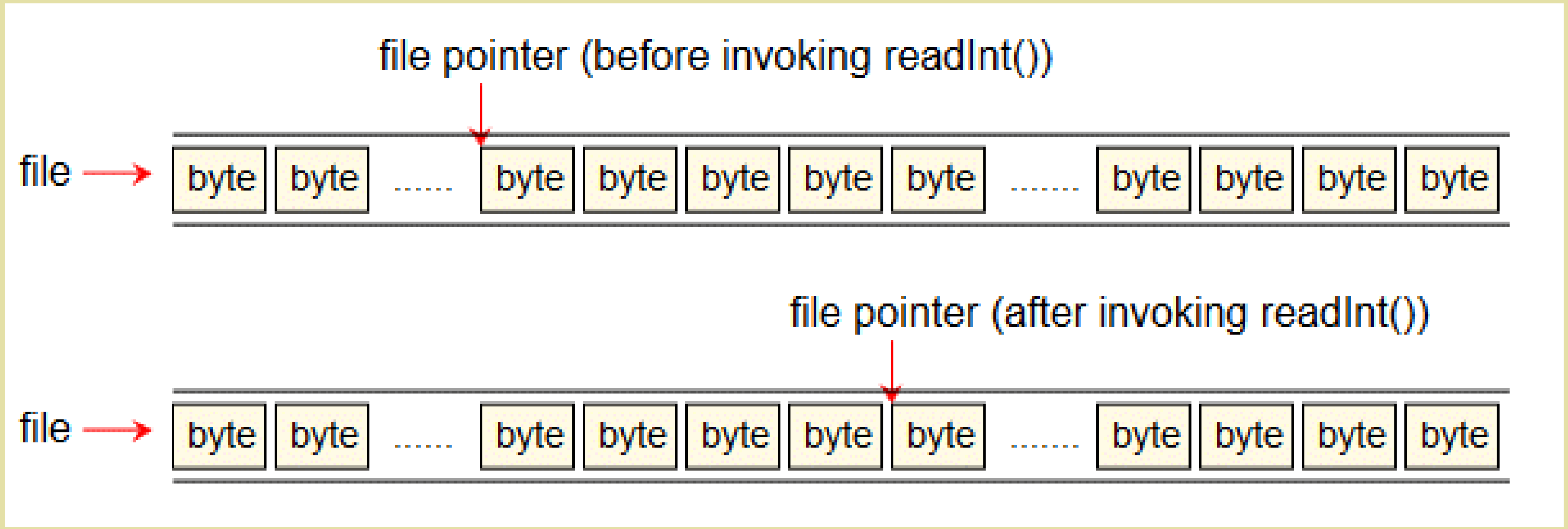
```
import java.io.*;

public class FileCopier {
   // command line arguments:
   // args[0] is source file
   // args[1] is target file
   public static void main(String[] args) {
      // check command line arguments
      if (args.length != 2) {
         System.out.println("Usage: java FileCopier source target");
         System.exit(1);
      }

      // check existence of source file
      File sourceFile = new File(args[0]);
      if (!sourceFile.exists()) {
         System.out.println("Source file " + args[0] + " does not exist.");
         System.exit(2);
      }

      // check existence of target file
      File targetFile = new File(args[1]);
      if (targetFile.exists()) {
         System.out.println("Target file " + args[1] + " already exists.");
         System.exit(3);
      }
      int numberOfBytesCopied = 0;
      try (  // create an input stream
          BufferedInputStream inpu = new BufferedInputStream(
              new FileInputStream(sourceFile));
          // create an output stream
          BufferedOutputStream output = new BufferedOutputStream(
              new FileOutputStream(targetFile)); ) {
        // continuously read a byte from input and write to output
        int r = 0;
        while ((r = input.read()) != -1) {
          output.write((byte) r);
          numberOfBytesCopied++;
        }

      } catch (IOException ex) {
        System.err.println("Exception!" + ex);
      }

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

Random-Access Files

- Streams to this point have been read-only or write-only (sequential streams)
  - a file opened using sequential stream is a sequential-access file
  - contents cannot be updated; instead, the entire file must be rewritten
- The RandomAccessFile class allows data to be read from and written to any location in a random-access file
  - The RandomAccessFile constructor includes a mode parameter to open the file as “r” (read-only) or “rw” (read-write).


«interface»
java.io.DataInput
«interface»
java.io.DataOutput
java.io.RandomAccessFile
+RandomAccessFile(file: File, mode: String)
+RandomAccessFile(name: String, mode: String)
+close(): void
+getFilePointer(): long
+length(): long
+read(): int
+read(b: byte[]): int
+read(b: byte[], off: int, len: int) : int
+seek(pos: long): void
+setLength(newLength: long): void
+skipBytes(int n): int
+write(b: byte[]): void
+write(b: byte[], off: int, len: int): void

- A random-access file consists of a sequence of bytes. The file uses a file pointer which points at the current read (or write) location (byte) of the file.
- When the file is opened, the point is set to the beginning of the file. Reading or writing data moves the pointer forward to the next item (based on the size of the item read or written).
- For example, if an int is read using readInt(), the pointer is moved 4 bytes ahead after reading the data.


file pointer (before invoking readInt())
file
byte byte ...... byte byte byte byte byte .... byte byte byte byte
file pointer (after invoking readInt())
file
byte byte ...... byte byte byte byte byte .... byte byte byte byte

- Given a RandomAccessFile **raf**, use raf.seek(position) to move the file pointer to a specified position
  - raf.seek(0) moves to the beginning of the file
  - raf.seek(raf.length()) moves it to the end of the file

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class TestRandomAccessFile {
   public static void main(String[] args) {
      final String FILENAME = "inout.dat";
      try ( // create a random access file
            RandomAccessFile inout = new RandomAccessFile(FILENAME, "rw"); ) {
         // clear the file contents
         inout.setLength(0);

         // write new integers
         for (int i = 0; i < 200; i++)
            inout.writeInt(i);

         // display the current file size
         System.out.println("current file size is " + inout.length());
```

```
            // get the first number
            inout.seek(0); // set file pointer to beginning
            System.out.println("the first number is " + inout.readInt());

            // get the second number
            inout.seek(1 * 4); // move to the second 4-byte int
            System.out.println("the second number is " + inout.readInt());

            // get the last number
            inout.seek(inout.length() - 4);
            System.out.println("the last number is " + inout.readInt());

            // modify the last number
            inout.seek(inout.length() - 4);
            inout.writeInt(555);

            inout.seek(inout.length() - 4);
            System.out.println("the last number is " + inout.readInt());

            // append a new number
            inout.seek(inout.length());
            inout.writeInt(999);

            // display the new file size
            System.out.println("new file size is " + inout.length());

            // get the new eleventh number
            inout.seek(inout.length() - 4);
            System.out.println("the last number is " + inout.readInt());
        } catch (IOException ex) {
            System.err.println("Exception!" + ex);
        }
    }
}
```

Object I/O

- Streams can be created for objects in addition to primitive-type values
- Classes must be **serializable**
  - Most library classes are already serializable
  - Custom classes must be declared to implement the Serializable interface
- Need to catch IOExceptions as usual
- Use try-with-resources to guarantee output files get closed
- When reading objects, **ClassNotFoundException** must also be caught

Writing and Reading Objects

```
import java.io.*;
import java.util.Date;

public class TestObjectStream {
   public static void main(String[] args) {
      final String FILENAME = "objects.dat";
      boolean keepGoing = true;

      // write an object stream to a file
      System.out.println("writing data");
      try (ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream(FILENAME)); ) {
         // write a UTF string, a double, and a Date object
         output.writeUTF("John Smith");
         output.writeDouble(85.5);
         output.writeObject(new Date());
      } catch (IOException ex) {
         System.err.println("Exception! " + ex);
         keepGoing = false;
      }

      // if we encountered an error, stop processing
      if (keepGoing == false)
         System.exit(1);
```

```
        // now read the objects back in
        System.out.println("reading data");
        try (ObjectInputStream input = new ObjectInputStream(new FileInputStream(FILENAME)); ) {
            // Read a UTF string, a double, and a Date object
            String s = input.readUTF();
            double d = input.readDouble();
            Date date = (Date)input.readObject();
            System.out.println("read " + s + ", " + d + ", " + date);
        } catch (IOException ex) {
            System.err.println("Exception! " + ex);
            keepGoing = false;
        } catch (ClassNotFoundException ce) {
            System.err.println("Class not found" + ce);
        }
    }
}
```

Serializing Arrays

- An array is serializable if all its elements are serializable
- writeObject() and readObject() can be used to store and recover an entire array

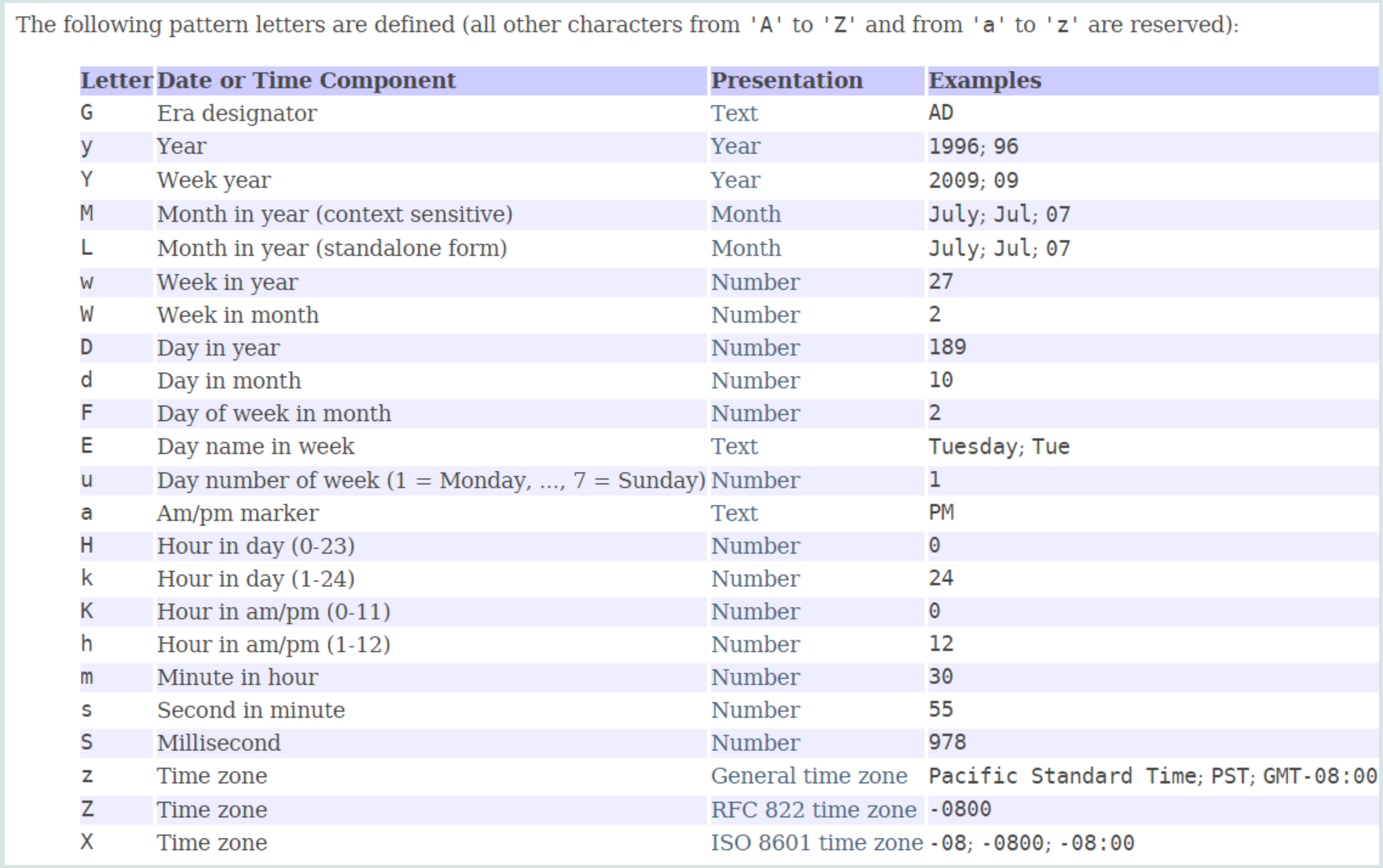
```
import java.io.*;

public class TestObjectStreamForArray {
    public static void main(String[] args) {
        final String FILENAME = "arrays.dat";
        int[] numbers = {1, 2, 3, 4, 5};
        String[] strings = {"John", "Susan", "Kim"};
        boolean keepGoing = true;

        try (  // create an output stream for the file
                ObjectOutputStream output = new ObjectOutputStream(
                        new FileOutputStream(FILENAME, true)); ) {
            // write arrays to stream
            output.writeObject(numbers);
            output.writeObject(strings);
        } catch (IOException ex) {
            System.err.println("Exception! " + ex);
            keepGoing = false;
        }

        // if we encountered an error, stop processing
        if (keepGoing == false)
            System.exit(1);
        try (  // create an input stream
             ObjectInputStream input = new ObjectInputStream(
                    new FileInputStream(FILENAME)); ) {
          // read arrays from stream
          int[] newNumbers = (int[])(input.readObject());
          String[] newStrings = (String[])(input.readObject());

          // display arrays
          for (int i = 0; i < newNumbers.length; i++)
             System.out.print(newNumbers[i] + " ");

          System.out.println();

          for (int i = 0; i < newStrings.length; i++)
             System.out.println(newStrings[i] + " ");
        } catch (IOException ex) {
          System.err.println("Exception! " + ex);
        } catch (ClassNotFoundException ce) {
          System.err.println("Class not found" + ce);
        }
     }
  }
```

Sidebar: Formatting Dates

- java.text.Format
  - abstract base class for formatting locale-sensitive information such as dates, messages, and numbers
- java.text.DateFormat
  - abstract class for date/time formatting subclasses which formats and parses dates or time in a language-independent manner
- java.text.SimpleDateFormat
  - <u>concrete</u> class for formatting and parsing dates in a locale-sensitive manner. It allows for formatting (date → text), parsing (text → date), and normalization.

SimpleDateFormat

- Date and Time Patterns



The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year (context sensitive)	Month	July; Jul; 07
L	Month in year (standalone form)	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

- Constructor: SimpleDateFormat(String pattern)
  - Constructs a SimpleDateFormat using the given pattern and the default date format symbols for the default FORMAT locale
- Method: public Date parse(String text, ParsePosition pos)
  - Parses text from a string to produce a Date
  - Must catch ParseException
- Method: public final String format(Date date) // inherited
  - Formats a Date into a date/time string

```
private static final SimpleDateFormat DATEFORMATTER = new SimpleDateFormat("MM/dd/yyyy");
...
Date newDate = DATEFORMATTER.parse("1/1/2016");
System.out.println(DATEFORMATTER.format(newDate));
```

Date Initialization Using the GregorianCalendar Class

- java.util.GregorianCalendar
  - GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
- Constructor:
  - GregorianCalendar(int year, int month, int dayOfMonth)
  - Constructs a GregorianCalendar with the given date set in the default time zone with the default locale.
- Method: public final Date getTime() // inherited
  - Returns a Date object representing this Calendar's time value (millisecond offset from the Epoch").

```
Date newDate = new GregorianCalendar(2016, 1, 1).getTime();
```

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.GregorianCalendar;

public class TestSimpleDateFormat {
    public static void main(String[] args) {
        final SimpleDateFormat DATEFORMATTER = new SimpleDateFormat("MM/dd/yyyy");
        try {
            Date newDate = DATEFORMATTER.parse("1/1/2016");
            System.out.println(DATEFORMATTER.format(newDate));

            newDate = new GregorianCalendar(2016, 1, 1).getTime();
            System.out.println(newDate);
        } catch (ParseException pe) {
            System.err.println("Exception!" + pe);
        }
    }
}
```