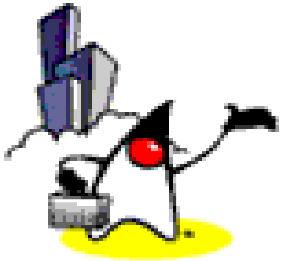
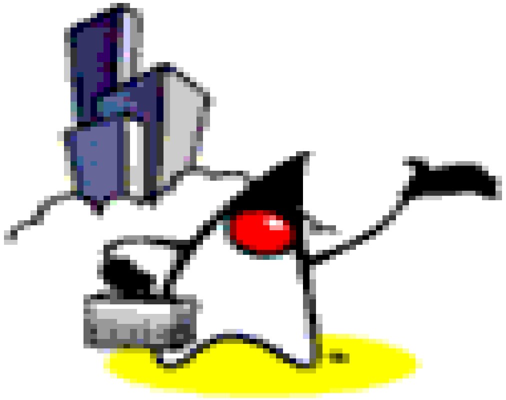
Java I/O Stream



Topics

- What is an I/O stream?
- Types of Streams
- Stream class hierarchy
- Control flow of an I/O operation using Streams
- Byte streams
- Character streams
- Buffered streams
- Standard I/O streams
- Data streams
- Object streams
- File class



What is an I/O Stream?

I/O Streams

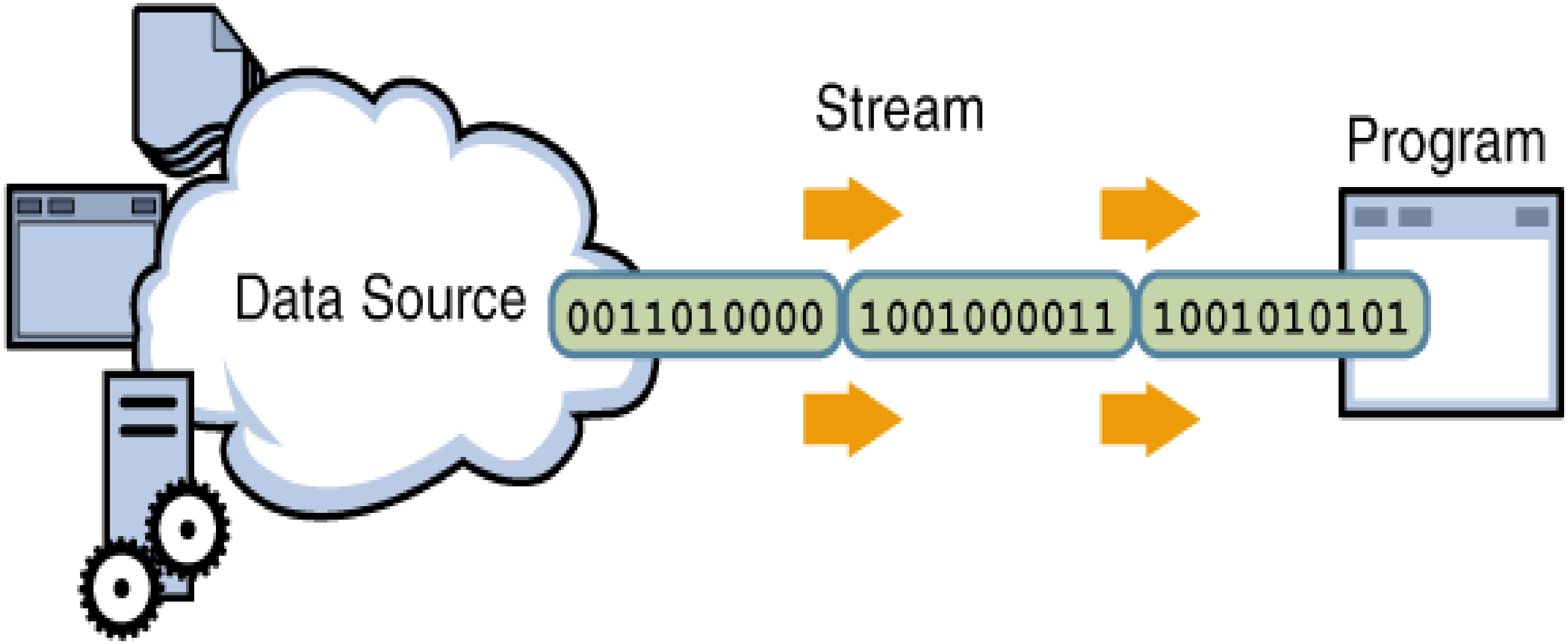
- An I/O Stream represents an input source or an output destination
- A stream can represent many different kinds of sources and destinations
  - disk files, devices, other programs, a network socket, and memory arrays
- Streams support many different kinds of data
  - simple bytes, primitive data types, localized characters, and objects
- Some streams simply pass on data; others manipulate and transform the data in useful ways.

I/O Streams

- No matter how they work internally, all streams present the same simple model to programs that use them
  - A stream is a sequence of data

Input Stream

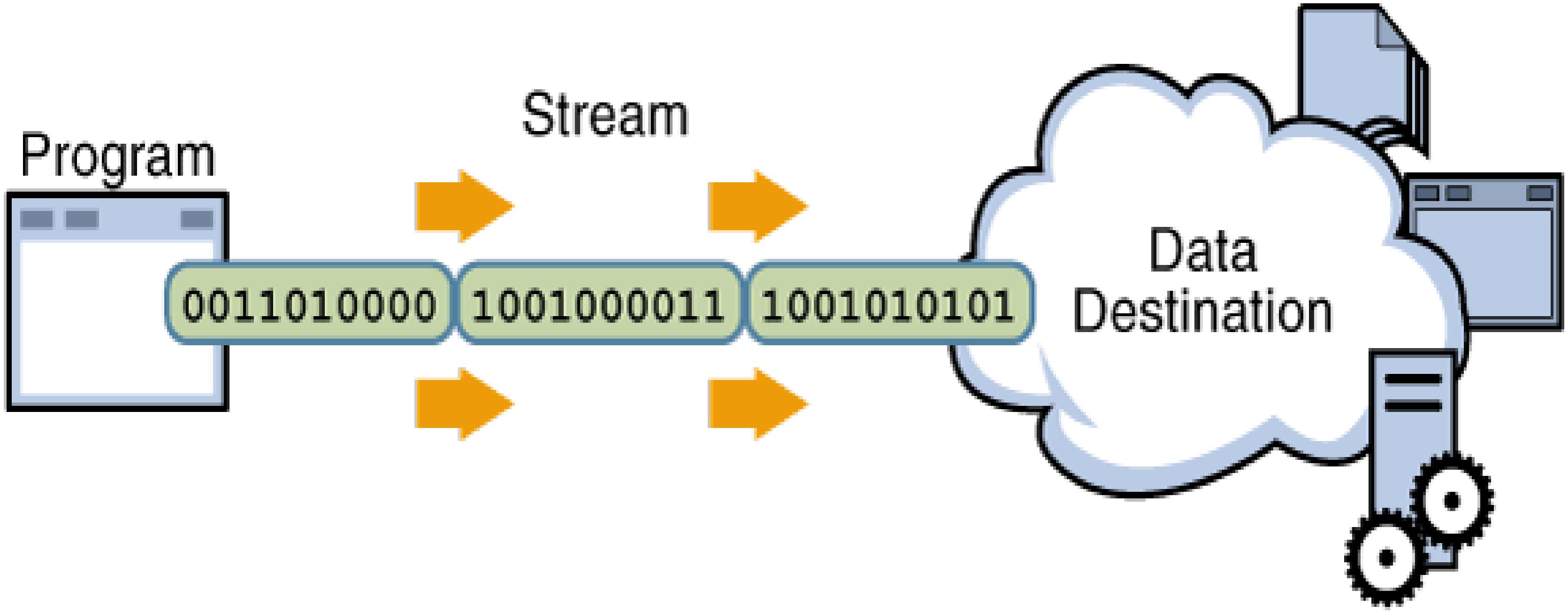
- A program uses an input stream to read data from a source, one item at a time

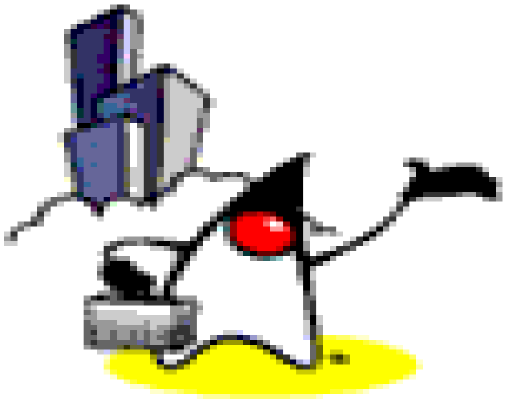


source:java.sun.com

Output Stream

- A program uses an output stream to write data to a destination, one item at time





Types of Streams

General Stream Types

- Character and Byte Streams
  - Character vs. Byte
- Input and Output Streams
  - Based on source or destination
- Node and Filter Streams
  - Whether the data on a stream is manipulated or transformed or not

Character and Byte Streams

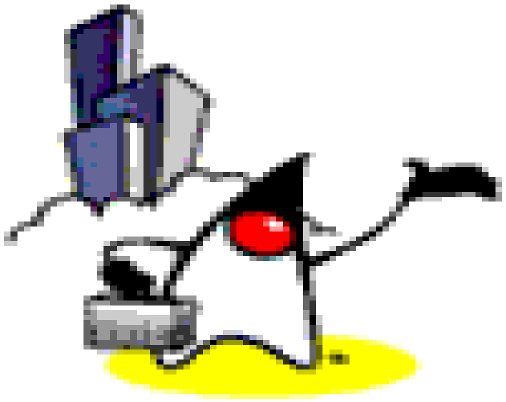
- Byte streams
  - For binary data
  - Root classes for byte streams:
    - The *InputStream* Class
    - The *OutputStream* Class
    - Both classes are *abstract*
- Character streams
  - For Unicode characters
  - Root classes for character streams:
    - The *Reader* class
    - The *Writer* class
    - Both classes are *abstract*

Input and Output Streams

- Input or source streams
  - Can read from these streams
  - Root classes of all input streams:
    - The *InputStream* Class
    - The *Reader* Class
- Output or sink (destination) streams
  - Can write to these streams
  - Root classes of all output streams:
    - The *OutputStream* Class
    - The *Writer* Class

Node and Filter Streams

- Node streams (Data sink stream)
  - Contain the basic functionality of reading or writing from a specific location
  - Types of node streams include files, memory and pipes
- Filter streams (Processing stream)
  - Layered onto node streams between threads or processes
  - For additional functionality- altering or managing data in the stream
- Adding layers to a node stream is called stream chaining



Stream Class Hierarchy

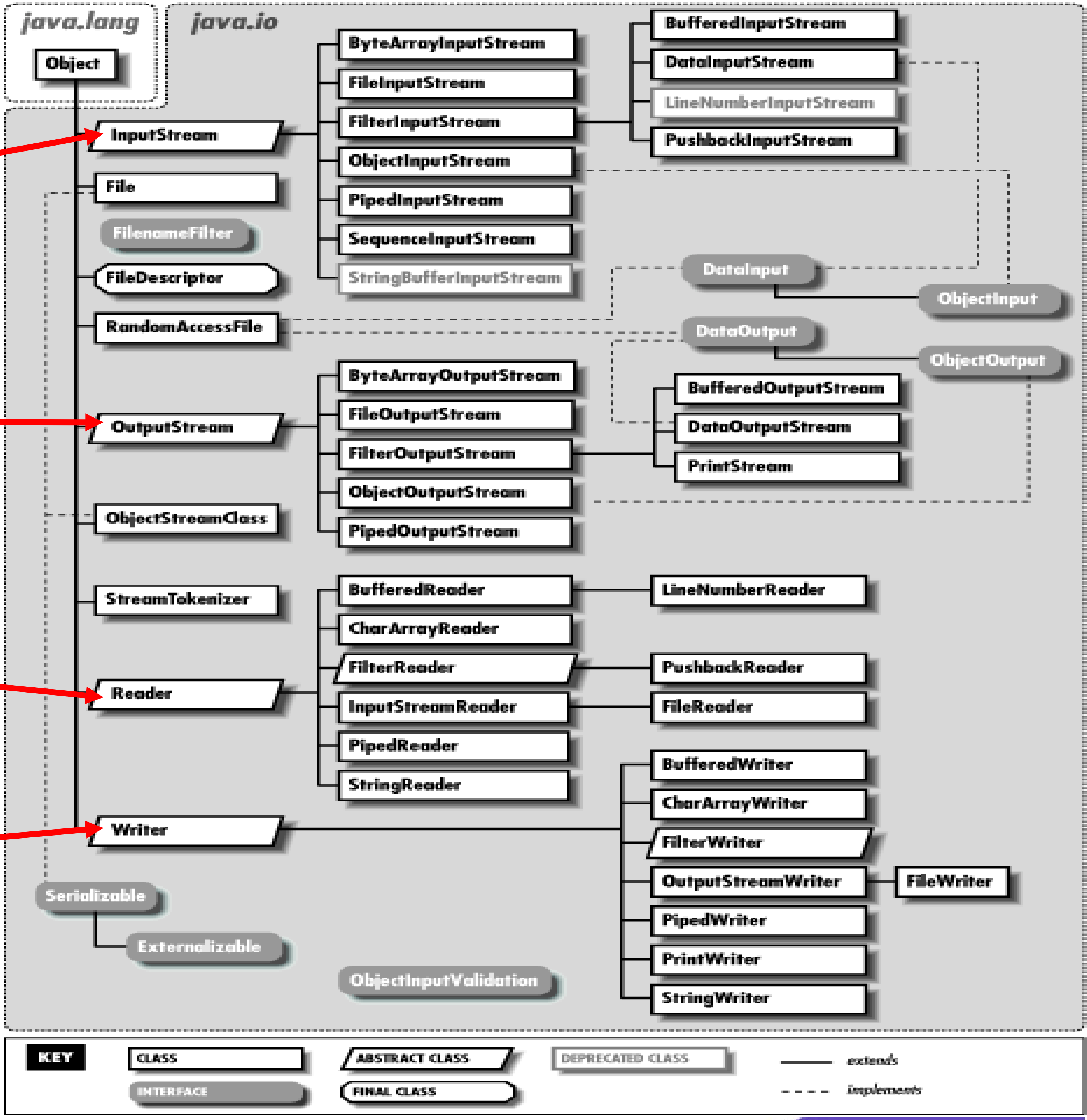
Streams

InputStream

OutputStream

Reader

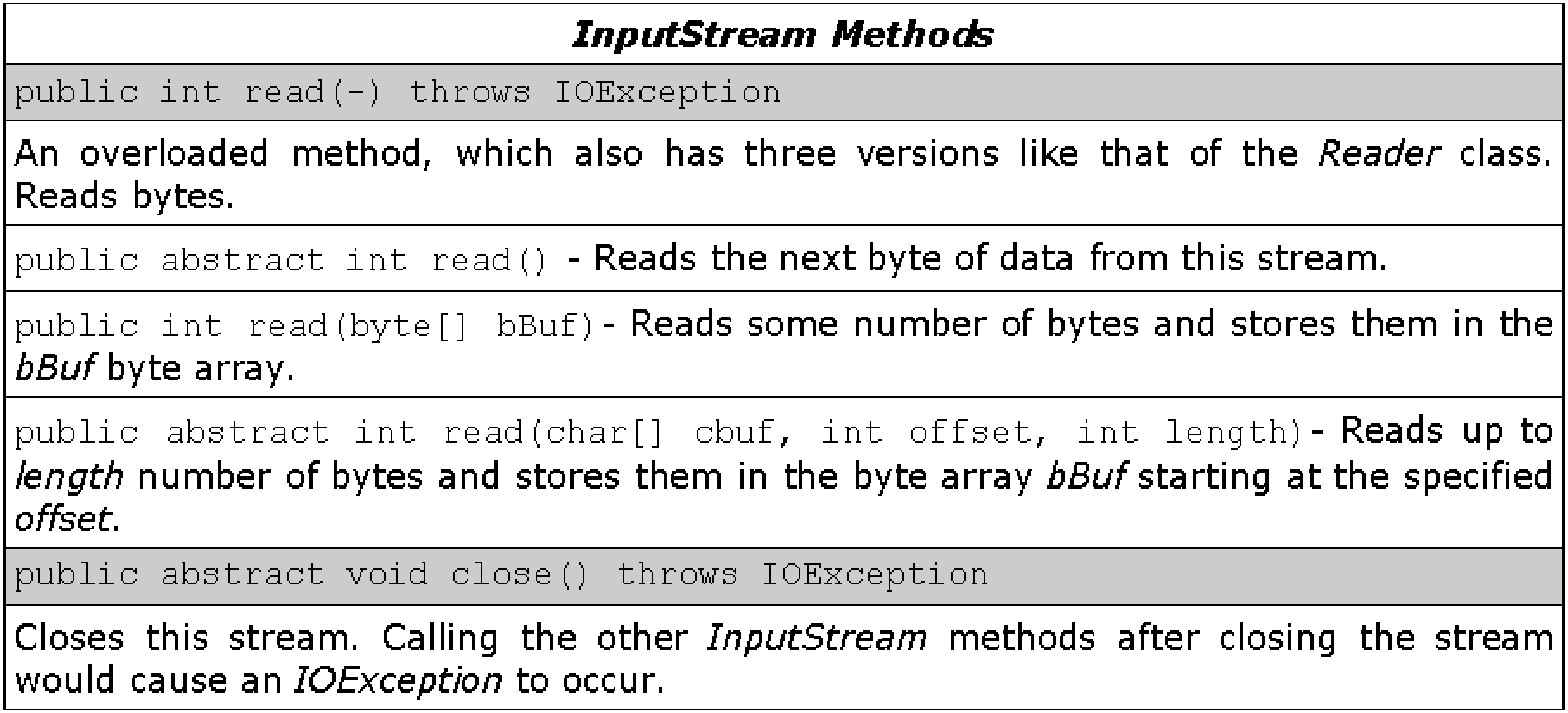
Writer



Abstract Classes

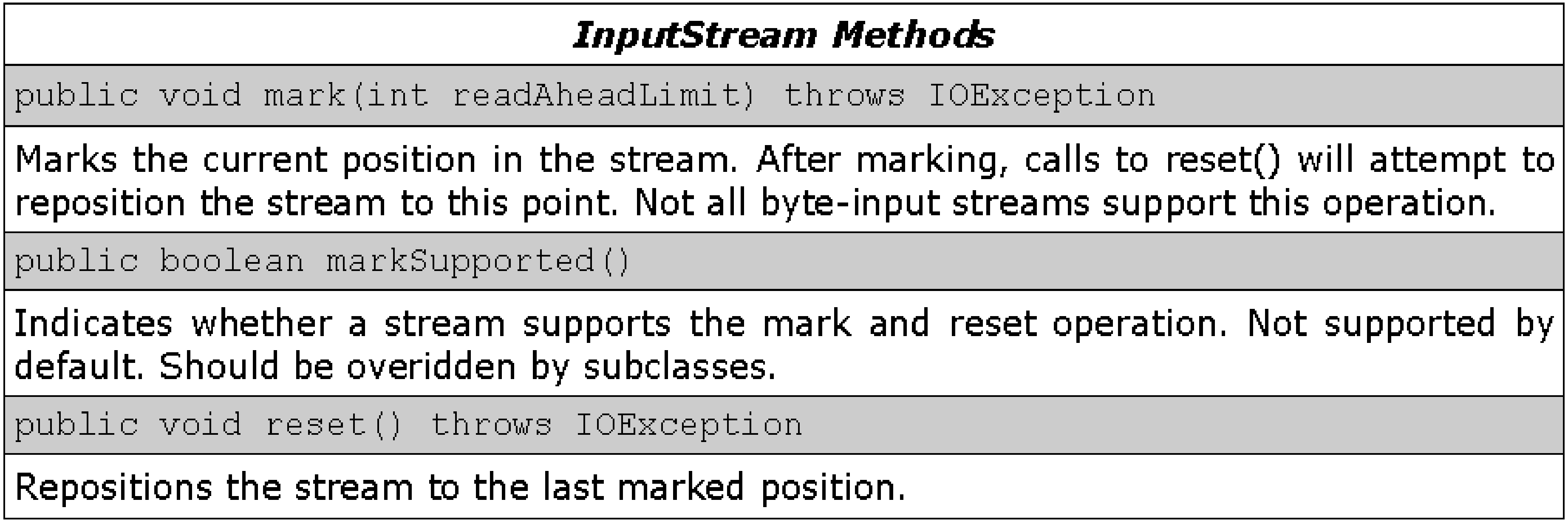
- InputStream & OutputStream
- Reader & Writer

InputStream Abstract Class



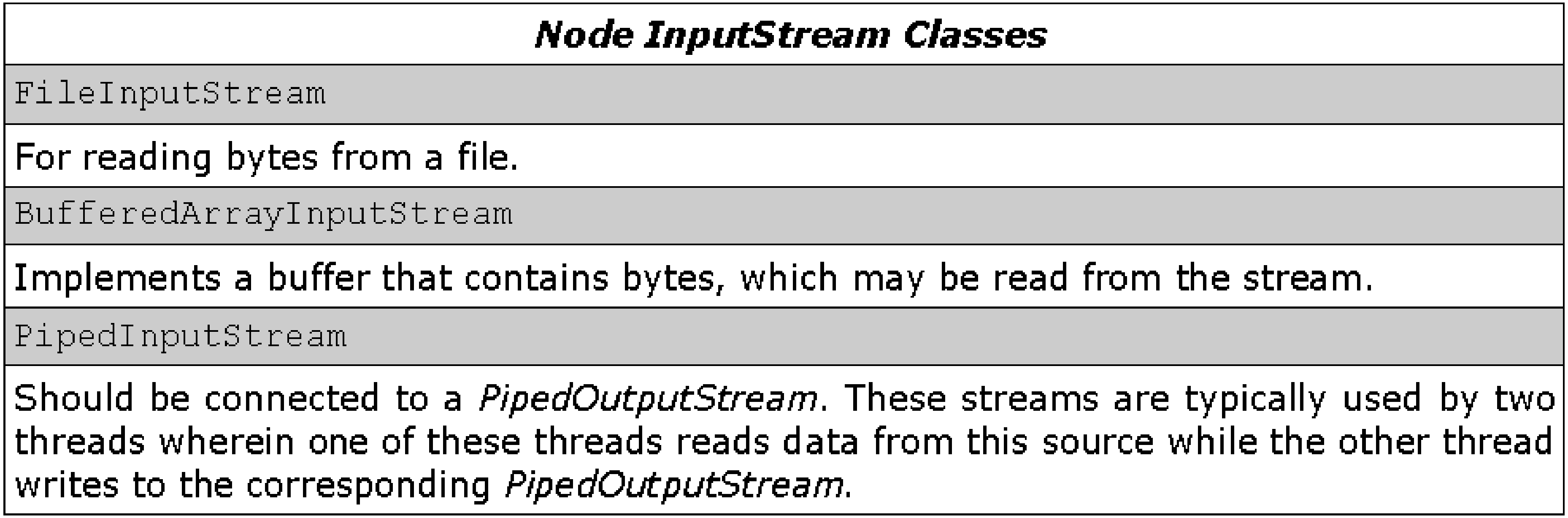
InputStream Methods
`public int read(-) throws IOException`
An overloaded method, which also has three versions like that of the *Reader* class. Reads bytes.
`public abstract int read()` - Reads the next byte of data from this stream.
`public int read(byte[] bBuf)` - Reads some number of bytes and stores them in the *bBuf* byte array.
`public abstract int read(char[] cbuf, int offset, int length)` - Reads up to *length* number of bytes and stores them in the byte array *bBuf* starting at the specified *offset*.
`public abstract void close() throws IOException`
Closes this stream. Calling the other *InputStream* methods after closing the stream would cause an *IOException* to occur.

InputStream Abstract Class



InputStream Methods
`public void mark(int readAheadLimit) throws IOException`
Marks the current position in the stream. After marking, calls to reset() will attempt to reposition the stream to this point. Not all byte-input streams support this operation.
`public boolean markSupported()`
Indicates whether a stream supports the mark and reset operation. Not supported by default. Should be overidden by subclasses.
`public void reset() throws IOException`
Repositions the stream to the last marked position.

Node *InputStream* Classes

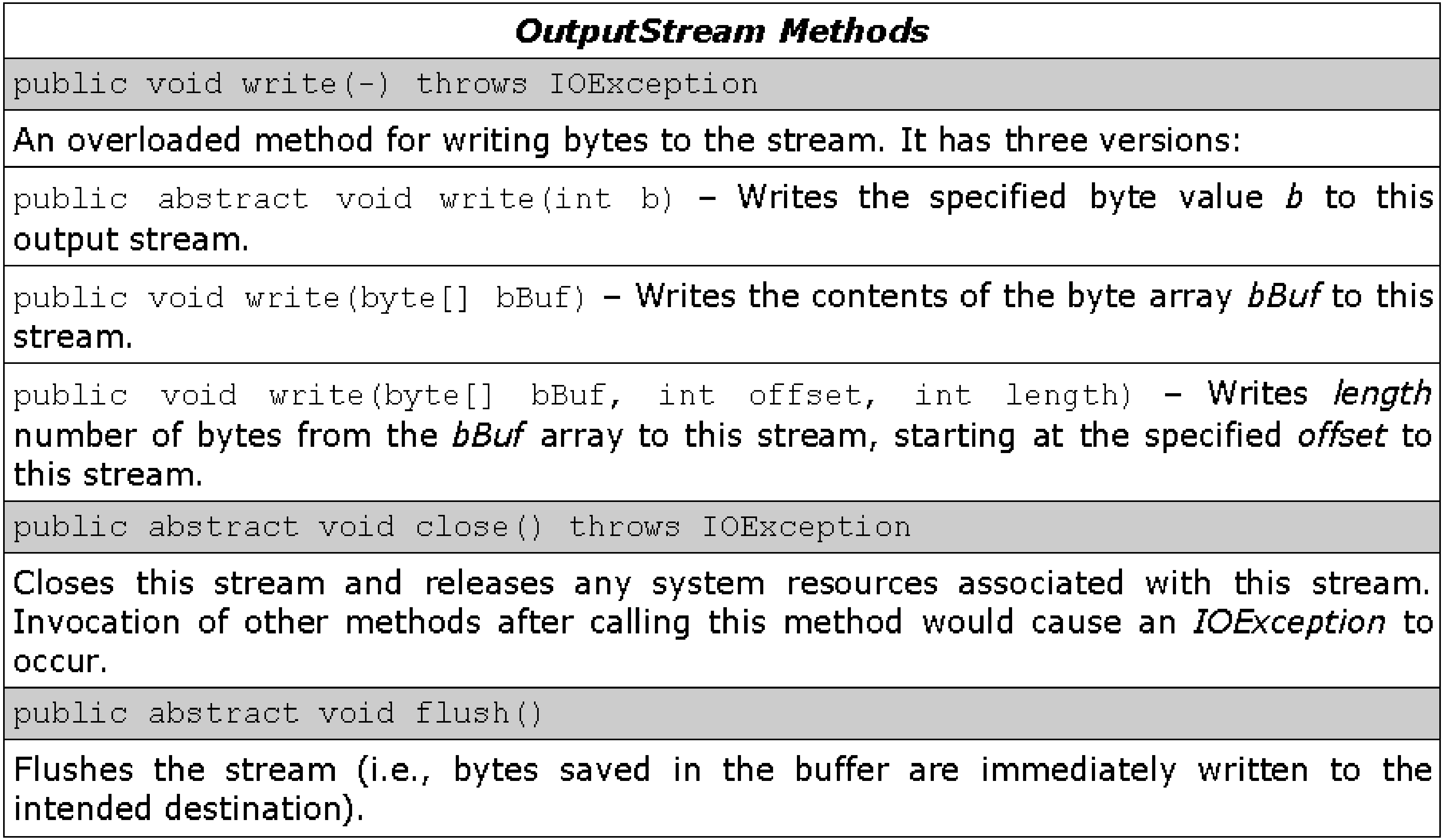


Node InputStream Classes
`FileInputStream`
For reading bytes from a file.
`BufferedArrayInputStream`
Implements a buffer that contains bytes, which may be read from the stream.
`PipedInputStream`
Should be connected to a *PipedOutputStream*. These streams are typically used by two threads wherein one of these threads reads data from this source while the other thread writes to the corresponding *PipedOutputStream*.

Filter *InputStream* Classes

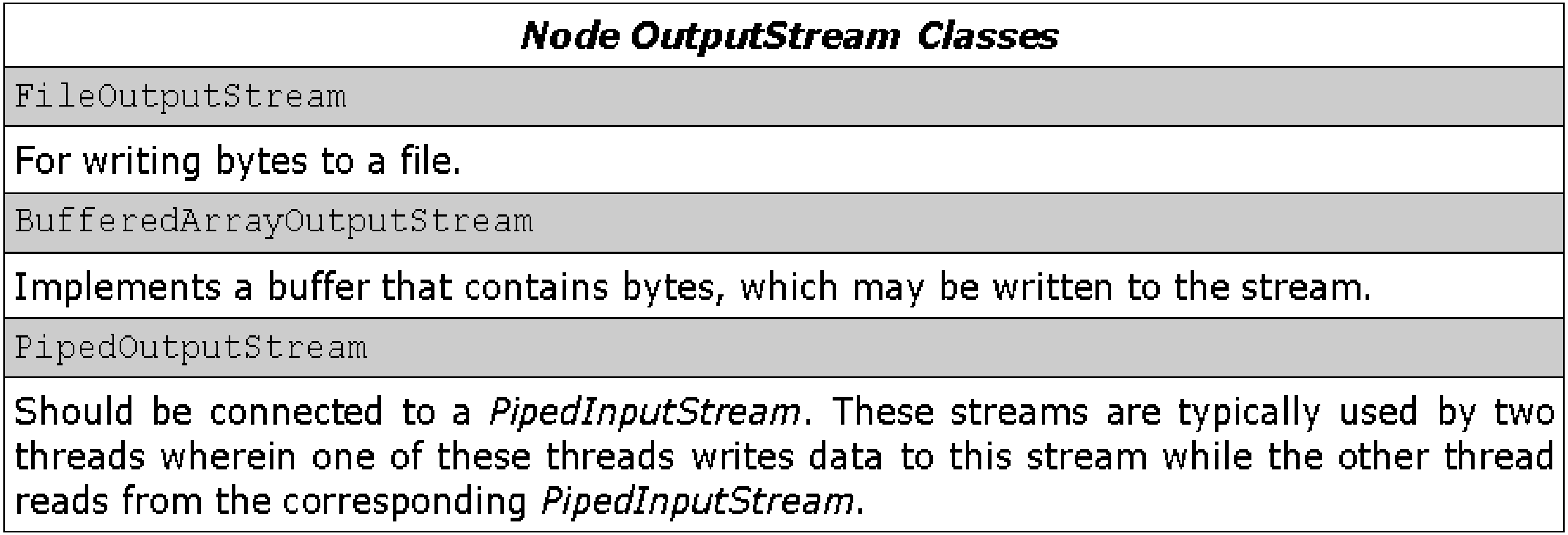
Filter InputStream Classes
`BufferedInputStream`
A subclass of *FilterInputStream* that allows buffering of input in order to provide for the efficient reading of bytes.
`FilterInputStream`
For reading filtered byte streams, which may transform the basic source of data along the way and provide additional functionalities.
`ObjectInputStream`
Used for object serialization. Deserializes objects and primitive data previously written using an *ObjectOutputStream*.
`DataInputStream`
A subclass of *FilterInputStream* that lets an application read Java primitive data from an underlying input stream in a machine-independent way.
`LineNumberInputStream`
A subclass of *FilterInputStream* that allows tracking of the current line number.
`PushbackInputStream`
A subclass of the *FilterInputStream* class that allows bytes to be pushed back or unread into the stream.

OutputStream Abstract Class



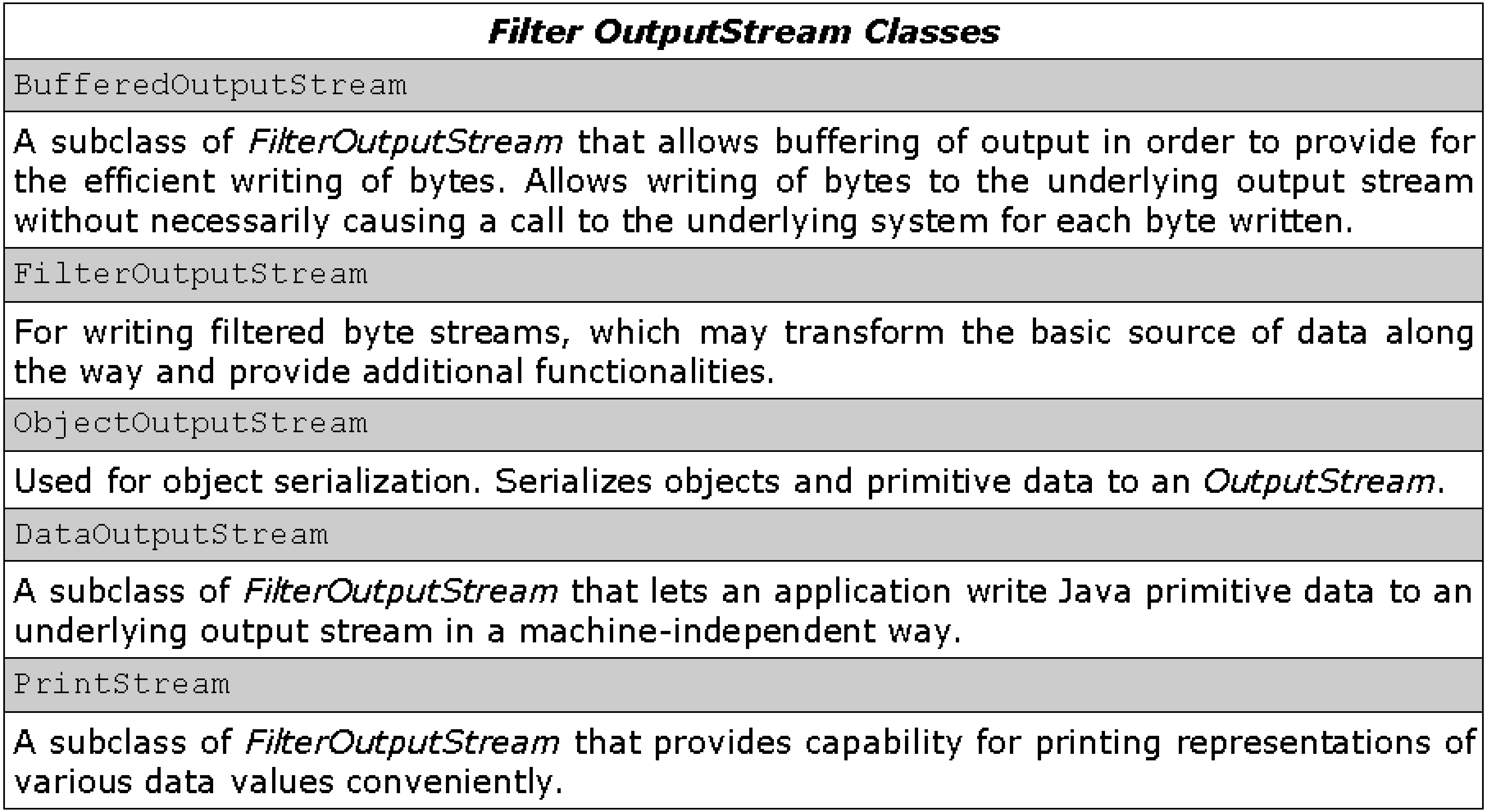
OutputStream Methods
`public void write(-) throws IOException`
An overloaded method for writing bytes to the stream. It has three versions:
`public abstract void write(int b)` – Writes the specified byte value *b* to this output stream.
`public void write(byte[] bBuf)` – Writes the contents of the byte array *bBuf* to this stream.
`public void write(byte[] bBuf, int offset, int length)` – Writes *length* number of bytes from the *bBuf* array to this stream, starting at the specified *offset* to this stream.
`public abstract void close() throws IOException`
Closes this stream and releases any system resources associated with this stream. Invocation of other methods after calling this method would cause an *IOException* to occur.
`public abstract void flush()`
Flushes the stream (i.e., bytes saved in the buffer are immediately written to the intended destination).

Node *OutputStream* Classes



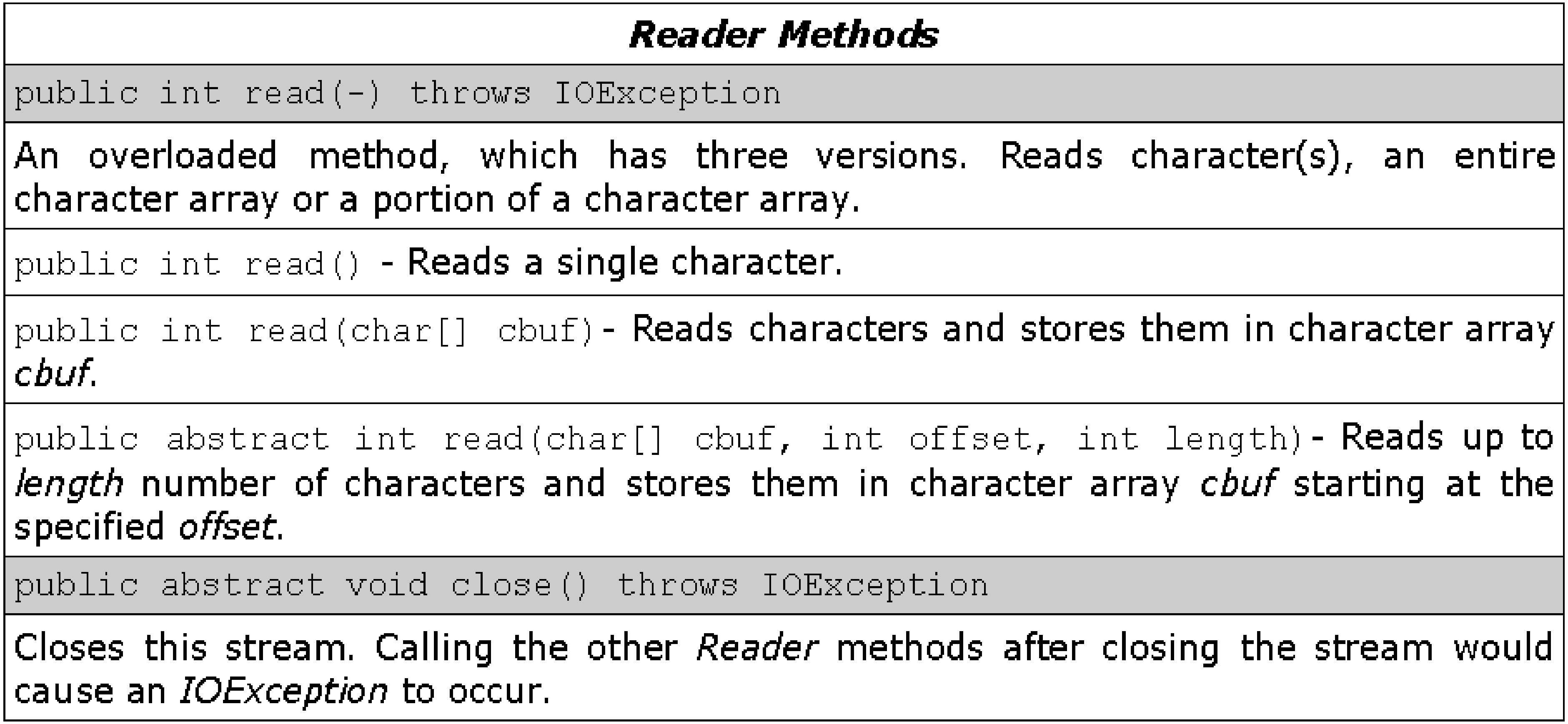
Node OutputStream Classes
`FileOutputStream`
For writing bytes to a file.
`BufferedArrayOutputStream`
Implements a buffer that contains bytes, which may be written to the stream.
`PipedOutputStream`
Should be connected to a *PipedInputStream*. These streams are typically used by two threads wherein one of these threads writes data to this stream while the other thread reads from the corresponding *PipedInputStream*.

Filter *OutputStream* Classes



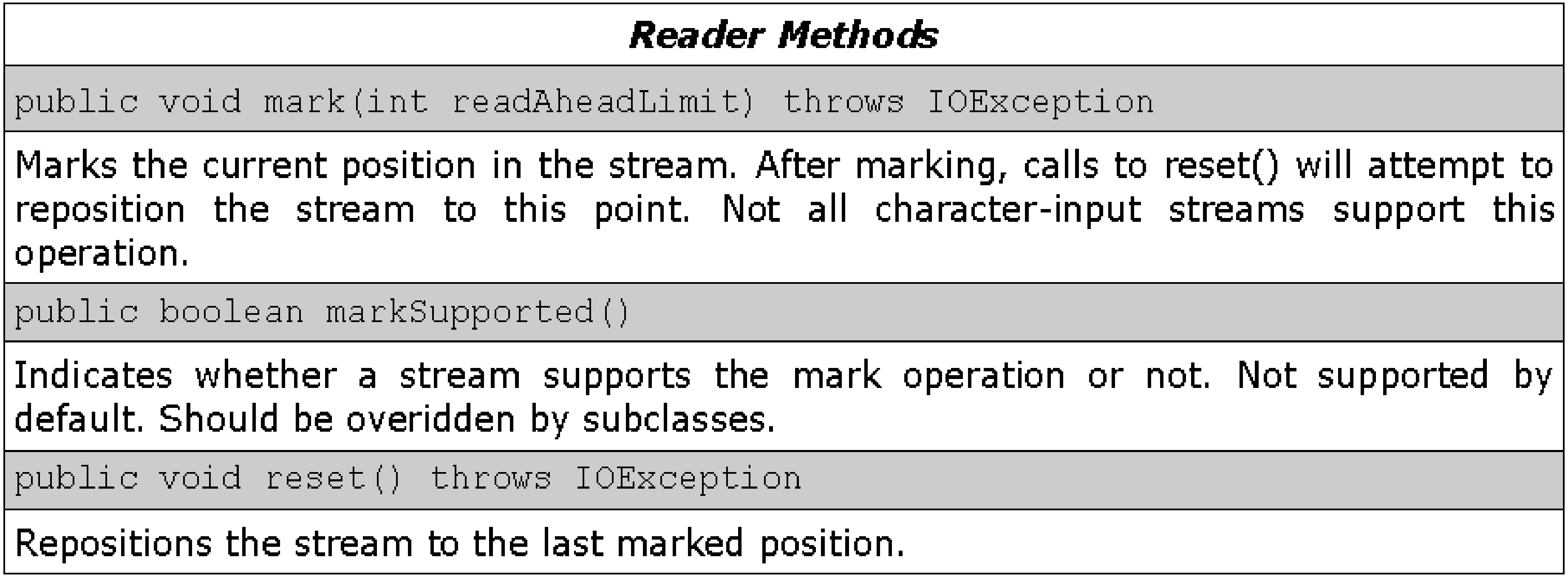
Filter OutputStream Classes
`BufferedOutputStream`
A subclass of *FilterOutputStream* that allows buffering of output in order to provide for the efficient writing of bytes. Allows writing of bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.
`FilterOutputStream`
For writing filtered byte streams, which may transform the basic source of data along the way and provide additional functionalities.
`ObjectOutputStream`
Used for object serialization. Serializes objects and primitive data to an *OutputStream*.
`DataOutputStream`
A subclass of *FilterOutputStream* that lets an application write Java primitive data to an underlying output stream in a machine-independent way.
`PrintStream`
A subclass of *FilterOutputStream* that provides capability for printing representations of various data values conveniently.

The *Reader* Class: Methods



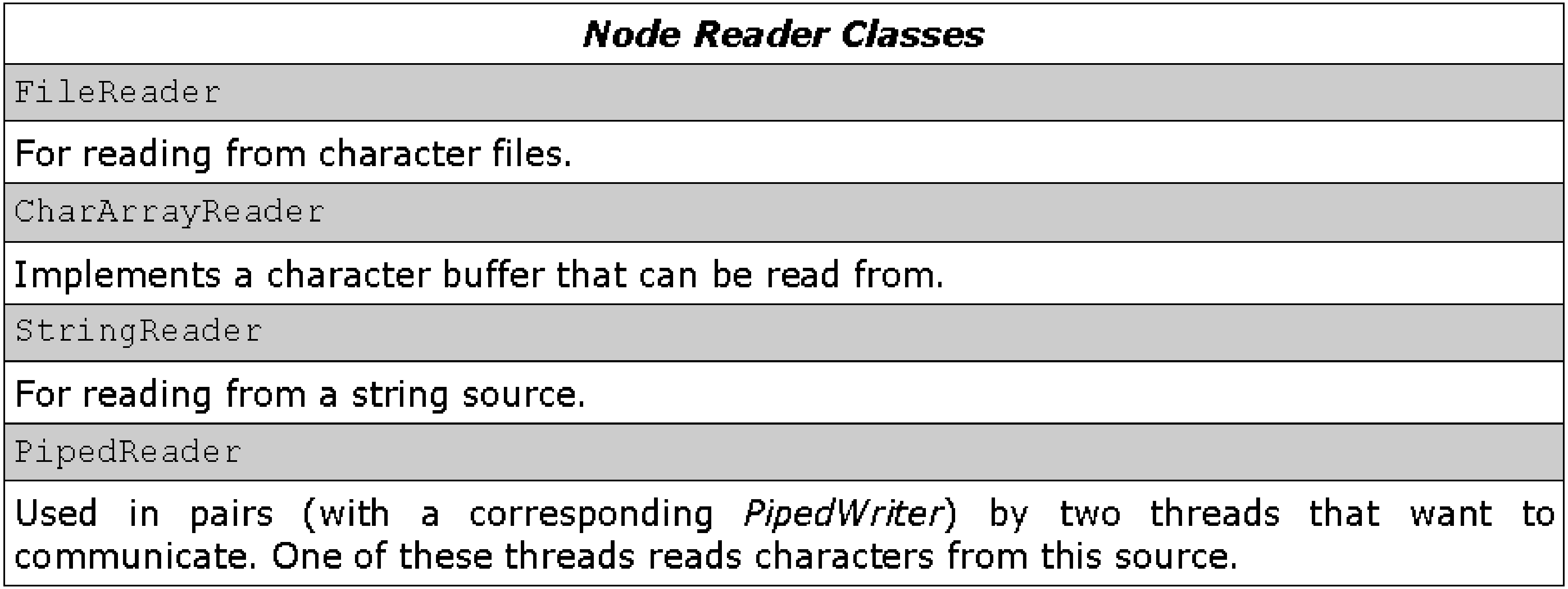
Reader Methods
`public int read(-) throws IOException`
An overloaded method, which has three versions. Reads character(s), an entire character array or a portion of a character array.
`public int read()` - Reads a single character.
`public int read(char[] cbuf)`- Reads characters and stores them in character array *cbuf*.
`public abstract int read(char[] cbuf, int offset, int length)`- Reads up to *length* number of characters and stores them in character array *cbuf* starting at the specified *offset*.
`public abstract void close() throws IOException`
Closes this stream. Calling the other *Reader* methods after closing the stream would cause an *IOException* to occur.

The *Reader* Class: Methods



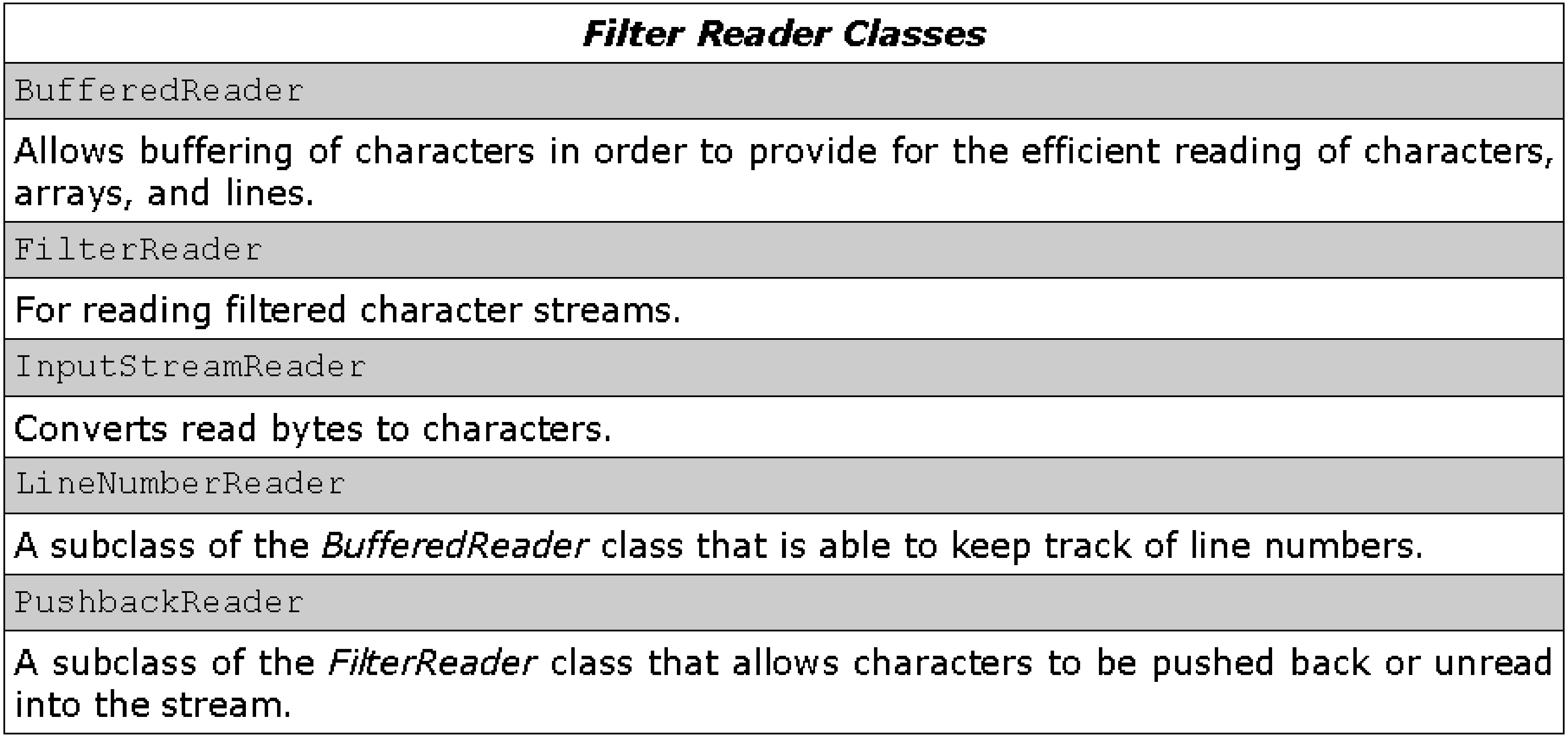
Reader Methods
`public void mark(int readAheadLimit) throws IOException`
Marks the current position in the stream. After marking, calls to reset() will attempt to reposition the stream to this point. Not all character-input streams support this operation.
`public boolean markSupported()`
Indicates whether a stream supports the mark operation or not. Not supported by default. Should be overidden by subclasses.
`public void reset() throws IOException`
Repositions the stream to the last marked position.

Node *Reader* Classes



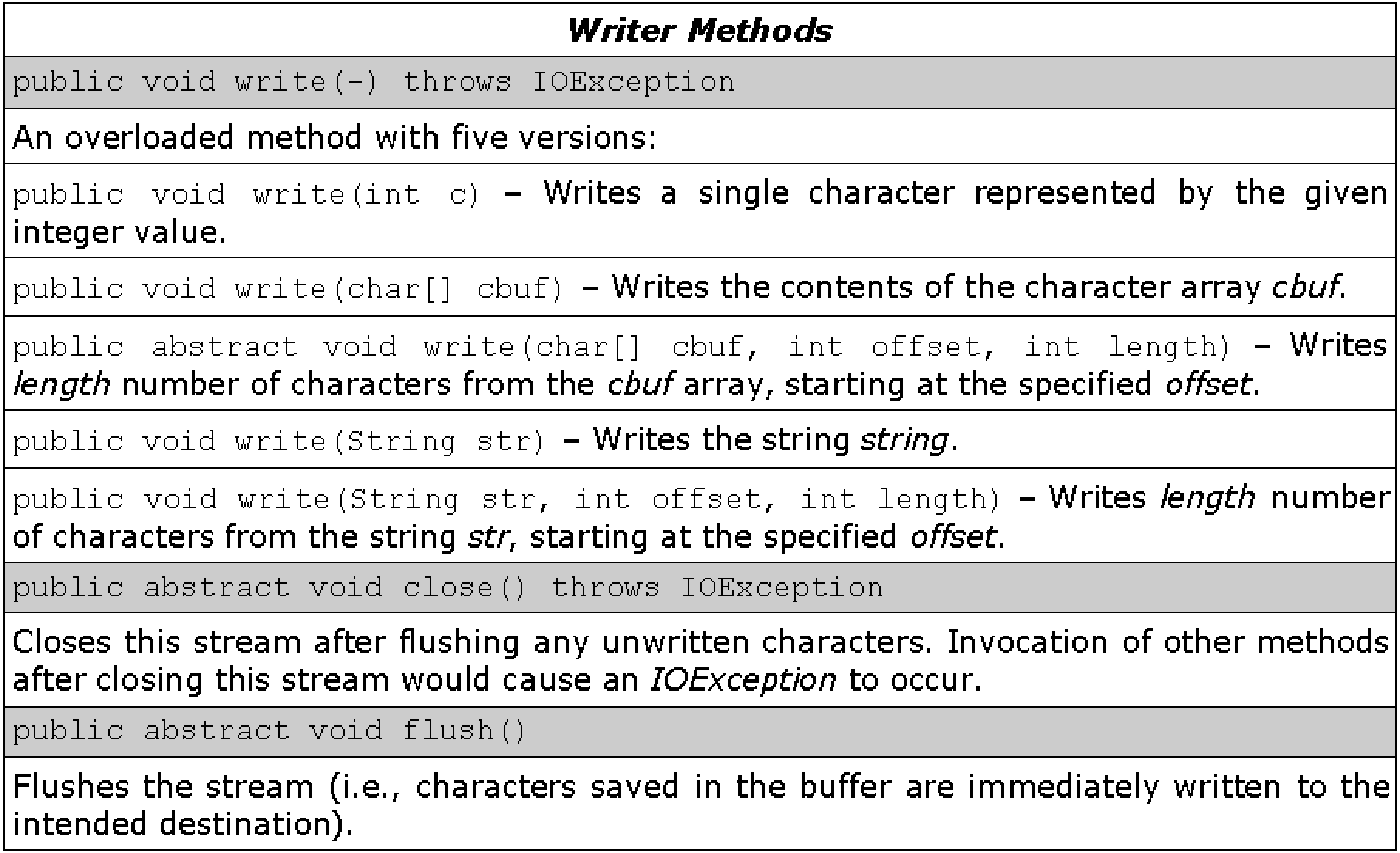
Node Reader Classes
`FileReader`
For reading from character files.
`CharArrayReader`
Implements a character buffer that can be read from.
`StringReader`
For reading from a string source.
`PipedReader`
Used in pairs (with a corresponding *PipedWriter*) by two threads that want to communicate. One of these threads reads characters from this source.

Filter *Reader* Classes



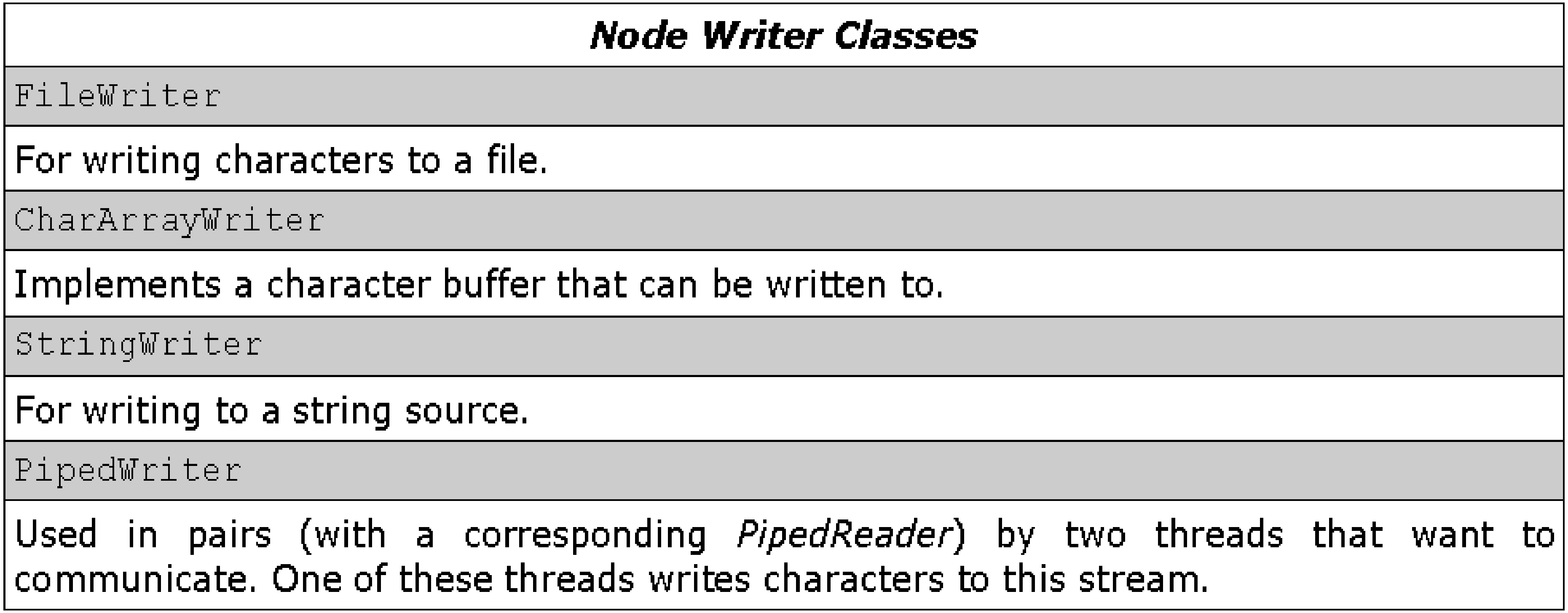
Filter Reader Classes
`BufferedReader`
Allows buffering of characters in order to provide for the efficient reading of characters, arrays, and lines.
`FilterReader`
For reading filtered character streams.
`InputStreamReader`
Converts read bytes to characters.
`LineNumberReader`
A subclass of the *BufferedReader* class that is able to keep track of line numbers.
`PushbackReader`
A subclass of the *FilterReader* class that allows characters to be pushed back or unread into the stream.

The *Writer* Class: Methods



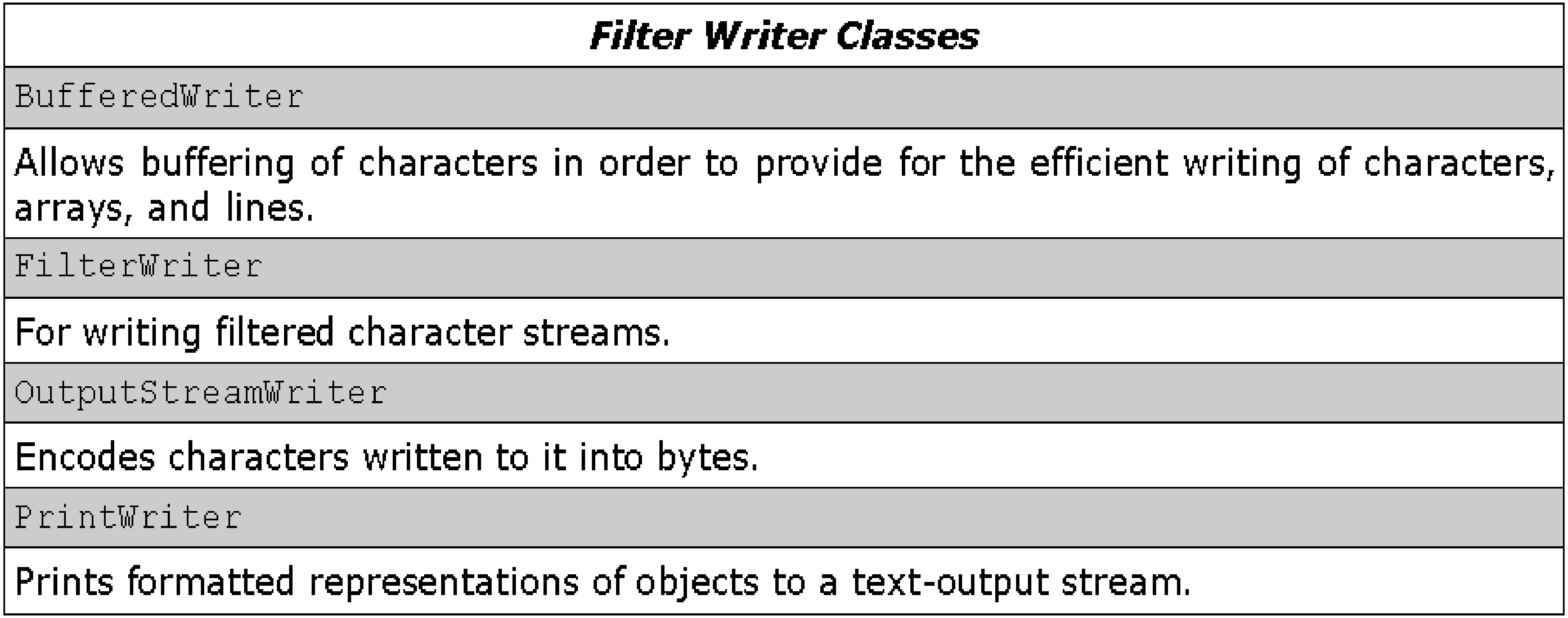
Writer Methods
`public void write(-) throws IOException`
An overloaded method with five versions:
`public void write(int c)` – Writes a single character represented by the given integer value.
`public void write(char[] cbuf)` – Writes the contents of the character array *cbuf*.
`public abstract void write(char[] cbuf, int offset, int length)` – Writes *length* number of characters from the *cbuf* array, starting at the specified *offset*.
`public void write(String str)` – Writes the string *string*.
`public void write(String str, int offset, int length)` – Writes *length* number of characters from the string *str*, starting at the specified *offset*.
`public abstract void close() throws IOException`
Closes this stream after flushing any unwritten characters. Invocation of other methods after closing this stream would cause an *IOException* to occur.
`public abstract void flush()`
Flushes the stream (i.e., characters saved in the buffer are immediately written to the intended destination).

Node *Writer* Classes

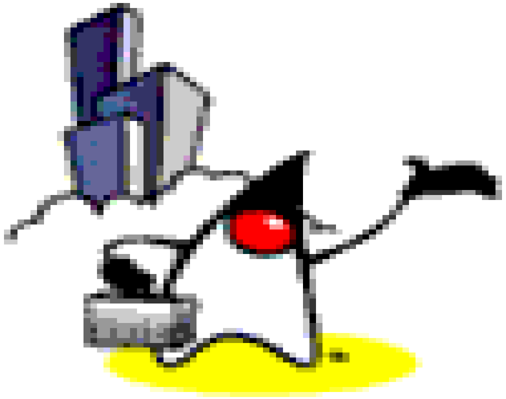


Node Writer Classes
`FileWriter`
For writing characters to a file.
`CharArrayWriter`
Implements a character buffer that can be written to.
`StringWriter`
For writing to a string source.
`PipedWriter`
Used in pairs (with a corresponding *PipedReader*) by two threads that want to communicate. One of these threads writes characters to this stream.

Filter *Writer* Classes



Filter Writer Classes
`BufferedWriter`
Allows buffering of characters in order to provide for the efficient writing of characters, arrays, and lines.
`FilterWriter`
For writing filtered character streams.
`OutputStreamWriter`
Encodes characters written to it into bytes.
`PrintWriter`
Prints formatted representations of objects to a text-output stream.



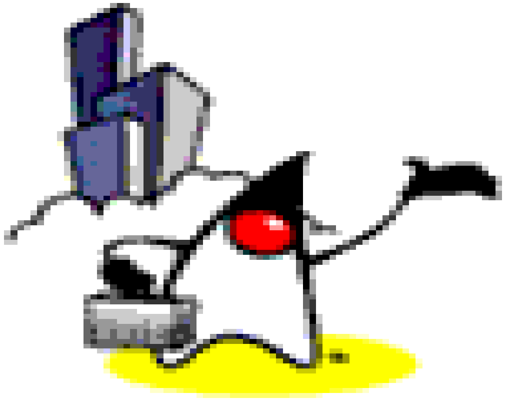
Control Flow of I/O Operation using Streams

Control Flow of an I/O operation

Create a stream object and associate it with a data-source (data-destination)

Give the stream object the desired functionality through stream chaining

```
while (there is more information)
    read(write) next data from(to) the stream
close the stream
```



Byte Stream

Byte Stream

- 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
  - FileInputStream and FileOutputStream
- They are used in much the same way; they differ mainly in the way they are constructed

When Not to Use Byte Streams?

- Byte Stream represents a kind of low-level I/O that you should avoid
  - If the data contains character data, the best approach is to use character streams
  - There are also streams for more complicated data types
- Byte streams should only be used for the most primitive I/O
- All other streams are based on byte stream

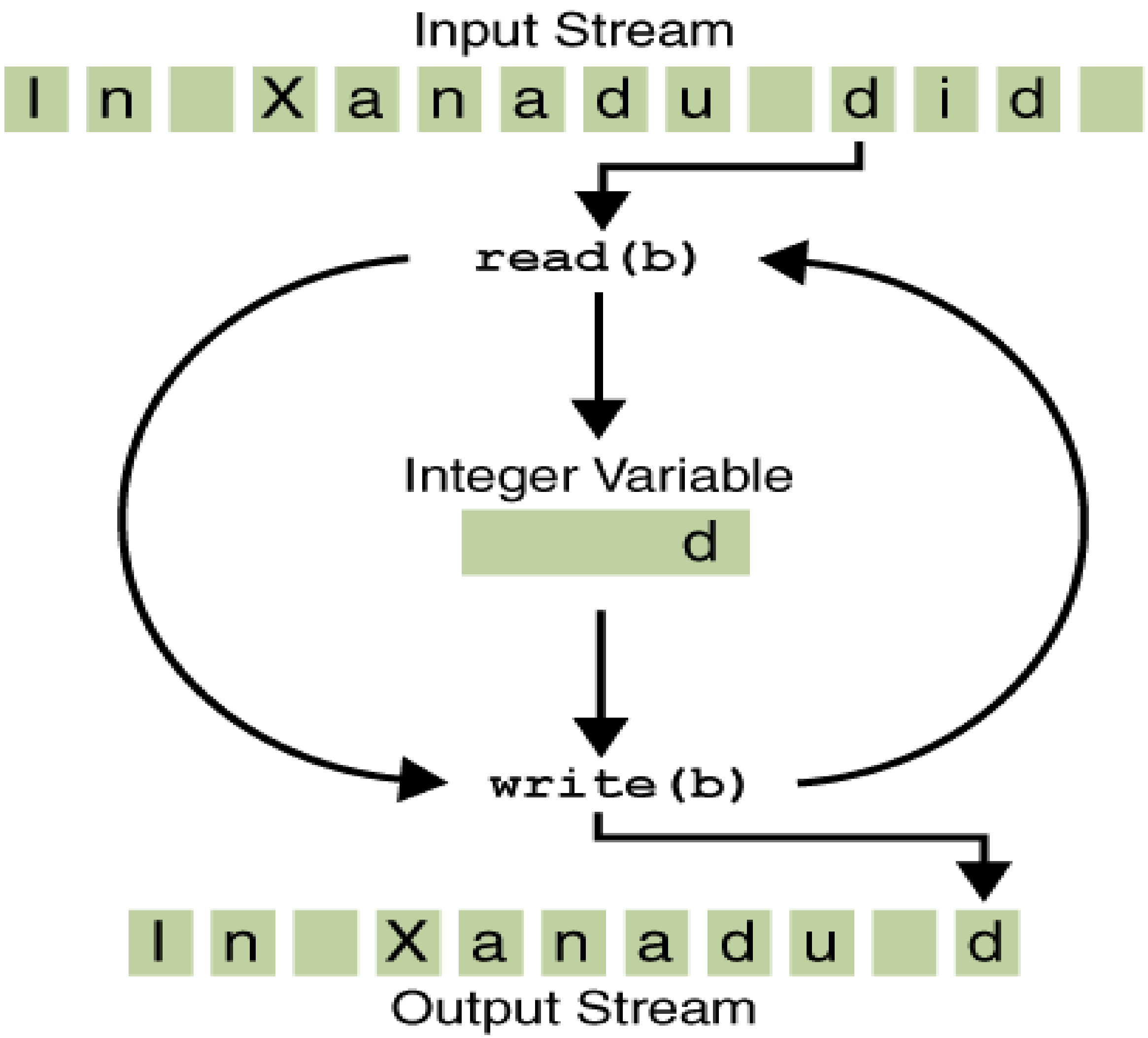
Example: FileInputStream & FileOutputStream

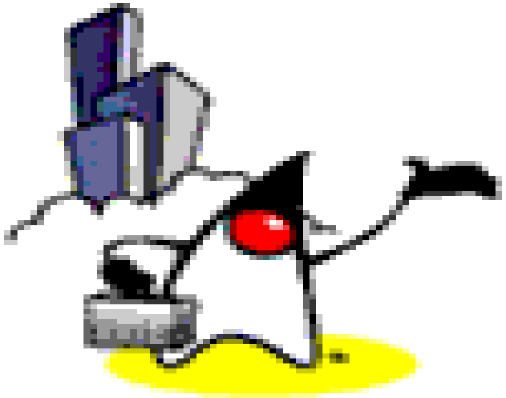
```
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }
        // More code
```

Example: FileInputStream & FileOutputStream

```
        finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Simple Byte Stream input and output





Character Stream

Character Stream

- The Java platform stores character values using Unicode conventions
- Character stream I/O automatically translates this internal format to and from the local character set.
  - In Western locales, the local character set is usually an 8-bit superset of ASCII.
- 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.

Character Stream

- For most applications, I/O with character streams is no more complicated than I/O with byte streams.
  - Input and output done with stream classes automatically translates to and from the local character set.
  - A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.
  - If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues.
  - Later, if internationalization becomes a priority, your program can be adapted without extensive recoding.

Example: FileReader & FileWriter

```
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);
            }
        }
        // More code
```

Example: FileReader & FileWriter

```
        finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Character Stream and Byte Stream

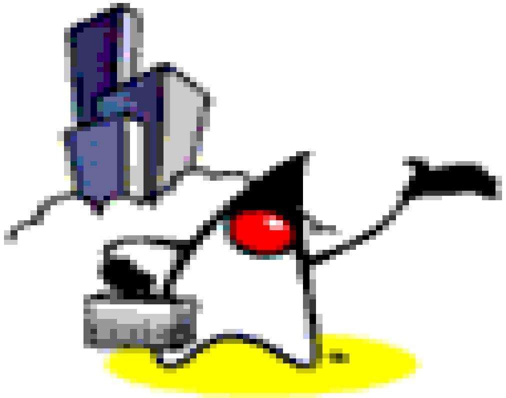
- Character streams are often "wrappers" for byte streams
- The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes.
  - FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream

Line-Oriented I/O

- Character I/O usually occurs in bigger units than single characters
  - One common unit is the line: a string of characters with a line terminator at the end
  - A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n").

Example: Line-oriented I/O

```
File inputFile = new File("farrago.txt");
File outputFile = new File("outagain.txt");

FileReader in = new FileReader(inputFile);
FileWriter out = new FileWriter(outputFile);

BufferedReader   inputStream = new BufferedReader(in);
PrintWriter    outputStream = new PrintWriter(out);

String l;
while ((l = inputStream.readLine()) != null) {
    System.out.println(l);
    outputStream.println(l);
}

in.close();
out.close();
```



Buffered Stream

Why Buffered Streams?

- An unbuffered I/O 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.

How to create Buffered Streams?

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

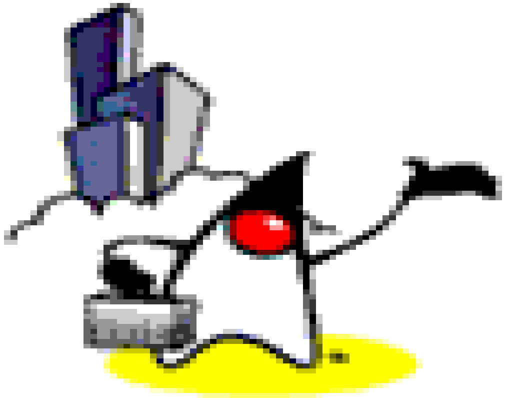
```
inputStream =
    new BufferedReader(new FileReader("xanadu.txt"));
outputStream =
    new BufferedWriter(new FileWriter("characteroutput.txt"));
```

Buffered Stream Classes

- *BufferedInputStream* and *BufferedOutputStream* create buffered byte streams
- *BufferedReader* and *BufferedWriter* create buffered character streams

Flushing Buffered Streams

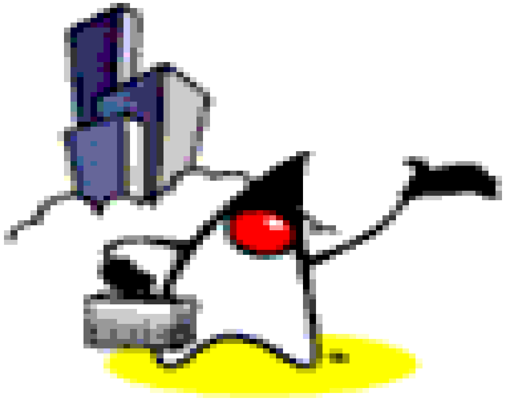
- It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer.
- Some buffered output classes support autoflush, specified by an optional constructor argument.
  - When autoflush is enabled, certain key events cause the buffer to be flushed
  - For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format.
- To flush a stream manually, invoke its flush method
  - The flush method is valid on any output stream, but has no effect unless the stream is buffered.



Standard Streams

Standard Streams on Java Platform

- Three standard streams
  - Standard Input, accessed through *System.in*
  - Standard Output, accessed through *System.out*
  - Standard Error, accessed through *System.err*
- These objects are defined automatically and do not need to be opened
- System.out and System.err are defined as PrintStream objects



Data Streams

Data Streams

- Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values
- All data streams implement either the *DataInput* interface or the *DataOutput* interface
- DataInputStream and DataOutputStream are most widely-used implementations of these interfaces

DataOutputStream

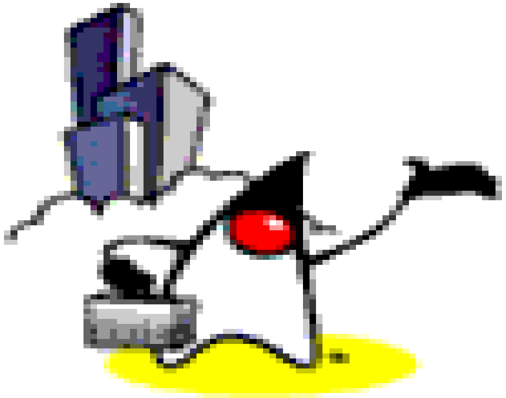
- *DataOutputStream* can only be created as a wrapper for an existing byte stream object

```
out = new DataOutputStream(
        new BufferedOutputStream(
        new FileOutputStream(dataFile)));
for (int i = 0; i < prices.length; i ++) {
   out.writeDouble(prices[i]);
   out.writeInt(units[i]);
   out.writeUTF(descs[i]);
}
```

DataInputStream

- Like *DataOutputStream*, *DataInputStream* must be constructed as a wrapper for a byte stream
- End-of-file condition is detected by catching *EOFException*, instead of testing for an invalid return value

```
in = new DataInputStream(
        new BufferedInputStream(
        new FileInputStream(dataFile)));

try{
        double price = in.readDouble();
        int unit = in.readInt();
        String desc = in.readUTF();
}  catch (EOFException e){
}
```



Object Streams

Object Streams

- Object streams support I/O of objects
  - Like Data streams support I/O of primitive data types
  - The object has to be *Serializable* type
- The object stream classes are ObjectInputStream and ObjectOutputStream
  - These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput
  - An object stream can contain a mixture of primitive and object values

Input and Output of Complex Object

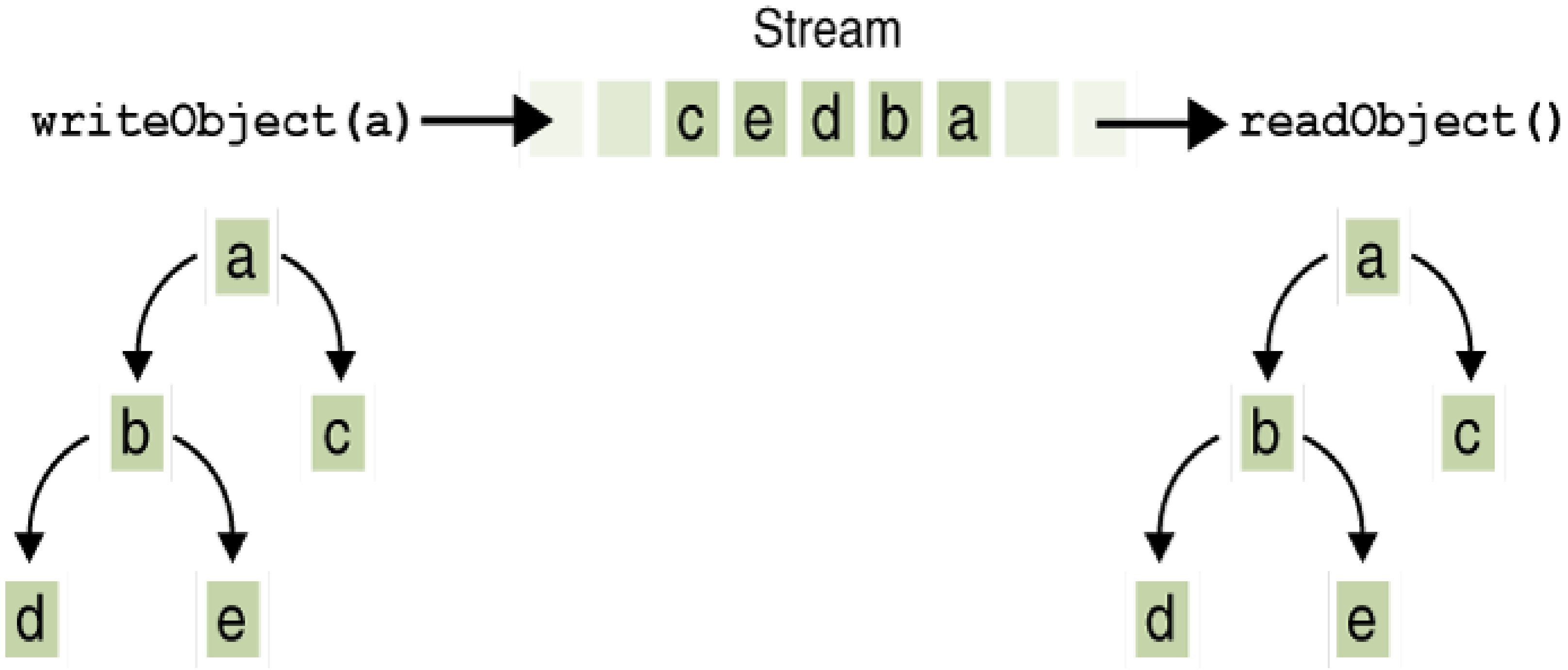
- The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic
  - This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects.
- If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to.
  - These additional objects might have their own references, and so on.

WriteObject

- The writeObject traverses the entire web of object references and writes all objects in that web onto the stream
- A single invocation of writeObject can cause a large number of objects to be written to the stream.

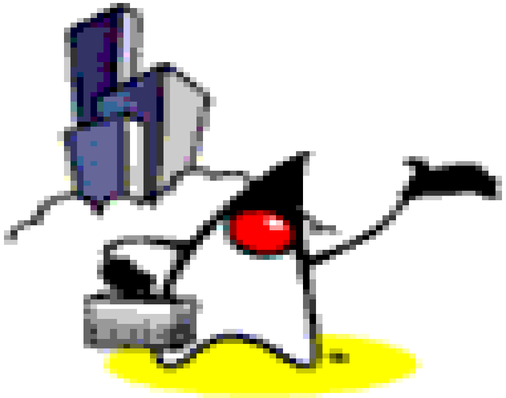
I/O of multiple referred-to objects

- Object a contains references to objects b and c, while b contains references to d and e



I/O of multiple referred-to objects

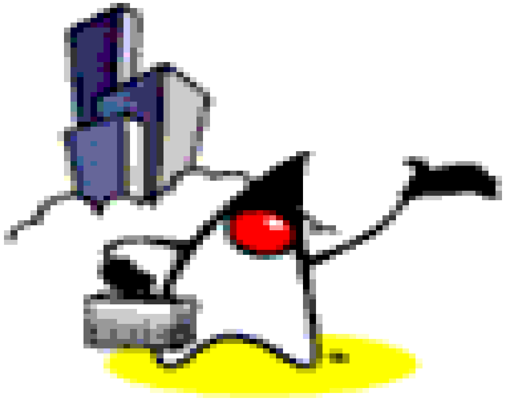
- Invoking writeObject(a) writes not just a, but all the objects necessary to reconstitute a, so the other four objects in this web are written also
- When a is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.



Closing Streams

Always Close Streams

- Closing a stream when it's no longer needed is very important — so important that your program should use a finally block to guarantee that both streams will be closed even if an error occurs
  - This practice helps avoid serious resource leaks.



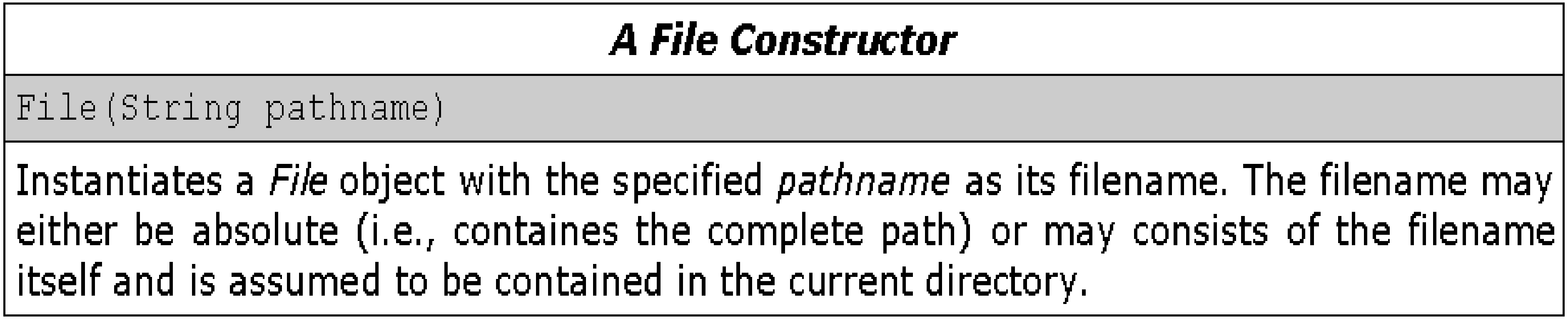
File Class

The *File* Class

- Not a stream class
- Important since stream classes manipulate *File* objects
- Abstract representation of actual files and directory pathname

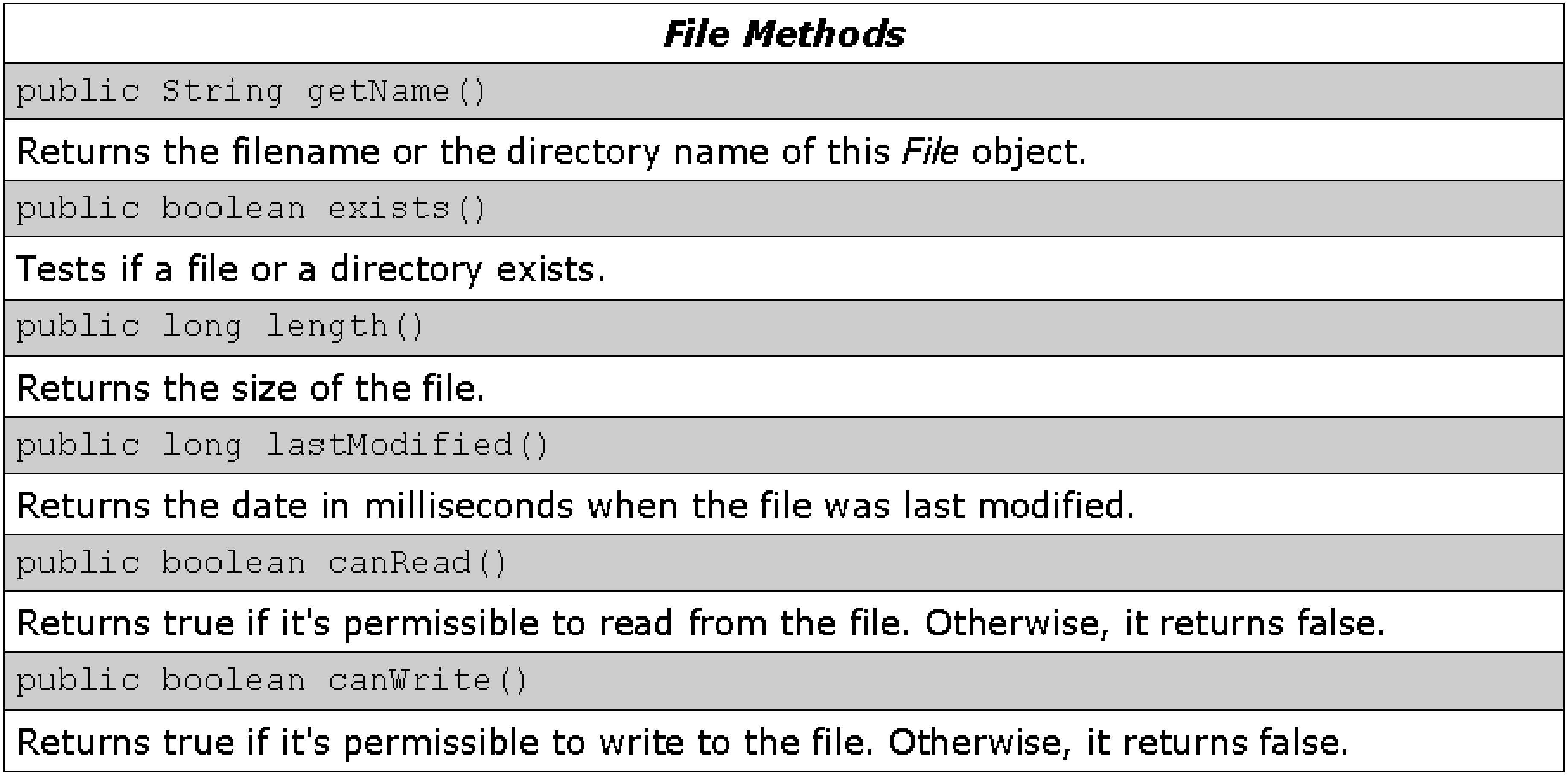
The *File* Class: Constructors

- Has four constructors



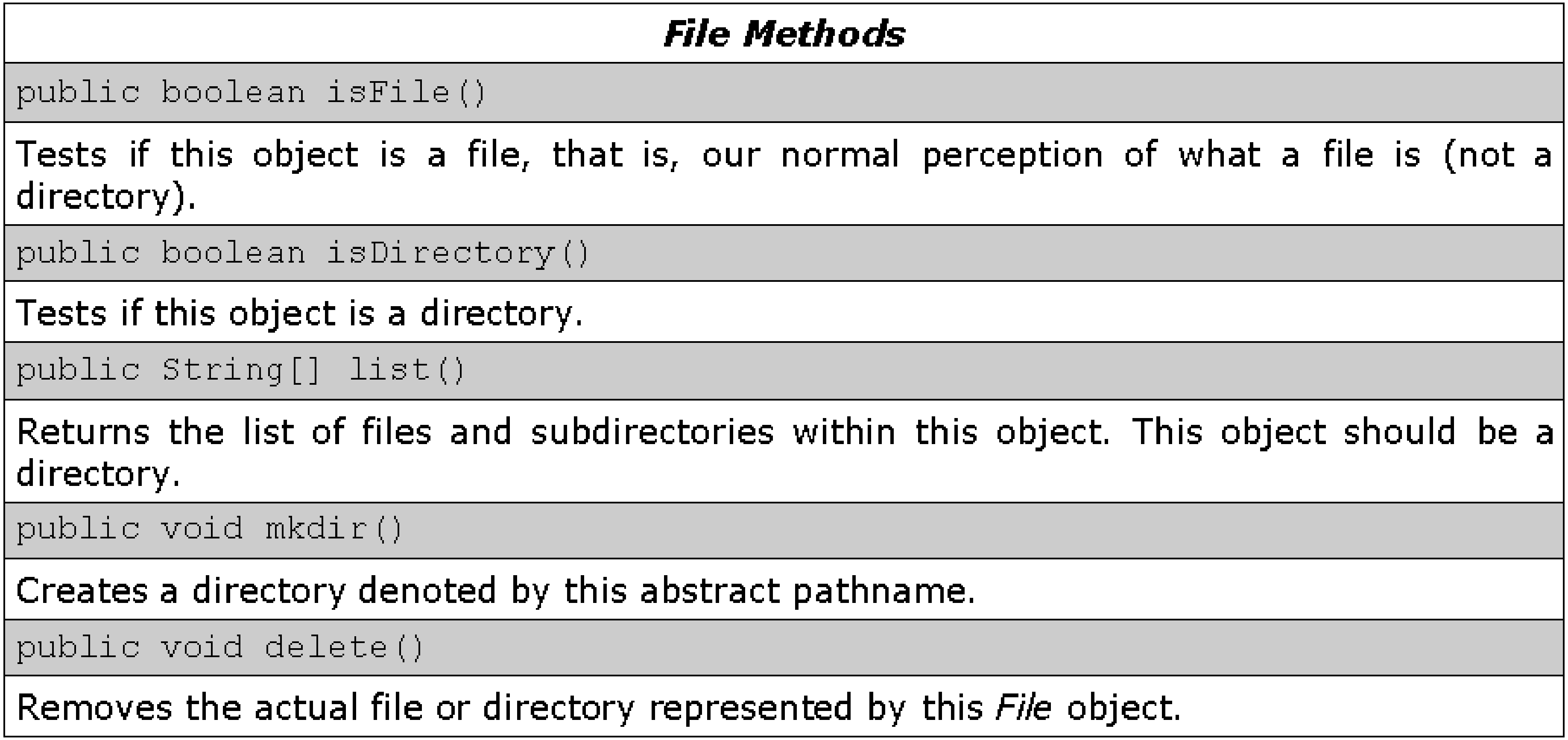
A File Constructor
`File(String pathname)`
Instantiates a *File* object with the specified *pathname* as its filename. The filename may either be absolute (i.e., containes the complete path) or may consists of the filename itself and is assumed to be contained in the current directory.

The *File* Class: Methods



File Methods
`public String getName()`
Returns the filename or the directory name of this *File* object.
`public boolean exists()`
Tests if a file or a directory exists.
`public long length()`
Returns the size of the file.
`public long lastModified()`
Returns the date in milliseconds when the file was last modified.
`public boolean canRead()`
Returns true if it's permissible to read from the file. Otherwise, it returns false.
`public boolean canWrite()`
Returns true if it's permissible to write to the file. Otherwise, it returns false.

The *File* Class: Methods



File Methods
`public boolean isFile()`
Tests if this object is a file, that is, our normal perception of what a file is (not a directory).
`public boolean isDirectory()`
Tests if this object is a directory.
`public String[] list()`
Returns the list of files and subdirectories within this object. This object should be a directory.
`public void mkdir()`
Creates a directory denoted by this abstract pathname.
`public void delete()`
Removes the actual file or directory represented by this *File* object.

The *File* Class: Example

```
import java.io.*;

public class FileInfoClass {
   public static void main(String args[]) {
      String fileName = args[0];
      File fn = new File(fileName);
      System.out.println("Name: " + fn.getName());
      if (!fn.exists()) {
         System.out.println(fileName
                            + " does not exists.");
//continued...
```

The *File* Class: Example

```
/* Create a temporary directory instead. */
System.out.println("Creating temp directory...");
fileName = "temp";
fn = new File(fileName);
fn.mkdir();
System.out.println(fileName +
    (fn.exists()? "exists": "does not exist"));
System.out.println("Deleting temp directory...");
fn.delete();
//continued...
```

The *File* Class: Example

```
      System.out.println(fileName + " is a " +
                 (fn.isFile()? "file." :"directory."));

      if (fn.isDirectory()) {
         String content[] = fn.list();
         System.out.println("The content of this directory
         for (int i = 0; i < content.length; i++) {
            System.out.println(content[i]);
         }
      }

//continued...
```

The *File* Class: Example

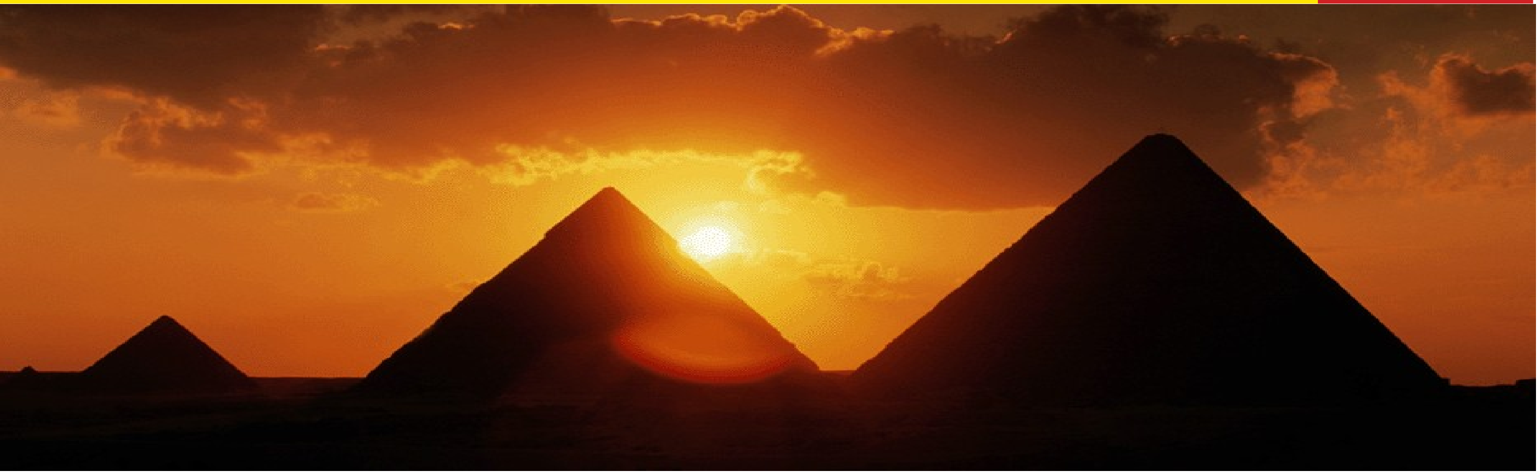
```


      if (!fn.canRead()) {
         System.out.println(fileName
                            + " is not readable.");
         return;
      }
//continued...
```

The *File* Class: Example

```
      System.out.println(fileName + " is " + fn.length()
                         + " bytes long.");
      System.out.println(fileName + " is " +
                    fn.lastModified() + " bytes long.");

      if (!fn.canWrite()) {
         System.out.println(fileName
                             + " is not writable.");
      }
   }
}
```

Modified *InputStream*/ *OutputStream* Example

```
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      String inputFile = args[0];
      CopyFile cf = new CopyFile();
      cf.copy(inputFile);
   }
}
```



Thank You!