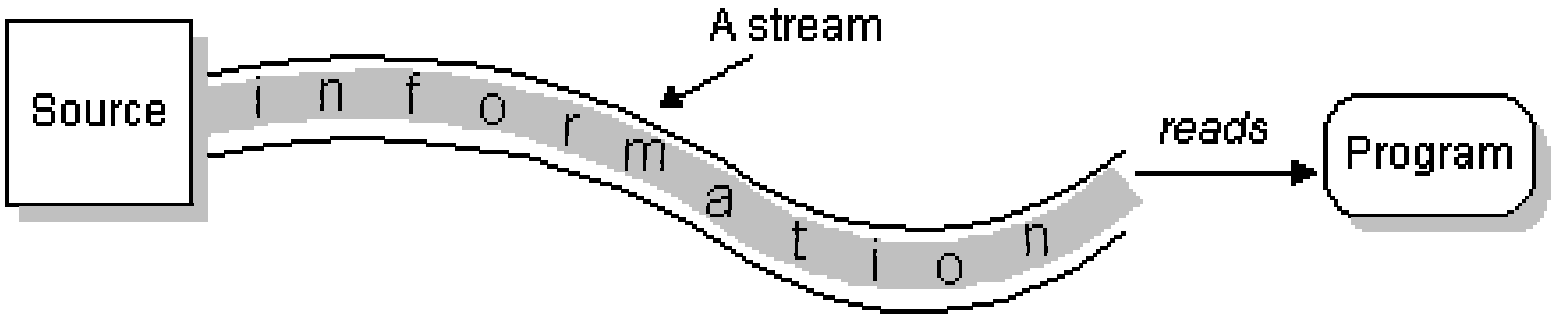
Java

Input/Output (I/O)

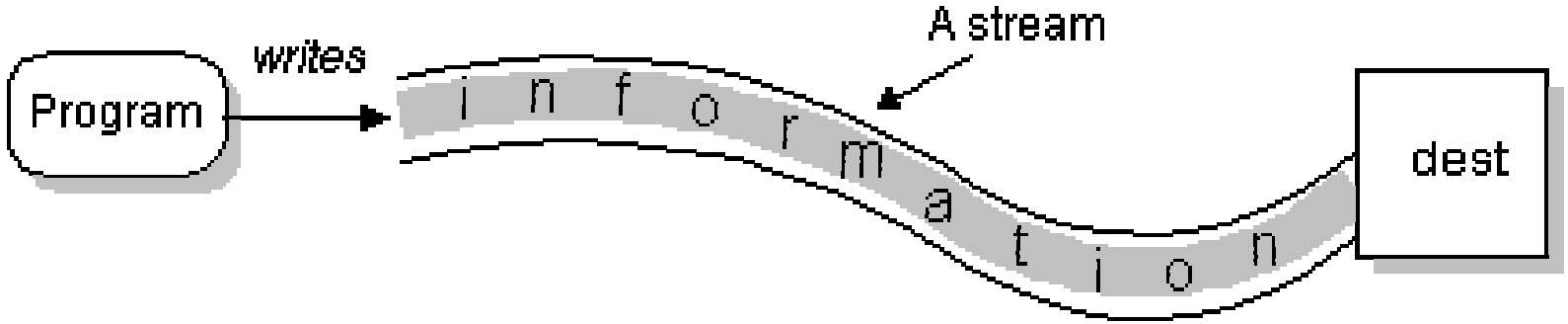
Presented by Bartosz Sakowicz

Overview of I/O streams

To bring in information, a program opens a *stream* on an information source (a file, memory, a socket) and reads the information sequentially:



Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out sequentially:



Overview of I/O streams(2)

The algorithms for sequentially reading and writing data are basically the same:

Reading	**Writing**
open a stream	open a stream
while more information	while more information
read information	**write** information
close the stream	close the stream

The java.io package contains a collection of stream classes that support these algorithms for reading and writing. To use these classes, a program needs to import the java.io package.

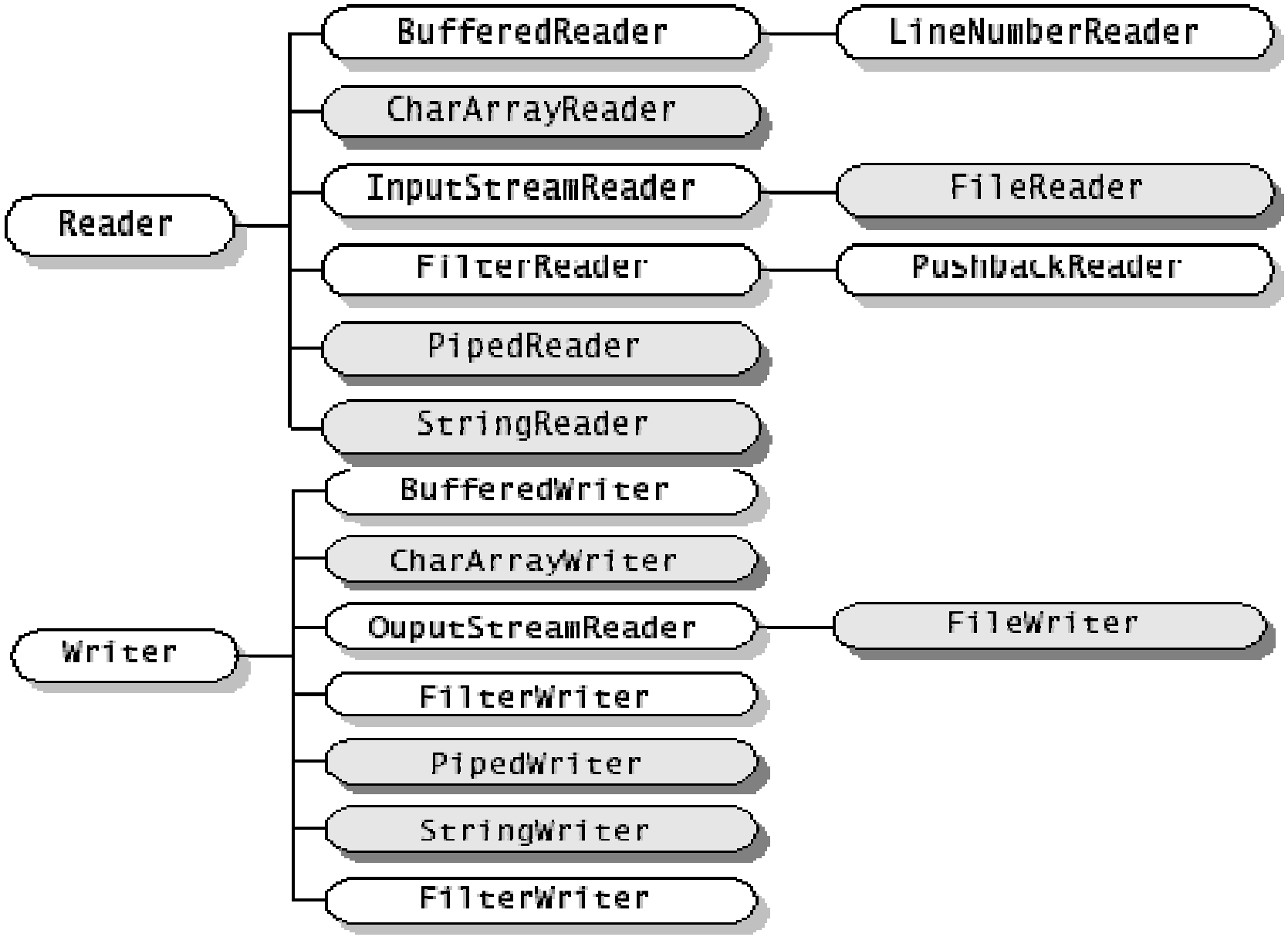
Character streams

The stream classes are divided into two class hierarchies, based on the data type (either characters or bytes) on which they operate.

Reader and Writer are the abstract superclasses for character streams in java.io. Reader provides the API and partial implementation for readers--streams that read 16-bit characters--and Writer provides the API and partial implementation for writers--streams that write 16-bit characters.

Most programs should use readers and writers to read and write textual information. The reason is that they can handle any character in the Unicode character set, whereas the byte streams are limited to ISO-Latin-1 8-bit bytes.

Character streams(2)



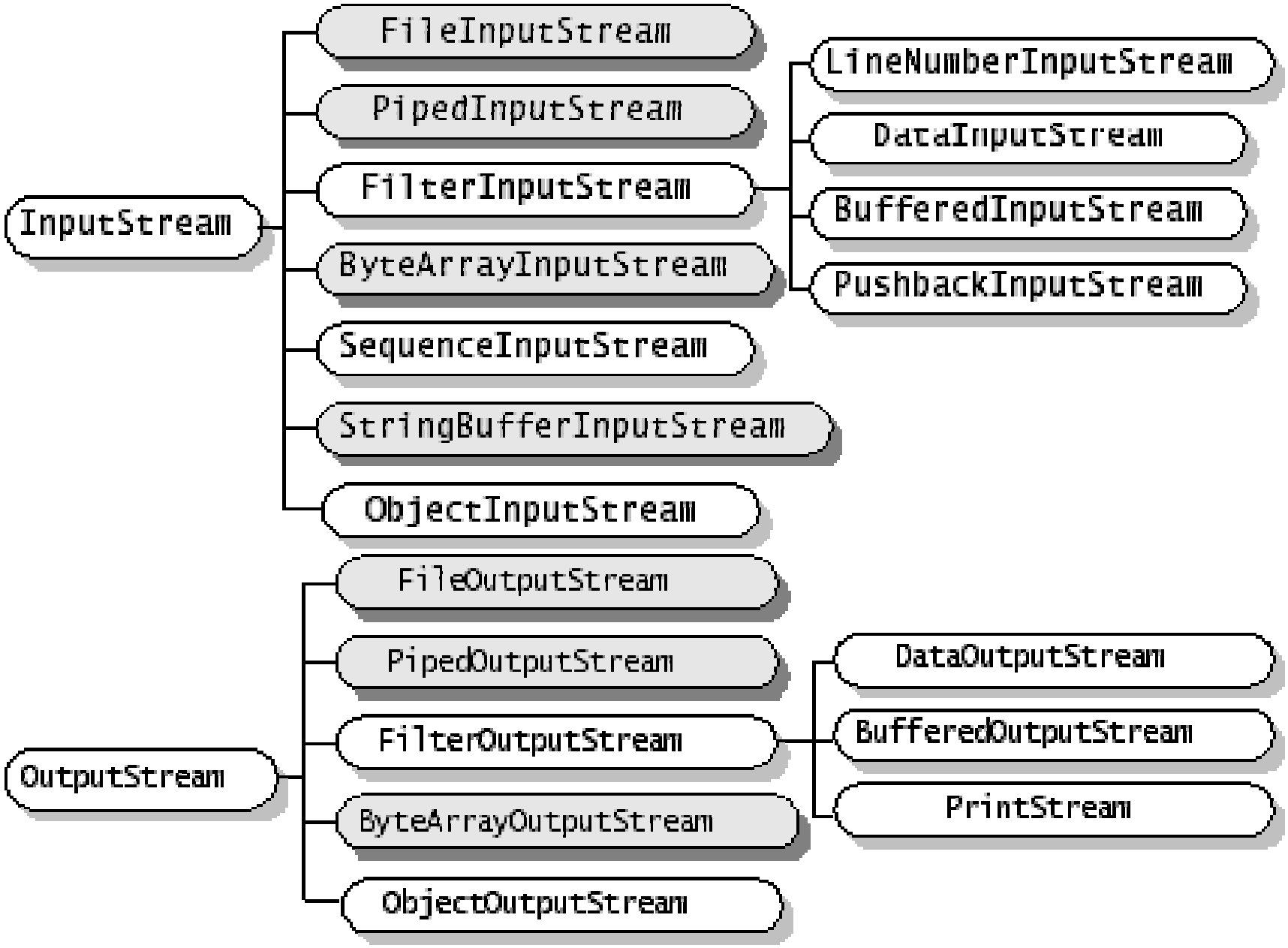
Byte streams

To read and write 8-bit bytes, programs should use the byte streams, descendants of InputStream and OutputStream.

InputStream and OutputStream provide the API and partial implementation for input streams (streams that read 8-bit bytes) and output streams (streams that write 8-bit bytes). These streams are typically used to read and write binary data such as images and sounds.

Two of the byte stream classes, ObjectInputStream and ObjectOutputStream, are used for object serialization.

Byte streams(2)



The I/O superclasses

Reader and InputStream define similar APIs but for different data types. For example, **Reader** contains these methods for reading characters and arrays of characters:

int read()

int read(char cbuf[])

int read(char cbuf[], int offset, int length)

InputStream defines the same methods but for reading bytes and arrays of bytes:

int read()

int read(byte cbuf[])

int read(byte cbuf[], int offset, int length)

Both streams provide methods for marking a location in the stream, skipping input, and resetting the current position.

The I/O superclasses (2)

Writer and OutputStream are similarly parallel. **Writer** defines these methods for writing characters and arrays of characters:

int write(int c)

int write(char cbuf[])

int write(char cbuf[], int offset, int length)

And **OutputStream** defines the same methods but for bytes:

int write(int c)

int write(byte cbuf[])

int write(byte cbuf[], int offset, int length)

All of the streams are automatically opened when created. You can close any stream explicitly by calling its *close* method or the garbage collector can implicitly close it, which occurs when the object is no longer referenced.

I/O streams – memory

CharArrayReader **CharArrayWriter**

ByteArrayInputStream **ByteArrayOutputStream**

Use these streams to read from and write to memory. You create these streams on an existing array and then use the read and write methods to read from or write to the array.

StringReader **StringWriter**

StringBufferInputStream

Use StringReader to read characters from a String in memory. Use StringWriter to write to a String. StringWriter collects the characters written to it in a StringBuffer, which can then be converted to a String.

StringBufferInputStream is similar to StringReader, except that it reads bytes from a StringBuffer.

I/O streams - pipe and file

PipedReader **PipedWriter**

PipedInputStream **PipedOutputStream**

Implement the input and output components of a pipe. Pipes are used to channel the output from one thread into the input of another.

FileReader **FileWriter**

FileInputStream **FileOutputStream**

Collectively called file streams, these streams are used to read from or write to a file on the native file system.

Presented by Bartosz Sakowicz

DMCS TUL

I/O streams - concatenation and serialization

SequenceInputStream

Concatenates multiple input streams into one input stream.

ObjectInputStream ObjectOutputStream

Used to serialize objects.

Presented by Bartosz Sakowicz

DMCS TUL

I/O streams - counting and peeking ahead

LineNumberReader **LineNumberInputStream**

Keeps track of line numbers while reading.

PushbackReader **PushbackInputStream**

These input streams each have a pushback buffer. When reading data from a stream, it is sometimes useful to peek at the next few bytes or characters in the stream to decide what to do next.

Presented by Bartosz Sakowicz

DMCS TUL

I/O streams - printing and buffering

PrintWriter **PrintStream**

Contain convenient printing methods. These are the easiest streams to write to, so you will often see other writable streams wrapped in one of these.

BufferedReader **BufferedWriter**

BufferedInputStream **BufferedOutputStream**

Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source. Buffered streams are typically more efficient than similar nonbuffered streams and are often used with other streams.

Presented by Bartosz Sakowicz

DMCS TUL

I/O streams - filtering

FilterReader **FilterWriter**

FilterInputStream **FilterOutputStream**

These abstract classes define the interface for filter streams, which filter data as it's being read or written.

I/O streams - converting betweeng bytes and characters

InputStreamReader **OutputStreamWriter**

A reader and writer pair that forms the bridge between byte streams and character streams.

An InputStreamReader reads bytes from an InputStream and converts them to characters, using the default character encoding or a character encoding specified by name.

An OutputStreamWriter converts characters to bytes, using the default character encoding or a character encoding specified by name and then writes those bytes to an OutputStream.

You can get the name of the default character encoding by calling *System.getProperty("file.encoding").*

Presented by Bartosz Sakowicz

DMCS TUL

Using file streams

```
import java.io.*;
public class Copy {
public static void main(String[] args) throws IOException {
        File inputFile = new File("in.txt");
        File outputFile = new File("out.txt");
        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;
        while ((c = in.read()) != -1) out.write(c);
        in.close(); out.close();
} }  //copying bytes by analogy
```

Concatenating files

```
import java.io.*;

public class Concatenate {

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

        ListOfFiles mylist = new ListOfFiles(args);
        SequenceInputStream s = new
        SequenceInputStream(mylist);

        int c;

        while ((c = s.read()) != -1) System.out.write(c);

        s.close();

} }     //ListOfFiles implements Enumeration.

        // nextElement() returns FileInputStream
```

Using filtering streams

• The java.io package provides a set of abstract classes that define and partially implement filter streams.

• A filter stream filters data as it's being read from or written to the stream.

• A filter stream is constructed on another stream (the underlying stream).

• The read method in a readable filter stream reads input from the underlying stream, filters it, and passes on the filtered data to the caller. The write method in a writable filter stream filters the data and then writes it to the underlying stream. The filtering done by the streams depends on the stream. Some streams buffer the data, some count data as it goes by, and others convert data to another form.

DataInputStream and DataOutputStream example

```
public class DataIODemo {

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

        DataOutputStream out = new DataOutputStream(new
                                        FileOutputStream("invoice1.txt"));

        double[] prices = { 19.99, 9.99, 15.99 };

        int[] units = { 12, 8, 13};

        String[] descs = { "Java T-shirt", "Java Mug", "Java Pin" };
```

Data ... example(2)

```
for (int i = 0; i < prices.length; i ++) {
   out.writeDouble(prices[i]);    out.writeChar('\t');
   out.writeInt(units[i]);        out.writeChar('\t');
   out.writeChars(descs[i]);      out.writeChar('\n');
}
out.close();
```

```
/* In the invoice1.txt should be:
19.99  12      Java T-shirt
9.99    8      Java Mug
15.99  13      Java Pin */
```

Data ... example(3)

```
// read it in again
DataInputStream in = new DataInputStream(new
                                FileInputStream("invoice1.txt"));


double price;
int unit;
StringBuffer desc;
double total = 0.0;
```

Data ... example(4)

```
try {
    while (true) {
        price = in.readDouble();
        in.readChar();      // throws out the tab
        unit = in.readInt();
        in.readChar();      // throws out the tab
            char chr;
            desc = new StringBuffer(20);
            char lineSep =
System.getProperty("line.separator").charAt(0);
```

Data ... example(5)

```
        while ((chr = in.readChar()) != lineSep)
                desc.append(chr);
            System.out.println("You've ordered " +
                                unit + " units of " +
                                desc + " at $" + price);
            total = total + unit * price;
        }
    } catch (EOFException e) {      }
    System.out.println("For a TOTAL of: $" + total);
    in.close();
}}
```

Data ... example(6)

The output should be:

You've ordered 12 units of Java T-shirt at $19.99

You've ordered 8 units of Java Mug at $9.99

You've ordered 13 units of Duke Juggling Dolls at $15.99

For a TOTAL of: $500,97

Writing filtering streams

1. Create a subclass of FilterInputStream and FilterOutputStream. Input and output streams often come in pairs, so it's likely that you will need to create both input and output versions of your filter stream.

2. Override the read and write methods, if you need to.

3. Provide any new methods.

4. Make sure that the input and output streams work together.

Object serialization

Reading and writing objects is a process called **object serialization**.The key to writing an object is to represent its state in a serialized form sufficient to reconstruct the object as it is read. Object serialization is essential to building all but the most transient applications. You can use object serialization in the following ways:

- Remote Method Invocation (RMI) - communication between objects via sockets
- Lightweight persistence - the archival of an object for use in a later invocation of the same program.

Writing to an ObjectOutputStream

Example:

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
 s.writeObject(new Date());
s.flush();
```

Writing to an ObjectOutputStream(2)

- ObjectOutputStream must be constructed on another stream.
- ObjectOutputStream implements the DataOutput interface that defines many methods for writing primitive data types, such as writeInt, writeFloat, or writeUTF.
- The writeObject method throws a *NotSerializableException* if it's given an object that is not serializable.
- **An object is serializable only if its class implements the Serializable interface.**

Reading from ObjectInputStream

Once you've written objects and primitive data types to a stream, you can read them out again and reconstruct the objects:

FileInputStream in = new FileInputStream("theTime");

ObjectInputStream s = new ObjectInputStream(in);

String today = (String)s.readObject();

Date date = (Date)s.readObject();

Reading from ObjectInputStream(2)

- ObjectInputStream must be constructed on another stream.
- In previous example, the objects were archived in a file, so the code constructs an ObjectInputStream on a FileInputStream. Next, the code uses ObjectInputStream's readObject method to read the String and the Date objects from the file.
- The objects must be read from the stream in the same order in which they were written.
- The return value from readObject is an object that is cast to and assigned to a specific type.
- ObjectInputStream stream implements the DataInput interface that defines methods for reading primitive data types (analogous to DataOutput Interface).

Providing serialization for a class

Complete definition of the Serializable interface:

```
package java.io;

public interface Serializable {  };
```

Making instances of your classes serializable is:

```
public class MyClass implements Serializable { ... }
```

Providing serialization for a class(2)

You don't have to write any methods. The serialization of instances of this class are handled by the defaultWriteObject method of ObjectOutputStream. This method automatically writes out everything required to reconstruct an instance of the class, including the following:

- Class of the object
- Class signature
- Values of all non-transient and non-static members, including members that refer to other objects

You can deserialize any instance of the class with the defaultReadObject method in ObjectInputStream. For many classes, this default behavior is good enough.

Customing serialization

You can customize serialization for your classes by providing two methods for it: writeObject and readObject.

The writeObject method controls what information is saved and is typically used to append additional information to the stream.

The readObject method either reads the information written by the corresponding writeObject method or can be used to update the state of the object after it has been restored.

Customing serialization(2)

The writeObject and readObject methods must be declared exactly as shown:

private void writeObject(ObjectOutputStream s) throws IOException {

s.defaultWriteObject();

// customized serialization code }

The readObject method must read in everything written by writeObject in the same order in which it was written:

private void readObject(ObjectInputStream s) throws IOException {

s.defaultReadObject();

// customized deserialization code }

File and StreamTokenizer

File class

Represents a file on the native file system. You can create a File object for a file on the native file system and then query the object for information about that file, such as its full path name.

StreamTokenizer class

Breaks the contents of a stream into tokens. Tokens are the smallest unit recognized by a text-parsing algorithm (such as words, symbols, and so on). A StreamTokenizer object can be used to parse any text file. For example, you could use it to parse a source file into variable names, operators, and so on, or to parse an HTML file into HTML tags. Similar class is **StringTokenizer**.