# The Java I/O System

- Binary I/O streams (ASCII, 8 bits)
  - **`InputStream`**
  - **`OutputStream`**
- The decorator design pattern
- Character I/O streams (Unicode, 16 bits)
  - **`Reader`**
  - **`Writer`**
- Comparing binary I/O to character I/O
- Files and directories
  - The class **`File`**

# Overview of The Java I/O System

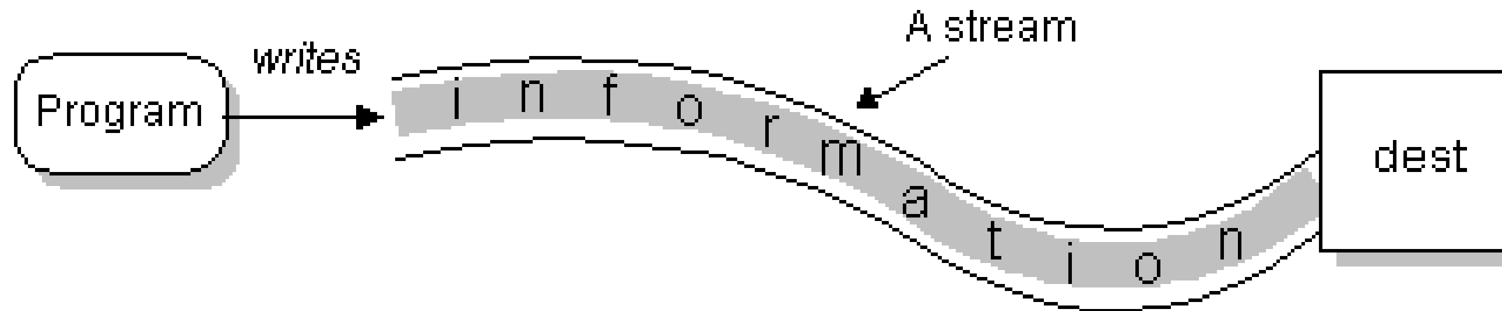
- *Goal*: To provide an abstraction of all types of I/O
  - Memory
  - File
  - Directory
  - Network

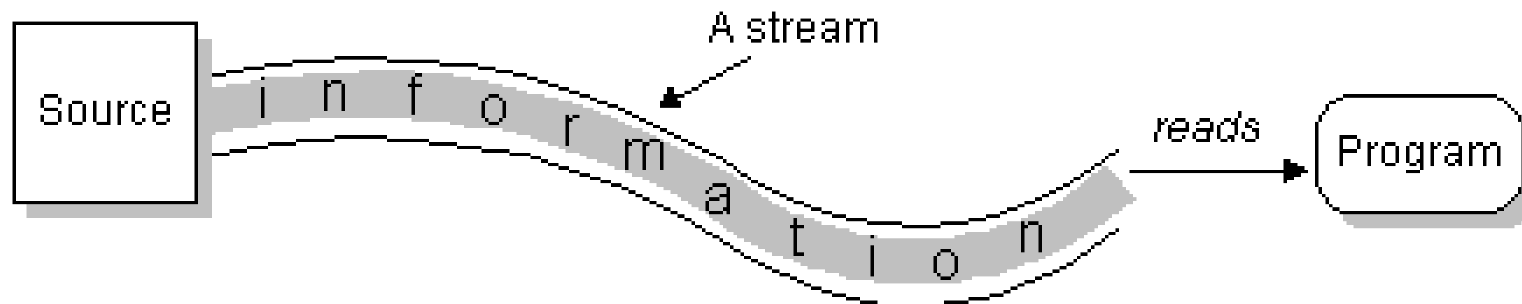
- Express all configurations
  - Character, binary, buffered, etc.

- Different kinds of operations
  - Sequential, random access, by line, by word, etc.

# The Stream Concept

- A *stream* is a sequential source of information used to transfer information from one source to another.



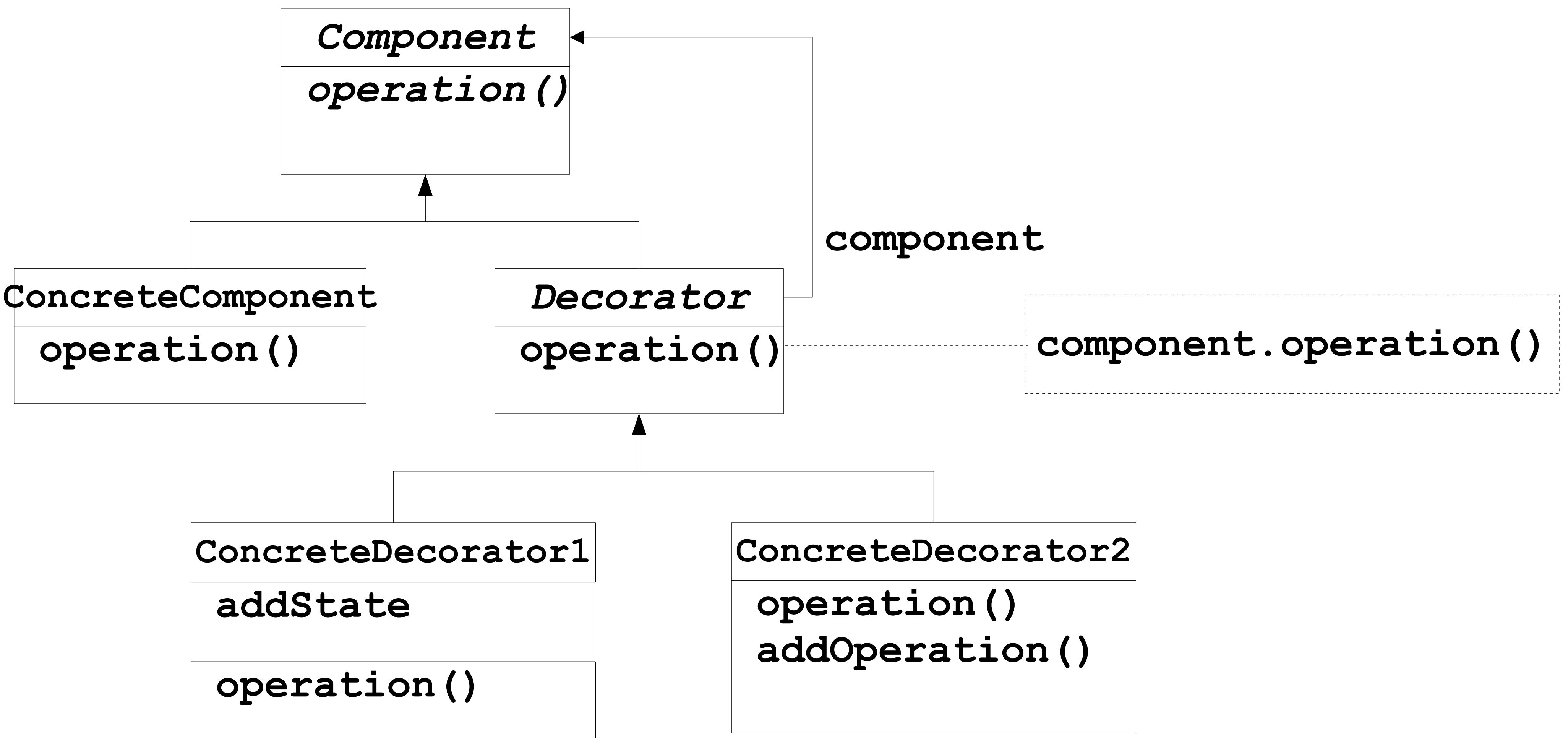

[Source: java.sun.com]

# Streams in Java

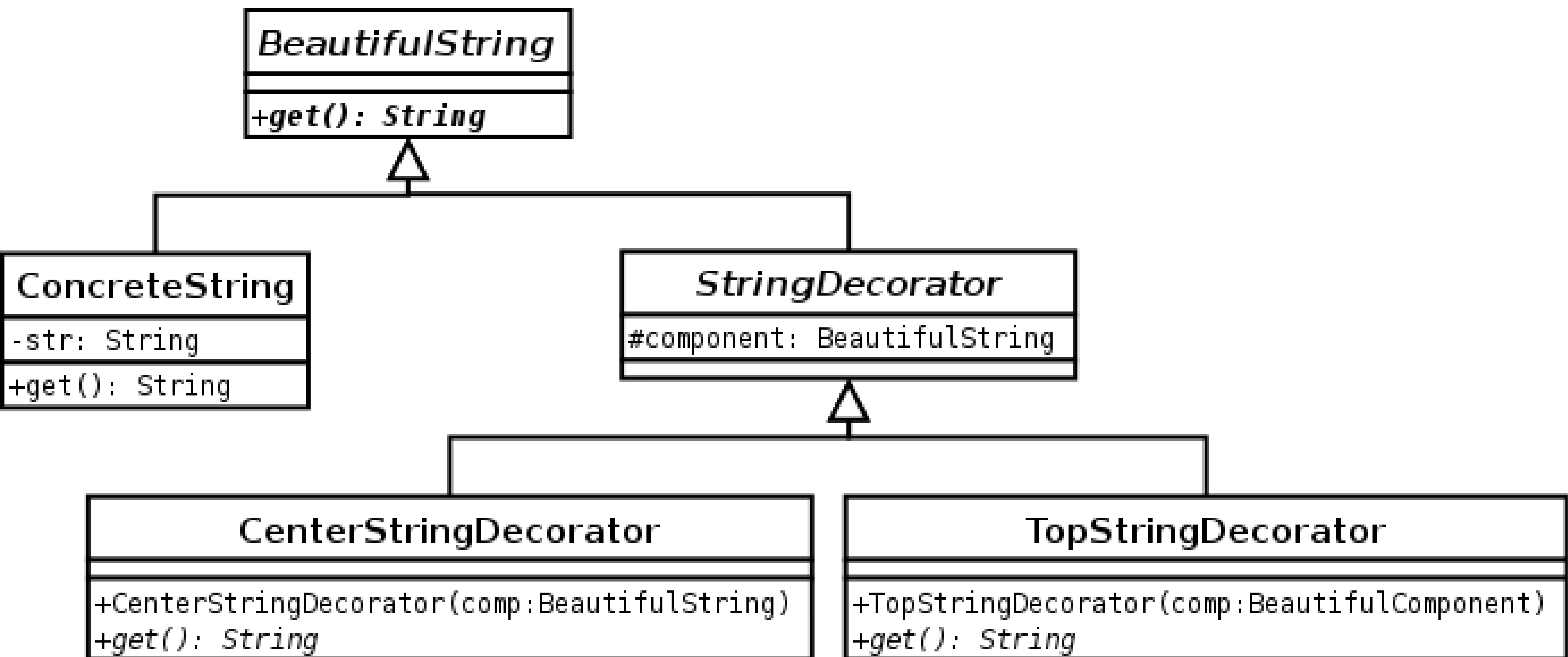
- There is a huge (and complicated) hierarchy of stream classes in Java.
- Overview classes of the stream hierarchy
  - **`InputStream`** (input + binary)
  - **`OutputStream`** (output + binary)
  - **`Reader`** (input + Unicode)
  - **`Writer`** (output + Unicode)
- All abstract classes.

# The Decorator Design Pattern

- Wrapper classes in “decorators” to add functionality.



# The Decorator Design Pattern, cont




```
Hello
```

```
-->Hello<--
```

```
-----
Hello
```

# The Decorator Design Pattern, cont

```
public abstract class BeautifulString {
    // encapsulated nice string
    protected String text;
    // The abstract methods that get a beautiful string
    public abstract String get();
}

public abstract class StringDecorator extends BeautifulString {
    /** The component that is decorated */
    protected BeautifulString component;
}

public class CenterStringDecorator extends StringDecorator {
    public CenterString(BeautifulString comp){
        component = comp; }
    // the decorator method
    public String get(){
        String temp = component.get();
        temp = "-->" + temp + "<--";
        return temp;
    }
}
```

# The Decorator Design Pattern, cont

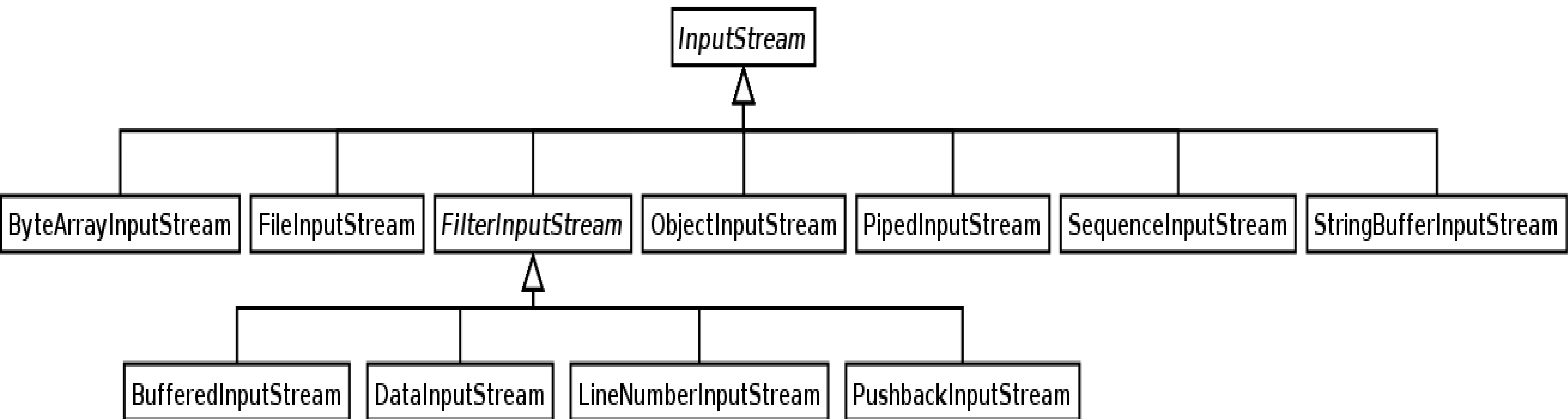
```
public static void main(String[] args) {
  System.out.println("Concrete");
  BeautifulString bs1 = new ConcreteString("Hello");
  System.out.println(bs1.get());


  System.out.println("Center + Concrete");
  BeautifulString bs2 = new CenterStringDecorator(
                                new ConcreteString("Hello"));
  System.out.println(bs2.get());


  System.out.println("Top + Center + Concrete");
  BeautifulString bs3 = new TopStringDecorator(
                                new CenterStringDecorator(
                                  new ConcreteString("Hello")));
  System.out.println(bs3.get());
```

# Decorator Pattern and Java I/O

- Two issues with I/O
  - What are you talking to ($n$).
  - The way you are talking to it ($m$).

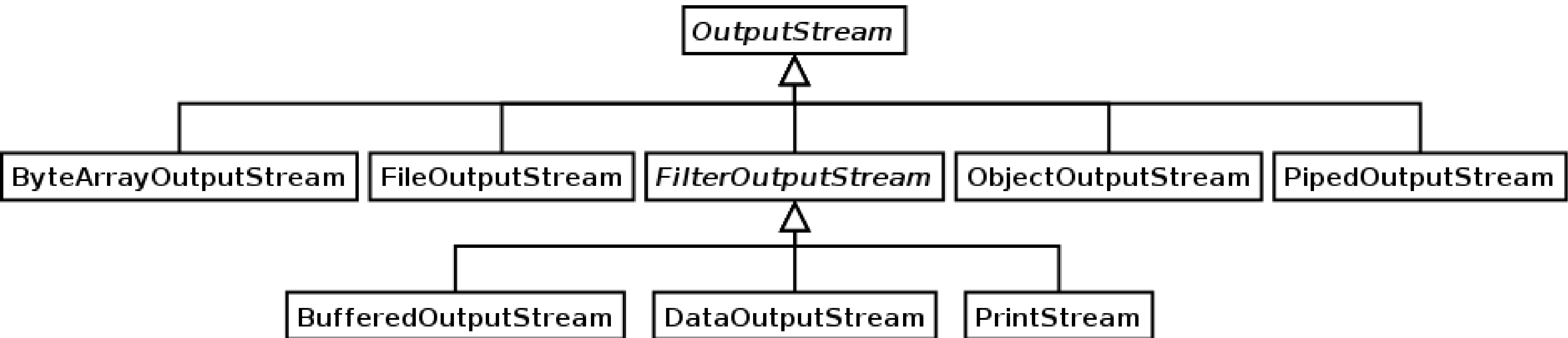
- Solution no. 1
  - Make a class for every combination
  - $n * m$ classes, not flexible, hard to extend
- Solutions no. 2
  - Java filter streams (decorators) are added dynamically to create the functionality needed.
  - $n + m$ classes
  - Input decorator: **`FilterInputStream`**
  - Output decorator: **`FilterOutputStream`**

# **InputStream** Hierarchy




- **InputStream**, the abstract component root in decorator pattern
- **FileInputStream**, etc. the concrete components
- **FilterInputStream**, the abstract decorator
- **LineNumberInputStream**, **DataInputStream**, etc. concrete decorators

# **`OutputStream`** Hierarchy




- **`OutputStream`**, the abstract component root in decorator pattern
- **`FileOutputStream`**, etc. the concrete components
- **`FilterOutputStream`**, the abstract decorator
- **`PrintStream`**, **`DataOutputStream`**, etc. concrete decorators

# **`InputStream`** Types

| *Type of InputStream* | *Reads From* |
| --- | --- |
| **`ByteArrayInputStream`** | Block of memory |
| **`StringBufferInputStream`** | **`String`** (note *not* **`StringBuffer`**) |
| **`PipedInputStream`** | Pipe (in another thread) |
| **`FileInputStream`** | File |
| **`SequencedInputStream`** | Combines **`InputStream`**s |
| **`ObjectInputStream`** | Objects from an **`InputStream`** |

Concrete Components

# **OutputStream** Types

| *Type of OutputStream* | *Writes To* |
|---|---|
| **ByteArrayOutputStream** | Block of memory |
| **PipedOutputStream** | Pipe (in another thread) |
| **FileOutputStream** | File |
| **ObjectOutputStream** | Objects to a **OutputStream** |

Concrete Components

# FilterInputStream

- **`DataInputStream`**
  - Full interface for reading built-in types
  - For portable reading of data between different OS platforms
- **`BufferedInputStream`**
  - Adds buffering to the stream (do this by default)
- **`LineNumberInputStream`**
  - Only adds line numbers
- **`PushbackInputStream`**
  - One-character push pack for scanners (lexers)

Concrete Decorators

# FilterOutputStream

- **`DataOutputStream`**
  - Full interface for writing built-in types
  - For portable writing of data between different OS platforms
  - Example: **`System.out.println`**
- **`PrintStream`**
  - Allows primitive formatting of data for display
  - Not for storage use **`DataOutputStream`** for this
- **`BufferedOutputStream`**
  - Adds buffering to output (do this by default!)

Concrete Decorators

# OutputStream, Example

```
import java.io.*; // [Source: java.sun.com]

public class DataIODemo {

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

       // where to write to
       DataOutputStream out =
            new DataOutputStream(
              new FileOutputStream("invoice1.txt"));

       // alternative also using a buffer decorator
       DataOutputStream out =
            new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("invoice1.txt")));
```

# OutputStream, Example, cont.

```
import java.io.*; // [Source: java.sun.com]
public class DataIODemo {
  public static void main(String[] args) throws IOException {
      //snip
      double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
      int[] units = { 12, 8, 13, 29, 50 };
      String[] descs = { "Java T-shirt",
                         "Java Mug",
                         "Duke Juggling Dolls",
                         "Java Pin",
                         "Java Key Chain" };


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

# InputStream, Example

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

// alternative also using a buffer decorator
DataInputStream in =
    new DataInputStream(
        new BufferedInputStream (
            new FileInputStream("invoice1.txt")));


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

# `InputStream`, Example, cont.

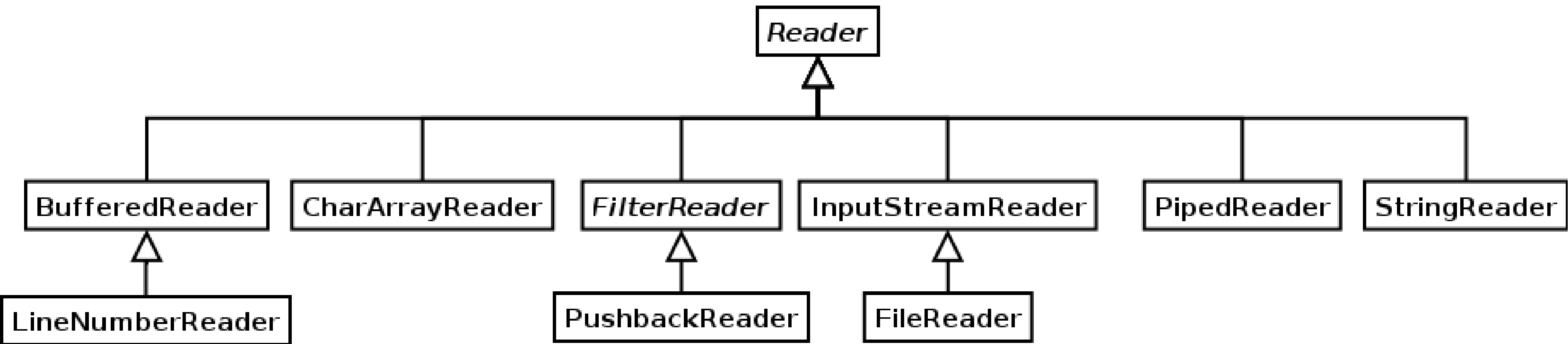
```
  try {
     while (true) {
        price = in.readDouble();
        in.readChar();        // throws out the tab
        unit = in.readInt();
        in.readChar();        // throws out the tab
        char chr;
        desc = new StringBuffer(20);
        char lineSep =
            System.getProperty("line.separator").charAt(0);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

# **`Reader`** and **`Writer`** Classes

- Added in Java 1.1
- Not meant to replace **`InputStream`** and **`OutputStream`**
- Internationalization Unicode support
- Designed to solved efficiency problems

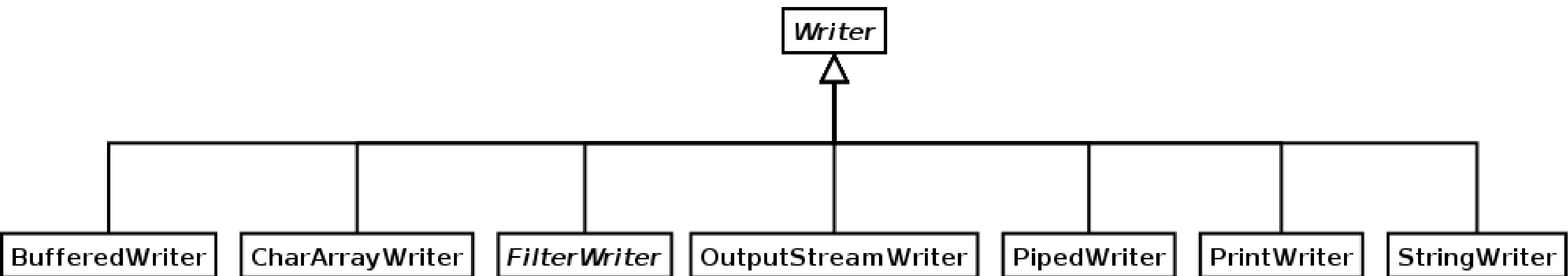
- Structured in class hierarchies similar to the **`InputStream`** and **`OutputStream`** hierarchies
  - Uses the decorator design pattern

# **Reader** Class Hierarchy




- **Reader**, the abstract component root in decorator pattern
- **BufferedReader**, etc. the concrete components
- **FilterReader**, the abstract decorator
- **PushbackReader**, concrete decorators

# **Writer** Class Hierarchy




- **Writer**, the abstract component root in decorator pattern
- **BufferedWriter**, etc. the concrete components
- **FilterWriter**, the abstract decorator
- No concrete decorators

# **Reader** and **Writer** Types

- Transport to and from main memory
  - **CharArrayReader**, **CharArrayWriter**
  - **StringReader**, **StringWriter**
- Transport to and from pipelines (networking)
  - **PipedReader**, **PipedWriter**
- Transport to and from files
  - **FileReader**, **FileWriter**
- **DataOutputStream** unaltered from Java 1.0 to 1.1

# Character Based Streams

- **`InputStreamReader`**
  - Reads platform characters and delivers Unicode characters to the Java program.
- **`OutputStreamWriter`**
  - Writes Unicode characters to platform dependent characters.
- **`PrintWriter`**
  - Writes Java primitive data types to file.

# FileReader and FileWriter, Example

```
import java.io.*;

public class Copy {
    public static void main(String[] args) throws IOException
    {
        FileReader in = new FileReader(new File(args[0]));
        FileWriter out = new FileWriter(new File(args[1]));
        int c;
        do{
            c = in.read();
            if(c != -1) {
                out.write(c);
            }
        } while (c != -1);

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

# Binary vs. Character Based I/O Overview

- InputStream
- OutputStream
- FileInputStream
- FileOutputStream
- StringBufferedInputStream
- N/A
- ByteArrayInputStream
- ByteArrayOutputStream
- PipedInputStream
- PipedOutputStream

- Reader
  convert: InputStreamReader
- Writer
  convert: OutputStreamWriter
- FileReader
- FileWriter
- StringReader (better name)
- StringWriter
- CharArrayReader
- CharArrayWriter
- PipedReader
- PipedWriter

# Binary vs. Character Filter Overview

- FilterInputStream
- FilterOutputStream
- BufferedInputStream
- BufferedOutputStream
- DataInputStream
- PrintStream
- LineNumberInputStream
- PushbackInputStream

- FilterReader
- FilterWriter (abstract class)
- BufferedReader (has a readline())
- BufferedWriter
- Use DataInputStream or BufferedReader
- PrintWriter
- LineNumberReader
- PushbackReader

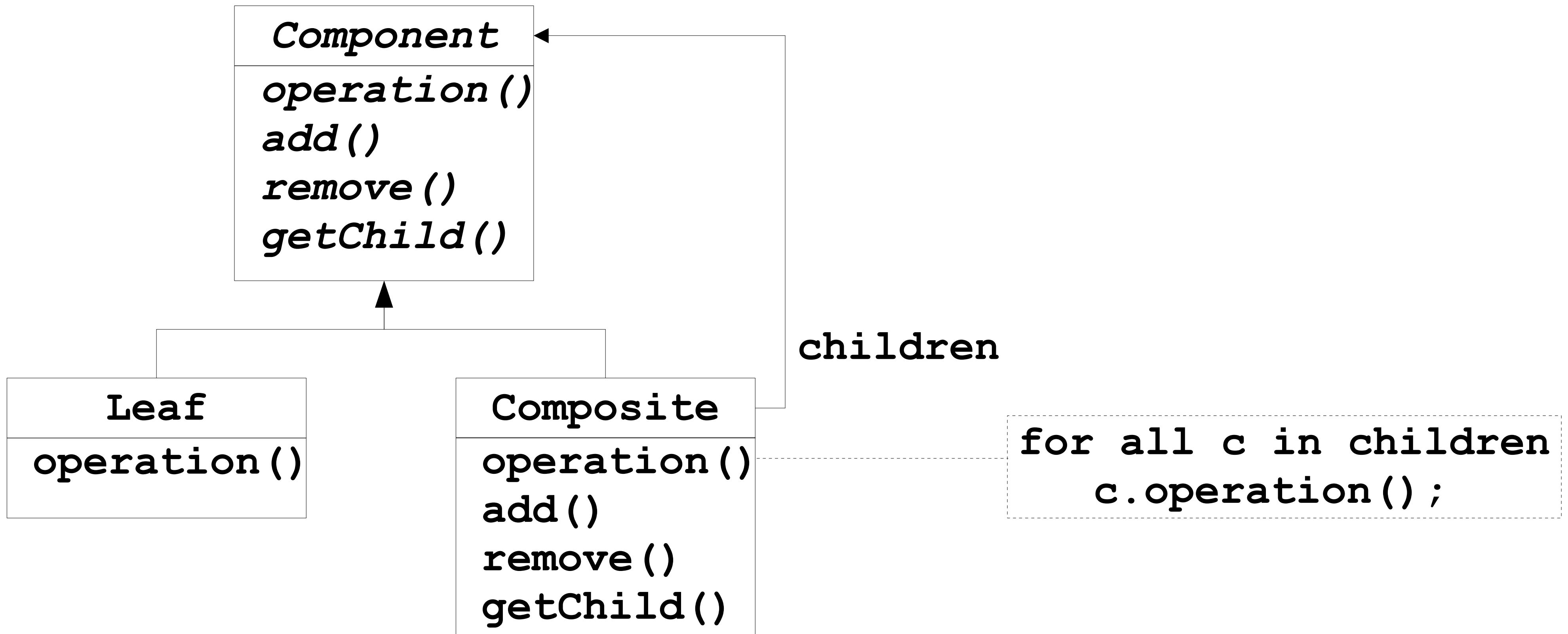
# Representing the File System

- File systems varies between operating system, i.e.,
  - Path separators
  - Permissions in Unix
  - Directories on the Mac
  - Drive letters on Windows
- Needs an abstraction to hide the differences
  - To make Java program platform independent.

# The `File` Class

- Refers to one or more file names, i.e., not a handle to a file
  - Composite design pattern
- To get an array of file names. Call the `list()` method.

# The Composite Design Pattern, Again



# The **File** Class, Example

```
import java.io.*;

public class DirectoryList {
   public static void main(String[] args) throws IOException{
      File dir = new File(args[0]);

      if(dir.isDirectory() == false) {
         if (dir.exists() == false)
            System.out.println("There is no such dir!");
         else
            System.out.println("That file is not a dir.");
      }
      else {
         String[] files = dir.list();
         System.out.println
            ("Files in dir \"" + dir + "\":");
         for (int i = 0; i < files.length; i++)
            System.out.println("  " + files[i]);
      }
   }
}
```

# Summary

- Streams a large class hierarchy for input and output.
  - The decorator pattern is the key to understanding it
- The decorator design pattern may seem strange
  - Very flexible, but requires extra coding in clients.
- Scanner class for input/output very versatile
- For objects to live between program invocations use the `Serializable` interface. Covered later in course.
- `java.nio` packages goal speed
  - Look at it if you needed it in your projects