Contrasting Java I/O Streams with Java Streams

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

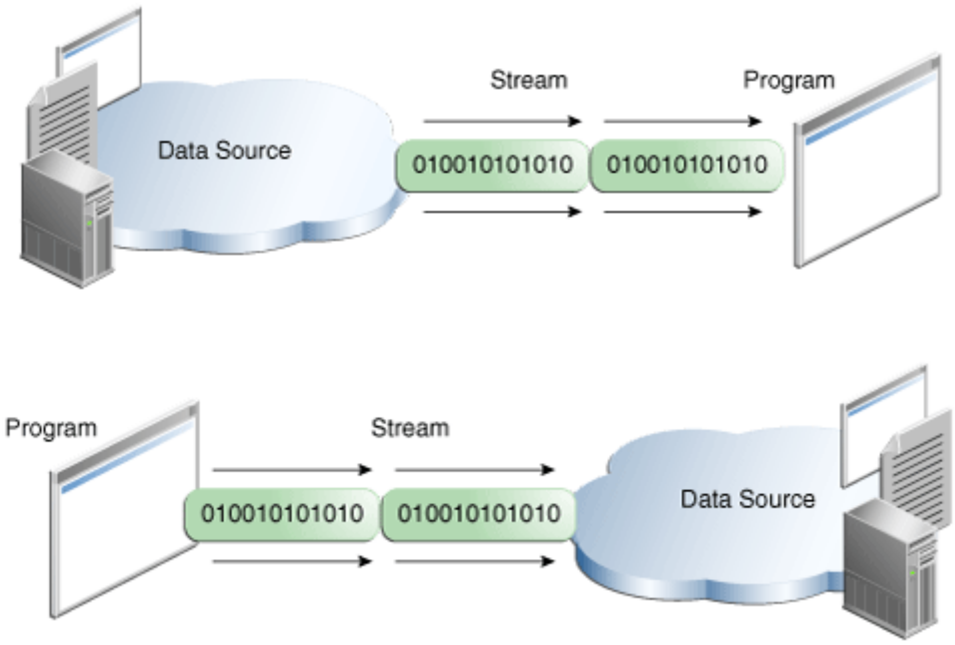
Professor of Computer Science

Institute for Software Integrated Systems

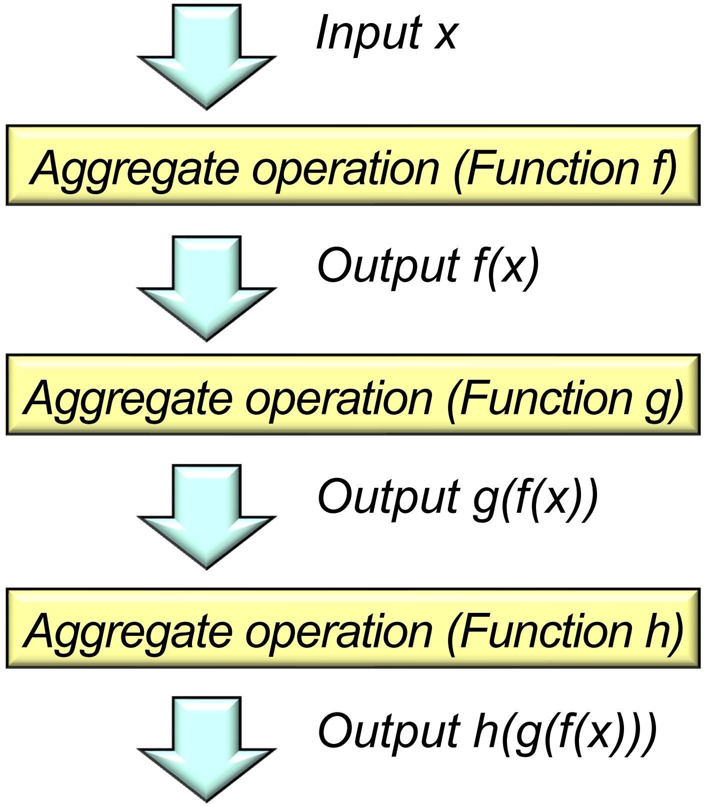
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand how Java I/O streams contrast with Java streams



$\neq$


Input x
Aggregate operation (Function f)
Output f(x)
Aggregate operation (Function g)
Output g(f(x))
Aggregate operation (Function h)
Output h(g(f(x)))

Learning Objectives in this Part of the Lesson

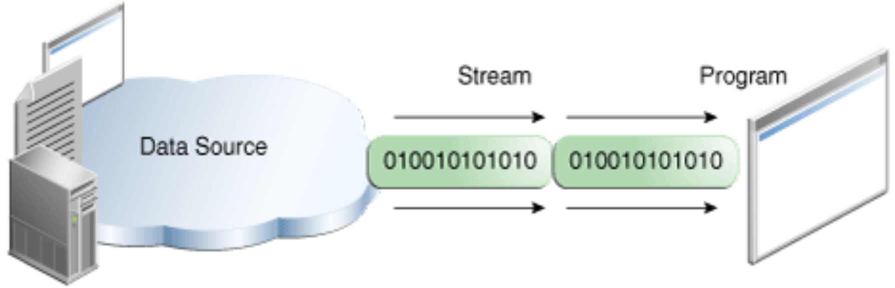
- Understand how Java I/O streams contrast with Java streams
  - Know how to program with Java I/O streams & Java streams

```
try (Stream<String> lines = Files.lines(Paths.get(path))) {
  return lines

    .skip(1)

    .map(line -> line.split(";"))

    .map(s -> new SimpleEntry<>(s[0], parseVector(s[1])))

    .collect(toMap(SimpleEntry::getKey,
                   SimpleEntry::getValue,
                   (x, y) -> x));
```

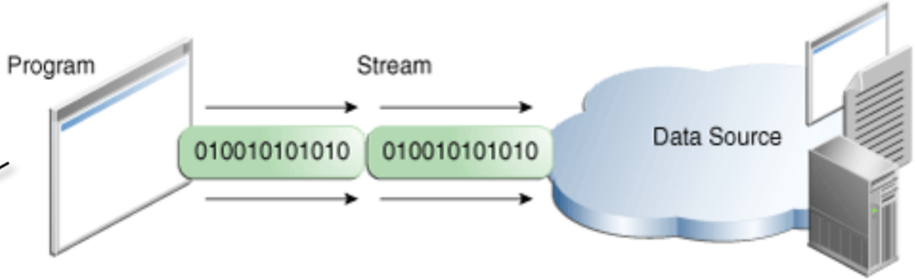
Contrasting Java I/O Streams with Java Streams

Contrasting Java I/O Streams & Java Streams

- A Java *I/O Stream* represents an input source or an output destination



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

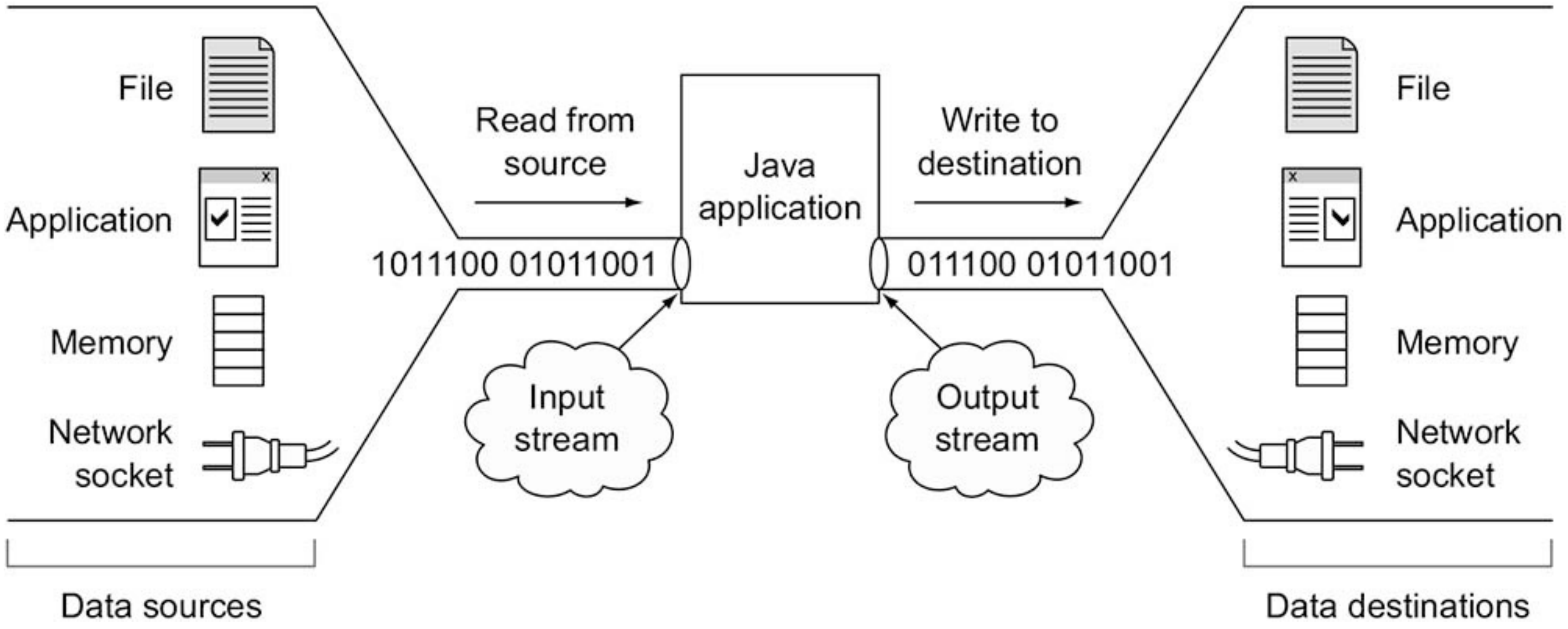


A program uses an output stream to write data to a destination, one item at time

See docs.oracle.com/javase/tutorial/essential/io/streams.html

Contrasting Java I/O Streams & Java Streams

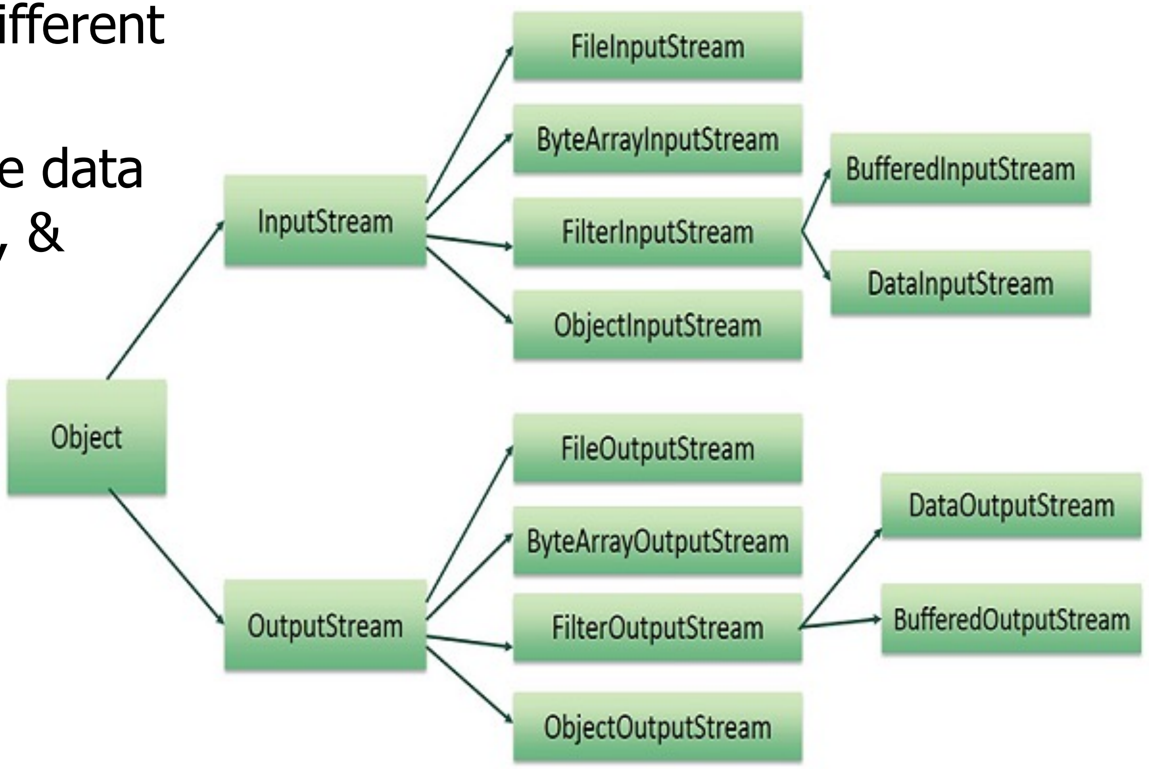
- An I/O stream can represent different sources & destinations
  - e.g., disk files, devices, other programs, & memory arrays



See www.javatpoint.com/java-io

Contrasting Java I/O Streams & Java Streams

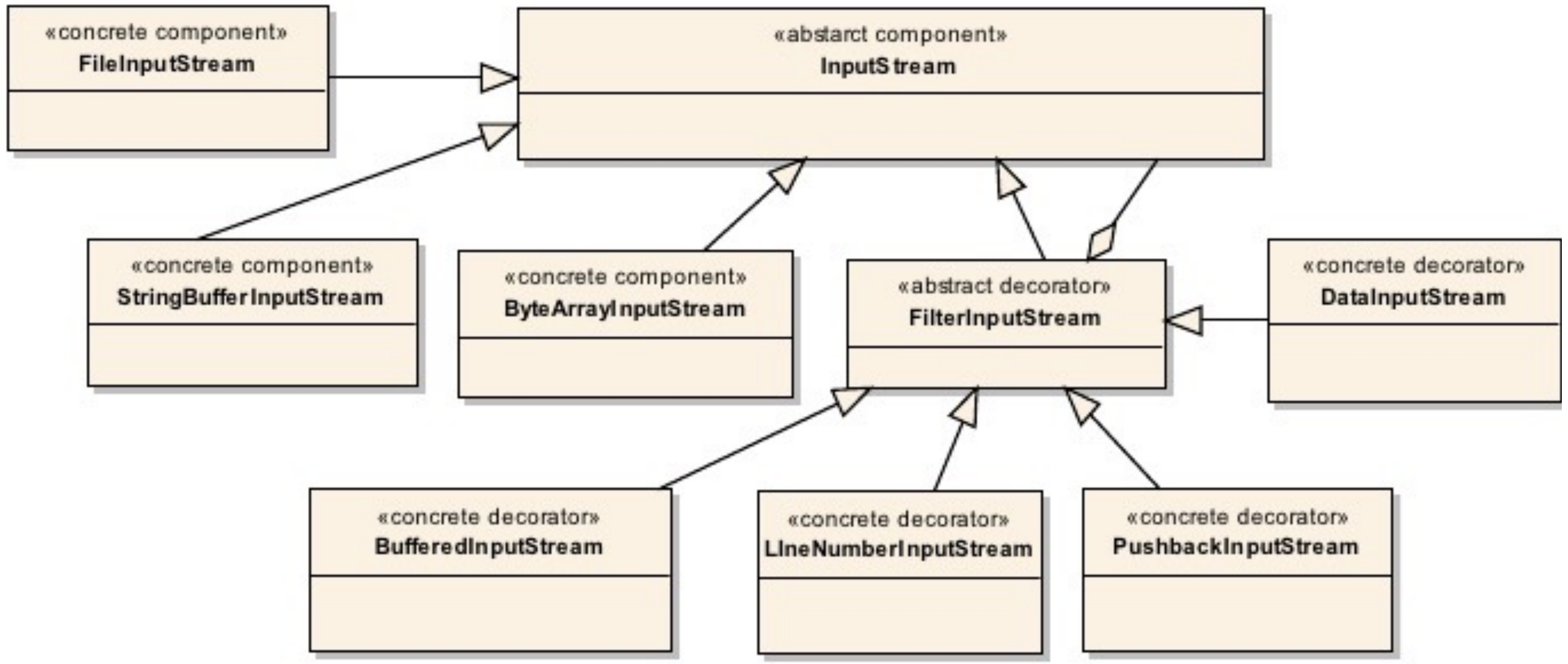
- I/O streams support many different types of data
  - e.g., simple bytes, primitive data types, localized characters, & objects



See www.tutorialspoint.com/java/java_files_io.htm

Contrasting Java I/O Streams & Java Streams

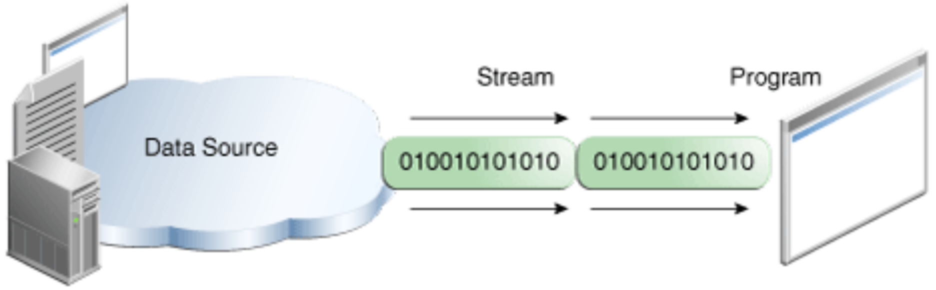
- Some I/O streams simply pass on data, whereas others manipulate & transform the data in useful ways

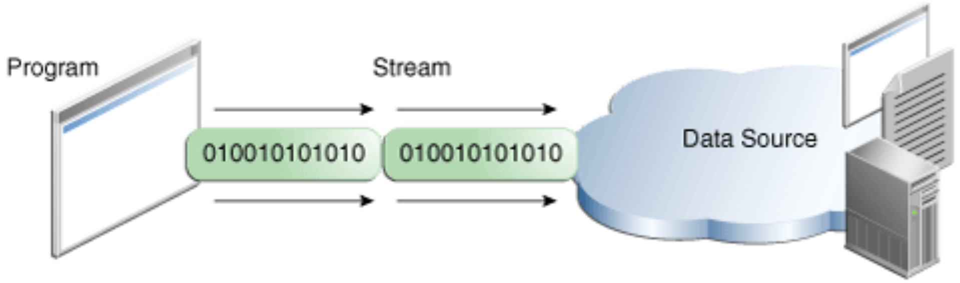


See kymr.github.io/2016/11/27/Decorator-Pattern

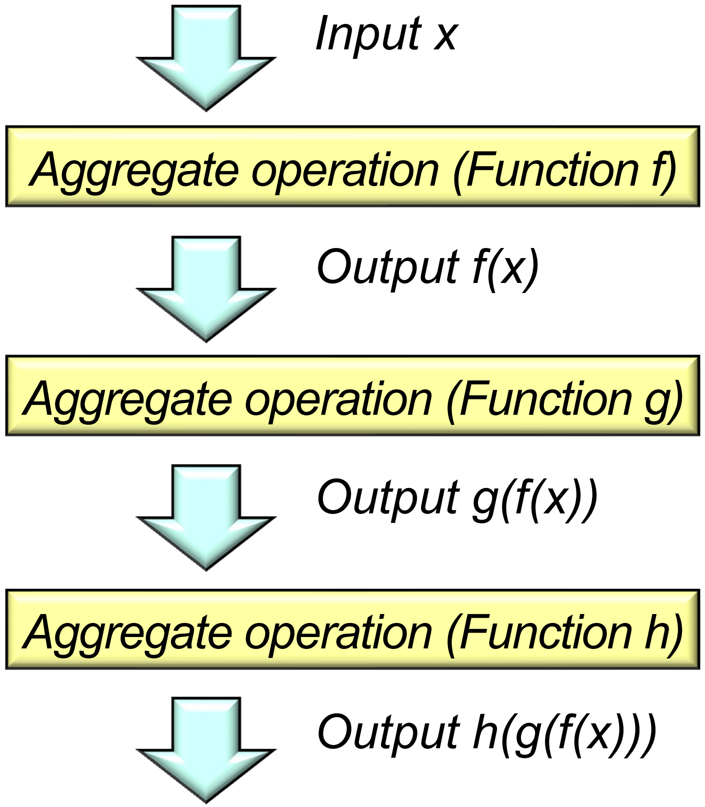
Contrasting Java I/O Streams & Java Streams

- Java I/O streams are different from Java streams!





$\neq$

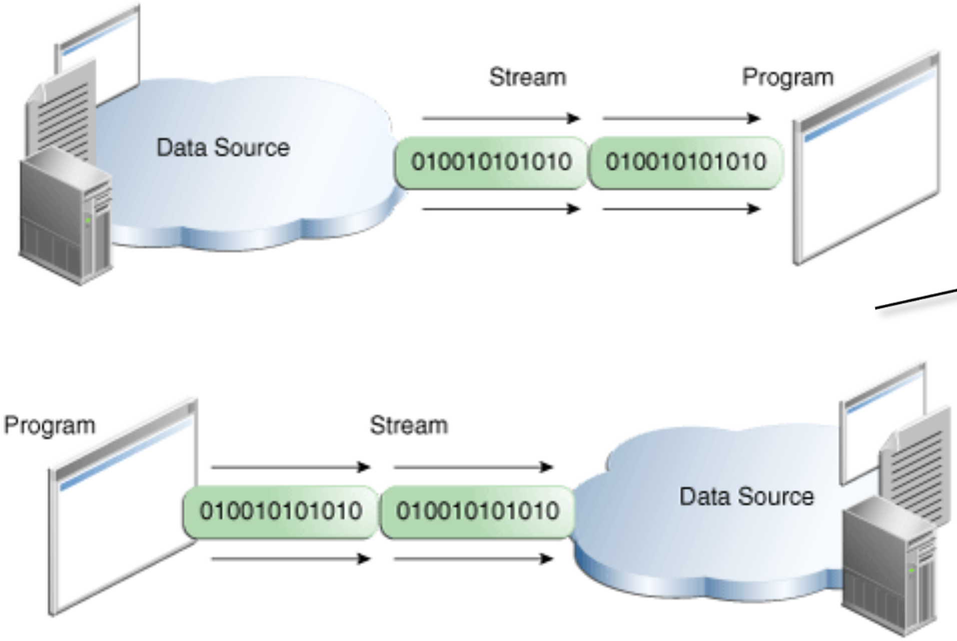


Input x
Aggregate operation (Function f)
Output f(x)
Aggregate operation (Function g)
Output g(f(x))
Aggregate operation (Function h)
Output h(g(f(x)))

See stackoverflow.com/questions/39550670

Contrasting Java I/O Streams & Java Streams

- Java I/O streams are different from Java streams!

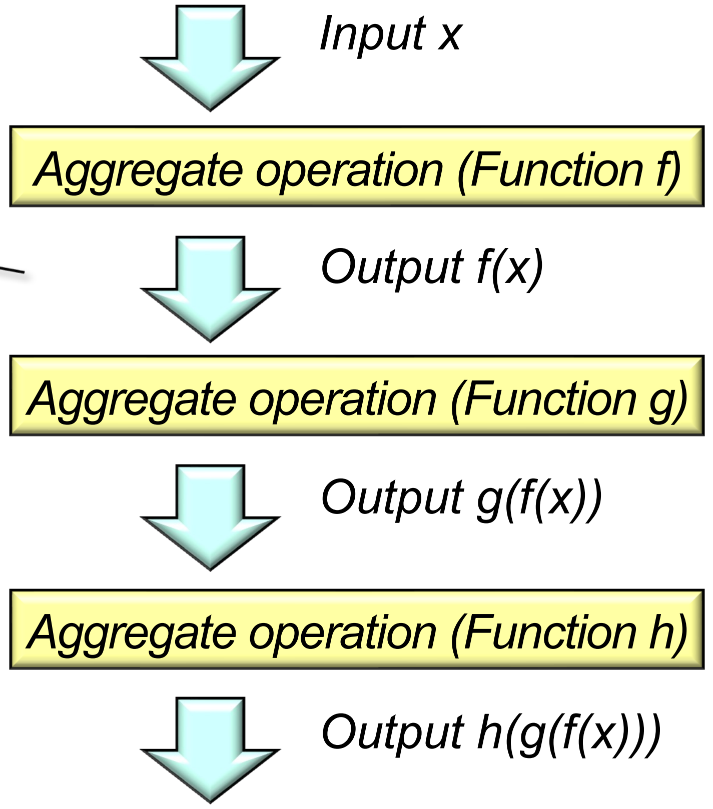


I/O streams are for reading content from a source, or writing the content to a destination

Contrasting Java I/O Streams & Java Streams

- Java I/O streams are different from Java streams!

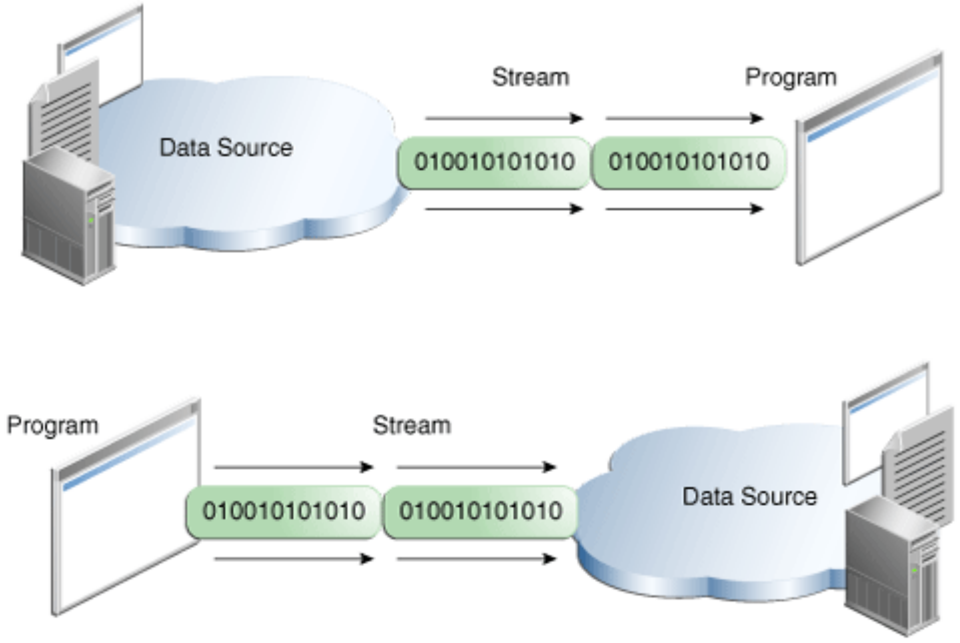
Java streams enable programs to manipulate a collection of data in a declarative way (i.e., functional-style operation)



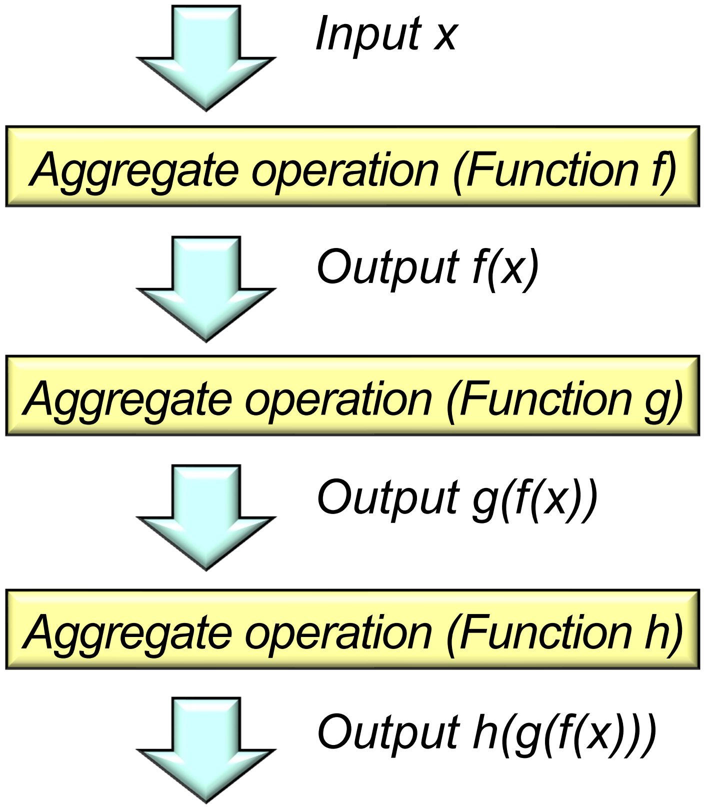
Input x
Aggregate operation (Function f)
Output f(x)
Aggregate operation (Function g)
Output g(f(x))
Aggregate operation (Function h)
Output h(g(f(x)))

Contrasting Java I/O Streams & Java Streams

- Java I/O streams & Java streams can be used together!



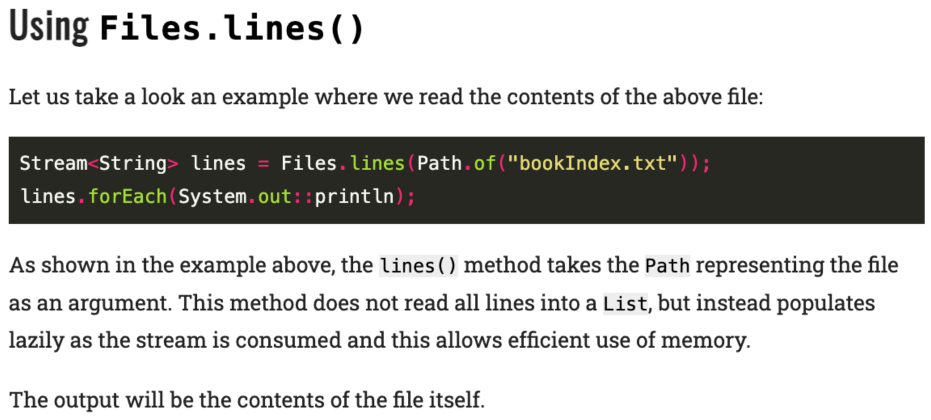
+


Input x
Aggregate operation (Function f)
Output f(x)
Aggregate operation (Function g)
Output g(f(x))
Aggregate operation (Function h)
Output h(g(f(x)))

Combining Java Streams & Java I/O Streams

Combining Java Streams & Java I/O Streams

- Modern Java integrates Java streams & Java I/O streams together nicely!



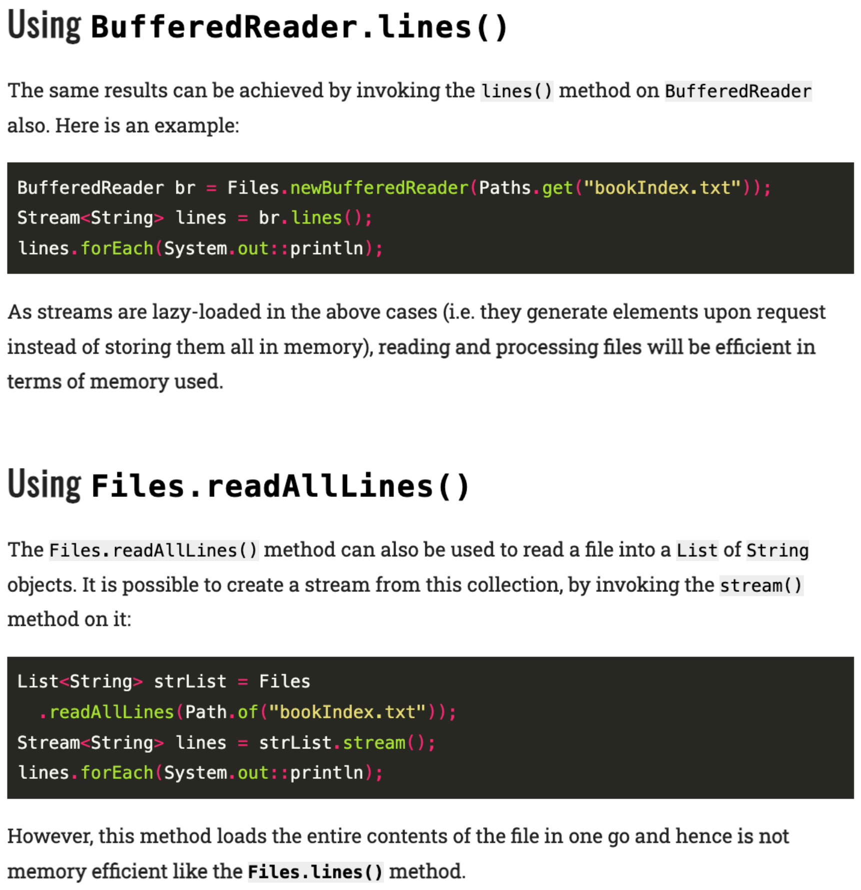
Using `Files.lines()`

Let us take a look an example where we read the contents of the above file:

```
Stream<String> lines = Files.lines(Path.of("bookIndex.txt"));
lines.forEach(System.out::println);
```

As shown in the example above, the `lines()` method takes the `Path` representing the file as an argument. This method does not read all lines into a `List`, but instead populates lazily as the stream is consumed and this allows efficient use of memory.

The output will be the contents of the file itself.



Using `BufferedReader.lines()`

The same results can be achieved by invoking the `lines()` method on `BufferedReader` also. Here is an example:

```
BufferedReader br = Files.newBufferedReader(Paths.get("bookIndex.txt"));
Stream<String> lines = br.lines();
lines.forEach(System.out::println);
```

As streams are lazy-loaded in the above cases (i.e. they generate elements upon request instead of storing them all in memory), reading and processing files will be efficient in terms of memory used.

Using `Files.readAllLines()`

The `Files.readAllLines()` method can also be used to read a file into a `List` of `String` objects. It is possible to create a stream from this collection, by invoking the `stream()` method on it:

```
List<String> strList = Files
  .readAllLines(Path.of("bookIndex.txt"));
Stream<String> lines = strList.stream();
lines.forEach(System.out::println);
```

However, this method loads the entire contents of the file in one go and hence is not memory efficient like the **`Files.lines()`** method.

See reflectoring.io/processing-files-using-java-8-streams

Combining Java Streams & Java I/O Streams

- This program demonstrates how to use modern Java I/O streams & streams to build a cosine vector Map from a CSV file containing movie cosine values

```
try (Stream<String> lines = Files.lines(Paths.get(path))) {
  return lines

    .skip(1)

    .map(line -> line.split(";"))

    .map(s -> new SimpleEntry<>(s[0], parseVector(s[1])))

    .collect(toMap(SimpleEntry::getKey,
                   SimpleEntry::getValue,
                   (x, y) -> x));
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex40

End of Contrasting Java Streams with Java I/O Streams