Principles of Software Construction: Objects, Design, and Concurrency
(Part 3: Design Case Studies)

Java I/O

Christian Kästner **Charlie Garrod**

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 4b due Thursday
- 2nd midterm exam Thursday, March 26th
  - Review session on Tuesday or Wednesday?

Key concepts from last Thursday

Key concepts from last Thursday

- Java Collections
  - Use of design patterns to achieve various design goals
    - Template method, strategy, adapter, decorator, ***iterator***, ***marker interface***, ***factory method***
  - For widespread use:
    - Design for reuse
    - Design for change

Design patterns we have seen so far

Iterator

Composite

Template Method

Façade

Adapter

Strategy

Observer

Marker Interface

Decorator

Model-View-Controller

Factory Method

Learning goals for today

- Understand design aspects of the stream abstractions in Java
- Recognize the underlying design patterns:
  - Adapter
  - Decorator
  - Template Method
  - Marker Interface
  - Iterator

A Java aside

- What is a `byte`?
  - Answer: a signed, 8-bit integer (-128 to 127)
- What is a `char`?
  - Answer: a 16-bit Unicode-encoded character

The I/O design challenge

- Identify a generic and uniform way to handle I/O in programs
  - Reading/writing files
  - Reading/writing from/to the command line
  - Reading/writing from/to network connections
- Reading bytes, characters, lines, objects, …
- Support various features
  - Buffering
  - Encoding (utf8, iso-8859-15, …)
  - Encryption
  - Compression
  - Line numbers
- Refer to files
  - Paths, URLs, symbolic links, directories, files in .jar containers, searching, …

The stream abstraction

- A sequence of **bytes**
- May read 8 bits at a time, and close

`java.io.InputStream`

```
void          close();
abstract int  read();
int           read(byte[] b);
```

- May write, flush and close

`java.io.OutputStream`

```
void          close();
void          flush();
abstract void write(int b);
void          write(byte[] b);
```

The reader/writer abstraction

- A sequence of **characters** in some encoding
- May read one character at a time and close

`java.io.Reader`

```
void          close();
abstract int  read();
int           read(char[] c);
```

- May write, flush and close

`java.io.Writer`

```
void          close();
void          flush();
abstract void write(int c);
void          write(char[] b);
```

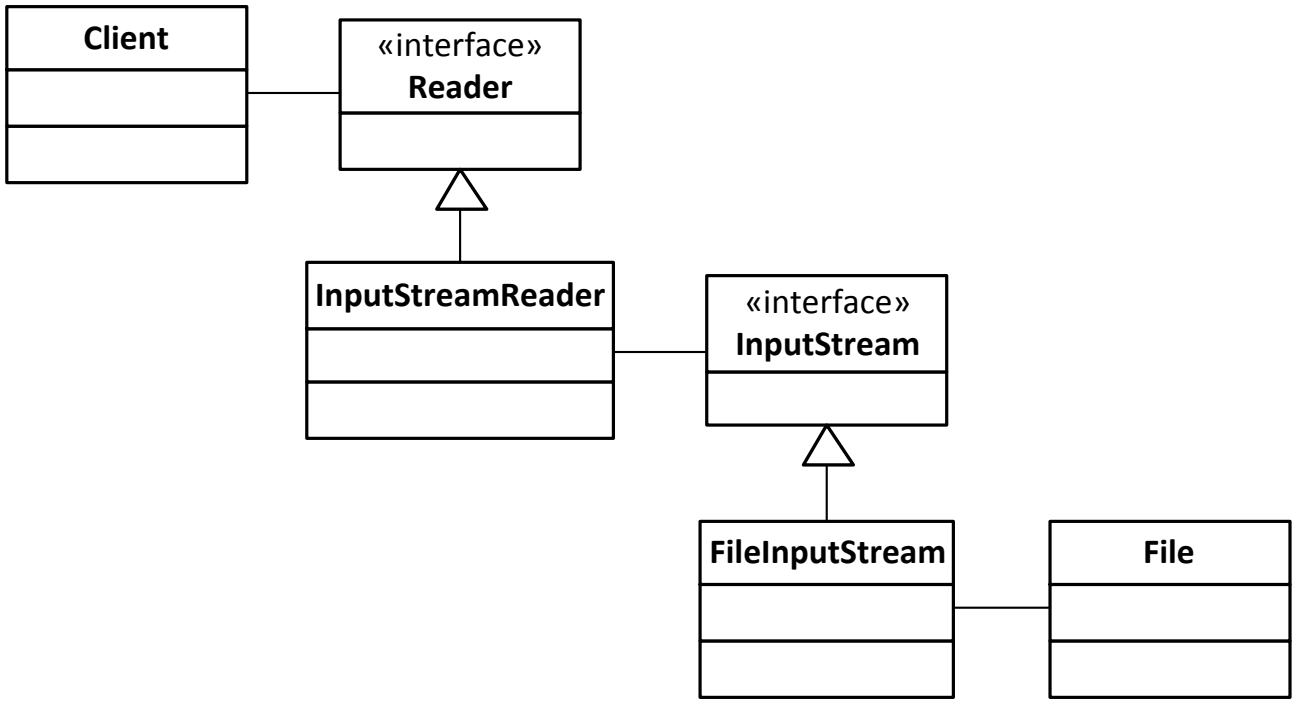
Implementing streams

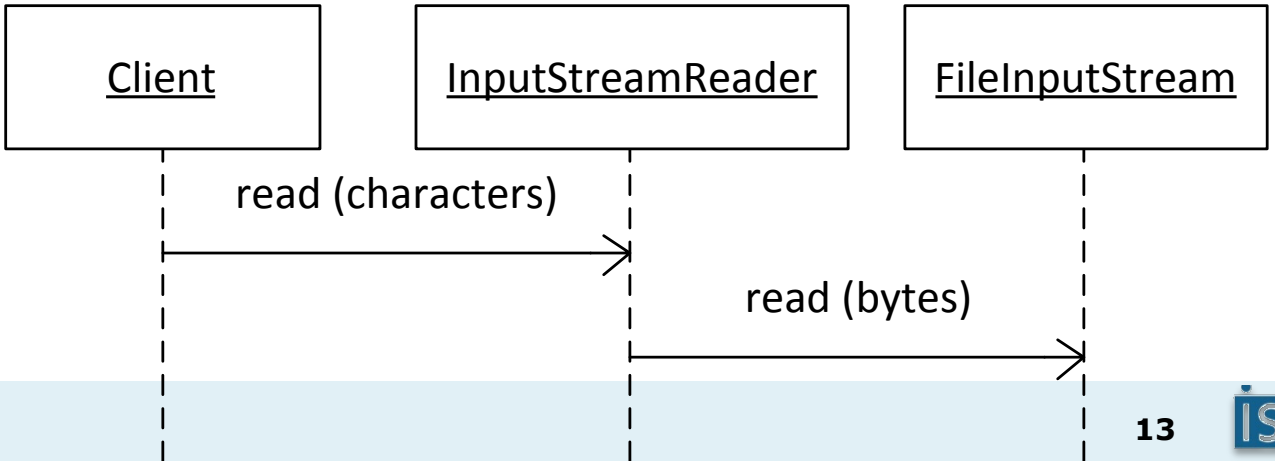
- `java.io.FileInputStream`
  - Reads from files, byte by byte
- `java.io.ByteArrayInputStream`
  - Provides a stream interface for a `byte[]`
- Many APIs provide streams for network connections, database connections, …
  - e.g., `java.lang.System.in`, `Socket.getInputStream()`, `Socket.getOutputStream()`, …

Implementing readers/writers

- `java.io.InputStreamReader`
  - Provides a `Reader` interface for any `InputStream`, adding additional functionality for the character encoding
    - Read characters from files/the network using corresponding streams
- `java.io.CharArrayReader`
  - Provides a `Reader` interface for a `char[]`
- Some convenience classes: `FileReader`, `StringReader`, …

Readers and streams



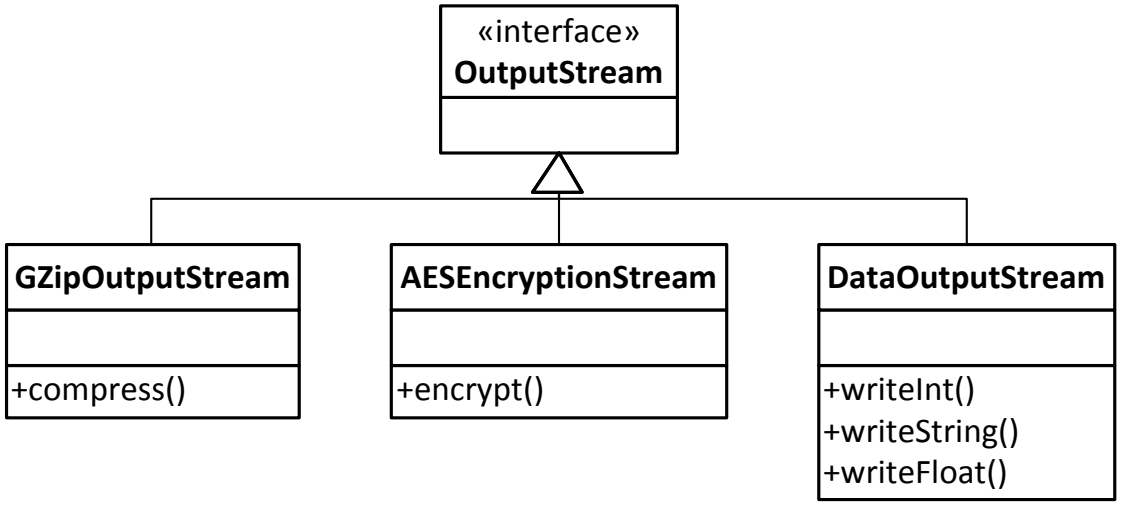


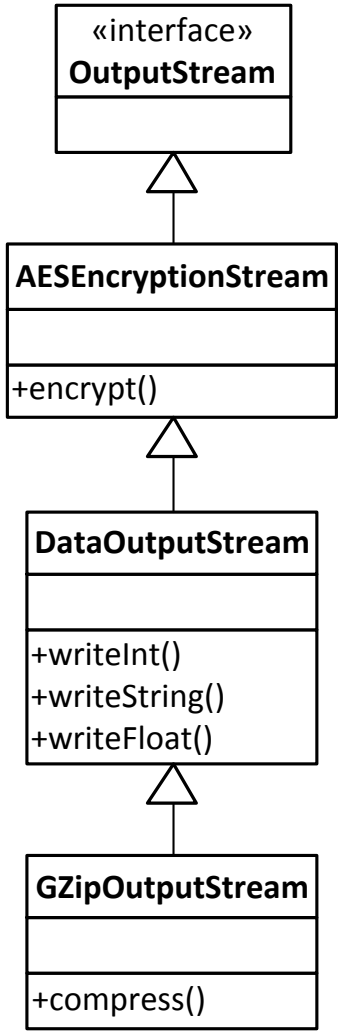
Writers and streams

- See FileExample.java

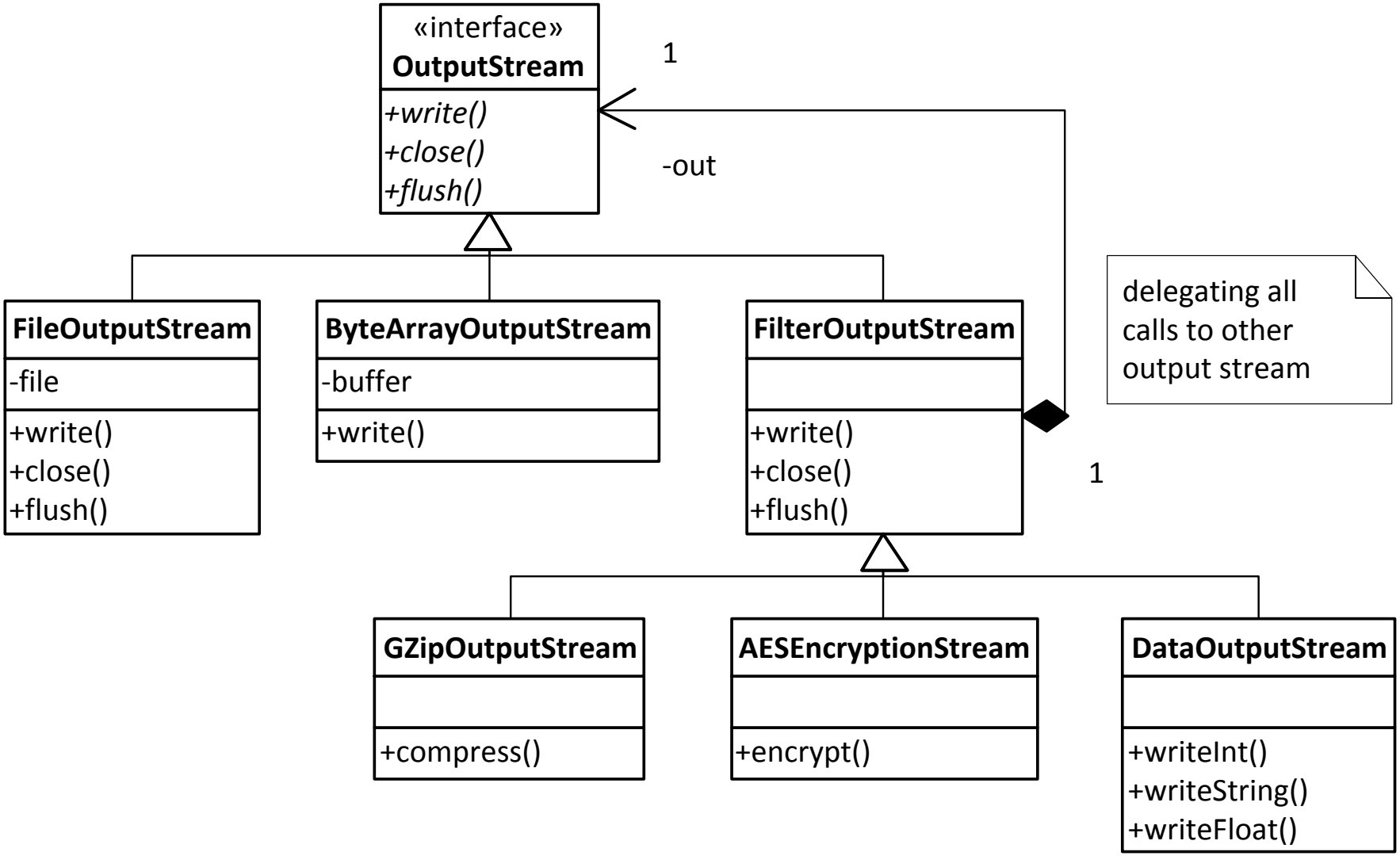
Adding functionality to streams

- E.g. encryption, compression, buffering, reading formatted data such as objects, numbers, lists, …
  - Two possible solutions?:





A better design to add functionality to streams



To read and write arbitrary objects

- Your object must implement the `java.io.Serializable` interface
  - Methods: none
- If all of your data fields are themselves `Serializable`, Java can automatically serialize your class
  - If not, will get runtime `NotSerializableException`
- Can customize serialization by overriding special methods

See QABean.java and FileObjectExample.java

The java.util.Scanner

- Provides convenient methods for reading from a stream

java.util.Scanner:

```
Scanner(InputStream source);
Scanner(File source);
void    close();
boolean hasNextInt();
int     nextInt();
boolean hasNextDouble();
double  nextDouble();
boolean hasNextLine();
String  nextLine();
boolean hasNext(Pattern p);
String  next(Pattern p);
…
```

A challenge for you

- Identify the design patterns in this lecture
  - For each design pattern you recognize, write:
    - The class name
    - The design pattern
    - If you have time: At least one design goal or principle achieved by the pattern in this context
  - Hints:
    - Use the slides online to review the lecture
    - Design patterns include at least:
      - Adapter
      - Decorator
      - Iterator
      - Marker Interface
      - Template Method

Warning: A subtlety of serializability

- Implement `Serializable` judiciously
  - Making a class `Serializable` violates the principle of information hiding
  - (*Effective Java* by Josh Bloch, 2nd edition, p. 274)

Summary

- `java.io` provides general abstractions for streams and readers
  - Standard implementations, convenience implementations
- Many optional features: compression, encryption, object serialization, …
- Convenience and flexibility via the Adapter pattern and Decorator pattern