Basics of programming 3

Java serialization

Serialization basics

- Problem
  - let's save our objects and later read them back
  - what should be stored?
    - objects' attributes
    - associations
    - static fields?
- Simple solution: *serialization*
  - built-in Java feature
  - converts objects' data and their associations to and from byte-streams

Serialization concepts

- Serialization
  - converting objects into binary form (byte stream)
    - also called marshalling, deflating
  - binary data can be stored, transmitted, etc.
- Deserialization
  - converting binary data (byte stream) into objects
    - also called unmarshalling, inflating
  - binary data can be read from a file, got from a network connection, etc.

Serialization example: write

```
import java.io.Serializable;
public class SerializableClass implements Serializable
{ ... }
```

```
//import java.io.*;
...
SerializableClass ser = ...;
try {
        FileOutputStream f =
                        new FileOutputStream(”filename”);
        ObjectOutputStream out =
                        new ObjectOutputStream(f);
        out.writeObject(ser);
        out.close();
}
catch(IOException ex) { ... }
```

Serialization example: read

```
//import java.io.*;
...
SerializableClass ser2;
try {
        FileInputStream f =
                        new FileInputStream("filename");
        ObjectInputStream in =
                        new ObjectInputStream(f);
        ser2 = (SerializableClass)in.readObject();

        in.close();

} catch(IOException ex) {
        ...
} catch(ClassNotFoundException ex) {
        ...
}
```

Rules of serialization

- Only classes implementing interface *Serializable* can be serialized
  - if superclass implements, it's OK
  - arrays, String, Integer, Double, etc. OK
- Interface *Serializable*
  - no methods
  - just formal notification for the compiler
- *Not serializable*
  - *Object, Socket, Input/OutputStream, System, etc.*

Rules of serialization

- What is serialized?
  - primitive attributes
  - serializable attributes
    - recursively
- What is not serialized?
  - static fields
  - *transient* fields

```
public class Serial implements Serializable {
        transient private String secret;
        private String other;
        ...
```

Serialization process

- Serialization is recursive
  - how to avoid cyclic graphs?
  - object's data are written only once
  - consecutively only reference is written

```
out.writeObject(a);
out.writeObject(b);
out.writeObject(a); // only reference!
```

- Serialization of inherited members
  - starts from topmost serializable superclass

Serialization process 2

- Let's have the following classes!

```
class Student implements
    Serializable {
  String name;
  University uni;
  Address a;
  double average;
  // ctr, set, get
}
```

```
class Address implements
    Serializable {
  String country,
    city, street;
  // ctr, set, get
}
```

```
class University implements
    Serializable {
  String name;
  Address a;
  // ctr, set, get
}
```

Serialization process 3

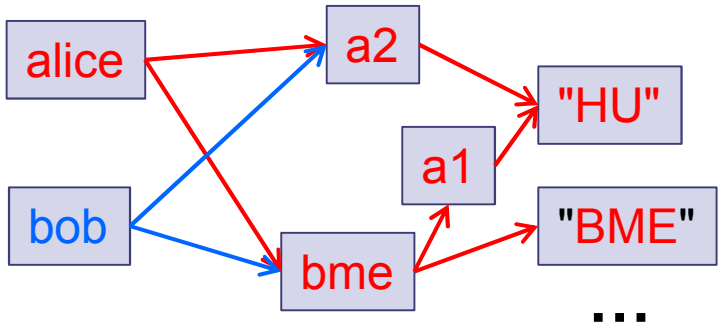
- Let's have the following objects!

```
Address a1 = new Address("HU", "Bp", "Műegyetem rkp 3");
Address a2 = new Address("HU", "Bp", "Alkotás u. 10");

University bme = new University("BME", a1);

Student alice = new Student("Alice", bme,  a2);
Student bob   = new Student("Bob", bme, a2);
```

```
ObjectOutputStream oos = ...;
oos.writeObject(alice);
oos.writeObject(bob);
oos.close();
```



Writing own serialization

- Use

```
private void writeObject(ObjectOutputStream out)
        throws IOException
private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException
```

- Super/subclass data handled automatically
- Default implementation
  - `out.defaultWriteObject(), in.defaultReadObject()`
- *out* and *in* have helper methods
  - reading/writing primitive and serializable types

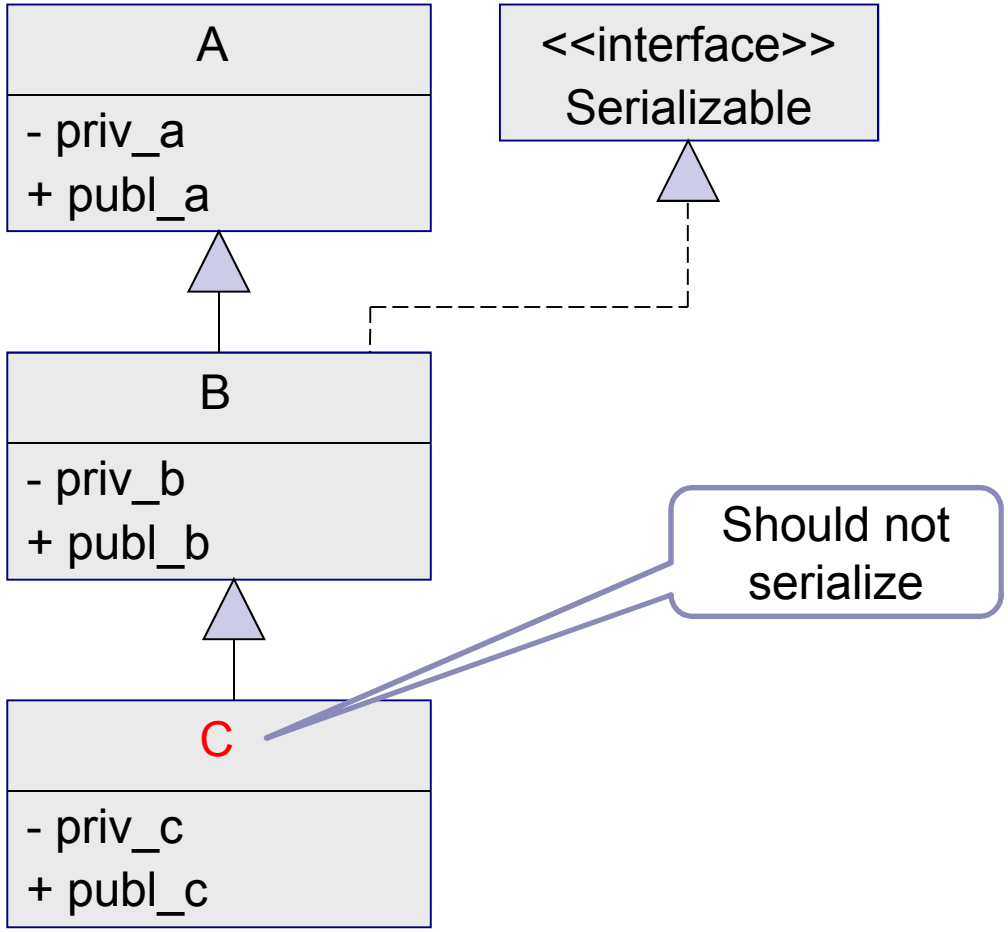
Writing own serialization 2

- For total control implement

```
interface Externalizable extends Serializable {
  public void writeExternal(ObjectOutput out)
    throws IOException;
  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException;
}
```

- Handle super/subclass data explicitly

Stopping serialization



Stopping serialization

```
private void writeObject(ObjectOutputStream out)
throws IOException {

      throw new NotSerializableException("C");
}


private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoundException {

      throw new NotSerializableException("C");
}
```

Versioning

- Each class has a unique version ID
  - showing it: `serialver ClassName`
- Ensuring compatibility
  - use same version ID

```
static final long serialVersionUID
 = 1027553947283749 5L;
```

- Compatible changes
  - method/field addition or access modification change
  - static/transient → persistent

Other serialization formats

- Serialization is low level
  - handling serialized data is hard
- JSON
  - **J**ava**S**cript **o**bject **n**otation
  - http://www.json.org/
- XML
  - own protocol
  - JAXB
    - *javax.xml.bind* package
- …

JSON

- Lightweight data-interchange format
  - easy for humans to read and write
    - mostly…
  - easy for machines to parse and generate
  - caveat: no metadata, extra testing is needed!
- Two structures
  - *Object*: unordered set of name/value pairs
    - like object, struct, etc.
  - *Array*: ordered list of values
    - like array, vector, list, or sequence.

JSON syntax

- Object
  - begins with **{**
  - ends with **}**
  - each name is followed by **:**
  - name/value pairs are separated by **,**

```
{
    "name" : "alice",
    "university" : {
        "name" : "BME"
    },
    "average" : 4.2
}
```

- Array
  - begins with **[**
  - ends with **]**
  - values are separated by **,**

```
[ "mon", "tue", "wed",
  "thu", "fri", "sat",
  "sun"
]
```

JSON syntax 2

- Value
  - *string* in double quotes
  - *number*
  - true or false or null
  - *object*
  - *array*
- Structures can be nested.

```
{
    "name" : "alice",
    "university" : {
        "name" : "BME"
    },
    "average" : 4.2,
    "siblings" : [
        "bob",
        "charlie"
    ],
    "degree" : null
}
```

JSON handling in Java

- Libraries
  - org.json (e.g. in Android)
    - types mapped to classes
    - build element by element
  - javax.json (JEE 7)
    - builder-based building of objects
  - Google gson
  - …

javax.json: creating JSON object

```
// sometimes trivial
// nesting is used

{ "student" : {
    "name" : "alice",
    "university" : {
        "name" : "BME"
    },
    "average" : 4.2
  }
}
```

```
JsonObject value =
  Json.createObjectBuilder()
  .add("name", "alice")
  .add("university",
     json.createObjectBuilder()
     .add("name", "BME")
  )
  .add("average", 4.2)
  .build();

JsonObject student =
  Json.createObjectBuilder()
  .add("student", value).build();
```

javax.json: reading and writing

```
// src can be InputStream or Reader
JsonReader reader = Json.createReader(src);
JsonObject obj = reader.readObject();
// JsonArray arr = reader.readArray();
reader.close();
```

```
// target can be OutputStream or Writer
JsonWriter w = Json.createWriter(target);
JsonObject o = ...;
w.writeObject(o);
w.close();
```

javax.json: accessing data

- In *JsonObject* and *JsonArray* getters for
  - boolean, int, String (also with def. value)
  - JsonArray, JsonNumber, JsonObject, JsonString
  - *isNull*
- Indexing
  - JsonObject: *String*
  - JsonArray: *int*

```
JsonObject obj = ...;
String name = obj.getString("name");
String uni_name =
  obj.getJsonObject("university")
  .getString("name");
```

Java IO Specialties

java.io.File

- Meta-information about files and directories
  - trivial constructors
    - from String or other File objects
    - with path and filename
  - OS dependent info
    - path separator, directory separator
    - don't use "/", "\", ";", ":"
  - file operations
    - delete, create tmp file, access right modification
  - information
    - exists, name, parent, access rights, type, length, content

File example

- List recursively current directory

```
void lsdir(File f, String tab) {

   File[] list = f.listFiles();

   for (int i = 0; i < list.length; i++) {
      System.out.println(tab+list[i].getName());

      if (list[i].isDirectory()) {
         lsdir(list[i], tab+" ");
      }

   }
}
```

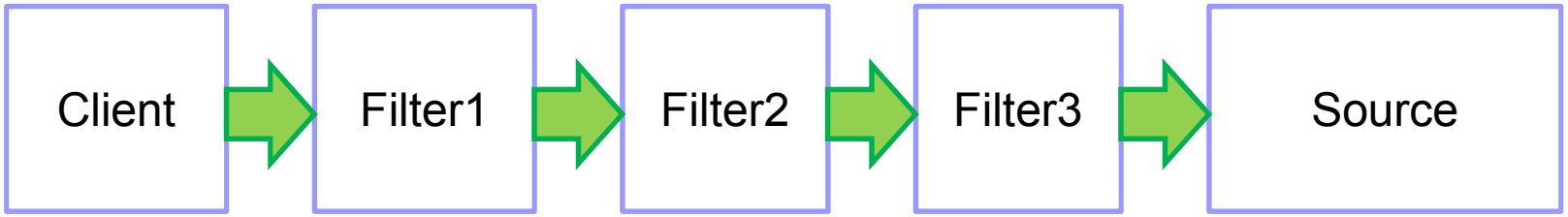
Piped streams

- Filtering is not always convenient
  - e.g. implement unix grep command
    - read lines, print those matching a pattern
    - String.matches helps
- PipedReader, PipedWriter

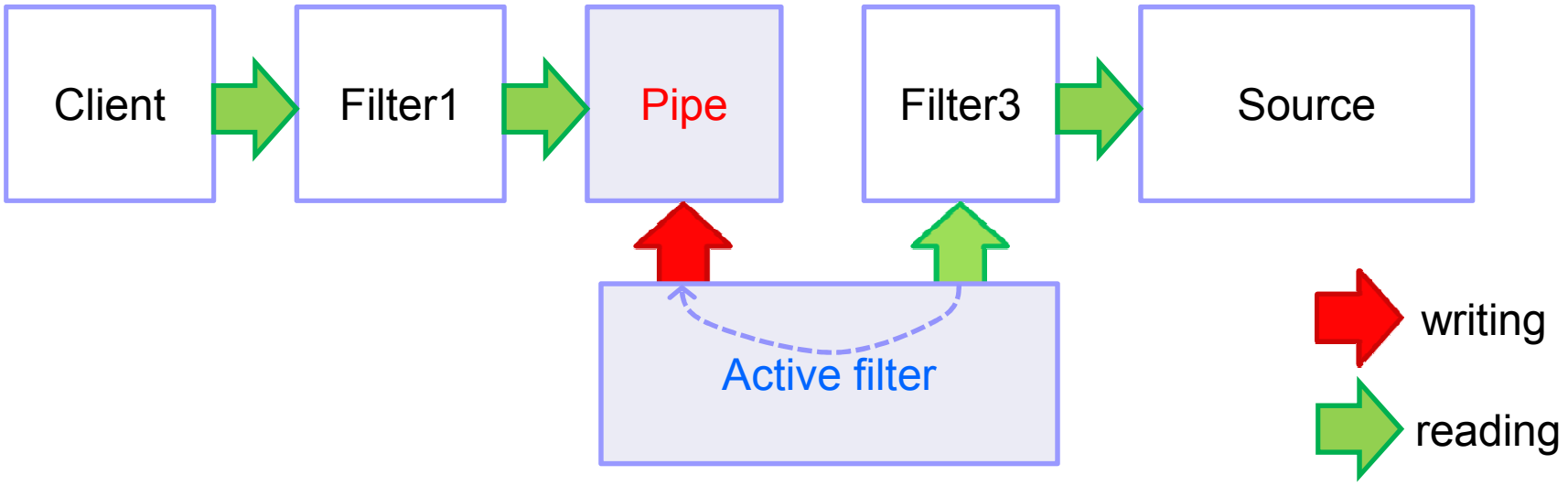
  PipedInputStream, PipedOutputStream
  - two objects are connected
  - what's written into the writer, can be read from the reader

Pipes explained

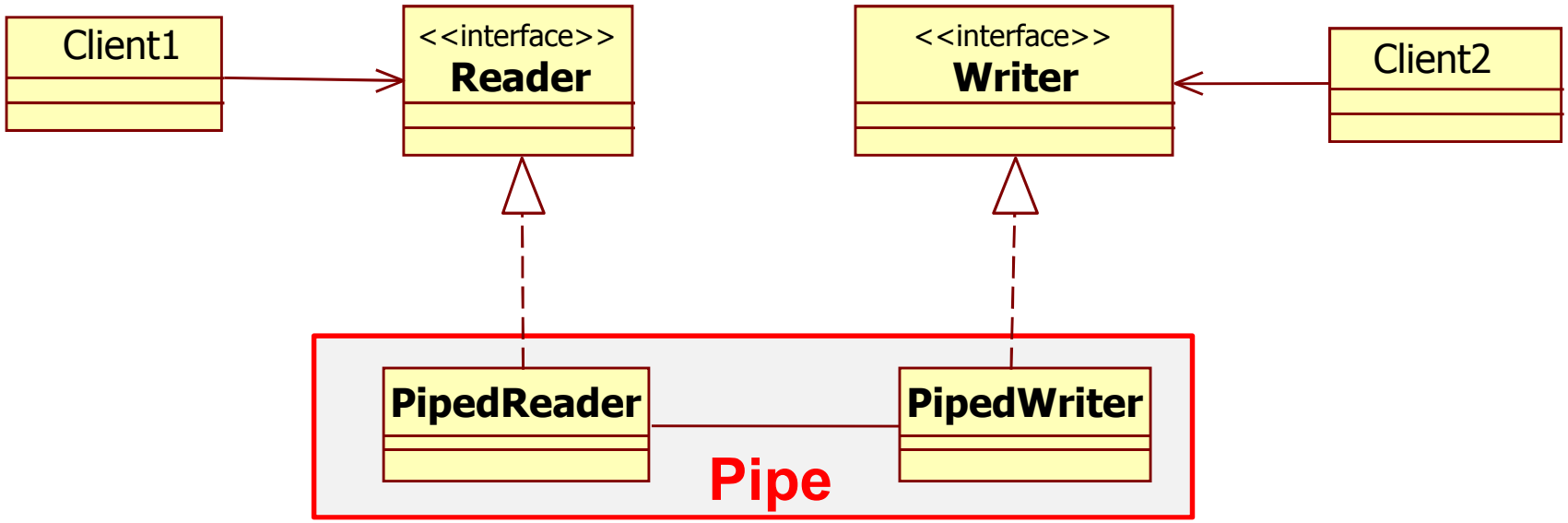
- Filter model



- Pipe model



Pipe classes in detail



Example: *grep*

- Classic unix command
  - filters lines from file or standard input
  - if specified pattern is found in a line, prints the line
- e.g.

```
> grep "apple" words.txt
Newton sat under an apple tree.
An apple was offered to Snow White.
>
```

Grep implementation

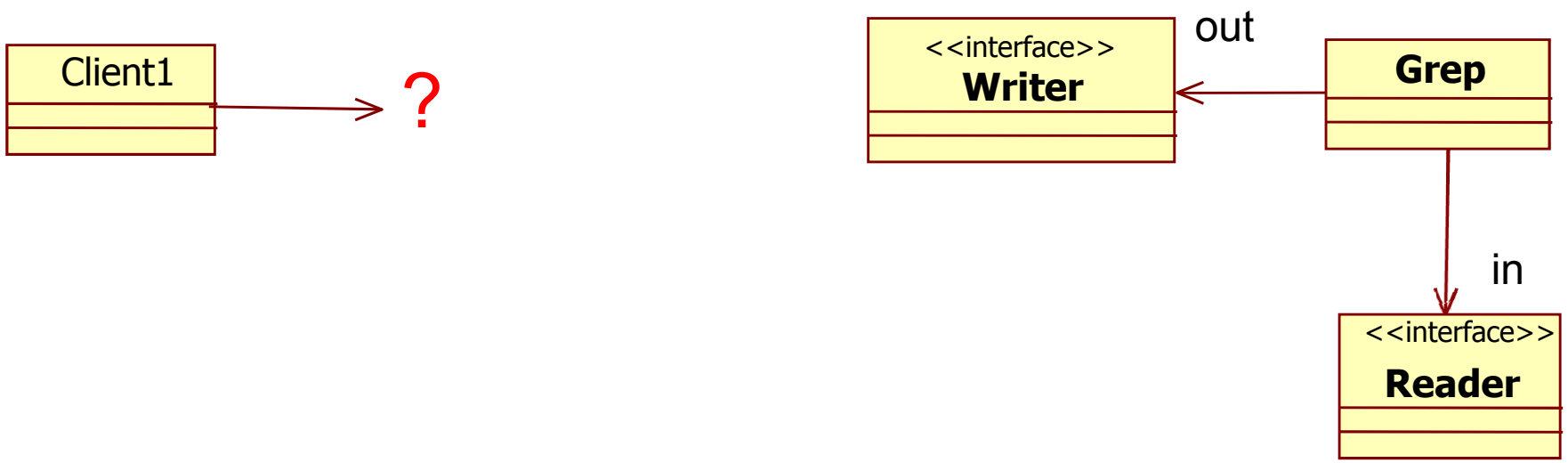
```
// acts as an active filter, hence in and out
public class Grep {
    Reader in; // reading lines from in
    Writer out; // writing lines to out
    String pattern; // the pattern to match lines against

    // constructor
    public Grep(Reader in, Writer out, String pat) {
       this.in = in;
       this.out = out;
       this.pattern = pat;
    }
...
```

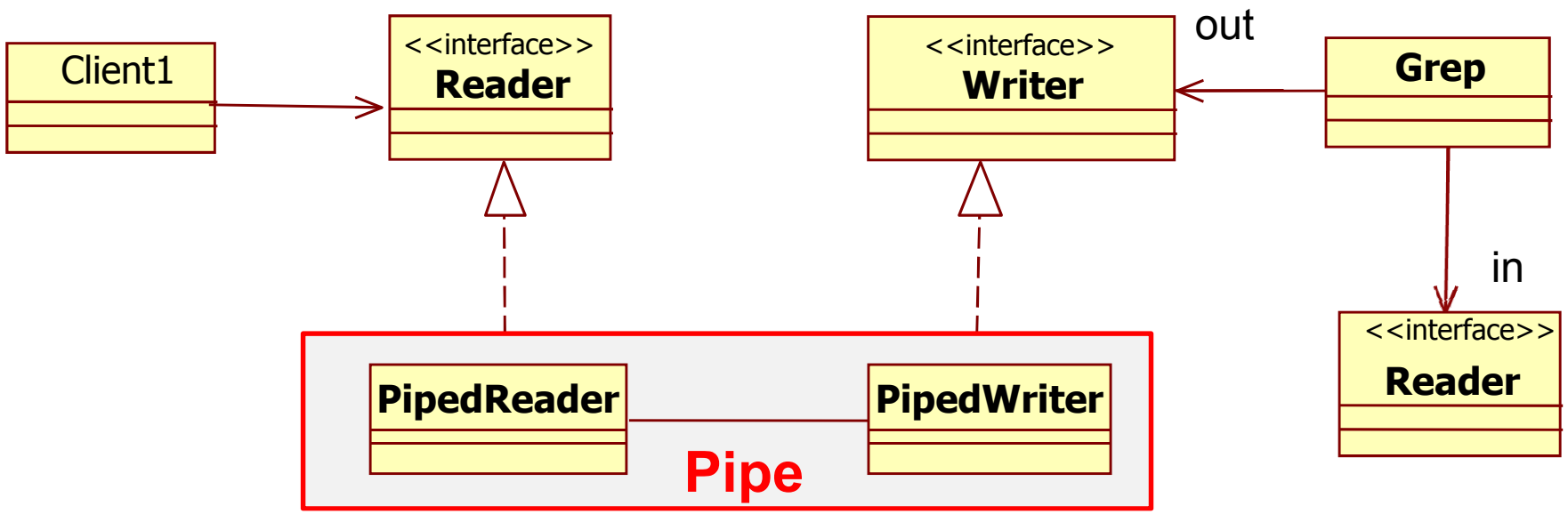
Grep implementation

```
...
        // the active filtering is done here
        public void run() {
                BufferedReader br = new BufferedReader(in);
                PrintWriter pw = new PrintWriter(out);
                while (true) {
                        String line = br.readLine();
                        if (line != null) {
                                if (line.matches(pattern)) {
                                        pw.println(line);
                                }
                        } else break;
                }
                pw.close();
        }
...
```

Grep so far, class diagram



Grep with pipes, class diagram



Grep with pipes, Java

```
...
    // returning a Reader that provides access to the
    // lines written to out
    static public Reader grep(Reader r, String pattern) {
       // creating a pipe
       PipedWriter pw = new PipedWriter();
       PipedReader pr = new PipedReader(pw);
       // the writer end (pw) of the pipe is given to a
       // Grep object (and also the source of lines, r)
       Grep grep = new Grep(r, pw, “hello”);
       // starting grep in the background
       // must actively handle both r and pw
       grep.run();
       return pr;
    }
}
```