Input/Output

Object Oriented Programming

Version 3.2.0 - May 2016

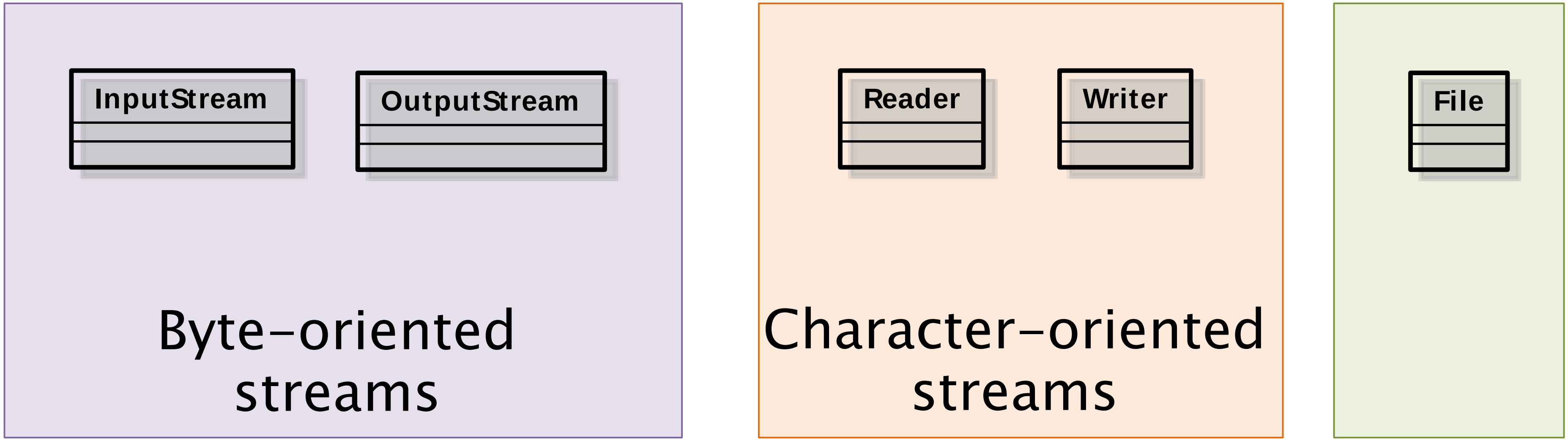
Stream

- All I/O operations rely on the abstraction of stream (flow of elements)
- A stream can be linked to:
    - A file on the disk
    - Standard input, output, error
    - A network connection
    - A data-flow from/to whichever hardware device
- I/O operations work in the same way with all kinds of stream

Stream

- Package: **`java.io`**
- **`Reader`** / **`Writer`**
  - stream of chars (Unicode chars – 16 bit)
    - All characters
- **`InputStream`** / **`OutputStream`**
  - stream of bytes (8 bit)
    - Binary data, sounds, images
- All related exceptions are subclasses of **`IOException`**

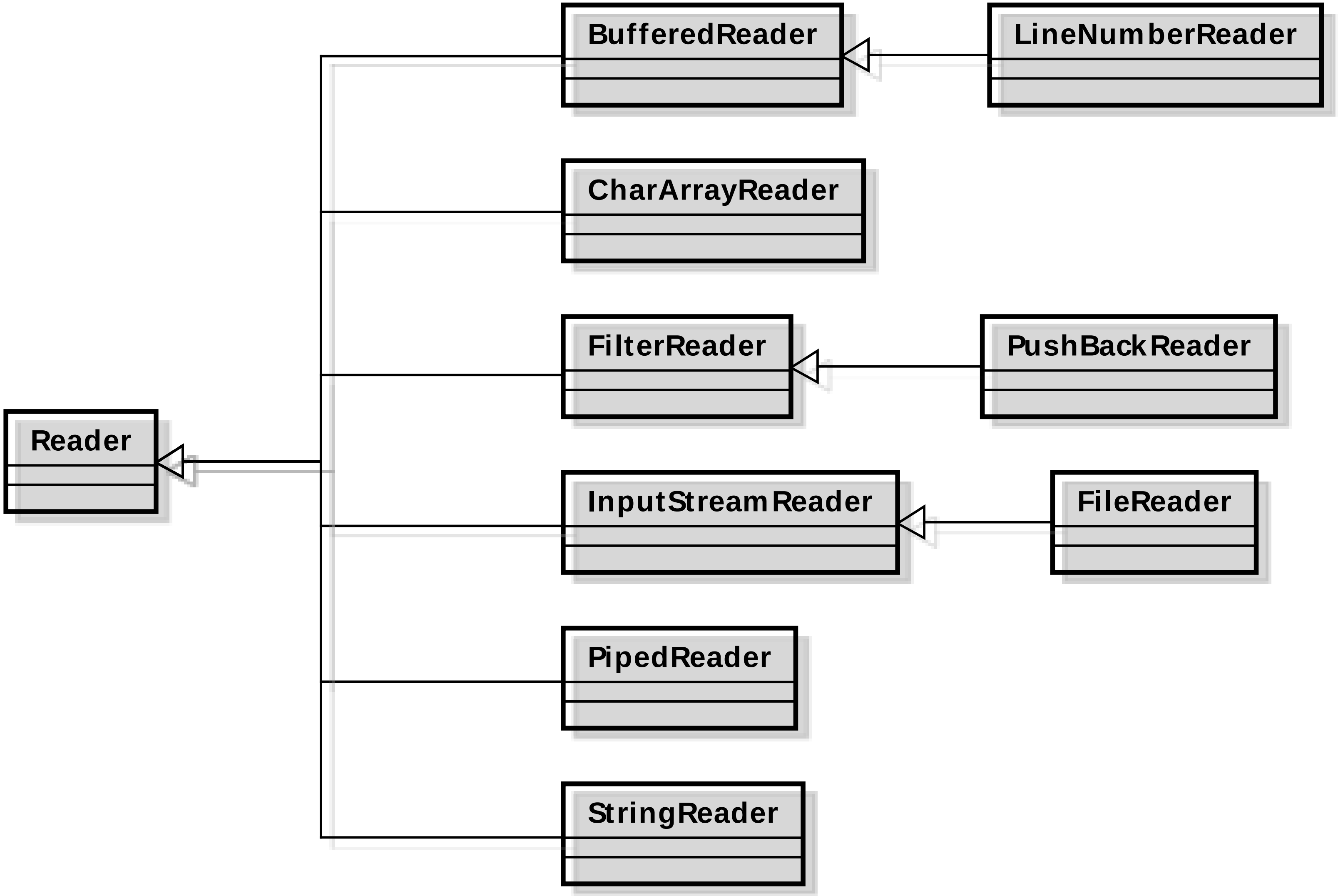
Base classes in `java.io`



Byte-oriented streams

Character-oriented streams

Readers


BufferedReader
LineNumberReader
CharArrayReader
FilterReader
PushBackReader
Reader
InputStreamReader
FileReader
PipedReader
StringReader

`Reader` (abstract)

`void close()`

- Close the stream.

`void mark(int readAheadLimit)`

- Mark the present position in the stream.

`boolean markSupported()`

- Tell whether this stream supports the mark() operation.

`int read()`

- Read a single character:
- Returns -1 when end of stream

`int read(char[] cbuf)`

- Read characters into an array.

`int read(char[] cbuf, int off, int len)`

- Read characters into a portion of an array.

Blocking methods, i.e. stop until
•data available,
•I/O error, or
•end of stream

Reader (abstract)

boolean ready()

- Tell whether this stream is ready to be read.

void reset()

- Reset the stream.

long skip(long n)

- Skip characters.

Read a char

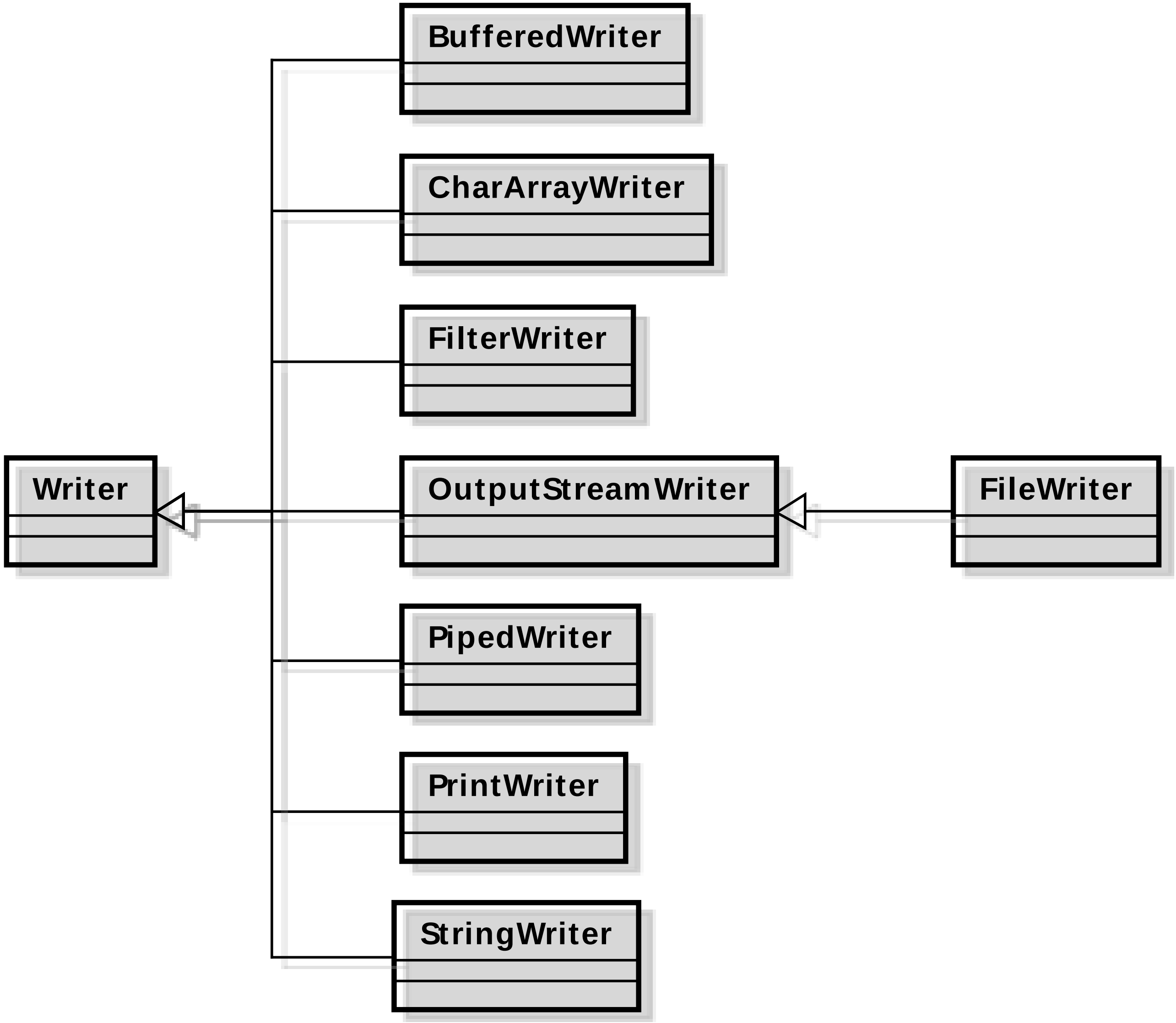
```
int ch = r.read();
char unicode = (char) ch;
System.out.print(unicode);
r.close();
```

Character	ch	unicode
‘A’	$0\ldots00000000\ 01000001_{bin} = 65_{dec}$	65
‘\n’	$0\ldots00000000\ 00001101_{bin} = 13_{dec}$	13
End of file	$1\ldots11111111\ 11111111_{bin} = -1_{dec}$	–

Read a line

```
public static String readLine(Reader r)
throws IOException{
 StringBuffer res= new StringBuffer();
 int ch = r.read();
 if(ch == -1) return null; // END OF FILE!
 while( ch != -1 ){
  char unicode = (char) ch;
  if(unicode == '\n') break;
  if(unicode != '\r’) res.append(unicode);
  ch = r.read();
 }
 return res.toString();
}
```

Writers


BufferedWriter
CharArrayWriter
FilterWriter
Writer
OutputStreamWriter
FileWriter
PipedWriter
PrintWriter
StringWriter

`Writer` (abstract)

`close()`
- close the stream, flushing it first.

`abstract void flush()`
- Flush the stream.

`void write(int c)`
- Write a single character.

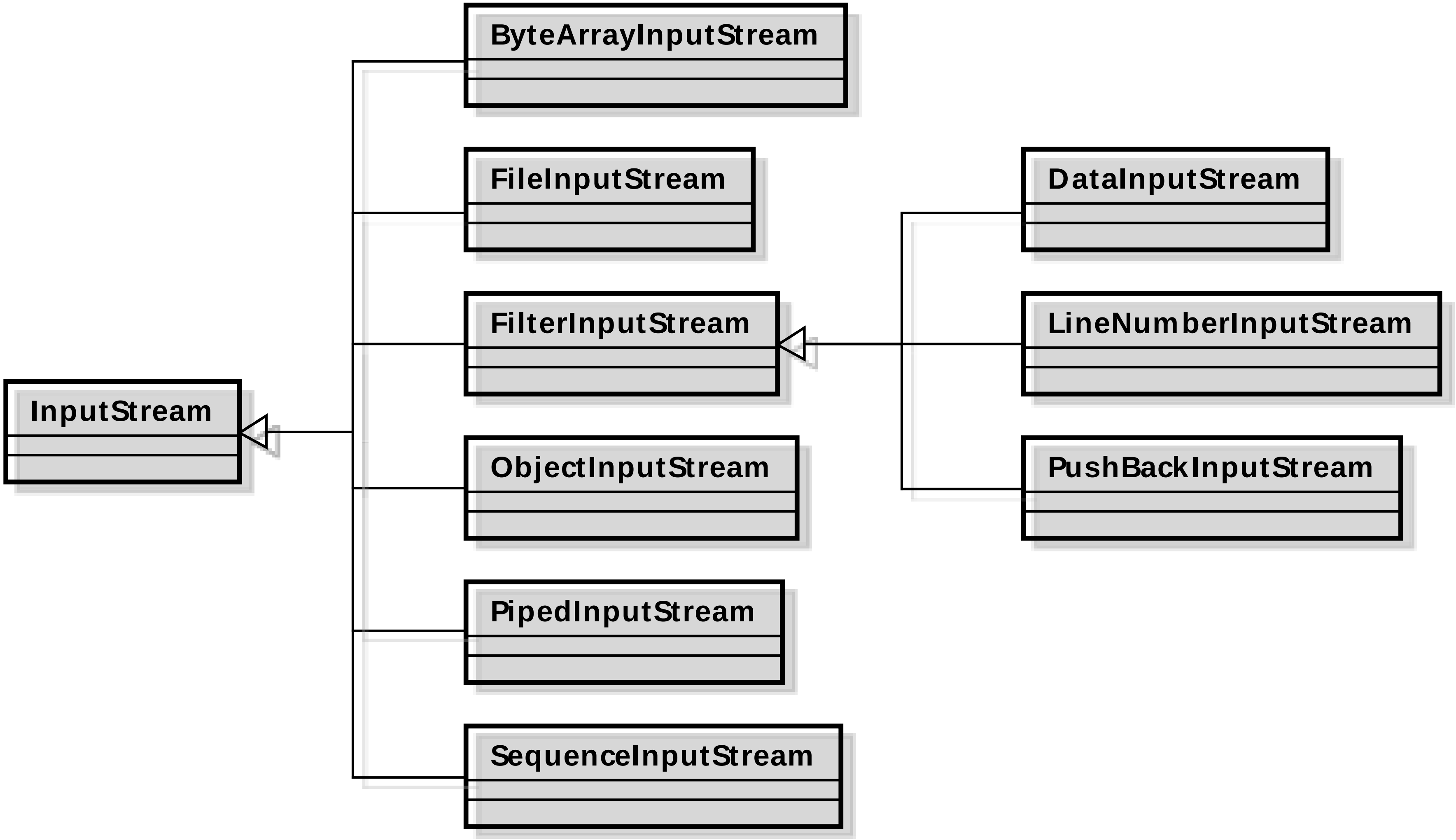
`void write(char[] cbuf)`
- Write an array of characters.

`void write(char[] cbuf, int off, int len)`
- Write a portion of an array of characters.

`void write(String str)`
- Write a string.

- `void write(String str, int off, int len)`
  - Write a portion of a string.

Input streams



InputStream

`void close()`

- Closes this input stream and releases any system resources associated with the stream.

`void mark(int readlimit)`

- Marks the current position in this input stream.

`boolean markSupported()`

- Tests if this input stream supports the mark and reset methods.

`int read()`

- Reads the next byte of data from the input stream.

`int read(byte[] b)`

- Reads some number of bytes from the input stream and stores them into the buffer array b.

`int read(byte[] b, int off, int len)`

- Reads up to len bytes of data from the input stream into an array of bytes.

InputStream

int available()

- Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.

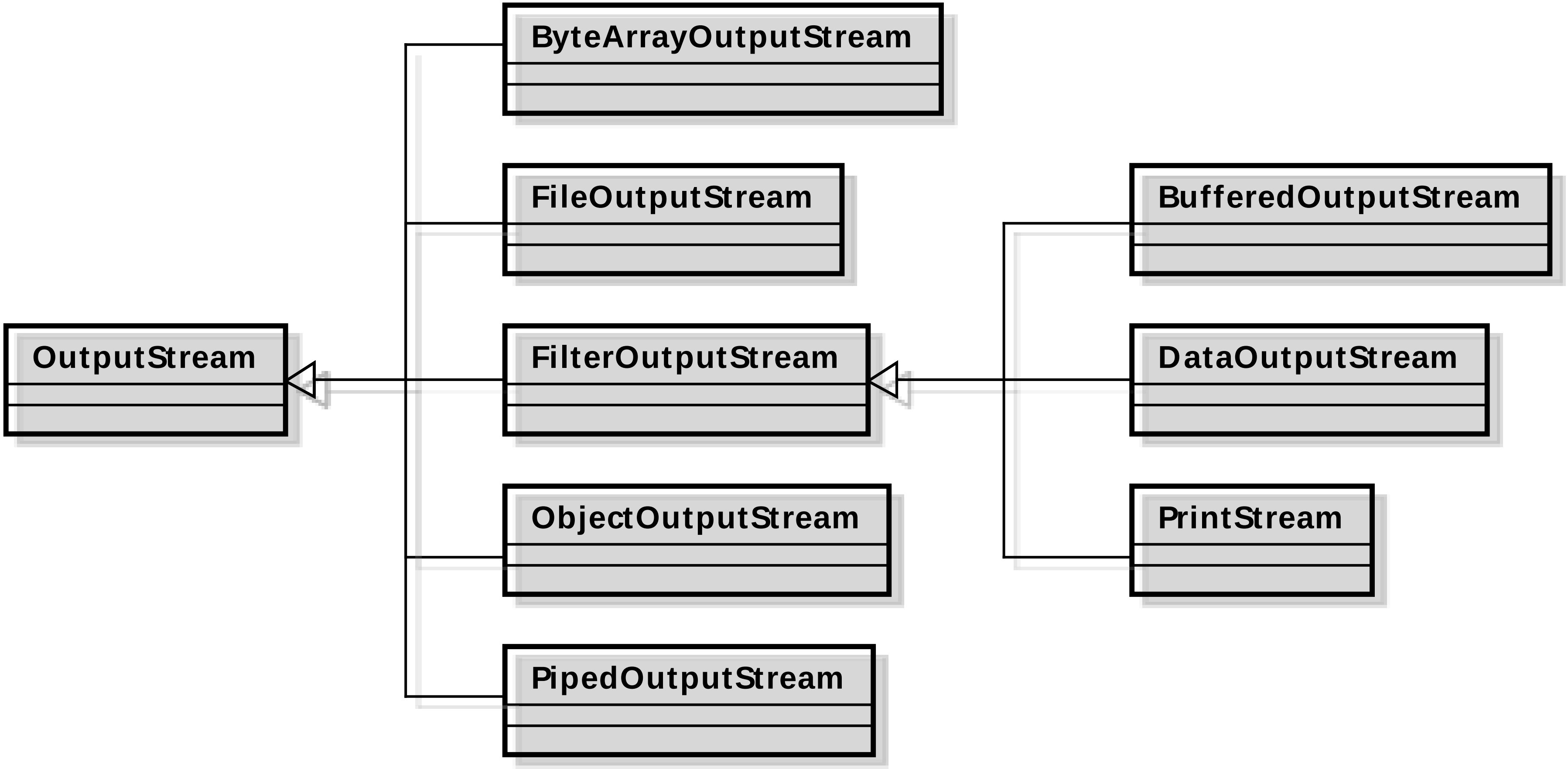
void reset()

- Repositions this stream to the position at the time the mark method was last called on this input stream.

long skip(long n)

- Skips over and discards n bytes of data from this input stream.

Output streams



OutputStream

- **`void close()`**
  - Closes this output stream and releases any system resources associated with this stream.
- **`void flush()`**
  - Flushes this output stream and forces any buffered output bytes to be written out.
- **`void write(byte[] b)`**
  - Writes b.length bytes from the specified byte array to this output stream.
- **`void write(byte[] b, int off, int len)`**
  - Writes len bytes from the specified byte array starting at offset off to this output stream.
- **`void write(int b)`**
  - Writes the specified byte to this output stream.

Stream specializations

- Memory
- Pipe
- File
- Buffered
- Printed
- Interpreted

Conversion byte <-> char

- **InputStreamReader**

  char ← byte

- **OutputStreamWriter**

  char → byte

Read/Write in memory

- **CharArrayReader**
- **CharArrayWriter**
- **StringReader**
- **StringWriter**
  - R/W chars from/to array or String
- **ByteArrayInputStream**
- **ByteArrayOutputStream**
  - R/W byte from/to array in memory

R/W of Pipe

- Pipes are used for inter-thread communication they must be used in connected pairs
- **`PipedReader`**
- **`PipedWriter`**
  - R/W chars from pipe
- **`PipedInputStream`**
- **`PipedOutputStream`**
  - R/W bytes from pipe

R/W of File

- Used for reading/writing files
- **`FileReader`**
- **`FileWriter`**
  - R/W chars from file
- **`FileInputStream`**
- **`FileOutputStream`**
  - R/W byte from file

Copy text file

```
Reader src = new FileReader(args[0]);
Writer dest = new FileWriter(args[1]);
int in;
while( (in=src.read()) != -1){
  dest.write(in);
}
src.close();
dest.close();
```

Higly inefficient!

Copy text file with buffer

```
Reader src = new FileReader(args[0]);
Writer dest = new FileWriter(args[1]);
char[] buffer = new char[4096];
int n;
while((n = src.read(buffer))!=-1){
  dest.write(buffer,0,n);
}
src.close();
dest.close();
```

Buffered

- **BufferedInputStream**

  `BufferedInputStream(InputStream i)`

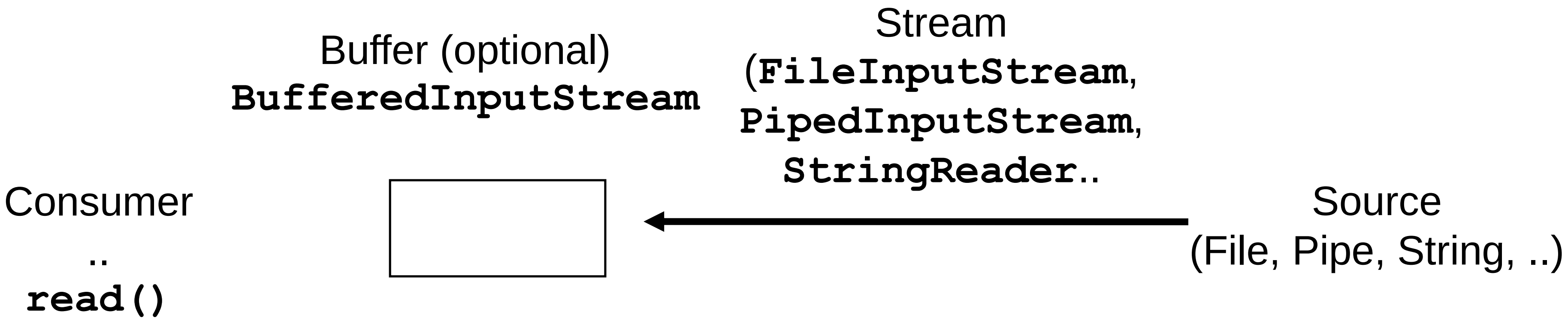
  `BufferedInputStream(InputStream i, int s)`

- **BufferedOutputStream**
- **BufferedReader**

  `readLine()`

- **BufferedWriter**

Buffered input



Buffer (optional)
`BufferedInputStream`

Stream
(**`FileInputStream`**,
`PipedInputStream`,
`StringReader`..

Consumer
..
`read()`

Source
(File, Pipe, String, ..)

```
File in = new File(“in.txt”);
BufferedInputStream b = new BufferedInputStream
                        (new FileInputStream( in));
.. while (b.read != -1 )  b.read();
```

Printed streams

- **`PrintStream`**(OutputStream o)
  - Provides general printing methods for all primitive types, String, and Object
    - **`print()`**
    - **`println()`**
  - Designed to work with basic byte oriented console
  - Does not throw **`IOException`**, but it sets a bit, to be checked with method **`checkError()`**

Standard in & out

- Default input and output streams are defined in class System

```
class System {
   //…
   static InputStream in;
   static PrintStream out;
   static PrintStream err;
}
```

Replacing standard streams

- Default streams can be replaced
  - `setIn()`, `setOut()`, `setErr()`

```
String input = "This is\nthe input\n";
InputStream altInput = new
    ByteArrayInputStream(input.getBytes());
InputStream oldIn = System.in;
System.setIn(altInput);
readLines();
System.setIn(oldIn);
```

Interpreted streams

- Translate primitive types into / from standard format
  - Typically on a file
- **DataInputStream**(InputStream i)
  - readByte(), readChar(), readDouble(), readFloat(), readInt(), readLong(), readShort(), ..
- **DataOutputStream**(OutputStream o)
  - like write()

URLs

- Streams can be linked to URL

```
URL page = new URL(url);
InputStream in = page.openStream();
```

  - Be careful about the type of file you are downloading.

Download file

```
URL home = new URL("http://…");
URLConnection con = home.openConnection();
String ctype = con.getContentType();
if(ctype.equals("text/html")){
  Reader r = new InputStreamReader(
                           con.getInputStream());
  Writer w = new OutputStreamWriter(System.out);
  char[] buffer = new char[4096];
  while(true){
    int n = r.read(buffer);
    if(n==-1) break;
    w.write(buffer,0,n);
  }
  r.close(); w.close();
}
```

Stream as resources

- Streams consume OS resources
  - Should be closed as soon as possible to release resources

```
String readFirstLine(String path)
                         throws IOException{
   BufferedReader br=new BufferedReader(
                     new FileReader(path));
   String l = br.readLine();
   br.close()
   return l
}
```

What happens in case of exception in **readLine** ?

Exception

```
String readFirstLine(String path) throws
IOException {
  BufferedReader br=new BufferedReader(
                     new FileReader(path));
    try {
       String l = br.readLine();
       br.close();
       return l
    } catch(IOException e){
       br.close();
       throw e;
    }
}
```

Complex and does not close in case of **`Error`**

Finally close

```
static String readFirstLine(String path)
throws IOException {
  BufferedReader br=new BufferedReader(
                         new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if(br!=null) br.close();
    }
}
```

Executed in any case before exiting the method

Try-with-resource

- Since Java 7:

```
static String readFirstLine(String path)
throws IOException {
   try(BufferedReader br=new
       BufferedReader(new FileReader(path))){
     return br.readLine();
   }
}
```

Must implement **Autocloseable**

```
public interface AutoCloseable{
   public void close();
}
```

SERIALIZATION

Serialization

- Read / write of an object imply:
  - read/write attributes (and optionally the type) of the object
  - Correctly separating different elements
  - When reading, create an object and set all attributes values
- These operations (serialization) are automated by
  - **`ObjectInputStream`**
  - **`ObjectOutputStream`**

Using Serialization

- Methods to read/write objects are:

```
void writeObject(Object)
Object readObject()
```

- ONLY objects implementing interface **Serializable** can be serialized
  - ♦ This interface is empty

  ⇒Just used to avoid serialization of objects, without permission of the class developer

Type recovery

- When reading, an object is created
- ... but which is its type?
- In practice, not always a precise downcast is required:
  - Only if specific methods need to be invoked
  - A downcast to a common ancestor can be used to avoid identifying the exact class

Saving Objects with references

- Serialization is applied recursively to object in references
- Referenced objects must implement the **`Serializable`** interface
- Specific fields can be excluded from serialization by marking them as **`transient`**

Saving Objects with references

- An **`ObjectOutputStream`** saves all objects referred by its attributes
  - objects serialized are numbered in the stream
  - references are saved as ordering numbers in the stream
- If two saved objects point to a common one, this is saved just once
  - Before saving an object, **`ObjectOutputStream`** checks if it has not been already saved
  - Otherwise it saves just the reference

Serialization

```
public class Student
implements Serializable {…}
```

```
List<Student> students=new LinkedList<>();
students.add( ... );
...
ObjectOutputStream serializer =
       new ObjectOutputStream(
           new FileOutputStream(”std.dat"));
serializer.writeObject(students);
serializer.close();
```

```
ObjectInputStream deserializer =
           new ObjectInputStream(
              new FileInputStream("std.dat"));
Object retrieved = deserializer.readObject();
deserializer.close();
List<Student> l = (List<Student>)retrieved;
```

FILE

File

- Abstract pathname
  - directory, file, file separator
  - absolute, relative
- convert abstract pathname <--> string
- Methods:
  - **`create() delete() exists() , mkdir()`**
  - **`getName() getAbsolutePath(), getPath(), getParent(), isFile(), isDirectory()`**
  - **`isHidden(), length()`**
  - **`listFiles(), renameTo()`**

Example: list files

- List the files contained in the current working folder

```
File cwd = new File(".");
for(File f : cwd.listFiles()){
  System.out.println(f.getName()+ " "
                     + f.length());
}
```

REGULAR EXPRESSIONS

Tokenizers

- **`StringTokenizer`**
  - Works on String
  - set of delimiters (blank, “,”, \t, \n, \r, \f)
  - Blank is the default delimiter
  - Divides a String in tokens (separated by delimiters), returning the token
  - **`hasMoreTokens()`**, **`nextToken()`**
  - Does not distinguish identifiers, numbers, comments, quoted strings

Tokenizers

- **`StreamTokenizer`**
  - Works on Stream (Reader)
  - More sophisticated, recognizes identifiers, comments, quoted string, numbers
  - use symbol table and flag
  - **`nextToken()`**, TT_EOF if at the end

Splitting text into tokens

- `String`
  - `public String[] split(String regex)`
  - Returns the array of strings computed by splitting this string around matches of the given regular expression
  - The regular expression `\\s` specifies a whitespace character: [\t \n \x0B \f \r]

Regular Expressions

- Represent a simple and efficient way to describe sets of character strings
- Operators allow representing:

characters	`c`
classes of characters	`[abc]` o `[a-c]`
optionality	*`exp`* `?`
repetition (0 o more)	*`exp`* `*`
repetition (1 o more)	*`exp`* `+`
alternatives	*`exp1`* `|` *`exp2`*
concatenation	*`exp1 exp2`*
grouping	`(` *`exp`* `)`

Examples of RE

- Positive integer number
  - `[0-9]+`
- Positive integer number w/o leading 0
  - `[1-9][0-9]*`
- Integer number positive or negative
  - `[+-]?[0-9]+`
- Floating point number

  -
    ```
    [+-]?(([0-9]+\.[0-9]*)|
                 ([0-9]*\.[0-9]+))
    ```

Regular expressions

- RE can be used to check whether an input string correspond to a given set
- RE describe sequence of characters and use a set of operators:
  - `" \ [ ] ^ - ? . * + | ( ) $ / { } % < >`
- Letters and numbers in the input text are described by themselves
  - **`val1` represents the sequence `‘v’ ‘a’` `‘l’ ‘1’`** in the input text

Regular expressions

- Non alphabetic characters must be preceded by the quotation character \
  - **`xyz\+\+`** represents the sequence **`‘x’ ‘y’ ‘z’ ‘+’ ‘+’`** in the input text
- Classes of characters are described by means of the operator `[]`:
  - `[0123456789]` represents a any number in the input text.
- When describing a class the symbol – indicates a range of characters:
  - `[0-9]` represents any numeric character in the input text.

Regular expressions

- To include the special character – in a class, it must be specified as the first or last in the sequence:
  - `[-+0-9]` **represents a number in the input text.**
- In the class description a ^ placed at the beginning of the character list indicates the characters to be excluded:
  - `[^0-9]` represents any non numeric character in the input text
- The set of all characters except new line can be described by a dot: “.”

Regular expressions

- The end-of-line is represented by `\n`.
- Any white space is described by `\s`.
- The operator `?` makes the preceding expression **optional**:
  - `ab?c` represents both `ac` and `abc`.
- The operator `*` indicates the preceding expression can be repeated 0 or more times:
  - `ab*c` represents all the sequences starting by `a`, terminating by `c`, and containing inside any number of `b`.

Regular expressions

- The operator **+** makes the preceding expression can be repeated 1 or more times:
  - **ab+c** represents all the sequences starting by **a**, terminating with **c**, and containing inside at least one **b**.
- The operator **|** represents an alternative between two expressions:
  - **ab|cd** represnts both the sequence **ab** and the sequence **cd**.
- The round parentheses allow expressing a grouping to define the priorities among operators
  - **(ab|cd+)?ef** represents such sequences as **ef**, **abef**, **cdddef**.

Recognizer

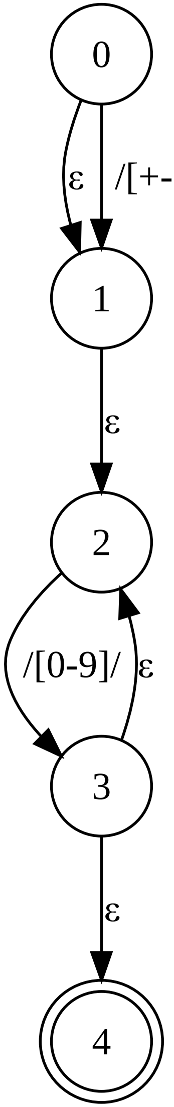
- An RE can be transformed into NFA (Non-deterministic Finite-state Automata)
  - Algorithm Thompson-McNaughton-Yamada
- Then an NFA can be transformed into a DFA (Deterministic)
- A DFS is encoded in a table that can be easily executed to recognize a sequence of characters

Recognizer example

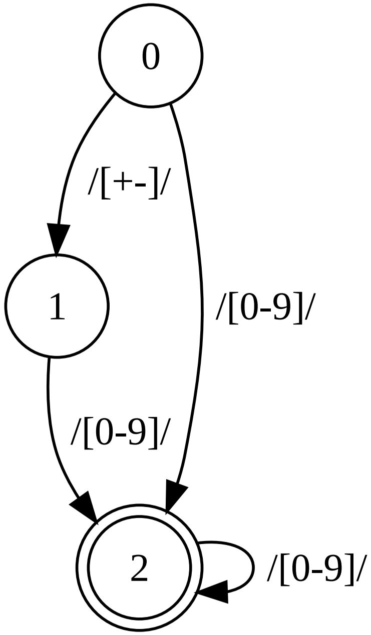
RE

`[+-]?[0-9]+`

NFA



DFA



http://hackingoff.com/compilers/regular-expression-to-nfa-dfa

RegExp in Java

- Package
  - `java.util.regex`
- **`Pattern`** represents the automata:

```
Pattern p=Pattern.compile("[+-]?[0-9]+");
```

- **`Matcher`** represents the recognizer

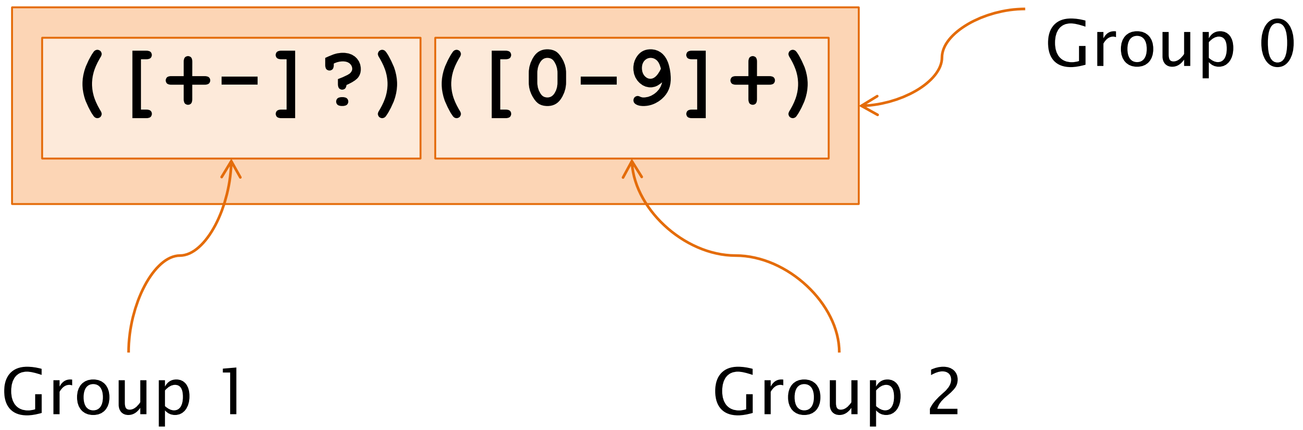
```
Matcher m = p.matcher("-4560");
boolean b = m.matches();
```

Matcher

- Three recognition modes
  - **`matches()`**
    - Attemp matching the whole string
  - **`lookintAt()`**
    - Attempt a partial matching starting from beginning
  - **`find()`**
    - Attempt matching any substring
- Recognized string:
  - **`group()`**

Capture groups

- Every pair of parentheses defines a capture group
  - Group 0 for the whole matched string



  - Non capturing group: `(?:E)`

Capture groups

```
m = p.matcher("-4560");
if(m.matches()){
  for(int i=0; i<=m.groupCount(); ++i){
    System.out.println("Group "+i+" : '"
                              + m.group(i) + "'");
  }
}
```

```
Group 0 : '-4560'
Group 1 : '-'
Group 2 : '4560'
```

CSV

- A comma-separated values (CSV) file stores tabular data (numbers and text) in plain text
- Each line of the file is a data record
- Each record consists of one or more fields, separated by delimiters
  - typically a single reserved character such as comma, semicolon, or tab
- It is used to import/export a table of data
  - i.e. from a spread-sheet or a database

Example CSV – Capture groups

```
\s*("([^"]*|"")*"|[^",]*)\s*(,|$)
```

```
\s*("(([^"]*|"")*)"|([^",]*))\s*(,|$)
```

Group 2 Group 4

- Pay attention to special characters
  - Backslash: \
  - Quotes: "

```
"\\s*(\"(([^\"]*|\"\")*)\"|([^\";]*))\\s*(;|$)"
```

Matches also Ɛ (empty string) at the end of the line

Example: CSV

```
Matcher m = p.matcher(line);
while(m.find()){
  String cell=m.group(2);
  if(cell!=null){
    cell=cell.replaceAll("\"\"", "\"");
  }else{
    cell = m.group(4);
  }
  System.out.println("content:" + cell);
}
```

May detect a spurious cell at the end of the line

Context

- Look-behind
  - `(?<=E)` means that **E** must precede the following RE, though E is not part of the recognized RE
  - `(?<!E)` means **E** must **not** precede
- Look-ahead
  - `(?=E)` means that **E** must follow the preceding RE, though E is not part of the recognized RE
  - `(?!E)` means that **E** must **not** follow

Example CSV – Context

- Java quoted RE:

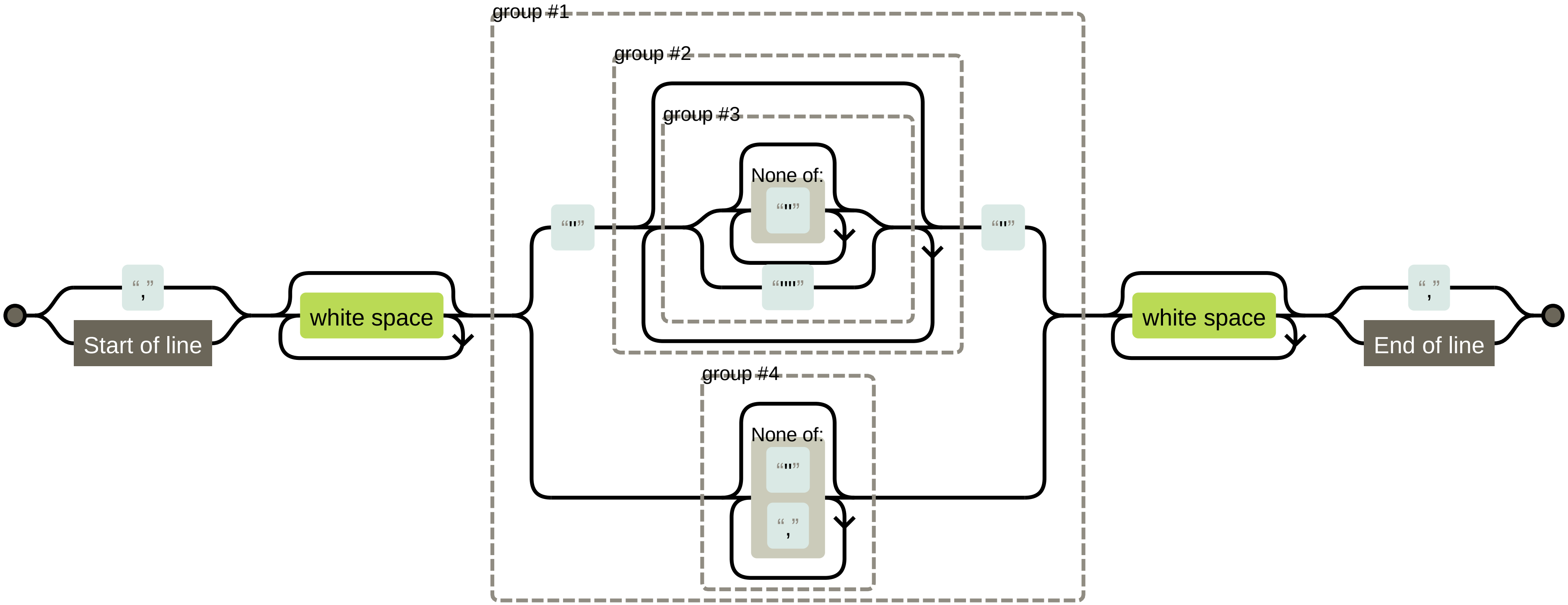
```
"(?<=,|^)\\s*(\"(([^\"]*|\"\")*)\"|([^\",]*))\\s*(,|$)"
```

- Unquoted RE:

```
(?<=,|^)      // look-behind context
\s*             // leading spaces
("(([^"]*|"")*)"  // quoted cell
|([^\",]*))     // normal cell
\s*           // trailing spaces
(?=,|$)       // look-ahead
```

Example CSV – Context

- Railroad diagram



Generated with: http://regexper.com

Named groups

- Capture groups can be named:
  - E.g. `(?<c>[^\",]*)`

- Named groups can be accessed using `group()` method:
  - E.g. `c = m.group("c");`

Example CSV – Named groups

- Java quoted RE:

```
"(?<=,|^)\\s*(?<qc>\"(([^\"]*|\"\")*)\"|(?<c>[^\",]*))\
\s*(,|$)"
```

- Unquoted RE:

```
(?<=,|^)                // look-behind context
\s*                     // leading spaces
("(?<qc>([^"]*|"")*)" // quoted cell
|(?<c>[^\",]*))// normal cell
\s*                     // trailing spaces
(?=,|$)                 // look-ahead
```

Scanner

- A basic parser that can read primitive types and strings using regular expressions
- Basic usage
  - Construction from a stream, file, or string
    - E.g. `new Scanner(new File("file.txt"))`
  - Check present of *next* token (optional)
    - E.g. `hasNextInt()`
  - Detection of *next* token:
    - E.g. `nextInt()`

Scanner

- Advanced use

```
try(Scanner fs = new Scanner(file)){
while(true){
  String c;
  while((c=fs.findInLine(pattern))!=null){
    System.out.println(c);
  }
  if(!fs.hasNextLine()) break;
  fs.nextLine();
}}
```

Summary

- Java IO is based on the stream abstraction
- Two main stream families:
  - Char oriented: Reader/Writer
  - Byte oriented: Input/OutputStream
- There are streams specialized for
  - Memory, File, Pipe, Buffered, Print

Summary

- Streams resources need to be closed as soon as possible
  - Try-with-resource construct guarantee resource closure even in case of exception
- Serialization means saving/restoring objects using Object streams
  - `Serializable` interface enables it

Summary

- Regular expression express complex sequences of characters
- Used to recognize parts of strings
  - `Pattern` contains the DFA
  - `Matcher` implements the recognizer
- RE are used extensively
  - String: `replaceAll()`, `split()`
  - Scanner: `findInLine()`