

Input/Output



Object Oriented Programming

<http://softeng.polito.it/courses/09CBI>



SoftEng
<http://softeng.polito.it>

Version 4.2.1

© Marco Torchiano, 2021



This work is licensed under the Creative Commons Attribution–NonCommercial–NoDerivatives 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-nd/4.0/>.

You are free: to copy, distribute, display, and perform the work

Under the following conditions:



Attribution. You must attribute the work in the manner specified by the author or licensor.



Non-commercial. You may not use this work for commercial purposes.



No Derivative Works. You may not alter, transform, or build upon this work.

For any reuse or distribution, you must make clear to others the license terms of this work.

- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

I/O Stream

- All I/O operations rely on the abstraction of **stream** (flow of elements)
- A I/O 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

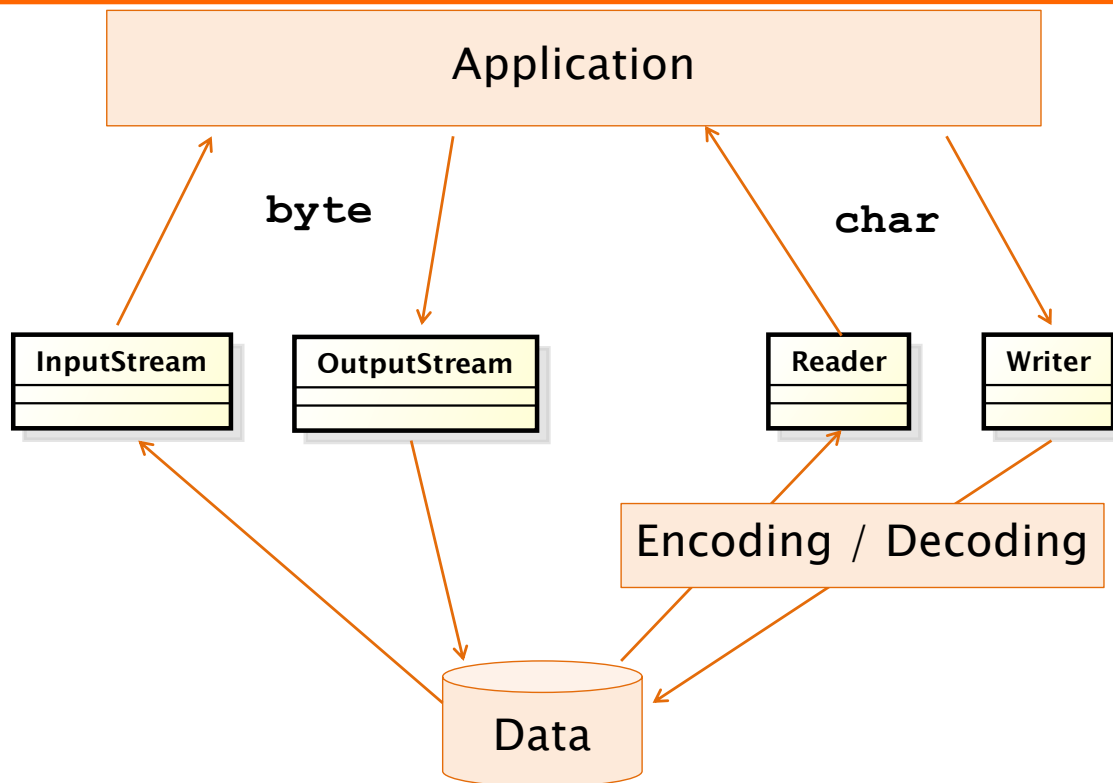
3

I/O Stream

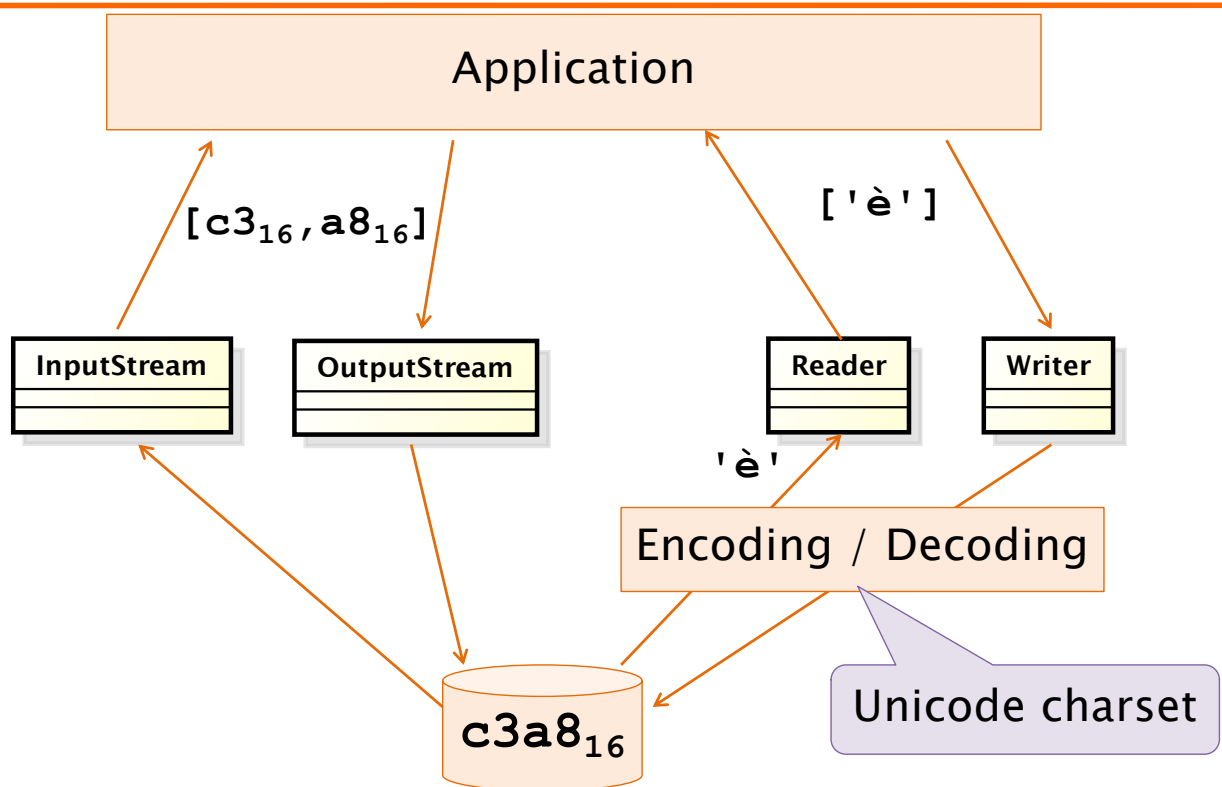
- Package: **java.io**
- Stream of **chars** (Unicode – 16 bit)
 - ♦ **Reader / Writer**
 - All characters
- Stream of **bytes** (8 bit)
 - ♦ **InputStream / OutputStream**
 - Binary data, sounds, images
- All related exceptions are subclasses of **IOException**

4

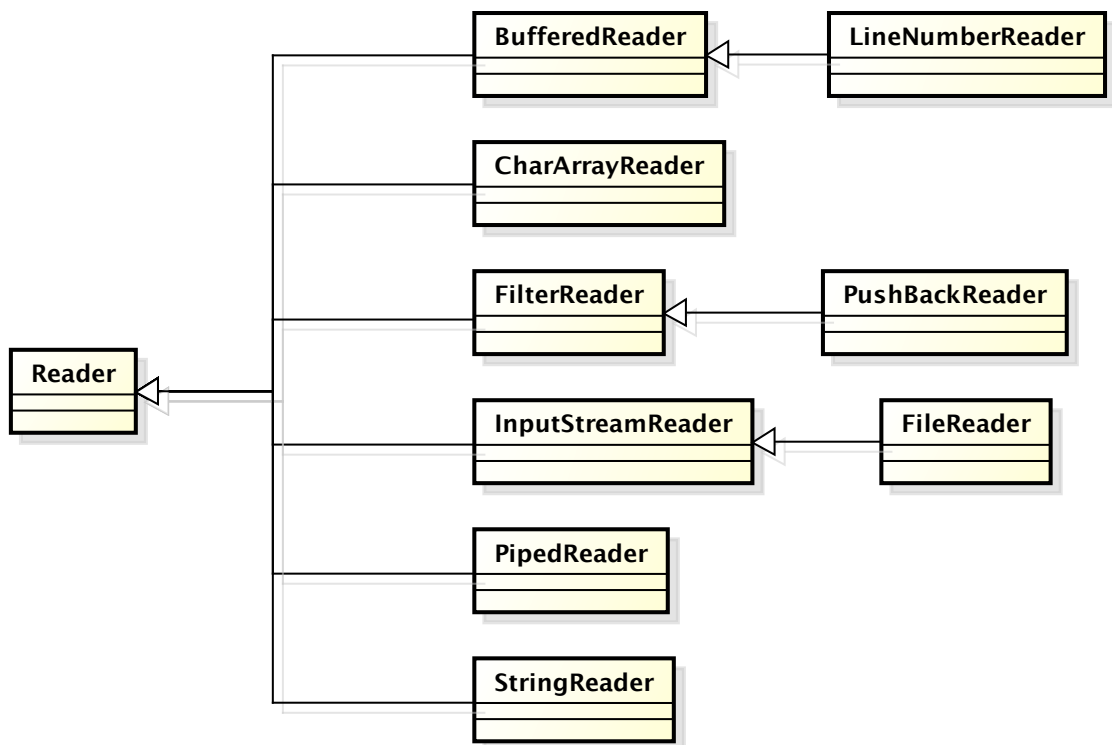
Byte vs. Char Oriented Streams



Byte vs. Char Example



Readers



Reader (abstract)

void close()

– Close the stream.

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 the stream is ready to be read.
- **void reset()**
 - Reset the stream, restart from beginning
- **long skip(long n)**
 - Skip n characters

Read a char

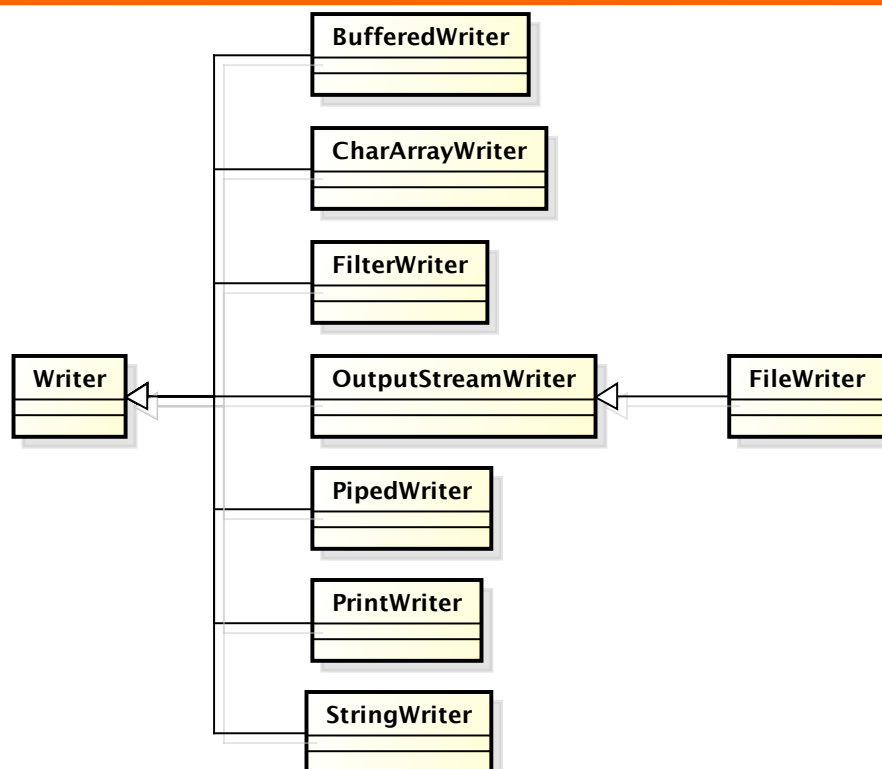
```
int ch = r.read();  
char unicode = (char) ch;  
System.out.print(unicode);  
r.close();
```

Character	ch	unicode
'A'	0...00000000 01000001 _{bin} = 65 _{dec}	65
'\n'	0...00000000 00001101 _{bin} = 13 _{dec}	13
End of file	1...11111111 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



Writer (abstract)

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.

close()

- ♦ Close the stream, flushing it first.

abstract void flush()

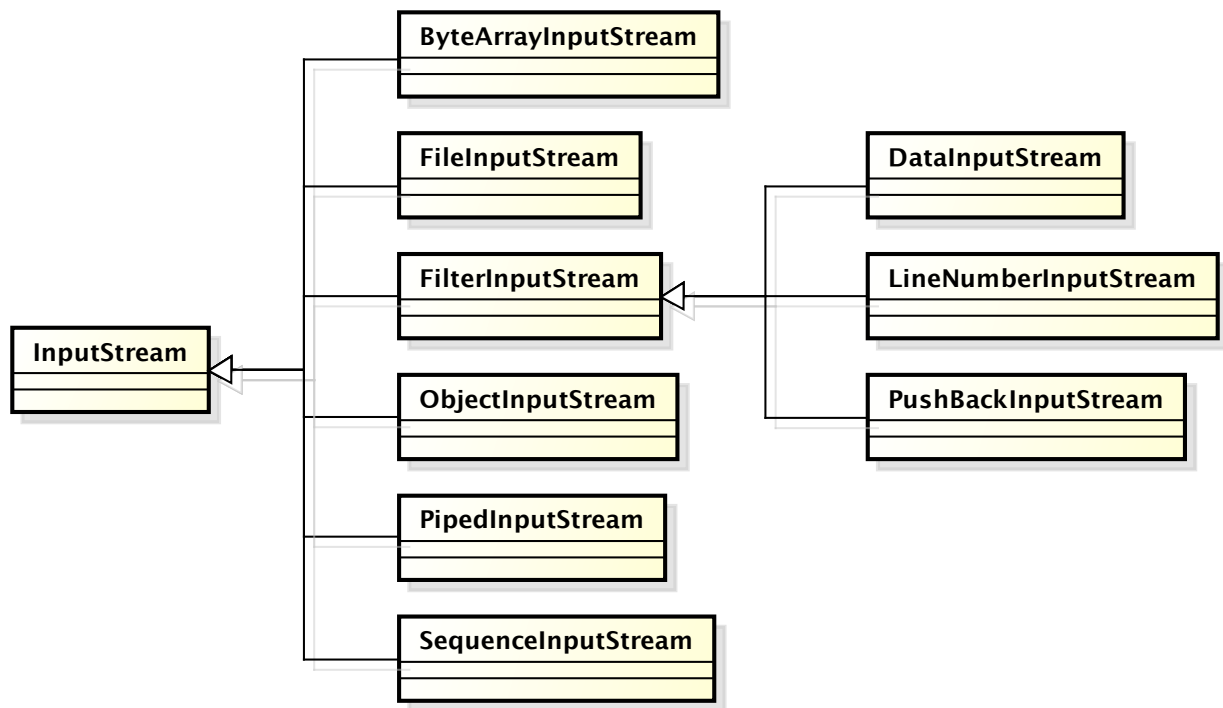
- ♦ Flush the stream.

13

Output buffers

- Output streams typically write to a memory buffer
 - ♦ Much faster than writing e.g. to file (leverage memory hierarchy)
 - ♦ When buffer is full a large chunk is written, as a whole, to the destination
 - Program termination wipes buffers
 - ♦ Must explicitly ensure buffers are flushed using method **close()** or **flush()**
-

Input streams



InputStream

void close()

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

int read()

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

int read(byte[] b)

- ◆ Reads some 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()

- ♦ Number of bytes that can be read (or skipped over) from this input stream without blocking.

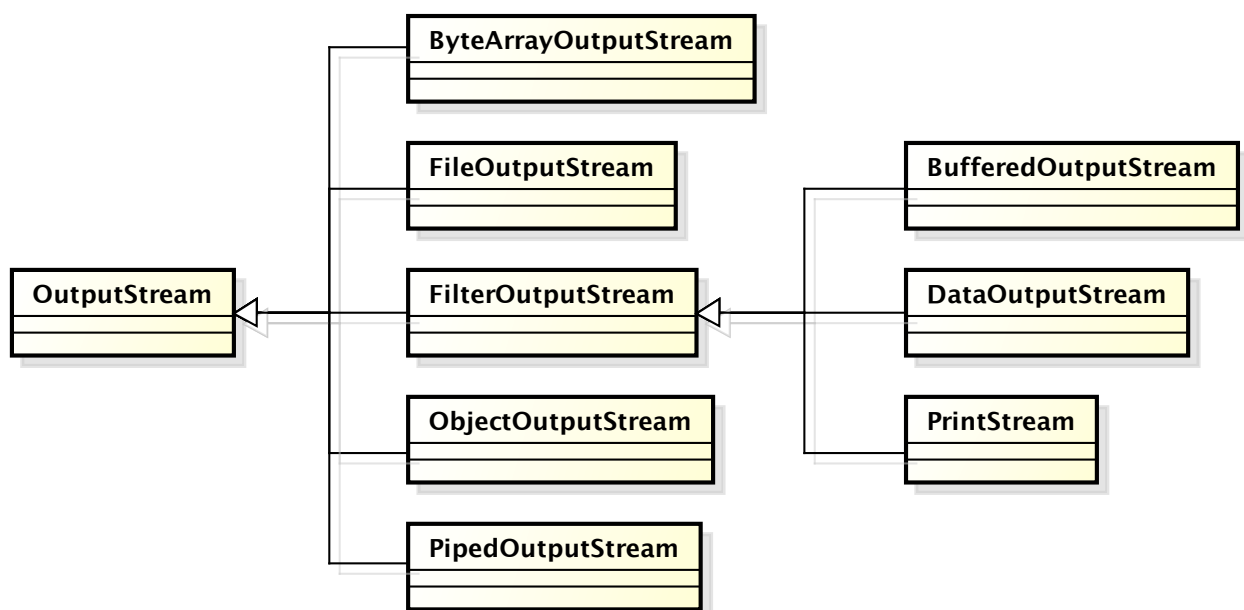
void reset()

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

long skip(long n)

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

Output streams



OutputStream

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.

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.

19

Stream specializations

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

20

Conversion byte \leftrightarrow char

- **InputStreamReader**
char \leftarrow byte
- **OutputStreamWriter**
char \rightarrow byte
- The constructors allow specifying a charset to decode/encode the byte to/from characters

21

Read/Write in memory

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

22

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

23

R/W of File

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

24

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();
```

One char at a time
is highly inefficient!

25

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();
```

The buffered version
is 10 times faster

26

Text file with encoding

- The text encoding of a stream can be defined using

- ◆ **InputStreamReader** for input

```
Reader r = new InputStreamReader(  
    new FileInputStream("file.txt"),  
    "ISO-8859-1");
```

- ◆ **OutputStreamWriter** for output

```
Writer w = new OutputStreamWriter(  
    new FileOutputStream("out.txt",  
    "ISO-8859-1");
```

Buffered

- **BufferedInputStream**

```
BufferedInputStream(InputStream i)
```

```
BufferedInputStream(InputStream i, int s)
```

- **BufferedOutputStream**

- **BufferedReader**

```
readLine()
```

- **BufferedWriter**
-

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()**

29

Standard in & out

- Default input and output streams are defined in class **System**

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

30

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()`

Streams and 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.

URL

- Represents a URL
 - ♦ Constructor may throw a **MalformedURLException**
 - Provide getters for URL portions:
 - Protocol, Host, Port
 - Path, Query, Anchor (Ref)
 - Method **openStream()** opens a connection and returns the relative **InputStream**
-

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

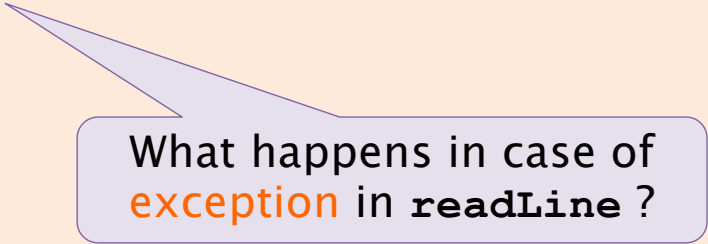
Stream as resources

- Streams consume OS resources
 - Such resources are limited
 - ♦ A program can open only a given number of file at once
 - Streams should be closed as soon as possible to release resources
-

Stream close and exceptions

```
String readFirstLine(String path)
                        throws IOException{
    BufferedReader br=new BufferedReader(
                        new FileReader(path));

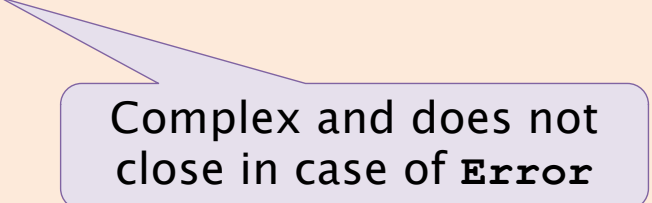
    String l = br.readLine();
    br.close();
    return l;
}
```



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

Catch and close

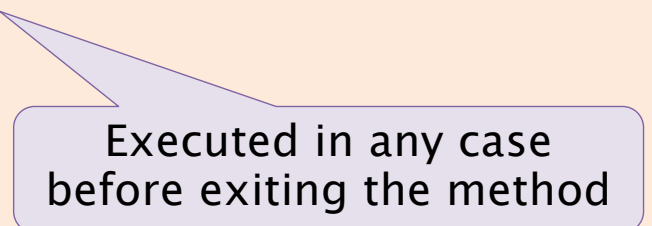
```
String readFirstLine(String path)
    throws IOException {
    BufferedReader br=new BufferedReader(
        new FileReader(path));
    try {
        String l = br.readLine();
        br.close();
        return l;
    } catch(IOException e){
        br.close();
        throw e;
    }
}
```



Complex and does not close in case of Error

Finally close

```
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

```
String readFirstLine(String path)
    throws IOException {

    try (
        BufferedReader br=new BufferedReader(
            new FileReader(path)) ) {
        return br.readLine();
    }
}
```

Works since `BufferedReader` implements `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`

43

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
 - ♦ IT is a marker interface
- ⇒ Just used to avoid serialization of objects, without permission of the class developer

44

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

47

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 SYSTEM

49

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()`

50

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());  
}
```

New IO (nio)

- Paths and Files
 - ♦ Abstract path manipulation
 - ♦ Static methods
 - Buffer and Channels
 - ♦ Buffer oriented IO
 - ♦ Leverages efficient memory transfers (DMA)
-

Class Path

- Represents path in the file system
 - ♦ Components extraction:
 - E.g. `getFileName()`
 - ♦ Navigation:
 - E.g. `getParent()` , `getRoot()`
 - ♦ Relative paths
 - `relativize()`
 - `isAbsolute()`
 - `resolve()`
-

Class Files

- Provides methods to operate on Paths
 - ♦ Copy content: `copy()`
 - ♦ Create: `createFile()`
 - ♦ Test properties: `isWritable()`
 - ♦ Navigate: `list()` , `find()`
 - ♦ Create stream: `newInputStream()`
 - ♦ Create channel: `newByteChannel()`
 - ♦ Read: `lines()`
 - ♦ Write: `write()`
-

Example

- Compute max line length

```
Path d = Paths.get("file.txt")
int maxLen = 0;
if (Files.exists(d)) {
    maxLen = Files.lines(d)
        .mapToInt(String::length)
        .max().getAsInt();
}
```

Tokenizers

- **StringTokenizer**
 - ♦ Works on
 - String to be tokenized
 - set of delimiters
 - Default: " \t\n\r\f"
- Divides a String in tokens (sequence of characters separated by delimiters), returning the tokens one by one
 - ♦ **hasMoreTokens()**: checks for more tokens
 - ♦ **nextToken()**: returns next token
 - Does not distinguish identifiers, numbers, comments, quoted strings

Tokenizers

- **StreamTokenizer**
 - ♦ Works on a stream (Reader)
 - ♦ More sophisticated, recognizes identifiers, comments, quoted string, numbers
 - ♦ use symbol table and flag
 - ♦ **nextToken()** , returns the type
 - **TT_EOF** if at the end of file
 - ♦ Attribute **val** contains the token

57

Summary

- Java IO is based on the stream abstraction
 - Two main stream families:
 - ♦ Char oriented: Reader/Writer
 - ♦ Byte oriented: InputStream/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
-