

# Software 1 with Java

Recitation No. 9

Java I/O

Ohad Barzilay and Lior Shapira

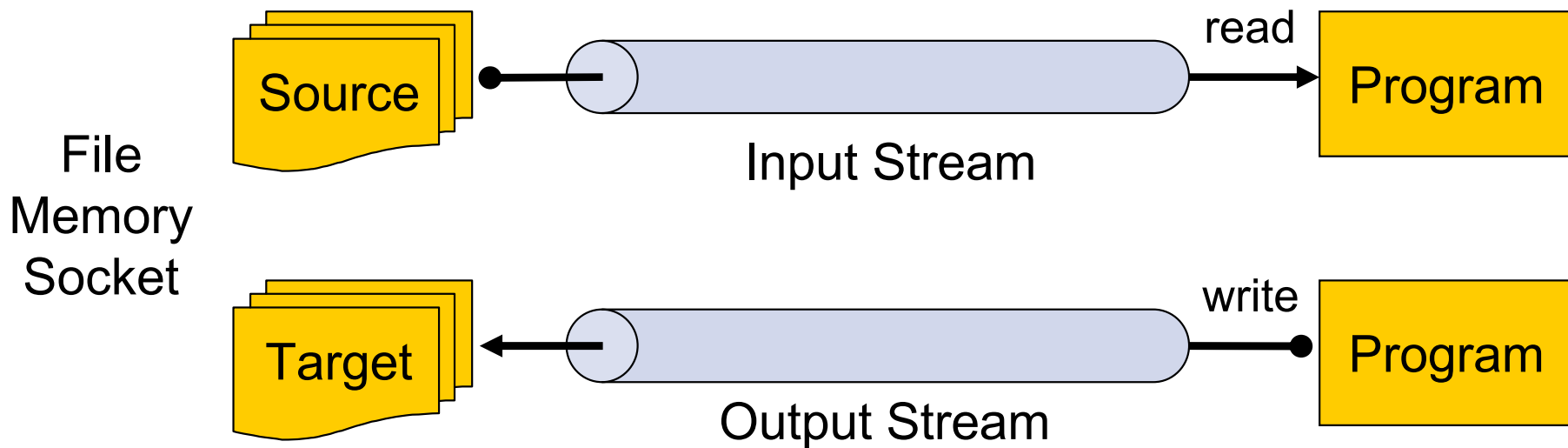
# The java.io package

---

- The java.io package provides:
  - Classes for reading input
  - Classes for writing output
  - Classes for manipulating files
  - Classes for serializing objects

# Streams

- A ***stream*** is a sequential flow of data
- Streams are one-way streets.
  - ***Input streams*** are for reading
  - ***Output streams*** are for writing



# Streams

- Usage Flow:

open a stream

while more information

    Read/write information

close the stream

- All streams are automatically opened when created.

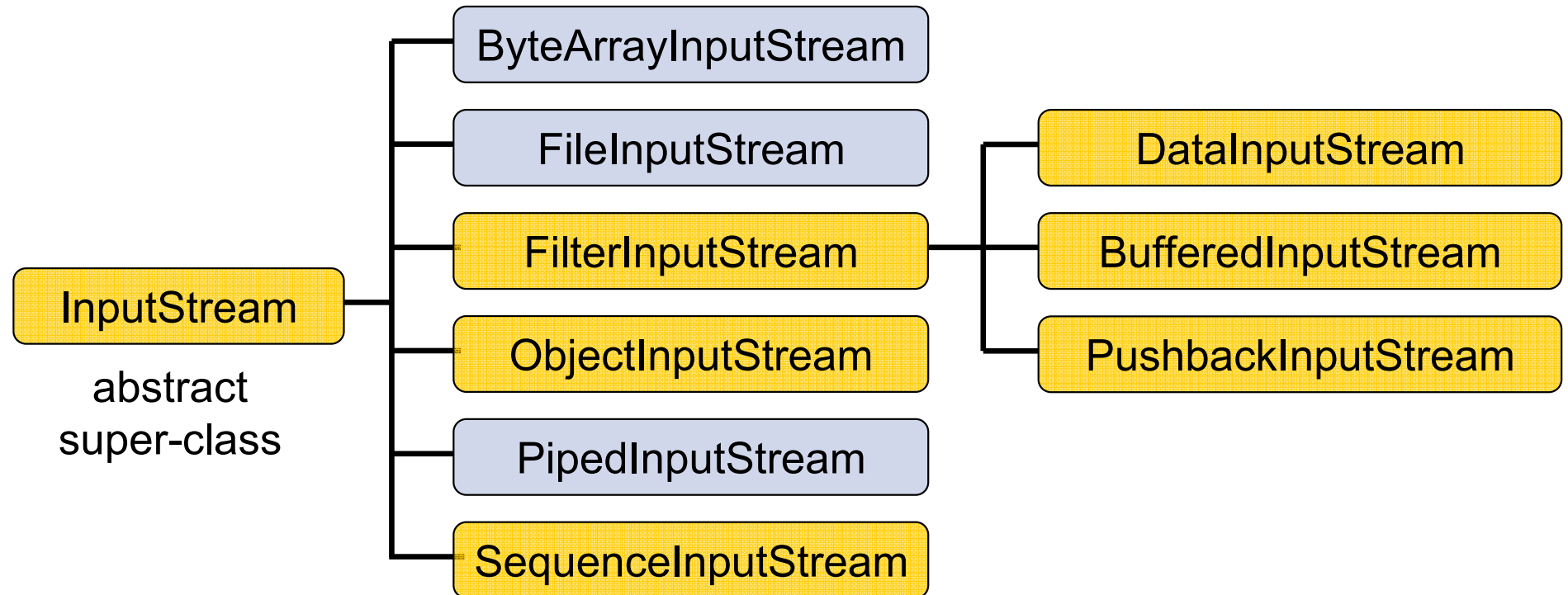
# Streams

- There are two types of streams:
  - **Byte streams** for reading/writing raw bytes
  - **Character streams** for reading/writing text

- Class Name Suffix Convention:

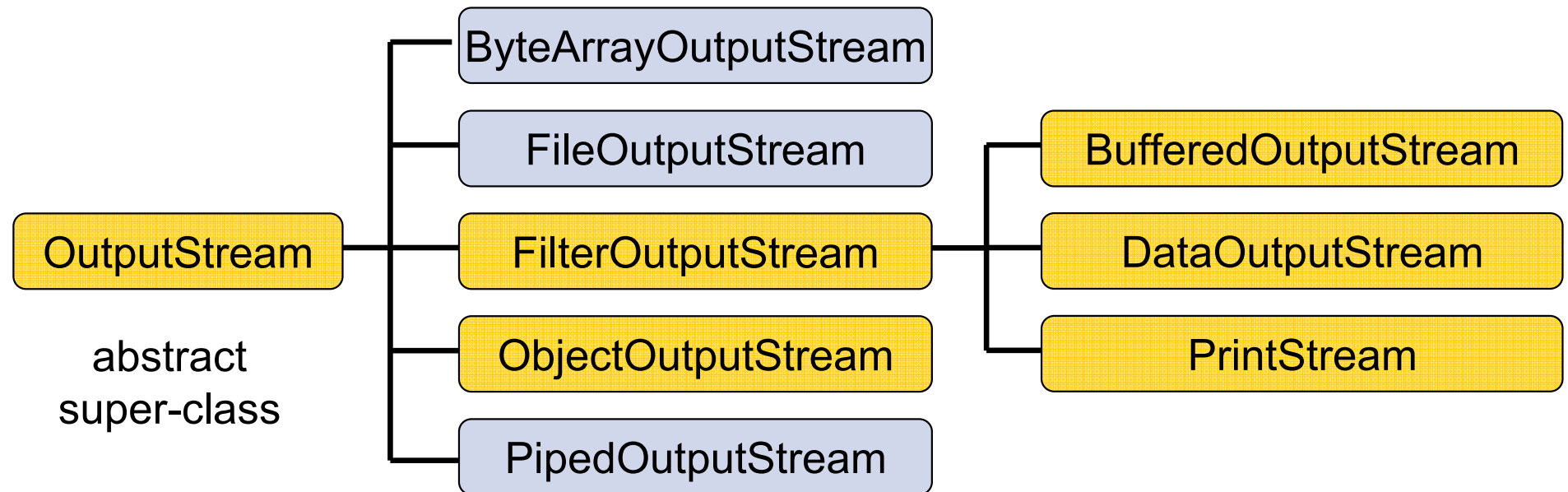
|        | Byte         | Character |
|--------|--------------|-----------|
| Input  | InputStream  | Reader    |
| Output | OutputStream | Writer    |

# InputStreams



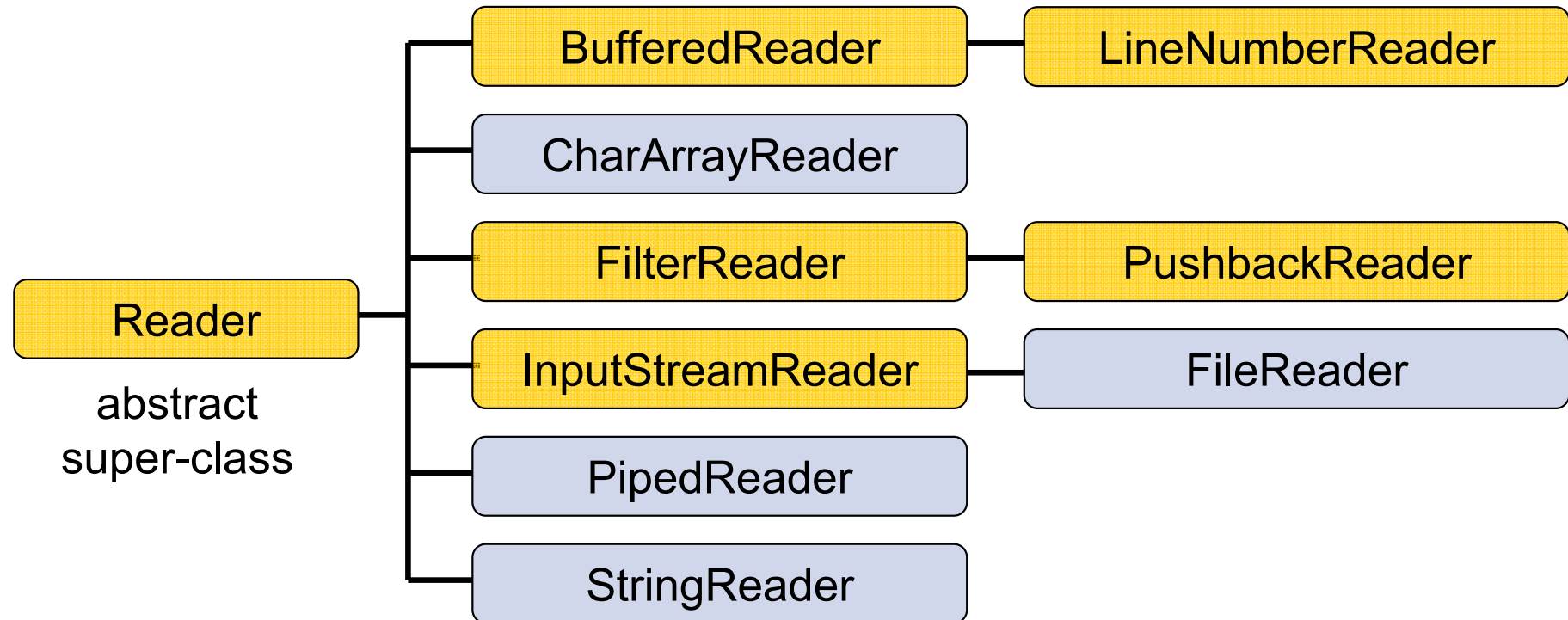
- - read from data sinks
- - perform some processing

# OutputStreams



- - write to data sinks
- - perform some processing

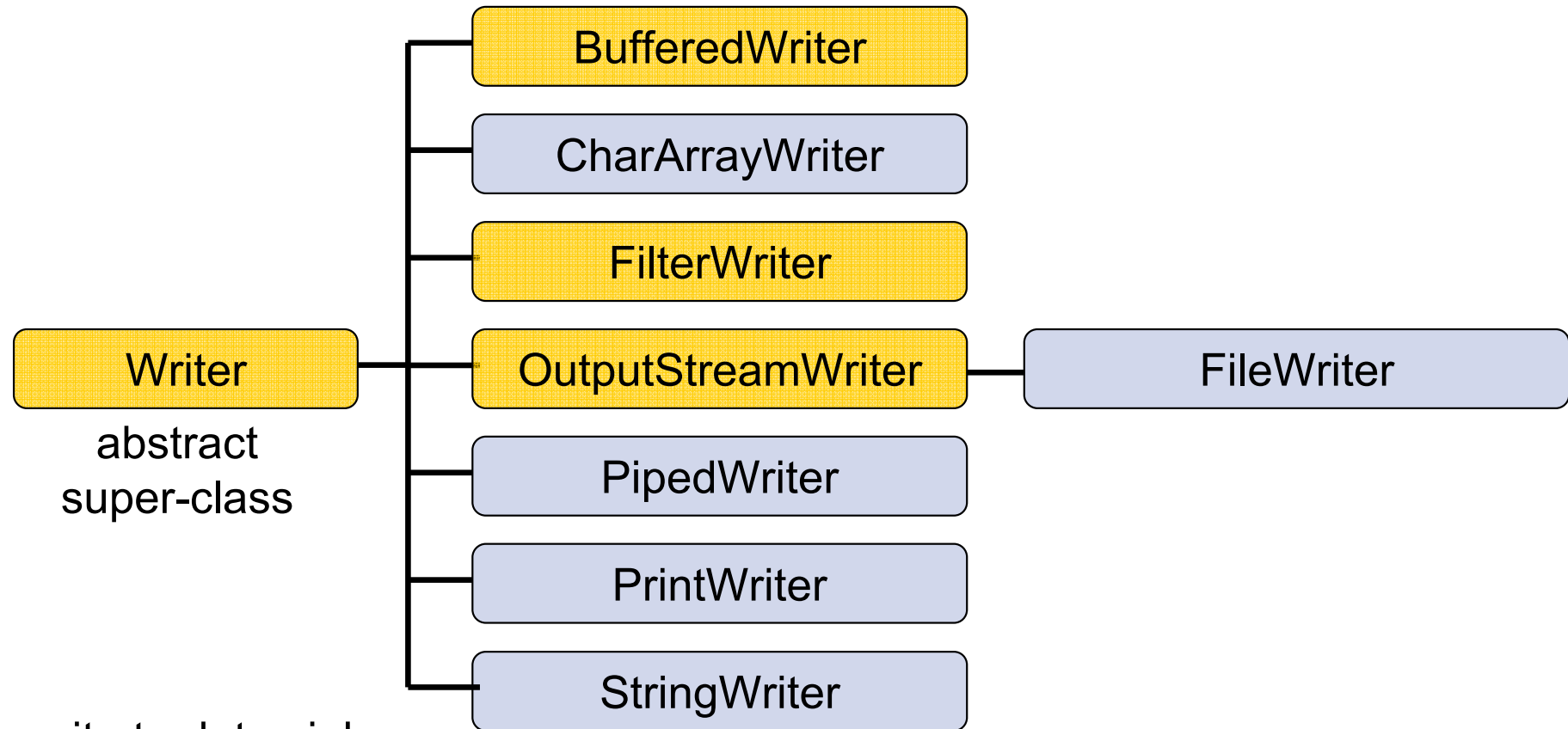
# Readers



- - read from data sinks
- - perform some processing



# Writers



- - write to data sinks
- - perform some processing

# Terminal I/O

- The `System` class provides references to the standard input, output and error streams:

```
InputStream stdin = System.in;
```

```
PrintStream stdout = System.out;
```

```
PrintStream stderr = System.err;
```

# InputStream Example

- Reading a single byte from the standard input stream:

```
try {  
    int value = System.in.read();  
    ...  
} catch (IOException e) {  
    ...  
}
```

an int with a byte information

is thrown in case of an error

returns -1 if a normal end of stream has been reached

# InputStream Example

- Another implementation:

```
try {  
    int value = System.in.read();  
    if (value != -1) {  
        byte bValue = (byte) value;  
        ...  
    }  
} catch (IOException e) {...}
```

end-of-stream  
condition

casting

# Character Stream Example

```
public static void main(String[] args) {  
  
    try {  
        FileReader in = new FileReader("in.txt");  
        FileWriter out = new FileWriter("out.txt");  
  
        int c;  
        while ((c = in.read()) != -1) {  
            out.write(c);  
        }  
  
        in.close();  
        out.close();  
  
    } catch (IOException e) {  
        // Do something  
    }  
}
```

# Stream Wrappers

- Some streams wrap other streams and add new features.
- A wrapper stream accepts another stream in its constructor:

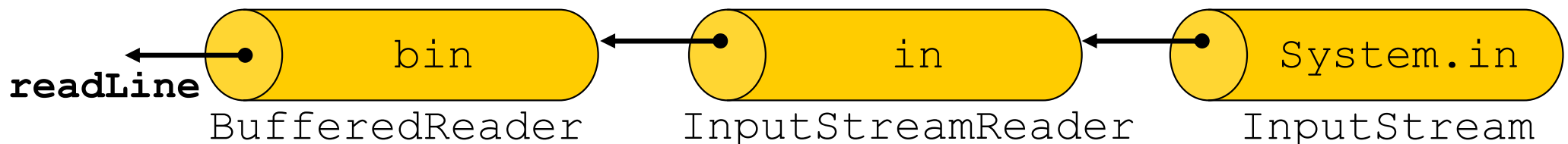
```
DataInputStream din =  
    new DataInputStream(System.in) ;  
double d = din.readDouble() ;
```



# Stream Wrappers (cont.)

- Reading a text string from the standard input:

```
try {  
    InputStreamReader in =  
        new InputStreamReader(System.in);  
  
    BufferedReader bin = new BufferedReader(in);  
  
    String text = bin.readLine();  
    ...  
} catch (IOException e) {...}
```



# The Scanner Class

- Breaks its input into tokens using a delimiter pattern (matches whitespace by default)
- The resulting tokens may then be converted into values

```
try {  
    Scanner s = new Scanner(System.in);  
    int anInt = s.nextInt();  
    float aFloat = s.nextfloat();  
    String aString = s.next();  
    String aLine = s.nextLine();  
  
    } catch (IOException e) {  
    // Do something  
}
```

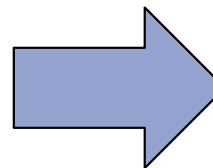


# The Scanner Class

- Works with any type of textual input
- We can change the delimiter and other options
- Another example:

```
String input = "1 fish 2 fish red fish blue fish";  
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");  
System.out.println(s.nextInt());  
System.out.println(s.nextInt());  
System.out.println(s.next());  
System.out.println(s.next());  
s.close();
```

↑  
Regular  
expression



```
1  
2  
red  
blue
```

# The File Class

---

- A utility class for file or directory properties (name, path, permissions, etc.)
- Performs basic file system operations:
  - removes a file: `delete()`
  - creates a new directory: `mkdir()`
  - checks if the file is writable: `canWrite()`
  - creates a new file: `createNewFile()`
- No direct access to file data
- Use file streams for reading and writing

# The File Class

## Constructors

- Using a full pathname:

```
File f = new File("/doc/foo.txt");  
File dir = new File("/doc/tmp");
```

- Using a pathname relative to the current directory defined in `user.dir`:

```
File f = new File("foo.txt");
```

**Note:** Use `System.getProperty('user.dir')` to get the value of `user.dir`

(Usually the default is the current directory of the interpreter. In Eclipse it is the project's directory)



# The File Class

## Pathnames

---

- Pathnames are system-dependent
  - `"/doc/foo.txt"` (UNIX format)
  - `"D:\doc\foo.txt"` (Windows format)
- On Windows platform Java accepts path names either with `'/'` or `'\'`
- The system file separator is defined in:
  - `File.separator`
  - `File.separatorChar`

# The File Class

## Directory Listing

---

- Printing all files and directories under a given directory:

```
public static void main(String[] args) {  
    File file = new File(args[0]);  
  
    String[] files = file.list();  
    for (int i=0 ; i< files.length ; i++) {  
        System.out.println(files[i]);  
    }  
}
```

# The File Class

## Directory Listing (cont.)

- Printing all files and directories under a given directory with ".txt" suffix:

```
public static void main(String[] args) {  
    File file = new File(args[0]);  
    FilenameFilter filter = new  
        SuffixFileFilter(".txt");  
  
    String[] files = file.list(filter);  
    for (int i=0 ; i<files.length ; i++) {  
        System.out.println(files[i]);  
    }  
}
```

# The File Class

## Directory Listing (cont.)

---

```
public class SuffixFileFilter
    implements FilenameFilter {

    private String suffix;

    public SuffixFileFilter(String suffix) {
        this.suffix = suffix;
    }

    public boolean accept(File dir, String name) {
        return name.endsWith(suffix);
    }
}
```



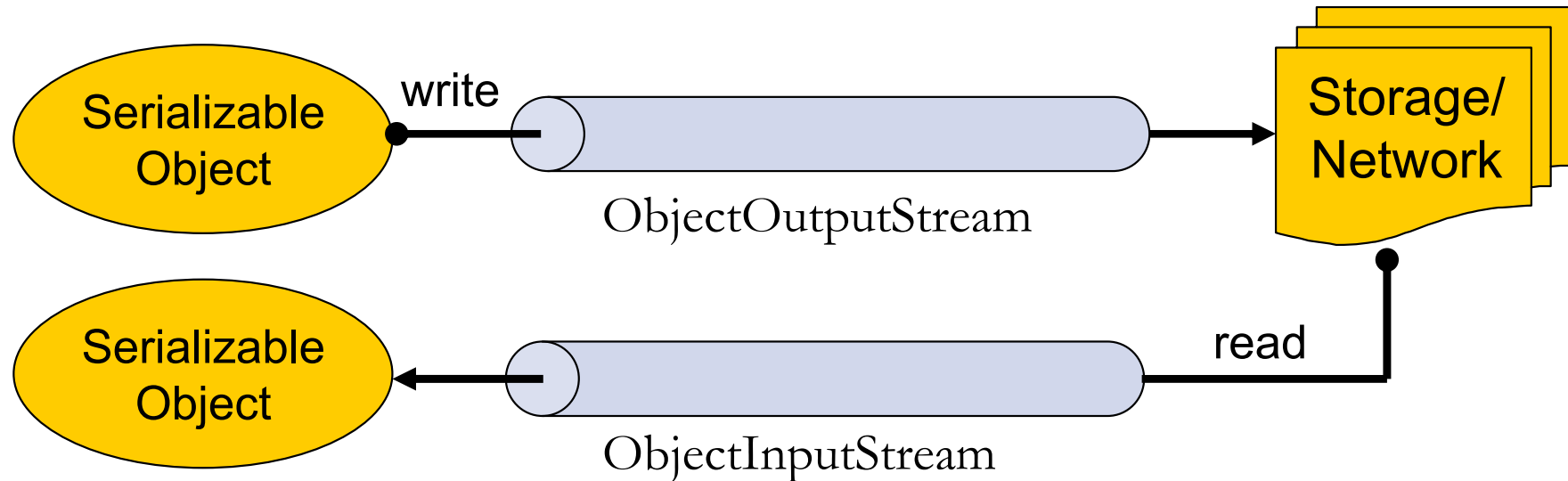
# Object Serialization

---

- A mechanism that enable objects to be:
  - saved and restored from byte streams
  - persistent (outlive the current process)
  
- Useful for:
  - persistent storage
  - sending an object to a remote computer

# The Default Mechanism

- The default mechanism includes:
  - The Serializable interface
  - The ObjectOutputStream
  - The ObjectInputStream



# The Serializable Interface

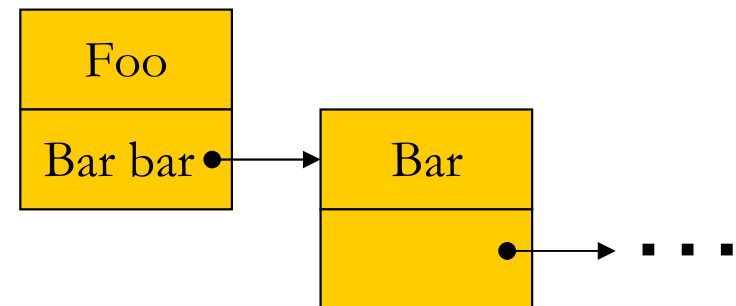
---

- Objects to be serialized must implement the `java.io.Serializable` interface
- An empty interface
- Most objects are `Serializable`:
  - Primitives, Strings, GUI components etc.
- Subclasses of `Serializable` classes are also `Serializable`

# Recursive Serialization

- Can we serialize a `Foo` object?

```
public class Foo implements Serializable {  
    private Bar bar;  
    ...  
}  
  
public class Bar {...}
```



- No, since `Bar` is not `Serializable`
- Solution:
  - Implement `Bar` as `Serializable`
  - Mark the `bar` field of `Foo` as `transient` (will not be discussed in the course)
  - And, so on recursively

# Writing Objects

- Writing a HashMap object (map) to a file\*:

```
try {  
  
    FileOutputStream fileOut =  
        new FileOutputStream("map.s");  
  
    ObjectOutputStream out =  
        new ObjectOutputStream(fileOut);  
  
    out.writeObject(map);  
  
} catch (Exception e) {...}
```

\* HashMap is Serializable

# Reading Objects

```
try {  
  
    FileInputStream fileIn =  
        new FileInputStream("map.s");  
  
    ObjectInputStream in =  
        new ObjectInputStream(fileIn);  
  
    Map h = (Map)in.readObject();  
  
} catch (Exception e) {...}
```