Object-Oriented Programming with Java

Recitations No. 4: Java IO

The java.io package

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

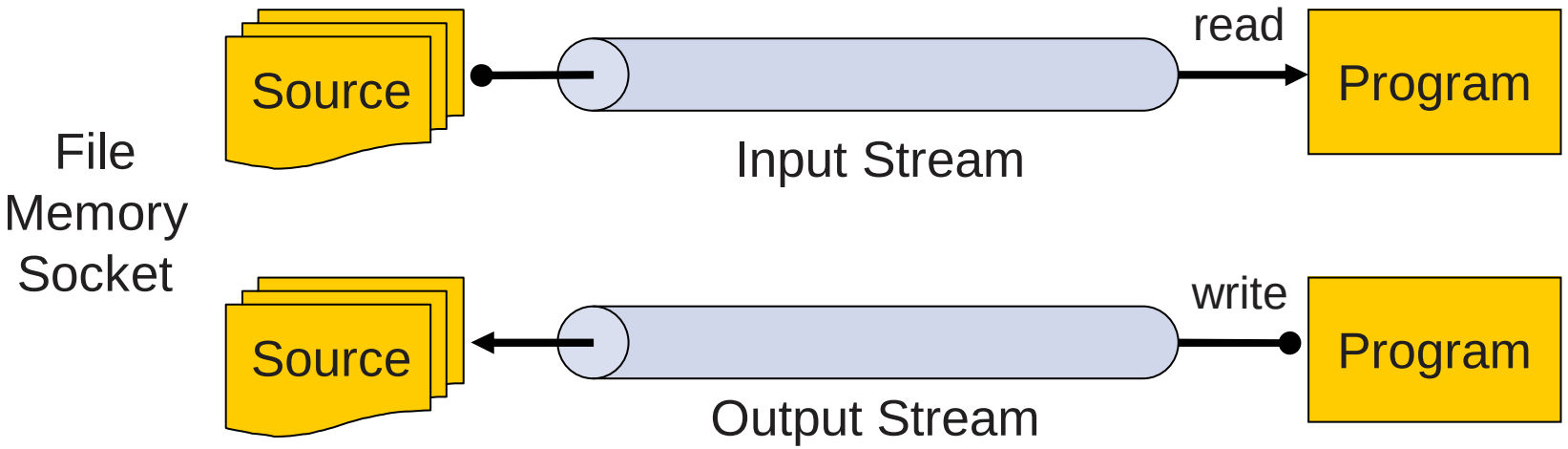
Online Books

- Java Fundamental Classes Reference (Oreilly)
- Exploring Java (Oreilly)
- The Java Tutorial (Sun)

  (http://java.sun.com/docs/books/tutorial/essential/io/)

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 (cont.)

- Usage Flow:

```
open a stream
while more information
  Read/write information
close the stream
```

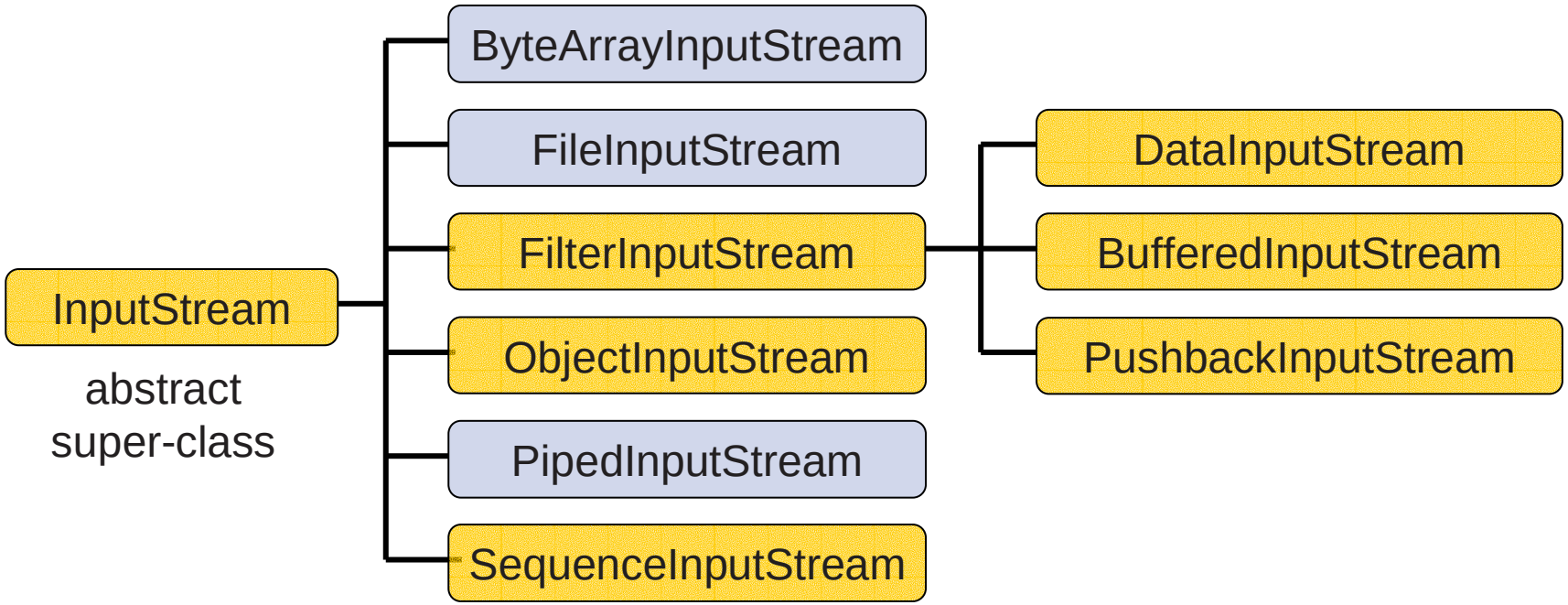
- All streams are automatically opened when created.

Streams (cont.)

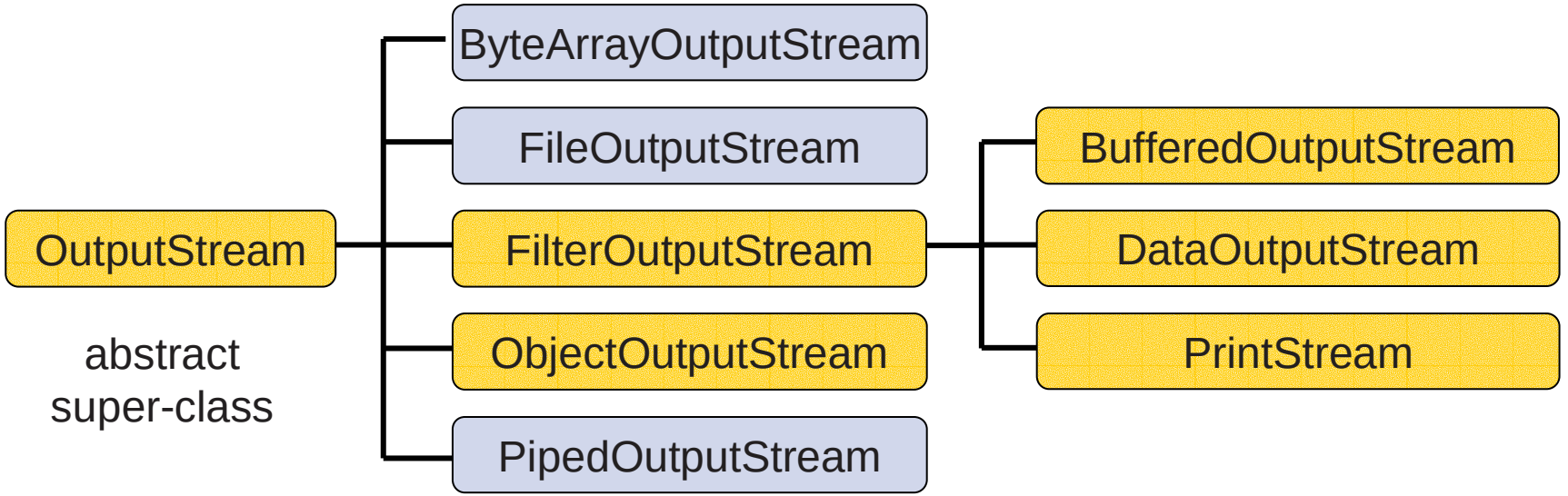
- There are two types of streams:
  - ***Byte streams*** for reading/writing raw bytes
  - ***Character streams*** for reading/writing text
- Class Name Suffix Conversion:

	Byte	Character
Input	InputStream	Reader
Output	OutputStream	Writer

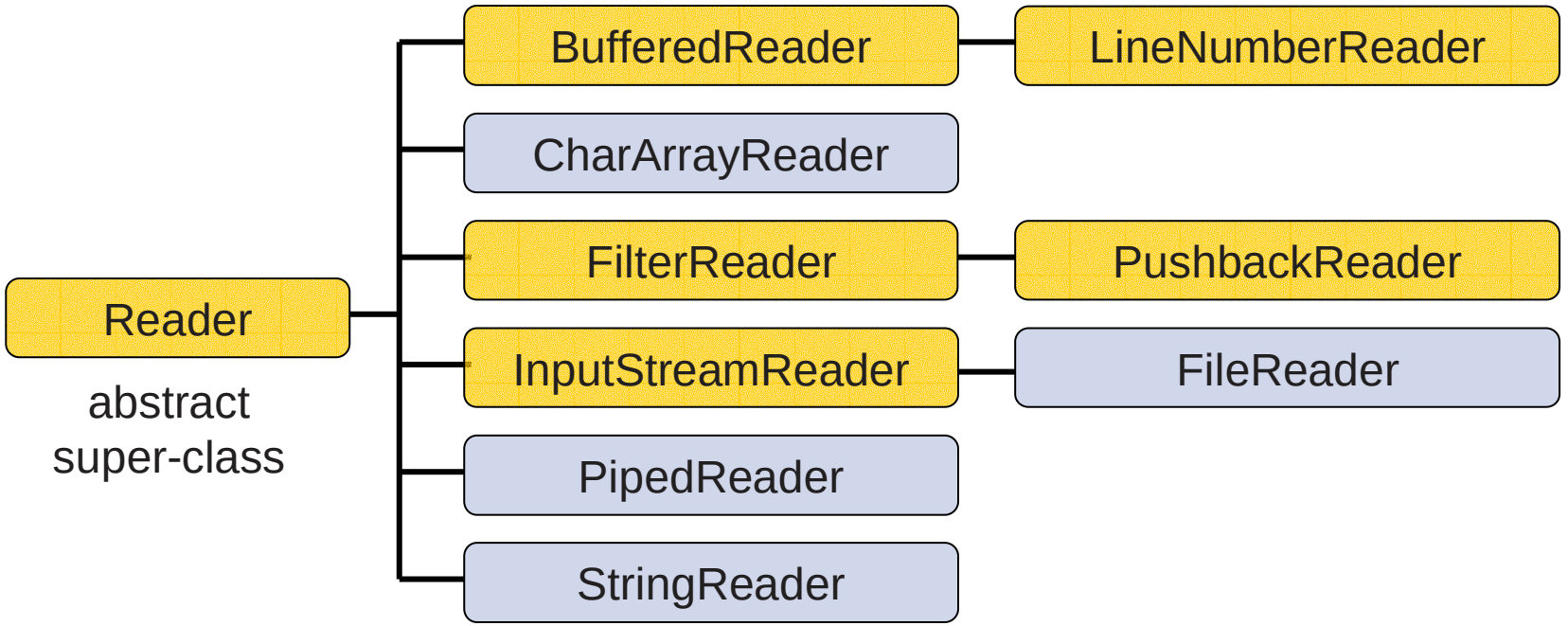
InputStreams



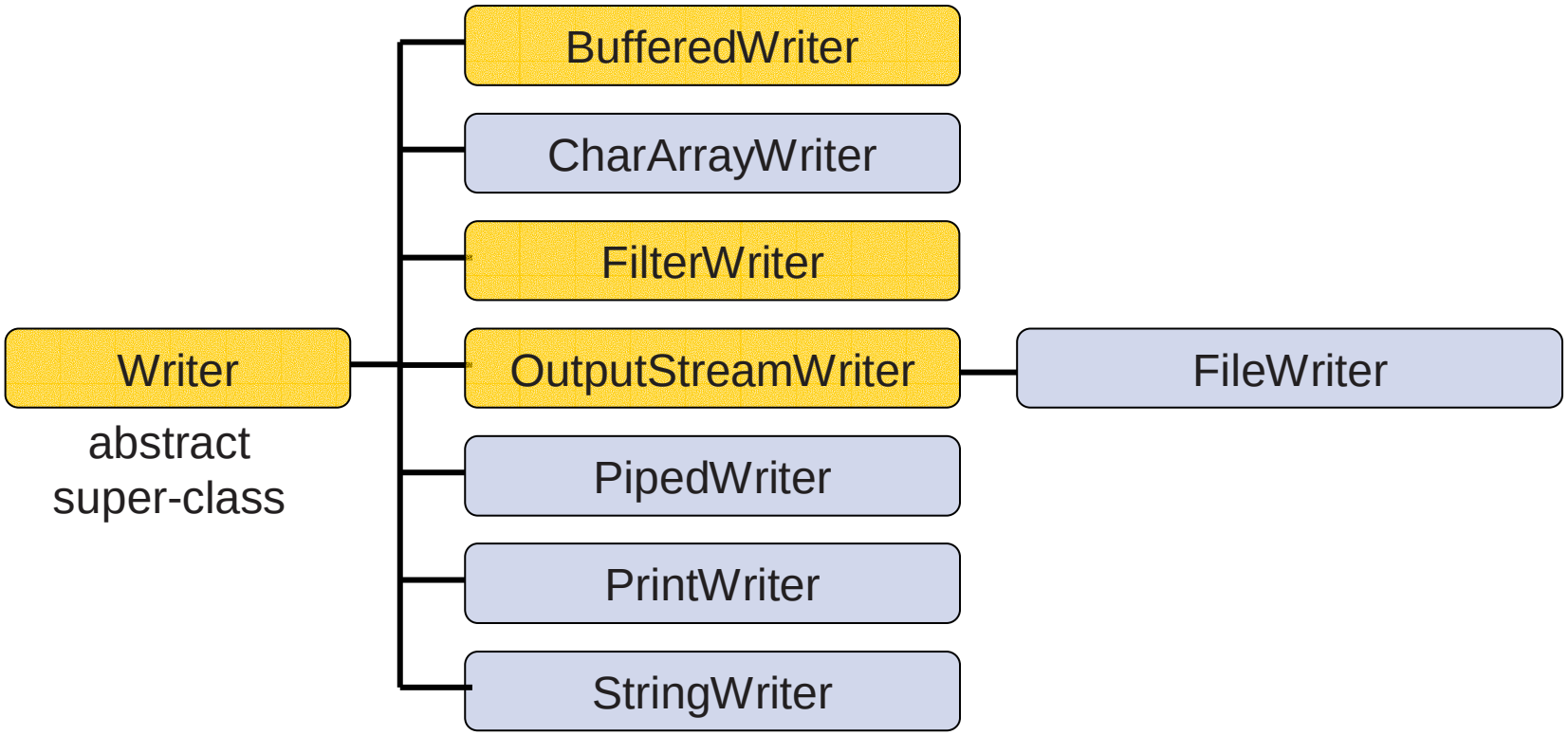
OutputStreams



Readers


Reader
abstract
super-class
BufferedReader
LineNumberReader
CharArrayReader
FilterReader
PushbackReader
InputStreamReader
FileReader
PipedReader
StringReader

Writers


BufferedWriter
CharArrayWriter
FilterWriter
Writer
OutputStreamWriter
FileWriter
abstract
super-class
PipedWriter
PrintWriter
StringWriter

Terminal I/O

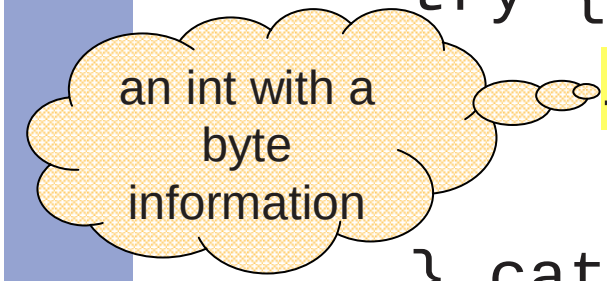
- The `System` class provides references to the standard input, output and error streams:

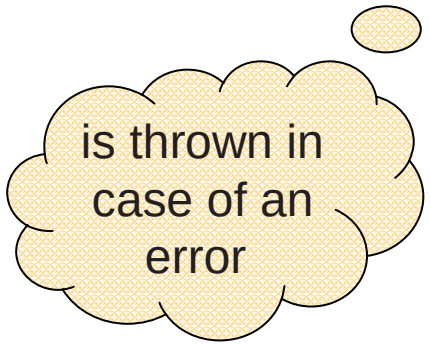
```
InputStream stdin = System.in;
OutputStream stdout = System.out;
OutputStream stderr = System.err;
```

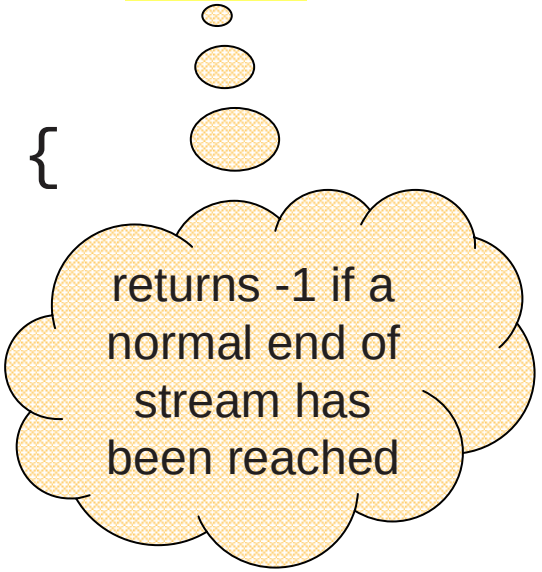
InputStream Example

- Reading a single byte from the standard input stream:

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







InputStream Example (cont.)

- Another implementation:

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

end-of-stream condition

casting

Stream Wrappers

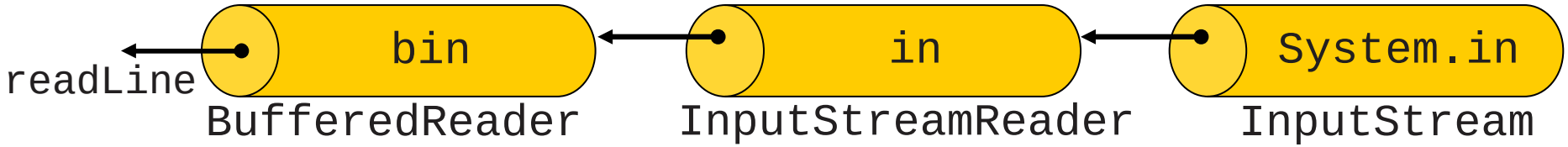
- Some streams wrap others streams and add new features.
- A wrapper stream accepts another stream in its constructor:

```
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 File Class

- Represents a file or directory pathname
- Performs basic file-system operations:
  - removes a file: `delete()`
  - creates a new directory: `mkdir()`
  - checks if the file is writable: `canWrite()`
- No method to create a new file
- 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 of the Java interpreter:

```
File f = new File(“foo.txt”);
```

Note: `System.getProperty('user.dir')` returns the current directory of the interpreter

The File Class
Constructors (cont)

- `File f = new File("/doc","foo.txt");`

- `File dir = new File("/doc");`
  `File f = new File(dir, "foo.txt");`
- A `File` object can be created for a non-existing file or directory
  - Use `exists()` to check if the file/dir exists

The File Class

Pathnames

- Pathnames are system-dependent
  - `"/doc/foo.txt"` (UNIX format)
  - `"D:\doc\foo.txt"` (Windows format)
- On Windows platform Java excepts 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);
    }
}
```

The RandomAccessFile Class

- permits random access to a file's data
- is used for both reading and writing files
- Constructors:
  - RandomAccessFile(File file, String mode)
  - RandomAccessFile(String name, String mode)

  Where:

  mode – specify the access mode (e.g. "r", "rw")

The RandomAccessFile Class

File Pointer

- indicates the current location in the file.
- Explicitly manipulating the file pointer:
  - `int skipBytes(int)`

    Moves the file pointer forward the specified number of bytes
  - `void seek(long)`

    Positions the file pointer before the specified byte
  - `long getFilePointer()`

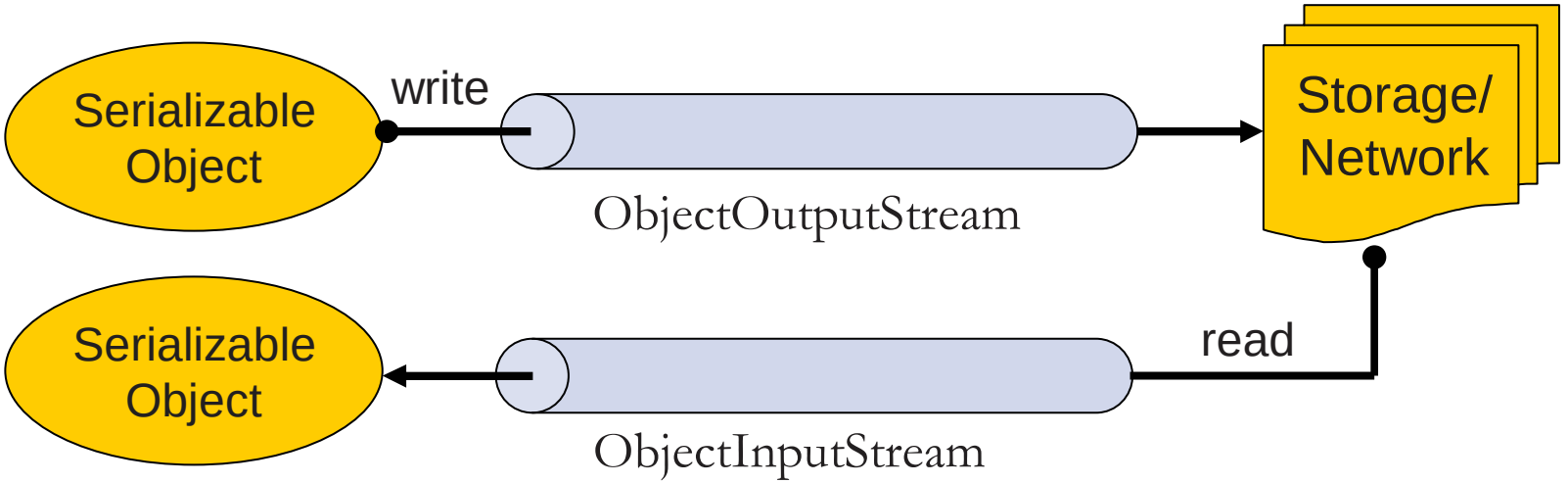
    Returns the current byte location of the file pointer

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
- Example:

```
public abstract class VersionedString
        implements Serializable {
        …
}
```

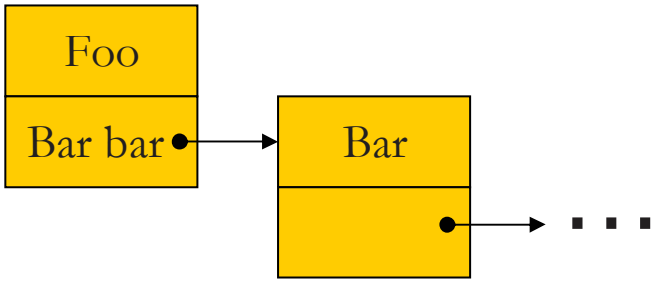
Nonserializable Objects

- Object does not implement Serializable
- Indeed, some system-level classes cannot be serialized:
  - Threads, DB connection, Network Sockets
- However, most objects are Serializable:
  - Primitives, Strings, GUI components etc.

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
  - And, so on recursively

Transient Fields

- Are not part of the object's persistent state
- Mark as transient any field that
  - cannot be serialized or
  - you do not want it to be serialized
- The other fields should hold references to Serializable objects

```
public class UserSession implements java.io.Serializable {
    private String userName;
    private transient String password;
}
```

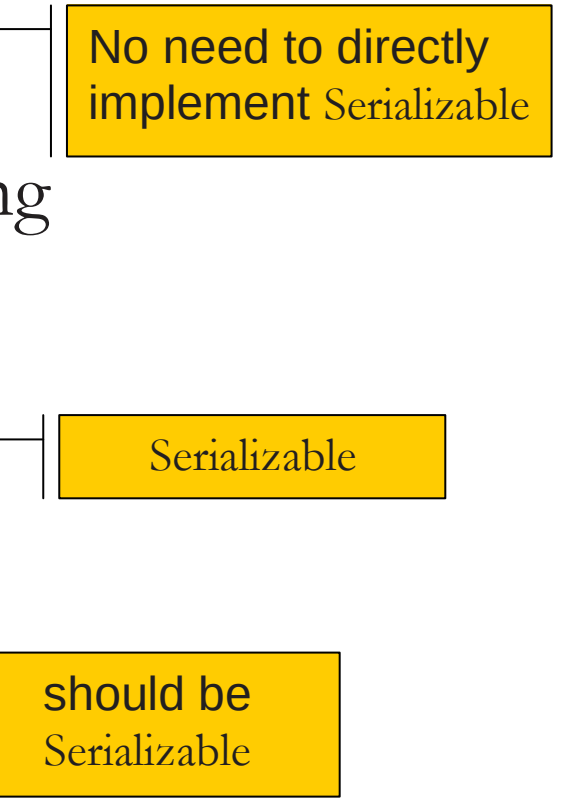
Strings are serializable

will be null after deserialization

Serialization and Inheritance

- Subclasses of Serializable classes are also Serializable

```
public class LinkedVersionedString
        extends VersionedString {
        protected int n;
        protected Version last;
        …
}
```



Serialization and Inheritance

- A Serializable subclass of a non serializable class:
  - is allowed only if the superclass has an accessible no argument constructor
  - During deserialization the fields of the non-serializable superclass will be initialized by invoking its no-argument constructor
  - Constructors of the serializable subclass will not be invoked during deserialization

Serialization and Inheritance

- Consider the following case:

```
public abstract class VersionedString {
    protected Date creationDate = new Date();
     public Date getCreationDate() {return creationDate;}
     public VersionedString() {}
     …
}

public class ArrayListVersionedString
     extends VersionedString implements Serializable {
     protected ArrayList list = new ArrayList();
    …
}
```

optional

Serialization and Inheritance

```
public static void main (String[] args) {
    VersionedString vstring = new ArrayListVersionedString();
    vstring.add("Version 1");
    System.out.println("date = " + vstring.getCreationDate());

    Thread.sleep(…)

    // Storing the vstring object in a file
    …

    // Restoring the vstring object from the file
    …

    System.out.println(“”after deserialization:”);
    System.out.println("date = " + vstring.getCreationDate());
    System.out.println(“last version = " + vstring.getLastVersion());
}
```

```
date = Tue Dec 14 10:33:57 IST 2004
after deserialization :
date = Tue Dec 14 10:34:02 IST 2004
last version = Version 1
```

Serialization and Inheritance

- Now, consider the following case:

```
public abstract class VersionedString implements Serializable {
    protected Date creationDate = new Date();
    public Date getCreationDate() {return creationDate;}
    …
}

public class ArrayListVersionedString extends VersionedString {
    protected ArrayList list = new ArrayList();
    …
}
```

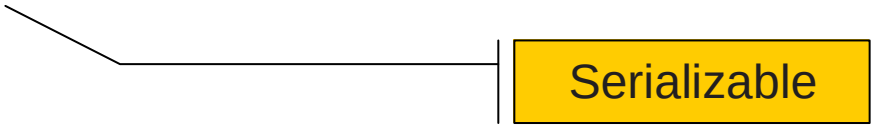
Serialization and Inheritance

```
public static void main (String[] args) {
    VersionedString vstring = new ArrayListVersionedString();
    vstring.add("Version 1");
    System.out.println("date = " + vstring.getCreationDate());

    Thread.sleep(…)

    // Storing the vstring object in a file
    …

    // Restoring the vstring object from the file
    …

    System.out.println(“”’after restoring:”);
    System.out.println("date = " + vstring.getCreationDate());
    System.out.println(“last version = " + vstring.getLastVersion());
}
```

```
date = Tue Dec 14 11:24:03 IST 2004
after deserialization : :
date = Tue Dec 14 11:24:03 IST 2004
last version = Version 1
```

Writing Serializable Objects

```
VersionedString vstring;
…
try {
    FileOutputStream fileOut =
                    new FileOutputStream("vstring.data");
   ObjectOutputStream objectOut =
                    new ObjectOutputStream(fileOut);
    objectOut.writeObject(vstring);
    objectOut.close();
} catch (IOException e) {…}
```



Reading Serializable Objects

```
VersionedString vstring;
…
try {
    FileInputStream fileIn =
                    new FileInputStream("vstring.data");
    ObjectInputStream objectIn =
                    new ObjectInputStream(fileIn);
    vstring = (VersionedString) objectIn.readObject();
   objectIn.close();
} catch (Exception e) {…}
```