Object – Oriented Design with UML and Java

PART VIII: Java IO

java.io.* & java.net.*

Java provides numerous classes for input/output:

- **`java.io.InputStream`** & **`java.io.OutputStream`** are abstract classes designed for reading and writing bytes.
- **`java.io.Reader`** & **`java.io.Writer`** are at the top of an analogous class hierarchy designed for reading and writing characters (16 bit unicode).
- **`java.io.File`** provides services for accessing native files and directories.
- **`java.nio.*`** "New IO" – Check out the *Apache MINA* project, a framework built using **`java.nio`** to facilitate building high performance, high scalability network applications over TCP/IP and UDP.
- NIO.2 File System released with JDK 7

The **`java.net`** package has useful classes that work in harmony with **`java.io`** ...

- **`java.net.URL`** for locating Files.
- **`java.net.Socket`** uses an **`InputStream`** and an **`OutputStream`** for distributed communications.

java.io.OutputStream

`java.io.OutputStream` has numerous subclasses, including:

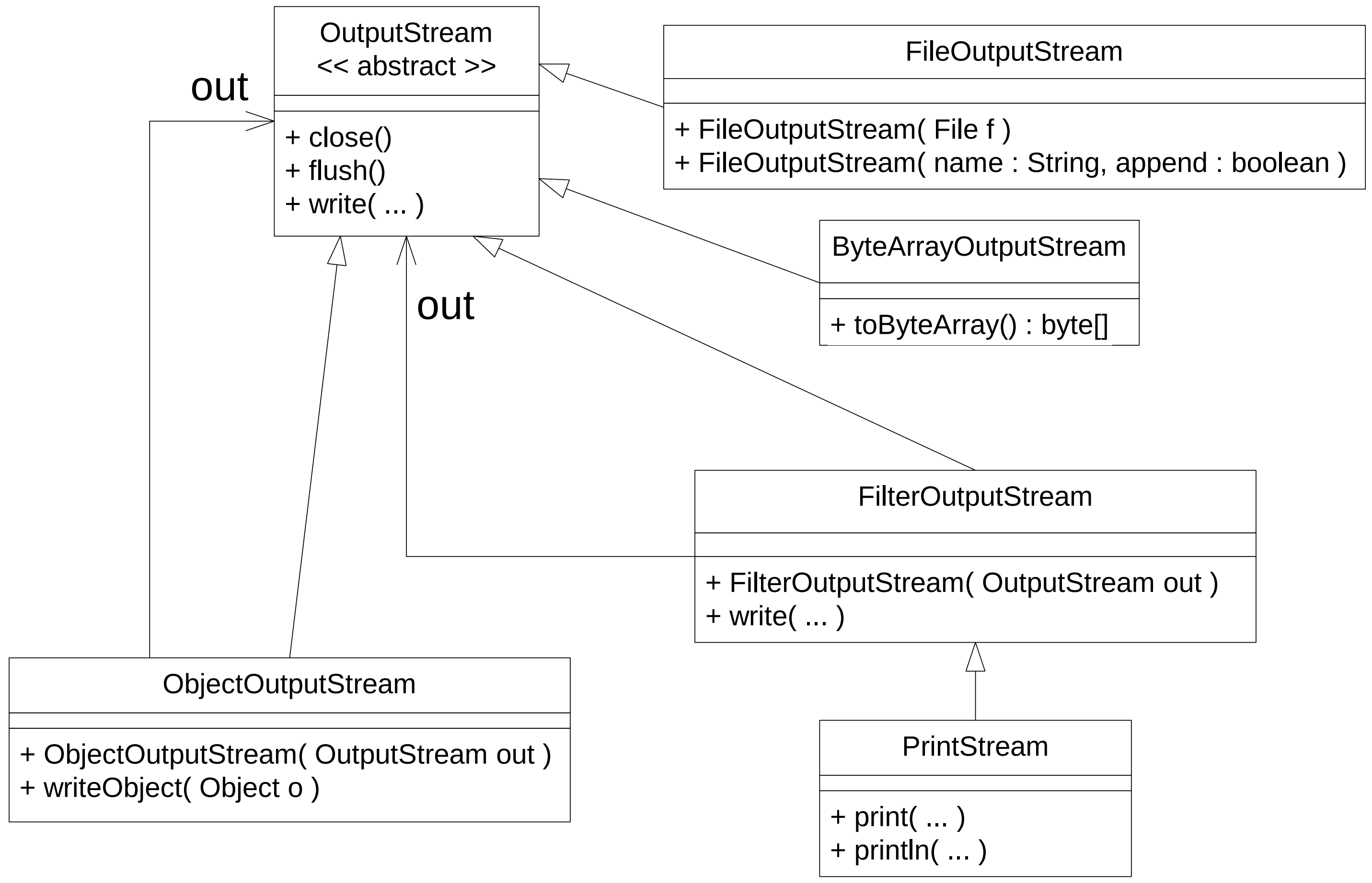
- **`ByteArrayOutputStream`** - Writes to a byte[] in memory.
- **`FileOutputStream`** - Writes to a File.
- **`FilterOutputStream`** - Decorates another OutputStream with filtering.
- **`ObjectOutputStream`** - Decorates another OutputStream with the ability to stream a 'serializable' Object (to a File, byte[], …, or another decorator).
- **`PipedOutputStream`** - Allows communication between Threads; one Thread writes to an OutputStream, while the other reads from an InputStream.
- **`PrintStream`** - Decorates another OutputStream, providing 'convenience' methods for writing Java's primitive data types.
- **`. . .`**

java.io.PrintStream

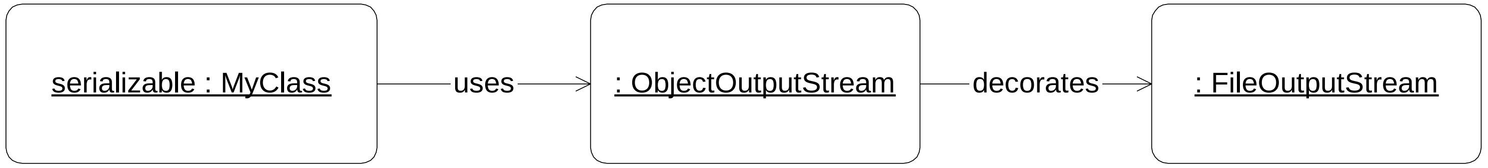
```
public class PrintStream extends FilterOutputStream
{
  public PrintStream( OutputStream out )
  public void close()
  public void flush()
  public void print( … )
  public void println( … )
  . . .
}
```

- **`System.out`** refers to an instance of class **`PrintStream.`**
- **`PrintStream`** translates your convenient **`print( … )`** method calls into the appropriate **`write( … )`** calls of its underlying **`OutputStream.`**
- Note: invoke **`flush()`** on an **`OutputStream`** whenever you want the output to be sent to the other end of the stream (**`byte[], File, Socket`**, etc.) as soon as possible (otherwise it might be ‘buffered’ - held in memory).

Decorator Design Pattern



Decorator Design Pattern (cont.)



In this example, an instance of MyClass (which implements the Serializable interface, discussed next) uses an ObjectOutputStream to serialize itself to a file. The ObjectOutputStream “decorates” the FileOutputStream, dynamically adding the new capability to be able to serialize an Object.

Decorator Design Pattern (cont.)

Intent: Extend the responsibilities of an object without subclassing. Decorators may be chained together dynamically, so that each performs some new function and then delegates to the next object for further processing.

- **`ObjectOutputStream`** 'decorates' **`OutputStream`**, by providing a way to *serialize* an Object, which then can be further streamed into a File, a byte[], or across a network Socket to another machine, …
- **`PrintStream`** 'decorates' **`OutputStream`**, by translating convenient calls to **`print( … )`** into the appropriate **`write( … )`** calls of its underlying **`OutputStream.`**
- Note the conspicuous *lack* of classes: **`ObjectFileOutputStream`**, **`ObjectByteArrayOutputStream`**, **`PrintFileOutputStream`**, …

Java Serialization

- Supported directly by language in Java and Smalltalk.
- Supported through vendor-specific toolkits in C++.

Pros:

- With language or toolkit support, it is simple and straightforward.
- Can be used for quick-and-dirty persistence on a project before the real database is up and running.
- Useful for things other than persistence, such as for passing Objects as parameters to distributed messaging.

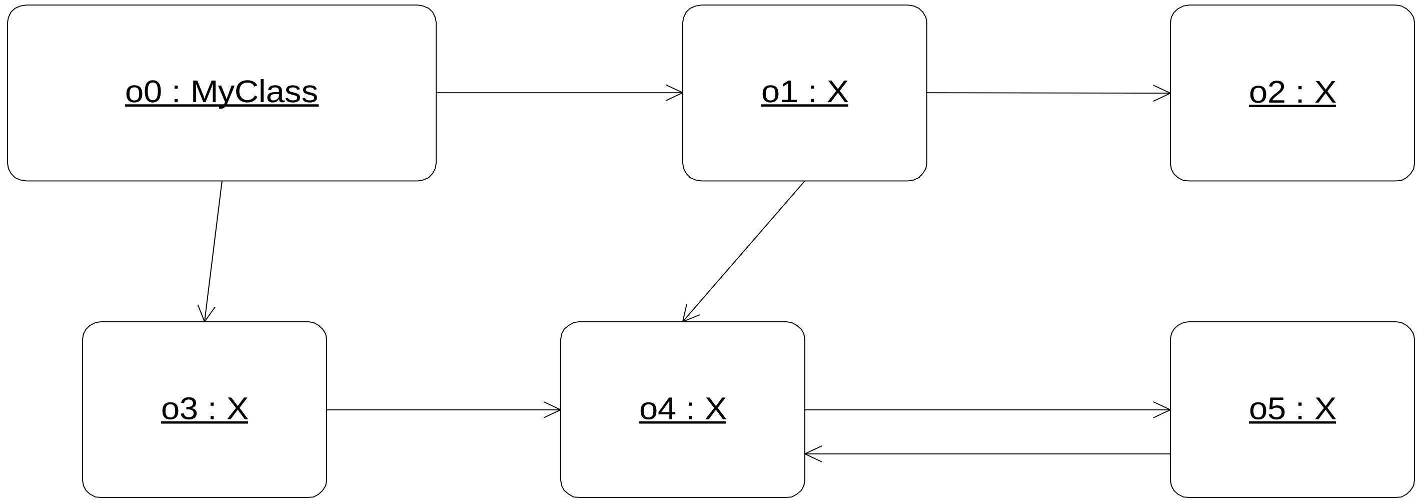
Cons:

- Startup or searching can be expensive if a large number of objects are stored.
- No transaction support.
- Not a substitute for a real database.

Serialization (cont.)

- Objects know how to write and read themselves as byte sequences.
  - The language run-time knows how to deal with base-type attributes.
  - Component objects are recursively serialized down to their base-type attributes, or the base class of the language.
- Issues:
  - Because of its recursive nature, storing or loading one object may pull in many, many others.
  - If a particular object is referenced by several others, we don't want multiple copies of the first object saved and restored. How might Graph Theory be used to work around this problem?
  - Not all fields of an object should be serialized (*transient*).
  - Versioning can be tricky.
  - Since we are serializing objects, not classes, ***static fields do not get serialized***; they remain unchanged, or they get initialized to null/0/false for the first instance of the class to be created.

Serialization (cont.)



- When we serialize o0, we want to also serialize o1, o2, o3, o4, and o5 (the entire *graph*), but we only want one serialized copy of each instance. Java serialization takes care of this automatically.

Java Serialization

- Provided as a marker interface.
  - No interface functions need to be implemented.

```
public interface java.io.Serializable {}
```

- Solves the multiple-instance Graph Theory problem.
- Can be customized.
  - Attributes declared as *transient* are not serialized.
  - The entire storage mechanism can be over-ridden.
    - » readObject() & writeObject()

Java Serialization (cont.)

```
// Write an object’s non-transient state to a file. . .
// For this to work, SerializationTest must implement Serializable
SerializationTest st = new SerializationTest( 1, 2, 3 );
FileOutputStream fileOut = new FileOutputStream( “st.sav” );
ObjectOutputStream output = new ObjectOutputStream( fileOut );
output.writeObject( st );
output.flush();
output.close();

// Read the object’s non-transient state from the file. . .
FileInputStream fileIn = new FileInputStream( “st.sav” );
ObjectInputStream input = new ObjectInputStream( fileIn );
st = (SerializationTest) input.readObject();
input.close();
```

Reading a file

- Use **`FileReader`**
- *Decorate* **`FileReader`** with **`BufferedReader`** to get **`readLine()`**

```
import java.io.*;
FileReader fr = new FileReader( “foo.txt” );
BufferedReader br = new BufferedReader( fr );
String lineFromFile = br.readLine();
while( lineFromFile != null )
{
  System.out.println( “New Line : “ + lineFromFile );
  lineFromFile = br.readLine();
}
br.close();
```

Example: StackTrace.toString()

- **`java.lang.Throwable's printStackTrace()`** writes to **`System.out:`**

```
java.lang.OutOfMemoryError
at fractal.FractalCalculator.getColorNumbers(
  FractalCalculator.java:147)
at fractal.FractalCalculator.calcFractal(FractalCalculator.java:66)
at fractal.FractalCalculator.run(FractalCalculator.java:184)
at java.lang.Thread.run(Thread.java:474)
```

How might we get this information into a **`String`**?

- If you look at Class **`Throwable's`** interface, you will see that **`printStackTrace()`** is overloaded, having a version that takes a **`PrintWriter.`** How does that help us?
- **`PrintWriter`**'s methods do not create **`String`**s. But we can construct a **`PrintWriter`** with an **`OutputStream`**…
- If we look through the possible **`OutputStream classes`** to choose from, we see that **`ByteArrayOutputStream`** has a **`toString()`** method...

StackTrace.toString()

- Bringing it all together…

```
import java.io.*;
public class StackTrace {
  public static String toString( Throwable t ) {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    PrintWriter pw = new PrintWriter( baos );
    t.printStackTrace( pw );
    pw.flush();
    String rc = baos.toString();
    pw.close();
    return rc ;
  }
}
```

Object cloning

- **`java.lang.Object`'s `clone()`** method has **`protected`** access…

Thus, an arbitrary block of code cannot **`clone`** any **`Object`**:

```
Foo foo = new Foo();
Foo foo2 = foo.clone(); // won't compile, unless Foo overrides
                        // clone() to be public.
```

So, how can an arbitrary block of code make a copy of any **`Object`**?

We could make a new interface: **`Copyable`** …

```
Foo foo = new CopyableFoo(); // implements Copyable
Foo foo2 = foo.copy(); // deep or shallow copy?
```

- What if the **`Object`** (and all of its delegates) implements **`java.io.Serializable`**?

Object cloning (cont.)

```
// Serialize the Object into a byte[]
Foo foo = new Foo(); // Foo implements java.io.Serializable
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream( baos );
oos.writeObject( foo );
oos.flush();
byte[] fooBytes = baos.toByteArray();
oos.close();

// De-Serialize the byte[] into a Foo
// deep or shallow copy?
ByteArrayInputStream bais = new ByteArrayInputStream( fooBytes );
ObjectInputStream ois = new ObjectInputStream( bais );
Foo foo2 = (Foo) ois.readObject();
ois.close();
```