

Java I/O

Christian Khoury

I/O Package

- *java.io* package
- 2 types of I/O classes
 - Byte streams called *input streams and output streams*
 - Character streams called *readers and writers*

Byte Streams

- Abstract class for input
 - *InputStream*
 - [Link to API](#)
- *Abstract class for output*
 - *OutputStream*
 - [Link to API](#)

Character Streams

- Abstract input
 - *Reader*
- Abstract output
 - *Writer*

InputStream/OutputStream

- Start by studying the [java.io.InputStream](#) and [java.io.OutputStream](#) API
- These are base classes for performing all **binary** i/o
- Note that these classes are **abstract** each with a single abstract method
 - abstract int read()
 - abstract void write(int)
- Concrete subclasses must provide implementation of read/write that can get/put a single byte to/from the relevant source

Concrete subclasses of InputStream/OutputStream

- Since InputStream/OutputStream are abstract, they cannot be used to create objects (of course, they can be used for typing).
- A very common non-abstract subclass is FileOutputStream/FileInputStream.
- These can be used in a simple way to do the most basic byte-based file io

Example with FileInputStream

```
/* class example DataInput1.java */
/* assumes each char is one byte -- dangerous
import java.io.FileInputStream;
public class DataInput1{
    public static void main(String[] args) throws Exception{
        String file = args[0];
        int input;
        FileInputStream fin = new FileInputStream(file);
        while ( (input = fin.read()) != -1){
            System.out.print((char) input);
        }
    }
}
```

Example with FileOutputStream

```
/* class example DataOutput1.java */
/* assumes each char is a single byte */
import java.io.FileOutputStream;
public class DataOutput1{
    public static void main(String[] args) throws Exception{
        String file = args[0];
        String output = "Hello World";
        FileOutputStream fout = new FileOutputStream(file);
        char[] outputAsChars = output.toCharArray();

        for (int i = 0; i < outputAsChars.length; ++i)
            fout.write(outputAsChars[i]);
    }
}
```

Higher-level functionality

- FileInputStream and FileOutputStream allow you to do pretty much any file i/o at a very low level.
- However, this is too low-level for Java.
- Java provides many more libraries to read/write higher-level constructs:
 - characters
 - Strings
 - native datatypes
 - arrays
 - arbitrary objects (serialization)

Decorator Pattern

- These capabilities are added using a design called the **Decorator Pattern**.
- InputStream/OutputStream instances are passed to a wrapper or decorator class that uses them and adds to their functionality.
- For example, floating point numbers can be read from a file by chaining together a FileInputStream and another class that assembles bytes into portable floating point.

Purpose of Decorator

- Best way to think of this is as follows:
 - There are two important issues when constructing an i/o library
 - Where the i/o is going (file, etc).
 - How the data is represented (String, native type, etc.)
 - Rather than create a class for each combination, Decorator classes allow you to mix and match, augment functionality of base classes.
 - This is a bit confusing but is very flexible.
 - Decorators can also add other capabilities, such as peek ahead, push back, write line number, etc.

Java i/o Decorator Classes

- All Java i/o decorator classes inherit from **FilterInputStream** and **FilterOutputStream**
- Look at the api for these classes and note a few things:
 - They wrap instances of `InputStream/OutputStream` respectively.
 - They inherit from `InputStream/OutputStream` respectively

More on Filter Streams

- Easiest way to think of the filter streams as wrapping an underlying class which they augment the functionality of.

Consider the respective constructors

- `FilterInputStream(InputStream in);`
- `FilterOutputStream(OutputStream out);`
- In each case, the FilterStreams use an underlying presumably simpler inputstream and augment its functionality.

Some FilterStream examples to clarify this

- Perhaps most common FilterInputStream is DataInputStream.
- Study the API and be sure you understand the inheritance hierarchy
- DataInputStream stores an InputStream and uses this to do higher-level i/o
 - readInt, readDouble, etc.
- DataOutputStream is analogous

Example of DataInputStream

```
/* DataInputStream2 example in course examples */  
import java.io.DataOutputStream;  
import java.io.FileOutputStream;  
  
public class DataOutput2{  
    public static void main(String[] args) throws Exception{  
        String file = args[0];  
        double[] data = {1.1,1.2,1.3,1.4,1.5};  
        DataOutputStream dout = new DataOutputStream  
            (new FileOutputStream(file));  
  
        for (int i = 0; i < data.length; ++i){  
            dout.writeDouble(data[i]);  
        }  
        dout.close();}}}
```

Example of DataInputStream

```
/* DataOutput2 example in course examples */
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.EOFException;
public class DataInput2{
    public static void main(String[] args) throws Exception{
        String file = args[0];
        DataInputStream din = new DataInputStream(new FileInputStream(file));
        double data;
        try{
            while (true){
                data = din.readDouble();
                System.out.println(data);
            }
        }
        catch (EOFException eofe){}
        din.close();}}}
```

Other Decorators

- Another common set of decorator classes is `BufferedInputStream` and `BufferedOutputStream`.
- Note that `java.util.Scanner` is a new and simpler alternative for a lot of high-level parsing tasks.
- These augment the functionality of the underlying stream by providing system buffering for higher-performance i/o
- They also add support for the `mark` method.
- Examples on next slide (notice how these classes can be multiply chained together in various ways).

BufferedInputStream Example

```
import java.io.*; /
public class DataInput3{
    public static void main(String[] args) throws Exception{
        String file = args[0];
        DataInputStream din = new DataInputStream
            (new BufferedInputStream
              (new FileInputStream(file)));
        double data;
        /* need an exception to know when end of file is hit */
        try{
            while (true){
                data = din.readDouble();
                System.out.println(data);
            }
        }
        catch (EOFException eofe){}
        din.close();}}
```

BufferedOutputStream example

```
import java.io.BufferedOutputStream;
import java.io.DataOutputStream;
import java.io.FileOutputStream;

public class DataOutput3{

    public static void main(String[] args) throws Exception{
        String file = args[0];
        double[] data = {1.1,1.2,1.3,1.4,1.5};
        DataOutputStream dout = new DataOutputStream
            (new BufferedOutputStream
              (new FileOutputStream(file)));

        for (int i = 0; i < data.length; ++i){
            dout.writeDouble(data[i]);
        }
        dout.close();}}}
```

Other output streams

- `FileOutputStream` is probably the most common.
- However, note that we could replace `FileOutputStream` with another `OutputStream` in these examples.
- In that case, the same decorated or undecorated data would be sent to some other device.
- Good example of this is thread communication, memory i/o, and socket i/o (using `Socket` class).
- I strongly encourage you to familiarize yourself with these classes.

Character-based IO

Reader and Writer classes

Reader/Writer

- Java maintains a second class hierarchy for performing higher-level character-based i/o.
- The two base classes in this case are
 - `java.io.Reader`
 - `java.io.Writer`
- Study the API for these classes.
- Very similar to `InputStream/OutputStream`
- Here I'll show how to do some common i/o tasks as examples

FileWriter Example

```
/* example Writer1.java in course examples */  
/* using a simple FileWriter for String-based i/o */  
import java.io.FileWriter;  
public class Writer1{  
    public static void main(String[] args) throws Exception{  
        String file = args[0];  
        String output = "Hello World!";  
        FileWriter fw = new FileWriter(file);  
        fw.write(output);  
        fw.close();  
    }  
}
```

Reading lines from stdin

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class Reader1{
    public static void main(String[] args) throws Exception{
        /* convert System.in, which is an InputStream,
           to a Reader by wrapping in InputStreamReader,
           then wrap everything in BufferedReader
        */
        String input;
        BufferedReader bin = new BufferedReader
            (new InputStreamReader
             (System.in));
        while ( (input = bin.readLine()) != null){
            System.out.println("you typed " + input);
        }
    }
}
```

converts an
InputStream
to a Reader



Reading by line from file

```
import java.io.BufferedReader; /*Reader2.java */
import java.io.InputStreamReader;
import java.io.FileInputStream;
public class Reader2{
    public static void main(String[] args) throws Exception{
        /* convert a FileInputStream, which is an InputStream,
           to a Reader by wrapping in InputStreamReader,
           then wrap everything in BufferedReader and call
           the readLine method to get a line at a time
        */
        String input;
        String file = args[0];
        BufferedReader bin = new BufferedReader
            (new InputStreamReader
             (new FileInputStream(file)));
        while ( (input = bin.readLine()) != null){
            System.out.println(input);
        }
    }
}
```

Serialization

Serialization

- You can read and write *objects* to files/network
- Object I/O goes by the awkward name of serialization/marshaling
- Serialization in other languages can be *very* difficult, because objects may contain references to other objects
- Java makes serialization (almost) easy

What happens ?

- Serialization
 - All attributes values are copied
 - Only *transient* attributes are not copied ! Such as data specific to the current system and that cannot be used to reify an object elsewhere (file handles, ...)
- Deserialization
 - A object of the same class/type is created/reified
 - Attributes are assigned the saved values

Conditions for serializability

- If an object is to be serialized:
 - The class must be declared as public
 - The class must implement *Serializable*
 - The class must have a no-argument constructor
 - All fields of the class must be serializable: either primitive types or serializable objects
 - The class should have a unique version ID !

Implementing the *Serializable* interface

- To “implement” an interface means to define all the methods declared by that interface, but...
- The *Serializable* interface does not define any methods!
 - Question: What possible use is there for an interface that does not declare any methods?
 - Answer: *Serializable* is used as a flag to tell Java it needs to do extra work with this class

Writing objects to a file

```
ObjectOutputStream objectOut =  
    new ObjectOutputStream(  
        new BufferedOutputStream(  
            new FileOutputStream(fileName)));  
  
objectOut.writeObject(serializableObject);  
  
objectOut.close( );
```

Reading objects from a file

```
ObjectInputStream objectIn =  
    new ObjectInputStream(  
        new BufferedInputStream(  
            new FileInputStream(fileName)));  
  
myObject = (itsType)objectIn.readObject( );  
  
objectIn.close( );
```

What else?

- Encrypted files, compressed files, files sent over internet connections, ...
- Exceptions! All I/O involves Exceptions!
- ```
try { statements involving I/O }
catch (IOException e) {
 e.printStackTrace ();
}
```

# Serializing Arrays ? (1/4)

- Arrays in Java are objects
  - Methods can be invoked on arrays (toString, equals, ...)
  - Copying an array into another one can be done using *System.arraycopy*
  - Arrays are serializable
  - Objects in arrays should also be serializable !

# Serializing Arrays ? (2/4)

```
// Serialize an int[]
```

```
ObjectOutputStream out = new ObjectOutputStream(new
 FileOutputStream("test.ser"));
```

```
out.writeObject(new int[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9});
```

```
out.flush();
```

```
out.close();
```

# Serializing Arrays ? (3/4)

```
// Deserialize the int[]
```

```
ObjectInputStream in = new ObjectInputStream(new
 FileInputStream("test.ser"));
```

```
int[] array = (int[]) in.readObject();
```

```
in.close();
```

# Serializing Arrays ? (4/4)

// Print out contents of deserialized int[]

```
System.out.println("It is " + (array instanceof Serializable) + "
 that int[] implements Serializable");
System.out.print("Deserialized array: " + array[0]);
for (int i=1; i<array.length; i++) { // foreach (int elt : array) ...
 System.out.print(", " + array[i]);
}
System.out.println();
```

# What about object references ? (1/2)

- Example : object containing a reference to another
  - Employee superclass
  - Manager subclass
  - Secretary subclass
- A Manager has a Secretary !
- What happens when we serialize a Manager object ?

# What about object references ? (2/2)

Employee Jean = new Employee(« Harry », ...)

Manager Paul = new Manager(« Paul .... », ...)

Paul.setSecretary(Jean);

- Algorithm used for serialization
  - All objects that are saved are given a serial number
  - When serializing an object, find out if the same object has already been serialized
  - If yes, just « write » its serial number, otherwise serialize it and « write » its serial number

# What if an object does not implement Serializable ? (1/3)

- Mark non serializable attributes as transient to avoid the *NotSerializableException*
- Overload the methods `writeObject` and `readObject`

# What if an object does not implement Serializable ? (2/3)

- Take a look at the *Serializable interface*

```
public class LabeledPoint implements Serializable {
 private String label;
 private transient Point2D.Double point;

}
```

# What if an object does not implement Serializable ? (3/3)

```
private void writeObject(ObjectOutputStream out) throws IOException
{
 out.defaultWriteObject();
 out.writeDouble(point.getX());
 out.writeDouble(point.getY());
}
```

```
private void readObject(ObjectInputStream in) throws IOException {
 in.defaultReadObject();
 double x = in.readDouble();
 double y = in.readDouble();
 point = new Point2D.Double(x, y);
}
```