OOP
Lecture 17
I/O and Graphics

Signe Ellegård Borch

Carsten Schuermann

IT University Copenhagen

Contents

- The challenges of I/O
- The design of the I/O library in Java
- Binary Streams
- Character Streams
- Random Access Files
- Class File - working with files and directories
- Object serialization and persistence
- Socket communication - streams over the network

The challenges of I/O

- I/O (short for Input/Output) is used whenever a program access sources external to itself - when it sends or recieves information from the outside.
- I/O is common to all programming languages, but designing a good I/O functionality is a very difficult task.
- The overall problem is to get the desired functionality without making the library overtly complex to comprehend and use.
- I/O is inherently complex, in that it has to face a number of different challenges:
- I/O should make it possible for the program to access data from different media, in different data formats, in sequence or random order, and provide means for adding buffering and other functionality.

The challenges of I/O continued

Different media

- **input from**: keybord, mouse, disc, own memory, other threads, other programs over the network
- **output to**: console, disc, own memory, other threads, other programs over the network

- Different data formats
  - Text (Unicode or some other encoding)
  - Binary data (pictures, sound, video, objects etc.)

- Sequential or random access
  - **Streams**: a stream has sequential access and is used for either writing or reading.
  - **Random access**: making it possible to both read and write in a random way.

- Different needs for buffering, filtering, etc.

The design of I/O in Java - the decorator pattern

- A lot of things might vary
- This could give a lot of classes - one for each combination of functionality.
- Instead, Java uses a so called *decorator pattern*, where the functionality is ”clicked together”.
- The idea is that you have a rather simple object with only the most basic funtionality. This object is then ”wrapped” with more functionality.
- Some decorators merely add new logic to the underlying stream but keep the same interface - others add new methods as well.
- By combining different objects you can ”customize” the streams for your particular needs.

Example with ”clicking” together streams:

- You first have to choose what media you want to work with: here we write to a file.
  - FileOutputStream
- If we want to write objects to that file:
  - ObjectOutputStream + FileOutputStream
- If we want to write primitive data:
  - DataOutputStream + FileOutputStream
- We might add buffering as well:
  - ObjectOutputStream + BufferedOutputStream + FileOutputStream
- Or:
  - DataOutputStream + BufferedOutputStream + FileOutputStream

Advantages and disadvantages of the design

- The advantage of Java I/O is that you can get complex behavior by combining a lot of rather simple classes.

- The trade-off is that this makes the library more difficult to use:

  – you have to understand the abstraction of ”clicking” together functionality

  – you have to know what combinations of functionality will suit your needs

  – you have to write more lines of code to accomplish even very common I/O tasks.

  – the order of clicking things together matters:
    - (OK) DataOutputStream + BufferedOutputStream + FileOutputStream
    - (NOT OK) BufferedOutputStream + DataOutputStream + FileOutputStream

Simple I/O example

```
import java.io.*;
public class WriteToFile{
    public static void main(String [ ] args) throws IOException{
        //WRITING TO FILE
        FileOutputStream fos = new FileOutputStream ( "f.dat");
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        DataOutputStream dos = new DataOutputStream(bos);
        dos.writeInt(30);
        dos.close( );
        //READING FROM FILE
        FileInputStream fis = new FileInputStream ("f.dat");
        BufferedInputStream bis = new BufferedInputStream(fis);
        DataInputStream dis = new DataInputStream(bis);
        int i = dis.readInt( );
        System.out.println( "Number read from file f:"+ i );
        dis.close();
    }
}
```

The Stream Abstraction

- A stream is an object which represents any data source you can read data from or any sink which can recieve data. The stream abstraction hides the details of what is actually happening to the data inside the I/O device.
- A stream has a sequential nature: you put data into streams in sequential order, and read data from streams in sequential order.
- There are basicly two kinds of streams: *binary streams* and *character streams*.
- Binary streams are used for writing and reading primitive data and objects, in machine-readable form (bytes). The Java objects used for this purpose are called InputStreams and OutputStreams .
- Character streams are used for reading and writing human readable text in some encoding (usually 16-bit Unicode). The corresponding objects in Java are called Readers and Writers.

Different media and their binary streams

The basic I/O classes in Java are made with respect to what media they communicate with.

These classes are seldomly used alone - they are usually decorated with some filter to get additional functionality.

Input

- Disk (FileInputStream)
- Own memory (ByteArrayInputStream)
- Other thread (PipedInputStream)
- Other program over the network (socket.getInputStream( ))
- Keybord (System.in)

Output

- Disk (FileOutputStream)
- Own memory (ByteArrayOutputStream)
- Other thread (PipedOutputStream)
- Other program over the network (socket.getOutputStream( ))
- Console (System.out, System.err)

ByteArrayInputStream & ByteArrayOutputStream

- ByteArrayInputStream and ByteArrayOutputStream allow a buffer in memory to be used as an inputstream or outputstream. All the data that you send to the stream is placed in this buffer.

- ByteArrayInputStream and ByteArrayOutputStream can e.g. be used when you make test cases:

  ```
  new ByteArrayInputStream("Read this in five words". getBytes( ));
  ```

  will make a stream with five words - this is easier than making a file, writing to it, and reading from it again.

- Also, it you have a program that is reading the same file many times, you might read it into a ByteArrayInputStream and read it from there.

FileInputStream & FileOutputStream

- FileInputStream and FileOutputStream are used when reading information from a file, or writing information to a file in bytes.
- This is a very common I/O task!
- Both FileInputStream and FileOutputStream need a reference to the file on which they are going to operate. This is given in the constructor as either the name of the file as a String, a File object or a FileDescriptor. We return to class File later on.
- A FileNotFoundException will be thrown if the file does not exist.

PipedInputStream & PipedOutputStream

- PipedInPutStream and PipedOutputStream are used when two threads want to communicate.
- They are always used together: a PipedInputStream is created from a PipedOutputStream or vice versa.
- A pipe end can only be connected once, and if one of the ends dies, the other end cannot be used anymore(an IOException will be thrown if anybody attempts to do so).
- If the two ends are not equally fast at writing or consuming data, the write() and read() methods block, waiting for the other end to provide or be ready to consume more data.

Decorating the basic binary streams

- Some of the objects you want to click onto your basic streams are the filter objects extending FilterInputStream.
- Most of them are very special purpose filters, that we will not go into detail with here.
- The most useful ones to know are:
  - BufferedInputStream and BufferedOutputStream
  - DataInputStream and DataOutputStream
  - PrintStream

BufferedInputStream and BufferedOutputStream

- Decorating a stream with a BufferedInputStream or a BufferedOutputStream will tell the stream to use buffering, so you don't get a physical read or write everytime you read from or write to the stream.
- This is very often used for efficiency reasons.
- In some situations the use of buffering will improve efficiency dramatically, in other situations it does not matter much.
- This is because some of the I/O classes provides some buffering themselves.
- Also, the streams writing to memory and other threads are not influenced much by buffering, since they operate direct on the memory, which is very fast.
- JP recommend that you always use buffering when working with files and sockets.

DataInputStream and DataOutputStream

- The DataOutputStream has methods for outputting primitive data (e.g. int and double) in bytes in a machine-independent way.
- The DataInputStream has methods for reading the bytes of a stream and converting them back into primitive types.
- The DataOutputStream put data elements on a stream in a way that DataInputStreams can portably reconstruct them.
- Implement the interfaces DataInput and DataOutput (see page 122 in JP).

PrintStream

- A PrintStream prints primitive data types and Strings in a viewable format.
- It has the methods print() and println(), which are overloaded to print all the various types.
- It does not throw IOExceptions like the other streams.
- System.out and System.err are examples of PrintStreams.
- If you need to print in a viewable format yourself, you should use a PrintWriter instead.

Basic Character Streams

- The classes for handling character streams are called Readers and Writers.
- One of their most important properties is that they are able to handle 16-bit Unicode characters (as well as other encodings).
- Like with the binary streams there exist some basic classes corresponding to the three basic way of communicating:
  - for thread communication, use PipedReader and PipedWriter
  - when working with files, use FileReader and FileWriter.
  - For input and output to memory, use StringReader and StringWriter, and CharArrayReader and CharArrayWriter (all of these are operating on buffers in memory).

Example with StringReader and StringWriter

```
import java.io.*;
public class WriteToString{
  public static void main(String [ ] args)throws IOException{
      //READING FROM A STRING
      Reader r = new StringReader("hey");
      System.out.println((char)r.read( )+ (char)r.read()  + (char)r.read( ) );
      //WRITING TO A STRING
      Writer sw = new StringWriter( );
       sw.write("h"); sw.write("e"); sw.write("y");
       System.out.println(sw.toString( ));
    }
}
```

Decorators on character streams

- Like the binary streams, the character streams also has a number of objects that can be ”clicked” onto them to provide additional functionality.

- There exists a number of special purpose character stream decorators. The ones that you will probably use the most are:
  - BufferedReader and BufferedWriter
  - InputStreamReader and OutputStreamWriter
  - PrintWriter

BufferedReader & BufferedWriter

- BufferedReader and BufferedWriter provides buffering, and as with the buffered streams, they are used for efficiency reasons.

- To make sure that the data is actually written to the stream, and not just remains in the buffer, you should always call flush( ) or close( ) when you are finished writing to the buffered stream.

- If anybody tries to write to a buffered binary stream or a buffered character stream after a call to close( ), an IOException will be thrown.

Wrapping binary streams as character streams

- Sometimes you want to convert a binary stream to a character stream.
- This is possible with the classes InputStreamReader and OutputStreamWriter.
- These classes are a sort of brigdes between the two basic kinds of streams: character streams and binary streams.
- In an OutputStreamWriter, chars are converted into bytes using a some encoding (either default or specified).
- In an InputStreamReader, a stream of bytes are converted into chars. A common example of this is when one wants to read characters from the standard input System.in.

```
import java.io.*;
public class StandardInputTest{
   public static void main(String [ ] args) throws
    IOException{
   System.out.println("Type some characters and
    press Enter");
   BufferedReader br = new BufferedReader(new
    InputStreamReader(System.in));
   String response = br.readLine( );
   System.out.println("You typed: " + response);
}}
```

Encodings

- With OutputStreamWriter and OutputStreamReader it is possible to change the 16-bit Unicode encoding which is default in Java.
- Not all systems support Unicode - therefore it is convenient to be able to convert between Unicode and other encodings.
- You can find out about the default encoding of the system in different ways:
  - You can call

```
System.getProperty ( ”file.encoding” );
```

  - Or you can make an OutputStreamWriter to check it:

```
OutputStreamWriter out = new OutputStreamWriter
(new ByteArrayOutputStream( ));
System.out.println(out.getEncoding( ));
```

Encodings continued

- You can convert from one encoding to Unicode by specifying the encoding you want to convert from in the constructor of the InputStreamReader.

- For example, to translate a text file in the UTF-8 encoding into Unicode, you create an InputStreamReader as follows:

```
FileInputStream fis = new FileInputStream ("test.txt");
InputStreamReader isr =
new InputStreamReader (fis, "UTF8");
```

- In this example, the encoding is given as a String. It can also be specified with a CharSet object representing the encoding.

- You can get an overview of the encodings supported by Java in the documentation of CharSet.

PrintWriter

- A PrintWriter is the writer you would normally use to print primitive data (int, double, char, boolean) and Strings in human readable form.
- A PrintWriter is somewhat similar to a PrintStream: both have overloaded print( ) and println( ) methods, corresponding to each type of primitive, and neither of them throws IOExceptions.
- The PrintWriter can be ”clicked” on either binary streams or character streams by giving it either an OutputStream or a Writer in its constructor.

StreamTokenizer

- There is no Reader in the I/O library that corresponds to PrintWriter (so that you can read primitive data and Strings instead of just characters from the character stream).

- Instead, use a StreamTokenizer with a Reader:

```
Reader r = new FileReader("f.txt);
StreamTokenizer st = new StreamTokenizer( r);
```

- A StreamTokenizer collects characters into tokens, which can then be printet out or further processed.

- The tokens are either numbers, words, strings, end-of-line, end-of-file.

StreamTokenizer example

```
import java.io.*;
public class StreamTokenizerExample {
  public static void main(String [ ] args){
    Reader r = new BufferedReader (new FileReader ( “f.txt”));
    StreamTokenizer stok = new StreamTokenizer ( r );
    stok.parseNumbers( );
¨   double sum = 0;
    stok.nextToken( );
    while ( stok.ttype != StreamTokenizer.TT_EOF) {
      if ( stok.ttype == StreamTokenizer.TT_NUMBER)
        sum +=stok.nval;
      else
        System.out.println (“Nonnumber: “ +  stok.sval );
      stok.nextToken( );
    }
    System.out.println ( “The file sum is “ + sum );
}}
```

Random Access Files

- The class RandomAccessFile is a bit off by itself in the I/O library:
  - it is used for both input and output.
  - it does not access the data in sequence like the streams, but can access it in any order.
  - It operates on a special kind of file called a random access file.
- It implements the interfaces DataInput and DataOutput
- A random access file is like a very big byte array stored in the file system.
- It has a pointer that can be moved to a specified point where the next read or write operation will then begin.
- The RandomAccessFile can be made in a read-only or a read-write mode.
- You will probably not use it very often (if ever).

Working with the file system - class File

- The class File is used as an abstraction over paths in the file system.
- It does not, as one might think, represent a file in the system, but can represent both *files* and *directories*.
- When making a File object, you give a pathname as a String in the constructor.
  - This name might be the path to a file or to a directory, or it migth not be the path anything
  - even if it is a file, it might not be accessable (it might be read or write protected).
  - Class File has got methods to check if the file is there, if it is a file or directory, and if it is accessable.

Class File - continued

- The String given in the constructor can be either the absolute or relative path name. Relative path names are resolved against the current directory.

- With class File you can perform a lot of the operations that you usually do with files and directories from inside your Java program:
  - You can create new files, delete files, and navigate the file system by getting the children or parents of the file.
  - You can also get the size of the file in bytes, and see when it was last modified.
  - You can make new directories, get a list of the subdirectories and files of a directory.

- It is possible to filter out the types files of your particular interest by implementing the FilenameFilter interface and giving this filter to the list ( ) or listFiles ( ) methods of class File

Object serialization 1

- It is possible to write objects to streams, and restore them from streams using an ObjectOutputStream or an ObjectInputStream.
- This is also known as *serialization* of objects.
- It is done by decorating some binary outputstream with an ObjectOutputStream:

```
FileOutputStream fos = new FileOuputStream (”objects.dat");
ObjectOutputStream oos = new ObjectOutputStream ( fos );
```

- For an object to be serialized, it is required that it implements the marker interface Serializable - otherwise, an NotSerializableException is thrown.

Object Serialization 2

- When an object is serialized, all its member fields (except the ones declared *transient*) are written to the stream.
- Static fields are *not* serialized !
- If the object has any references to other objects, these are serialized as well. The object reference graph is thus preserved. Remember that an inner object has a reference to its enclosing object.
- An ObjectOutputStream remembers which objects that have been written to it, and if the same object is written to it twice, it does not save it again, even when the state of the object has changed.
- If two objects have a reference to another third object are both serialized, the third object is only serialized once.

Object serialization 3

- Reading the object is also known as *de-serialization*.
- This is done by reading from an ObjectInputStream using the readObject( ) method. Remember to cast the object to its original type before you use it.
- In de-serialization, the values of the fields are restored, and the fields declared transient are given their default values.
- When de-serializing the object, the JVM needs the .class file of the object in order to be able to restore it properly. If it is not there, a ClassNotFoundException is thrown.

Object serialization 4

- De-serialization actually works like a deep-clone mechanism, in that you get an exact copy of the object originally written to the ObjectOutputStream AND a copy of all the objects that the original object referred to.
- Object serialization can be used for persistence of objects, so that their state can be preserved even when the program they belong to terminates.
- It is also used in distributed computing. In Remote Method Invocation (RMI) it is used to send object arguments and return values over the network.

Socket communication

- A socket is an abstraction representing one end of a connection between two processes.
- Sockets are used when different processes want to communicate, either on the same computer or over the network.
- The socket abstraction is not unique for Java, but is a common abstraction in networks.
- From a socket you can obtain both an OutputStream and InputStream.

Socket communication continued

- Sockets are often used in client/server architectures.
- When the client and server have established the socket connection, they can communicate in a bidirectional manner.
- In Java, client/server communication is based on two classes: a ServerSocket, that waits for a client to connect, and a Socket belonging to the client.
- When a client connects, the ServerSocket makes a new Socket that will serve the client.

Socket communication -a server

```
import java.io.*;
import java.net.*;
public class OOPServer {
   public static void main(String [ ] args) throws IOException{
      ServerSocket ss = new ServerSocket ( 2445);
      while ( true ){
          Socket s = ss.accept( );
          DataInputStream dis = new DataInputStream ( s.getInputStream( ));
          DataOutputStream dos = new DataOutputStream ( s.getOutputStream( ));
          int query = dis.readInt( );
          dos.writeInt(query * 2);
          dis.close( );
          dos.close( );
      }
   }
}
```

Socket communication - a client

```
import java.io.*;
import java.net.*;
public class OOPClient {
   public static void main( String [ ] args ) throws IOException{
        Socket s = new Socket ( “localhost”, 2445 );
         DataInputStream dis = new DataInputStream (s.getInputStream( ));
          DataOutputStream dos = new DataOutputStream (s.getOutputStream( ));
          dos.writeInt( 2 );
          int result = dis.readInt( );
          System.out.println(“Server replied: “ + result);
          dis.close( );
          dos.close( );
     }
}
```