JAVA I/O (SLR 201)

Patrick Bellot & Ada Diaconescu

Télécom ParisTech

The notion of stream

- **I/O in JAVA are based on the notion of stream.**
  - Input streams.
  - Output streams.

- **Streams allow a JAVA program to read from/ write to:**
  - Disk files on the computer file system.
  - Network sockets.
  - Program memory.

The notion of stream

- **Specialized streams allow to read/write several types of data:**
  - Byte streams for writing or reading raw binary data, i.e. Java bytes.
  - Character streams to read and write chars and strings, handling local character sets.
  - Data streams for writing and reading primitive data types in binary form.
  - Objects streams (serialization).

- **Streams can be buffered or not.**

Streams: the main principle

- All streams are basically byte streams.

- Then, it is possible to code characters, integers, floats, doubles, strings, serialized objects, etc. as sequences of bytes.

- Thus, the idea of JAVA I/O programming is to create a byte stream object (the underlying stream) and to wrap it inside a specialized byte coding or decoding object.

Example

- **If I want to write serialized objects:**

```
// Create a byte output stream (the underlying output stream)
FileOutputStream fileOutputStream = new FileOutputStream(“data.ser”) ;

// Create an object output stream from the byte output stream
ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream) ;

// Write an object
objectOutputStream.writeObject(object) ;

// Close the stream
objectOutputStream.close() ;
```

Byte streams

- To read / write bytes.

- Main read functions:
  - `int read() ;`
  - `int read(byte[] b) ;`
  - `int read(byte[] b, int off, int len) ;`

- Main write functions:
  - `void write(int b) ;`
  - `void write(byte[] b) ;`
  - `void write(byte[] b, int off, int len) ;`

Byte output streams

- `OutputStream` is the base class.

- Main subclasses are:
  - `ByteArrayOutputStream`
  - `FileOutputStream`
  - `ObjectOutputStream`

- Specialized subclasses are:
  - `PipedOutputStream` (for pipes between threads)
  - `FilterOutputStream` (wrapper for redefinitions)
  - `org.omg.CORBA.portable.OutputStream`

Byte output streams

- `FilterOutputStream` is a wrapper for `OutputStream`.

- Write methods of `FilterOutputStream` simply call the write methods of the underlying `OutputStream`.

- But these methods can be redefined in subclasses to provide specialized functions.

Byte output streams

```
public class FilterOutputStream extends OutputStream
{
   private OutputStream outputStream ;

   public FilterOutputStream(OutputStream outputStream)
   {
      this.outputStream  = outputStream  ;
   }

   public void write(int b)
   {
      outputStream.write(b) ;
   }

   public void write(byte[] b)
   {
      outputStream.write(b) ;
   }

   …

}
```

Byte output streams

- **Main subclasses of `FilterOutputStream`:**
  - `BufferedOuputStream`: buffered output.
  - `DataOutputStream`: for primitives types.
  - `CheckedOutputStream`: maintains a check sum.
  - `CipherOutputStream`: encrypted communications.
  - `DeflaterOutputStream`: compressed communication.
  - `PrintStream`: formatted data.

Byte output streams inheritance tree

- OutputStream
  - ByteArrayOutputStream
  - FileOutputStream
  - ObjectOutputStream
  - PipedOutputStream
  - org.omg.CORBA.portable.OutputSTream
  - FilterOutputStream
    - BufferedOutputStream
    - DataOutputStream
    - CheckedOutputStream
    - CipherOutputStream
    - DeflaterOutputStream
    - PrintStream

Example

- I want to write the portable binary representation of primitive data types in a file.
- I want the stream to be buffered because I will do many small write operations.

```
// Creates the base byte output stream
FileOutputStream fout = new FileOutputStream(“data.bin”) ;
// Wrap it to bufferize it
BufferedOutputStream bout = new BufferedOutputStream(fout)
// Wrap it to provide primitive data types write functions
DataOutputStream dout = new DataOutputStream(bout) ;
// Write…
dout.write(132) ;
dout.flush() ; // It is possible to flush.
dout.write(‘c’) ;
// Close
dout.close() ;
```

Character Output Streams

- Character output streams are used to write data of String and char types.

- JAVA internal memory character encoding uses Unicode.

- Character output streams automatically translate internal Unicode to the local character set. For us, it is usually utf-8 encoding or 8-bits encoding such as iso-latin1.

Character Output Streams

- The base class of all character output streams is the class `Writer`.

- Main write methods:
  - `void write(char c) ;`
  - `void write(char[] cbuf) ;`
  - `void write(char[] cbuf, int off, int len) ;`
  - `void write(String str) ;`
  - `void write(String str, int off, int len) ;`

Character Output Streams

- **Main sub-classes of `Writer`:**
  - `BufferedWriter`: bufferizes.
  - `CharArrayWriter`: write characters in memory.
  - `FilterWriter`: for redefinitions.
  - `OutputStreamWriter`: to write bytes.
  - `FileWriter`: to write bytes into a file.
  - `PipedWriter`: to write characters into a pipe.
  - `PrintWriter`: to write formatted data.
  - `StringWriter`: to build a String.

Example

- I want to write strings into a file using the local character set of my computer:

```
FileWriter fout = new FileWriter(“data.txt”) ;
```

- I want to write strings into a file using a different charset:

```
// byte output stream
FileOutputStream fout = new FileOutputStream(“data.txt”) ;

// wrapped in a character output stream
OutputStreamWriter sout = new OutputStreamWriter(fout,”UTF-8”) ;
```

Buffered Output Streams

- Most of the I/O we have seen are non-buffered.
- That means that all write method calls are turned into one of the underlying API calls.
- This could be inefficient when writing into a file or a network socket when the written data are small.
- Fortunately, and as usual, JAVA provides the appropriate tools.

Buffered Outputs Streams

- As we have already seen, `BufferedOutputStream` allows to bufferize byte output streams.

- And `BufferedWriter` allows to bufferize character output streams:

```
FileWriter     fout = new FileWriter(“data.txt”) ;

BufferedWriter bout = new BufferedWriter(fout) ;
```

Input Streams

- Did you understand the mechanisms and the principles of output streams?
- The mechanisms and principles of input streams are exactly the same!

Input Streams

- **Input streams can be:**
  - Byte streams for reading raw binary data, i.e. Java bytes.
  - Character streams to read chars and strings, handling local character sets.
  - Data streams for writing primitive data types in binary form.
  - Objects streams (serialization).

Byte Input Streams

- `InputStream`: **base class.**
  - `ByteArrayInputStream`.
  - `FileInputStream`.
  - `ObjectInputStream`.
  - `PipedInputStream`.
  - `StringBufferInputStream`.
  - `SequenceInputStream`: sequence of input streams.
  - `AudioInputStream`.
  - `org.omg.CORBA.portable.InputStream`.

  - `FilterInputStream`: the wrapper.

Byte Input Streams

- `FilterInputStream`: the wrapper.
  - `BufferedInputStream`.
  - `CheckedInputStream`.
  - `CipherInputStream`.
  - `DataInputStream`.
  - `InflaterInputStream`.
  - `ProgressMonitorInputStream`.
  - `LineNumberInputStream`.
  - Etc.

Character Input Stream

- The base class is `Reader`.
  - `BufferedReader`.
  - `InputStreamReader`: byte stream to character stream.
  - `CharArrayReader`.
  - `PipedReader`.
  - `StringReader`.
  - Etc.

Sockets

- Sockets are used for network connexions between two programs on one or two machines.
- Once created, a socket connection is very similar to everything we have seen.
- On each side of the connexion, we have an input stream and an output stream.

Sockets

- **Principle:**
  - One of the two programs called the server waits for a connexion on a given port ranging from 0 to 65535, unsigned 16 bits integer.
  - The other program called the client connects to the server on the given port.
  - The connexion demand is accepted by the server and each side has input and output streams.
  - Ports:
    - well-known ports 0-1023 are reserved to the admin.
    - registered ports 1024-49151 are used by registered applications.
    - dynamic ports: 49152 to 65535 can be freely used.

Server Socket

- Create the server socket:

```
ServerSocket serverSocket = new ServerSocket(port) ;
```

- Wait for a connexion:

```
Socket socket = serverSocket.accept() ;
```

- The Socket object contains all information on the connexion. It also contains the two byte streams:

```
InputStream  inputStream  = socket.getInputStream() ;
OutputStream outputStream = socket.getOutputStream() ;
```

- These byte streams can be used as any byte stream.

Client Socket

- Build address of the server:

```
InetAddress addr = InetAddress.getByName("java.sun.com");
```

- Create a socket:

```
Socket socket = new Socket(addr, port);
```

- Extract the byte streams from the socket. That’s all !

- Note that all socket operations must be enclosed in a try-catch block.

Timed Client Socket

- Build address of the server:

```
InetAddress addr = InetAddress.getByName("java.sun.com");
```

- Create a socket address object:

```
SockAddress sockAddr = new InetSockAddress(addr, port) ;
```

- Create a socket:

```
Socket socket = new Socket();
```

- Connect with a timeout:

```
socket.connect(sockAddr, 1000) ; // ms
```

Structure of a Web Server

```
class RequestProcessor extends Thread
{
   private Socket socket ;

   public void run()
   {
      try { // run() is not allowed to raise an exception

         // Buffered character input stream
         InputStream       is = socket.getInputStream() ;
         InputStreamReader ir = new InputStreamReader(is) ;
         BufferedReader    rd = new BufferedReader(ir);

         // Buffered character output stream
         OutputStream       os = socket.getOutputStream() ;
         OutputStreamWriter ow = new OuputStreamWriter(os) ;
         BufferedWriter     wr = new BufferedWriter(ow) ;

         // Then:
         // - read and decode input request
         // - build and write response
         // close the streams
      catch (Exception e) { … }
   }
}
```

Structure of a Web Server

```
public class WebServer
{
   public static void main(String args[])
   {
      ServerSocket serverSocket ;
      Socket socket ;
      do {
         try {
            serverSocket = new ServerSocket(80) ;

            socket = serverSocket.accept() ;

            RequestProcessor rp = new RequestProcessor(socket) ;
            rp.start() ;

         } catch (Exception e) { … }
         finally {
            socket.close() ;
            serverSocket.close() ;
         }
      } while (true) ;
   }
}
```

SSL Sockets

- It is possible to implement encrypted SSL sockets.
- A key store must be installed on your computer.
- JAVA provides the tools to build the key store.

```
keytool -keystore <fichier> -genkey -keyalg RSA
```

Server SSL Socket

```
// Initialization

System.setProperty("javax.net.ssl.trustStore",       “key file”) ;
System.setProperty("javax.net.ssl.keyStore",         “key file”) ;
System.setProperty("javax.net.ssl.keyStorePassword", “password”) ;

// Factory

SSLServerSocketFactory sf
        =(SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

// SSL server socket

SSLServerSocket serverSocket = (SSLServerSocket)sf.createServerSocket(port) ;

// Connexion

Socket socket = serverSocket.accept() ;
```

Client SSL Socket

```
// Factory

SSLServerSocketFactory sf
         =(SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

// Client socket

SSLSocket socket = (SSLSocket) sf.createSocket(”server", port);


// Protocol

socket.startHandShake() ;

// Go on !
```

- **That’s all folks !**