

Introduction to Computation and Problem Solving

Class 29: Introduction to Streams November 22, 2005

Prof. Steven R. Lerman
and
Dr. V. Judson Harward

Goals

Just as Java has a modern approach to error handling inherited from C++, Java communicates with the outside world using another C++ inspired technique called *streams*.

In this session, we will:

- Look at the classic stream code to read a file.
- Examine Java's architecture of stream classes
- Learn how to parse simple text input.

On Tuesday, we will have an active learning session on streams. Please review these notes before then.

JFileViewer

- `JFileViewer` is a small demonstration program that reads in a file and puts it up in a text area.
- It is composed of two classes:
 - `JFileViewer`: `main()` and main loop
 - `JTextViewer`: the (rudimentary) GUI

3

JFileViewer, core try block (in English)

```
try {  
    open a stream from the file  
    while there is more data  
        read it  
        append it to our GUI display  
    close the file  
}  
catch any I/O errors
```

4

JFileViewer, core try block (in Java)

```
try {
    FileReader in = new FileReader( args[ 0 ] );
    char [] buf = new char[ 512 ];
    int nread;
    while( ( nread = in.read( buf ) ) >= 0 )
        { view.append( new String( buf, 0, nread ) ); }
    in.close();
}
catch ( IOException e )
{
    handleErr( e );
}
```

5

Traditional I/O

The traditional approach uses different schemes depending on the type of the source or destination, e.g.,

- keyboard input
- screen output
- files
- interprocess pipes
- network sockets

6

Java I/O

- Java's preferred approach is to handle I/O using *streams* (pioneered in C++)
- Think of a stream as a data hose connecting the program to the outside world.
- It hides the details of the external data source or destination.
- Java allows you to treat some I/O distinctively, e.g. `RandomAccessFile`

7

Input vs Output Streams

- Streams are *unidirectional*
- An *input stream* controls data coming into the program from some source
- An *output stream* controls data leaving the program for some destination
- If you want to read and write the same destination, you use 2 streams

8



Streams and I/O Channels

Usually the other end of a stream leads to or arises from a platform-specific media service, for instance, a file system



9

Java Stream Classes

Java provides

- a set of abstract stream classes that define the stream *interfaces* for different kinds of streams:
 - `InputStream`: reads bytes
 - `OutputStream`: writes bytes
 - `Reader`: reads chars
 - `Writer`: writes chars
- a hierarchy of stream implementations:
 - that are tailored for a particular data source or destination, e.g., `FileReader` reads chars from a file
 - that add functionality to a preexisting stream (filter streams), e.g., `BufferedReader`

10

What Streams Share

- **Java Streams are FIFO queues**
 - Streams deliver information in the order it was inserted into the underlying channel
- **Basic Java streams only provide sequential access without rewind, backup, or random access**

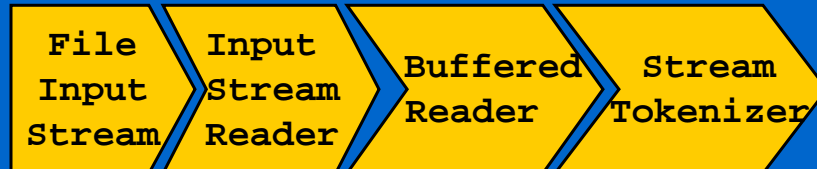
11

Coupling Streams

- **Java streams may be combined by using one stream as a constructor argument to another**
- **This is usually done to add functionality and/or convert the format or representation of the data**
- **Stream pipelines are constructed**
 - from the data source to the program or
 - from the data destination back to the program

12

Stream Pipeline, I



13

Stream Pipeline, II

- A `FileInputStream` reads “raw” bytes from a file
- An `InputStreamReader` converts a byte stream to characters
 - A `FileInputStream` coupled with an `InputStreamReader` equals a `FileReader`
- A `BufferedReader` buffers a character stream in memory for efficiency, and allows you to read line by line
- A `StreamTokenizer` parses a character stream into tokens

14

Constructing the Pipeline

```
FileInputStream f =  
    new FileInputStream( path );  
InputStreamReader i =  
    new InputStreamReader( f );  
BufferedReader b =  
    new BufferedReader( i );  
StreamTokenizer t =  
    new StreamTokenizer( b );
```

15

The 3 Flavors of Streams

In Java, you can read and write data to a file:

- as text using `FileReader` and `FileWriter`
- as binary data using `DataInputStream` coupled to a `FileInputStream` and as a `DataOutputStream` coupled to a `FileOutputStream`
- as objects using an `ObjectInputStream` coupled to a `FileInputStream` and as an `ObjectOutputStream` coupled to a `FileOutputStream`

16

Text Streams: Readers

FileReader methods:

- `public FileReader(String name)`
throws `FileNotFoundException`
- `public FileReader(File f)`
throws `FileNotFoundException`
- `public int read(char[] cbuf)` throws `IOException`
- `public int read()` throws `IOException`
- `public void close()` throws `IOException`

BufferedReader methods:

- `public BufferedReader(Reader in)`
- `public String readLine()` throws `IOException`

17

Text Streams: Writers

FileWriter methods:

- `public FileWriter(String name)` throws `IOException`
- `public FileWriter(File f)` throws `IOException`
- `public void write(char[] cbuf)` throws `IOException`
- `public void write(String s)` throws `IOException`
- `public void write(int c)` throws `IOException`
- `public void close()` throws `IOException`
- `public void flush()` throws `IOException`

BufferedWriter methods:

- `public BufferedWriter(Writer in)`
- `public void newLine()` throws `IOException`

18

DataInputStream Methods

- `public DataInputStream(InputStream in), e.g.`
`DataInputStream d = new DataInputStream(
 new FileInputStream("f.dat"));`
- `public boolean readBoolean() throws IOException`
- `public int readInt() throws IOException`
- etc

plus all the standard InputStream methods:

- `public int read() throws IOException`
- `public int read(byte[] b) throws IOException`
- `public int read(byte[] b, int off, int len)
 throws IOException`
- `public void close() throws IOException`

19

ObjectOutputStream Methods

- `public ObjectOutputStream(OutputStream out), e.g.`
`ObjectOutputStream d = new ObjectOutputStream(
 new FileOutputStream("f.dat"));`
- `public void writeBoolean(boolean b) throws
 IOException`
- `public void writeInt(int i) throws IOException, etc`
- `public void writeObject(Object obj) throws
 IOException`

plus all the standard OutputStream methods:

- `public void write(int i) throws IOException`
- `public void write(byte[] b) throws IOException`
- `public void write(byte[] b, int off, int len)
 throws IOException`
- `public void close() throws IOException`

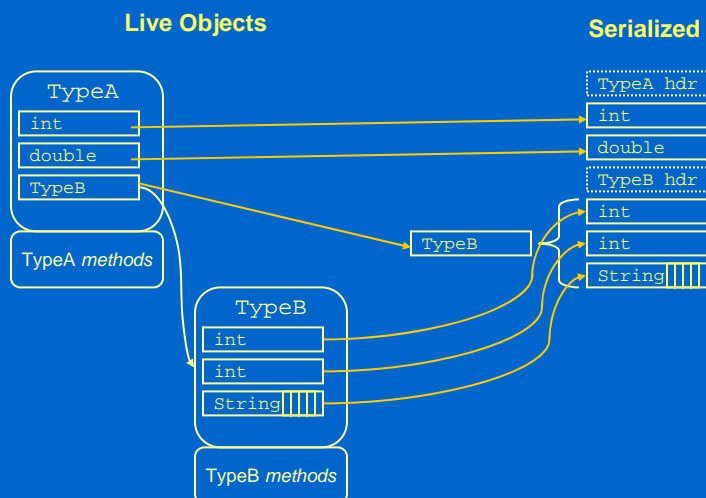
20

Serialization

- `ObjectInputStream` and `ObjectOutputStream` depend on a technique called object serialization.
- If you want to write out an object instance on a stream, you must write out the object's fields.
- The fields can contain native types or references to other object instances.
- You can recursively write out those references to contained instances.
- Eventually you can serialize any object instance (from a class that implements `Serializable`) into fields of native types

21

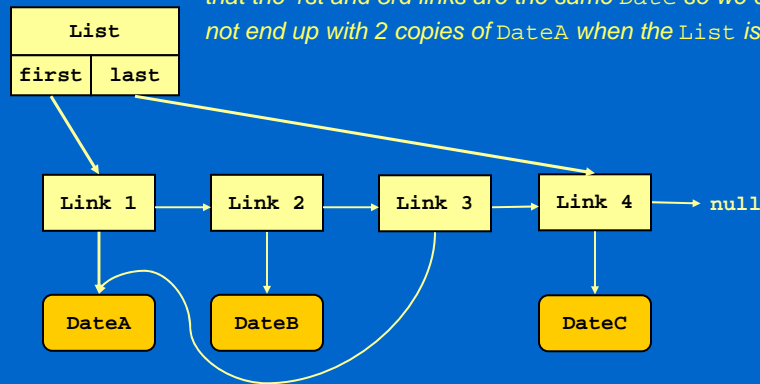
Serialization Diagram



22

Serializing a List

The serialized version must preserve the information that the 1st and 3rd links are the same Date so we do not end up with 2 copies of DateA when the List is restored.



23

StreamTokenizer

- Java supplies a special class called `StreamTokenizer` that accepts a `Reader` as a constructor argument.
- It breaks the input character stream into *tokens*, sequences of 1 or more contiguous characters that "belong" together.
- The user accesses these tokens by calling the `nextToken()` method, which always returns an `int` representing the *type* of the next token:
 - *word*,
 - *number*,
 - *end of file* (EOF),
 - *end of line* (EOL, optional), and
 - *otherCharacter*, returned as `int` value of the 16 bit character code

24

StreamTokenizer, 2

- When a `StreamTokenizer` recognizes a word, the public member `sval` contains the `String` representing the word.
- When a number token is recognized, public member `nval` contains its value.
- `StreamTokenizer` also ignores *whitespace* (blanks and tabs) and C, C++, and Java style comments by default.
- `StreamTokenizer` instance methods can change the definition of “word” or “number” and can turn on or off features like ignoring comments.

25

JPolygonPlotter

- `PolygonPlotter` is a sample program that illustrates how to handle formatted input.
- It reads a data file that describes a set of polygons and then plots them in a window.
- The command line accepts 3 arguments: width and height of the display window and the name of the data file.

26

JPolygonPlotter, 2

- The data file consists of a series of polygon definitions separated by blank lines.
- Each polygon definition consists of lines containing two integers each, an x- and a y-coordinate.
- The program checks for improperly formatted input. If it can not make sense of what it is reading, it throws an `DataFormatException`.
- In the `catch` clause it sends an error message to the console identifying where in the input file it got confused.

27

JPolygonPlotter, main()

```
public class PolygonPlotter {
    private static PolygonViewer view;
    private static int width = 0;
    private static int height = 0;
    private static FileReader fileRdr;

    public static void main( String[] args ) {
        . . .
        view = new PolygonViewer( width, height );
        readPolygons( args[ 2 ] );
        . . .
    }
}
```

28

JPolygonPlotter, readPolygons()

```
private static void readPolygons( String f ) {
    try {
        fileRdr = new FileReader( f );
        BufferedReader bufRdr =
            new BufferedReader( fileRdr );
        StreamTokenizer tokens =
            new StreamTokenizer( bufRdr );
        tokens.eolIsSignificant( true );

        main loop

        bufRdr.close();
    } catch ( IOException e ) {
        handleErr( e );
    }
}
29
```

JPolygonPlotter, readPolygons() main loop in English

```
try {
    get a token
    while we haven't reached the end of file
    if line is blank, skip it
    else if the line starts with a number
    extract the next polygon from the data
    else throw a DataFormatException
} catch ( DataFormatException e ) {
    write an error message and exit
}
```

30

JPolygonPlotter, readPolygons() main loop

```
try {
    tokens.nextToken();          // get a token
    while ( tokens.ttype != StreamTokenizer.TT_EOF ){
        if ( tokens.ttype == StreamTokenizer.TT_EOL ) {
            // if line is blank, skip it
            tokens.nextToken();
        } else if ( tokens.ttype == StreamTokenizer.TT_NUMBER ) {
            // if line starts with a number, parse polygon
            view.addPolygon( extractPolygon( tokens ) );
        } else
            throw new DataFormatException();
    }
} catch ( DataFormatException e ) {
    System.err.println( "Can't read polygon @ lineno " +
        tokens.lineno() + "." );
    System.exit( 1 );
}
```

31

JPolygonPlotter, extractPolygons()

```
private static Polygon extractPolygon( StreamTokenizer t )
    throws DataFormatException, IOException {
    Polygon p = new Polygon();
    do {
        int x = ( int ) t.nval;
        if ( t.nextToken() != StreamTokenizer.TT_NUMBER )
            throw new DataFormatException();
        int y = ( int ) t.nval; y = height - y - 1;
        if ( t.nextToken() == StreamTokenizer.TT_EOL ||
            t.ttype == StreamTokenizer.TT_EOF ) {
            p.addPoint( x, y );
        } else throw new DataFormatException();
    } while ( t.ttype == StreamTokenizer.TT_EOL &&
        t.nextToken() == StreamTokenizer.TT_NUMBER );
    return p;
}
```

32