

# Java libraries

## *Miscellany*

- lang
  - throwable errors and exceptions
- io
  - further aspects of the i/o libraries

# Throwables Errors & Exceptions

## Throwables

### Throwable

- Error
  - LinkError (10 subclasses), VirtualMachineError (4 subclasses), AWTError, ThreadDeath
- Exception
  - RuntimeException (>10 subclasses)
    - null 'pointer', invalid array index, ...
  - IOException (≈10 subclasses including those relating to networking)
  - NoSuchMethodException, ClassNotFoundException, InterruptedException, ...
  - *your exception classes*

## Throwable - Error

- ThreadDeath Errors
  - can occur when thread terminated by another thread
  - sometimes are caught to allow special tidying up; must be re-thrown to get termination to proceed
- Other Errors
  - not caught, should terminate interpretation of Java code

## Throwable - Exception - RuntimeException

- **RuntimeException**
  - NullPointerException, Security, NegativeArrayIndex, Arithmetic, IndexOutOfBounds, ...
  - *don't have to attempt to catch these*
  - *don't have to specify these in throws clause of function declaration*
- **Other Exception subclasses**
  - either
    - deal with exception in body of function where it may occur (try { ... } catch { ... } )
  - or
    - pass it on, but must then declare in throws part of function declaration



## Basic exception handling ...

- Code prototype

```
try {
    Operations that may fail
}
catch (exception type) { what to do for this type of failure }
catch (exception type) { what to do for other type of failure }
finally {
    odds and ends you want to do to tidy up irrespective of
    whether there were problems or succesful operation
}
```

## Basic exception handling ...

- Your catch clauses can exploit exception hierarchy to simplify code when you aren't interested in specific nature of failure ...

```
try { lots of things can go 'rong here ... }
catch (EOFException eofe) { ... }
catch (FileNotFoundException fnfe) { ... }
catch (IOException io) { ... }
catch (MyException me) { ... }
catch (Exception e) { ... }
```

*Try in order listed, IOException clause handles IO problems other than end of file or file not found*

## Basic exception handling ...

- Exception objects will
  - be able to print a stack trace so that you can see how the exception occurred
  - provide a message string with some details
- and may have other properties specific to particular subclasses.
- You can use these methods and properties in your catch() { ... } code.

**i/o**

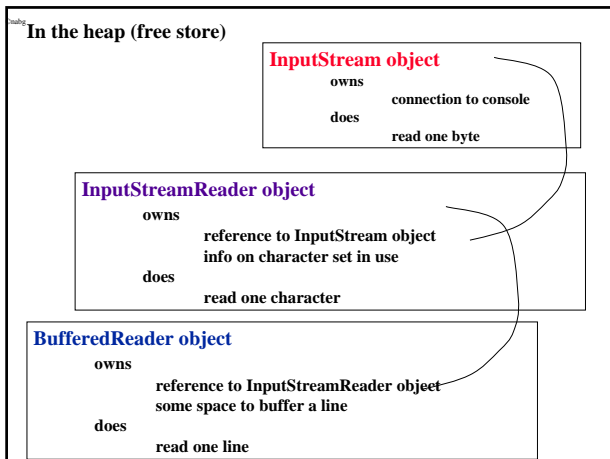
## Streams

- Horstmann claimed that there were 58 different “stream” classes in Java’s i/o package
  - *haven't counted myself, he was probably correct then; but more must have been added*
- Really two main groups
  - InputStream, OutputStream (byte oriented)
  - Reader, Writer (char oriented)
- Reader, Writer group is for Unicode text files.
- “Filters” (Leggo) model - build a stream with features you require by plugging streams together.

## Streams and filters – you’ve seen them in practice

```
BufferedReader input = new BufferedReader(
    new InutStreamReader(System.in);
```

- System.in – stream connected to keyboard, returns bytes
- Filters (Wrappers)
  - InputStreamReader (bytes to Unicode characters)
  - BufferedReader (readLine)



- Unicode related**  
(data type **char**)
- **Reader**
    - BufferedReader
      - LineNumberedReader
    - CharArrayReader
    - FilterReader
      - PushbackReader
    - InputStreamReader
      - FileReader
    - PipedReader
    - StringReader
  - **Writer** has similar hierarchy

- Unicode related - readers**
- Reading Unicode characters
  - Primary methods
    - read(char[], int, int)
    - read() returns **char**
    - close()
  - **Reader** *abstract class*
  - **BufferedReader**
    - give it a concrete Reader, BR adds more efficient buffering
    - supplies readLine() functionality

- Unicode related**
- **FilterReader** (abstract), **PushbackReader**
    - provide “unread()”
    - when writing things like parsers or lexical scanners, it is often useful to be able to “peek” at next character
      - can read next character and store somewhere special
        - may end up with lots of special case tests: *if previously read character then ... else ...*
      - can read next character, and put it back into stream (“unread”) if don’t want that character yet

- Unicode related**
- **StringReader**
    - gives you ability to “read” from a String
  - **PipedReader**
    - pipes are the recommended mechanism for passing character streams between threads
    - similar to pipes for interprocess communication on Unix,
    - pipe acts as buffer, one thread (process) writes to pipe (and block if buffer full), other thread (process) reads from pipe (and block if buffer empty)

- Unicode related**
- **InputStreamReader**
    - bridge between byte stream (other hierarchy) and Readers
    - reads and translates bytes
      - either
        - use default system encoding
        - chosen encoding (String to name “alphabet”)
          - 8559\_1 ISO Latin-1 (ordinary ASCII)
          - 8559\_6 ISO Latin/Arabic
          - Cp1257 Windows Baltic
          - Cp861 PC Icelandic
          - ...

## Unicode related

- If you are handling text (Strings etc) you should use the Unicode Reader/Writer classes.
- Some of the classes in other hierarchy have methods for reading Strings,
  - mostly “deprecated” methods
  - would fail with some Unicode files.

## Byte stream related

### *InputStream hierarchy (simplified!)*

- **InputStream**
  - ByteArrayInputStream
  - FileInputStream
  - FilterInputStream
    - approx 10 subclasses (some on next slide)
  - PipedInputStream
  - SequenceInputStream
    - *StringBufferInputStream* (don't use, instead use *StringReader*)
- OutputStream hierarchy is similar

### *InputStream hierarchy (simplified!)*

- **InputStream**
  - ByteArrayInputStream
  - FileInputStream
  - FilterInputStream
    - BufferedInputStream
    - CheckedInputStream (defined in java.util.zip)
    - DigestInputStream (defined in java.security)
    - InflaterInputStream (defined in java.util.zip)
      - GZIPInputStream
      - ZipInputStream
    - PushbackInputStream
    - DataInputStream
    - *LineNumberInputStream* (don't work properly, use *LineNumberReader* from Reader hierarchy)

## Byte stream related

- In addition to main stream part of hierarchy have interfaces such as:
  - **DataInput**
    - defines (abstract) methods like readShort, readChar, ...
  - **ObjectInput**
    - abstract method readObject
  - **Serializable** (and **Externalizable**) interfaces
    - for classes that want their objects to be persistent
- and special classes such as **ObjectInputStream** and **RandomAccessFile**

## Byte stream related

- Reading bytes
- Primary methods
  - read(byte[], int, int)
  - read() returns **byte**
  - close()
- **InputStream** abstract class

## Byte stream related

- **PipedInputStream**
  - similar to PipedReader but using byte data rather than char
  - possibly more useful as can package inside a *DataInputStream* and so have threads that read and write things like int, double
- **ByteArrayInputStream**
  - “read” from an array of bytes
- **StringBufferInputStream**
  - don't use; use *StringReader*
  - idea was same, give it a String, do read() on String;
  - it doesn't handle character codes correctly

## Byte stream related

- **FileInputStream**
  - InputStream reading bytes from a file or filedescriptor
- **SequenceInputStream**
  - Exotic! It concatenates several InputStreams, making them appear as single continuous stream (give it an Enumeration that works through the available streams in sequence)

## FilterInputStream

- **FilterInputStream**
  - BufferedInputStream
  - PushbackInputStream
  - CheckedInputStream (defined in java.util.zip)
  - InflaterInputStream (defined in java.util.zip)
    - GZIPInputStream
    - ZipInputStream
  - DigestInputStream (defined in java.security)
  - **DataInputStream**
  - *LineNumberInputStream (don't work properly, use LineNumberReader from Reader hierarchy)*

## Filters

- **BufferedInputStream**
  - like BufferedReader, adds buffering to existing stream for efficiency
- **PushbackInputStream**
  - Add unread() function to appropriate InputStreams

## util.zip streams

- These support compressed files – advantage when downloading data across network
  - InflaterInputStream
    - Created from an InputStream and an **Inflater** object that reverses compression applied to file
  - GZIPInputStream
  - ZipInputStream
  - *These specialized InflaterInputStreams handle variants on common Zip and GZIP compression formats.*

## util.zip streams

- **CheckedInputStream**
  - Will calculate checksum while you read the contents of a file
    - checksum provides a simple integrity check
    - can often detect changes to small number of bits such as might be caused by some form of electronic noise

## java.security stream

- **DigestInputStream**
  - a “message digest” provides a much more effective check on file integrity than does a simple checksum (digests are discussed briefly in lecture on “security”)
  - a digest will be a 128 bit (or 160 bit or similar) “signature” for file
    - each byte in file contributes to digest value
    - a change of 1-bit in a byte can result in changes to as many as half of the bits in the digest
  - *as stream is read, bytes combined; at end of stream, can get digest value*
  - *if can obtain original digest, comparison of original and calculated values will reveal any changes to data*

## DataInputStream

- **DataInputStream** extends `FilterInputStream` implements **DataInput**
- `DataInputStream` and `DataOutputStream` are used for “binary files”
- “Binary files”
  - avoid cost of converting internal to text representations
  - save space
    - eg a long will take 8-bytes (since a long could be as large as 9,223,372,036,854,775,807L it could take 20 bytes if translated to text)

## DataInputStream / DataOutputStream

- If you want *Java program to Java program* communication then use `DataInputStream`, and `DataOutputStream` combinations.
- First program writes to `DataOutputStream` (which utilises a `FileOutputStream`, or `PipedOutputStream` or a URL-connection)
- second program reads with a `DataInputStream` (which utilises a `FileInputStream`, or `PipedInputStream` or a URL-connection)

## DataInputStream

- Most functions straightforward:
- `read(byte[])`
  - Reads up to `byte.length` bytes of data from this data input stream into an array of bytes.
- `readBoolean()`
  - Reads a boolean from this data input stream.
- `readByte()`
  - Reads a signed 8-bit value from this data input stream.
- `readChar()`
  - Reads a Unicode character from this data input stream.
- `readDouble()`
  - Reads a double from this data input stream.
- ...

## DataInputStream

- `readFloat()`
  - Reads a float from this data input stream.
- `readInt()`
  - Reads a signed 32-bit integer from this data input stream.
- `readLong()`
  - Reads a signed 64-bit integer from this data input stream.
- `readShort()`
  - Reads a signed 16-bit number from this data input stream.
- `readUnsignedByte()`
  - Reads an unsigned 8-bit number from this data input stream.
- `readUnsignedShort()`
  - Reads an unsigned 16-bit number from this data input stream.
- `readLine()`
  - Reads the next line of text from this data input stream. *Deprecated.*

## DataInputStream

- `readUTF()`
  - Reads in a string that has been encoded using a modified UTF-8 format.
- `DataOutputStream` and `DataInputStream` have functions for dealing with Unicode (“16 bit character”) strings in a space efficient manner (using a format that is a slight modification of standard known as UTF-8).
- (Compression factor for coding strings makes it rather difficult to estimate how many bytes a string will occupy in file.)

## DataInputStream

- Modified UTF-8.
  - All characters in the range `\u0001` to `\u007F` are represented by a single byte:
    - 0, 7 bits of character
  - The null character `\u0000` and characters in the range `\u0080` to `\u07FF` are represented by a pair of bytes:
    - 110, bits 6-10
    - 10, bits 0-5
  - Characters in the range `\u0800` to `\uFFFF` are represented by three bytes:
    - 1110, bits 12-15
    - 10, bits 6-11
    - 10, bits 0-5

## DataInputStream, DataOutputStream – real values

- Note feature of following
  - DataOutputStream.writeDouble()
  - DataInputStream.readDouble()
  - and corresponding Float() functions
- Nominally, data value in file is long (or int); created via Double.doubleToLongBits()

## RandomAccessFile

- class RandomAccessFile extends Object implements **DataInput**, **DataOutput**
- Has all expected input and output functions from readBoolean() to writeUTF().
- Also
  - getFilePointer()
    - Returns the current offset in this file.
  - seek(long)
    - Sets the offset from the beginning of this file at which the next read or write occurs.

## RandomAccessFile

- Random access files convenient for “fixed size records”
  - variable length UTF strings are potential source of problems
  - solution A
    - Horstmann illustrates a scheme for padding so UTF string occupies a fixed block
  - solution B
    - only use in association with an index file, when create the RandomAccessFile can keep track of position with getFilePointer(); store record positions (along with any “key” if desired) in separate index file.

# Object i/o

## Object i/o

- Attractive if could simply say “*Save yourself on this file*” to an object - and have it save itself.
- Now Java compiler knows structure of an instance of class and so could (in principle) automatically generate the “saveMyself(DataOutputStream)” function

```
class RectItem {  
    ...  
    private int fX, fY;  
    private short fR, fG, fB;  
    private boolean fFilled;  
};  
  
    public void saveMyself(DataOutputStream do) {  
        do.writeInt(fX); do.writeInt(fY);  
        do.writeShort(fR); ...  
        do.writeBoolean(fFilled);  
    }
```

## Object i/o

- Well, *maybe*.
- It is obviously *easy* if all the data members are simple built in types.
- *But what about members that are instances of other classes?*
- *And what about heterogeneous collections?*



## Object i/o

### 1) heterogeneous collection

- You have “DataCollection” - a collection of RectItem, LineItem, TextItem etc
  - it actually uses a Vector to store the items
- We would want:
  - length of vector written
  - then for each item in vector
    - its class type
    - its data
- Could then read data back if could create instances of classes for names that we encounter

Object i/o 1) heterogeneous collection



Vector containing objects

- Expected output file:

```
4
PolygonItem      data defining fill colour, and peripheral points of poly
... ..
RectItem         data defining fill colour, and dimensions of rect
... ..
TextItem
avcd a           data defining fill colour, and location of text
... ..
LineItem         data defining end points of Line
... ..
```

Object i/o 1) heterogeneous collection

## reading the file

- Could deal with such a file in an “ad hoc” manner (similar to code in earlier examples):

```
read N
for(i = 0 ; i < N; i++)
  read nameString
  if nameString.equals("LineItem") {
    LineItem aline = new LineItem(); aLine.readMyself(in);
    myItems.add(aline);
  }
  else if nameString.equals("PolyItem") { ... }
  else ...
```

Object i/o 1) heterogeneous collection

## reading the file

- java.lang.Class
  - contains functions to make such code more systematic
    - public static native Class forName(String className) throws ...
      - get a “Class” object associated with className
    - public native Object newInstance() throws ...
      - get an instance object of given class

Object i/o 1) heterogeneous collection

## reading the file

- Using this standard functionality, can make the input code something along following lines (exception handling try ... catch blocks omitted)

```
read N
for(i = 0 ; i < N; i++)
  read nameString
  Class theClassObject = Class.forName(nameString);
  Object theObject = theClassObject.newInstance();
  DataItem di = (DataItem) theObject;
  di.readMyself(in);
  theItems.add(di);
```

Object i/o 1) heterogeneous collection

## Object i/o class identifiers in file

- Java compiler can arrange for something equivalent to those class names
  - when object to write itself to file, it starts by outputting class name; then saves data members
  - when reading a file: read class name, instantiate a suitable object, let it read the rest of the data.

Object i/o

## 2) references to other objects

- A simple (but inadequate) approach for a compiler to generate, automatically, the required “save” and “restore” functions would be for it to put write (and related read) requests for all referenced collaborators and components:

```

class RectItem {
    ...
    private String fName;
    private Point fPos;
    private Color fColor;
    private boolean fFilled;
};

void writeMyself(DataOutputStream o)
{
    "save my type"
    fName.writeMyself(o); fPos.writeMyself(o);
    ...
}

void readMyself(DataInputStream in)
{
    Object something = getNextObject(in);
    if(something is instanceof String) fName =
        (String) something;
    else throw UnexpectedInput ...;
}

```

Object i/o 2) references to other objects

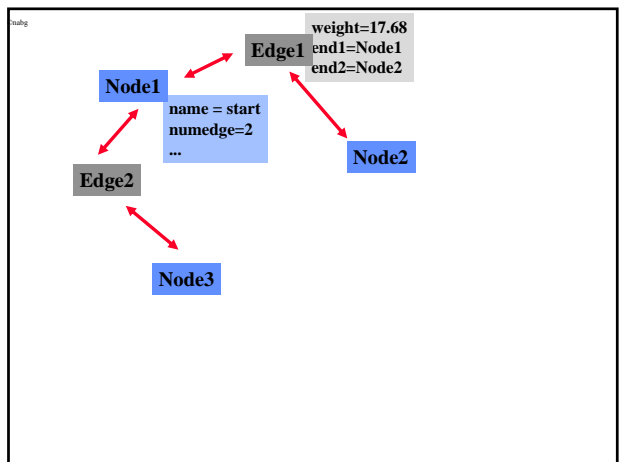
## references to other objects

- It isn't appropriate to simply rush off and save every referenced object.
- Some referenced objects are collaborators that really don't represent state data
  - eg a link to a Canvas that is to be told to “repaint()” if this data object changed
- You really wouldn't want to save the state of the Canvas object, if you read the data object back into memory it will be associated with a different Canvas held in a different window.
- Simple solution: **transient** type qualifier.

Object i/o 2) references to other objects

## references to other objects

- In other situations, you may find several references to the same object; consider for example
  - class Graph
    - owns a collection of Node objects
  - class Node
    - owns a collection of Edge objects
    - ...
  - class Edge
    - owns references to its start Node, its end Node,
    - owns a weight, ...



Process	Outputfile
Start saving:	
Node1.saveMyself	
output class	Node
output name	String "start"
output edge count	2
for i=0;i<edgecount;i++	
myEdge[i].saveMyself	
Edge1.saveMyself	
output class	Edge
output weight	17.68
end1.saveMyself	
Node1.saveMyself	
output class	Node
output name	String "start"
...	

Object i/o 2) references to other objects

## references to other objects

- Objects may have mutual references (like the Node and the Edge in the example) or may form parts of more complex directed graphs.
- You can't simply output each object at the point where you encounter a reference.

## References to objects: output

- The solution to this problem has been around for years (originating in Smalltalk, in C++ libraries since 1988)
- You perform i/o using special streams
  - streams have associated “dictionaries”
  - when find have to output an object (referenced via a pointer or “object reference variable”):
    - check with the stream: Has this object been output already?
    - If No, make entry in “dictionary” (sequence number, object id), then output the object
    - If Yes, simply output a tag “*previously written obje ct*” and the sequence number from dictionary

## References to objects: input

- Special input stream also associated with a “dictionary”
- This stream provides a “getNextObject()” function.
  - getNextObject() reads from the stream
    - if finds a class name,
      - create object of that class
      - tell the new object to read its data
      - register new object in “dictionary” along with index number
      - return reference to new object
    - if finds a “previously written object” tag, read object sequence number, return reference to existing object as found in dictionary

## C++ implementations

- C++ implementations of scheme have been around since Gorlen’s NIH class library (1988) and are available as part of common environments on PCs etc
  - use modified stream classes
  - require some form of run time type identification (since RTTI only recently standardized, C++ libraries did their own thing; *result: object i/o for different libraries is incompatible*)
  - generally required use of macros to insert standard functions in classes utilising object i/o
  - required the program to write both output and input functions for each class using object i/o
    - output statements for simple data members
    - invoking the “object save yourself” mechanism for pointer fields

## Java implementation

- Standardized.
- Automatic.
- Simply
  - define your class as one that implements **Serializable**
  - use transient type qualifier for any references that are not “state” data and which are not to be saved
- Use **ObjectInputStream, ObjectOutputStream**

## Java implementation

- Java adds some features not typical of earlier implementations –
  - data saved to define class contains signature information related to current class definition
  - this information prevents corruption when an old data file read with revised program
    - class in revised program will have changed signature (resulting from for instance the addition of an extra data member)
    - signature won’t match that in file
  - can even provide a “versioning” mechanism allowing revised program to recognise and handle old files

## Java implementation

```
DataThing myDataThing = new DataThing(345, "Hello World", 1.);
...
ObjectOutputStream outObjs = new ObjectOutputStream(
    FileOutputStream("xyst.dat"));
...
outObjs.writeObject(myDataThing);
```

- Simply define class DataThing as implements Serializable

## Java implementation

```
DataThing myDataThing = null;
...
ObjectInputStream inObjs = new ObjectInputStream(
    FileInputStream("xyst.dat"));
...
myDataThing = (MyDataThing) inObjs.readObject();
```

- *(Note need for cast on read)*

## Java implementation

- Programmer can intervene
  - define functions `readObject(...)` and `writeObject(...)` for own classes
    - these to call `defaultReadObject()` etc
    - can do extra work like validating data read
  - can take over almost the entire mechanism
    - define `readExternal()`, `writeExternal()`
    - automatic mechanism then deals solely with saving details of class of object
- Such intervention only needed in specialised contexts.

# Composing i/o streams

## Composing your stream

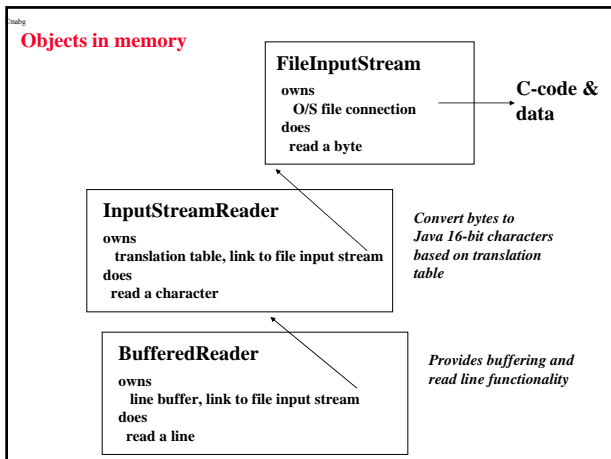
- You slot stream classes together to build the stream that you want –
  - reading Unicode from a file
    - `FileInputStream`
    - `InputStreamReader`
    - `BufferedReader`
  - reading binary data from a compressed file
    - `FileInputStream`
    - `ZipInputStream`
    - `DataInputStream`

## Composing your stream

- Create the object that connects to source of bytes,
- Package inside object that provides additional functionality

## Examples

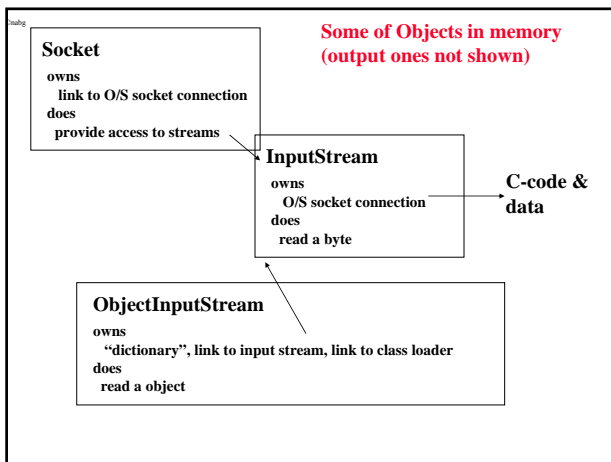
```
File f = new File(filename);
if(!f.canRead()) { ... /* exit */ }
FileInputStream fStream = null;
try {
    fStream = new FileInputStream(f);
}
catch (FileNotFoundException e) {
    System.out.println("No such file?");
    System.exit(1);
}
BufferedReader datareader = new
    BufferedReader(new InputStreamReader(fStream));
```



## Examples

```

public ClientHandler(Socket s)
{
    try {
        input = new
            ObjectInputStream(s.getInputStream());
        output = new
            ObjectOutputStream(s.getOutputStream());
    }
    catch(IOException ioe) {
        ...
    }
}
  
```



## “Feature” when opening ObjectStreams for sockets

- Build the ObjectOutputStream first
- flush the new output stream
- Then build the ObjectInputStream
- Otherwise Java I/O subsystem may “hang”

```

output = new
    ObjectOutputStream(s.getOutputStream());
output.flush();
input = new
    ObjectInputStream(s.getInputStream());
  
```