Effective Java Programming

efficient file handling

Structure

- efficient file handling
  - streams (input-output)
  - buffering streams
  - free access
  - buffers and channels (new input-output NIO)
  - memory mapped files
  - serialization

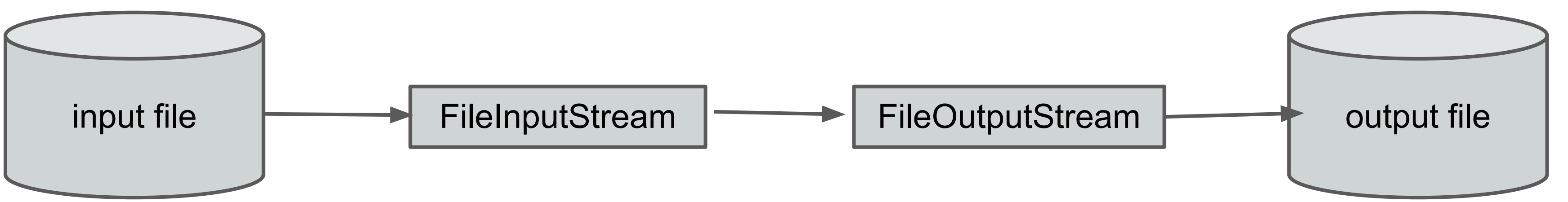
Streams - basics

- direction: input or output
- binary operations - *InputStream, OutputStream*
- low-level - work on the resource
  - file, array of bits, socket, pipe ...
- high level - additional functionality
  - serialization, audio, caching ...
- text operations - *Reader, Writer*
- operations on the directory structure - *File*

Streams - example

- rewriting data from one file to another

```
String from = "SOURCE-PATH";
String to = "DESTINATION-PATH";
InputStream in = new FileInputStream(from);
OutputStream out = new FileOutputStream(to);
int data;
while ((data = in.read()) != -1) {
  out.write(data);
}
in.close();
out.close();
```

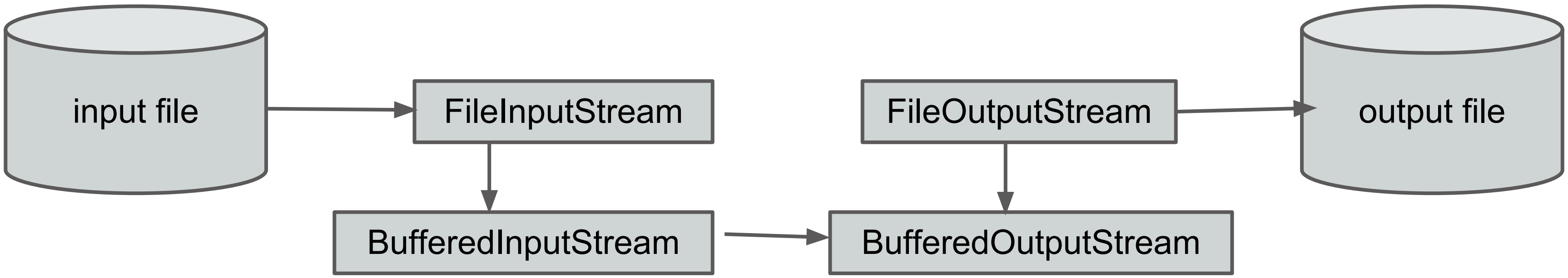


Streams - better example

- inefficient - byte by byte
- THERE IS NO DEFAULT CACHING
- a better solution?
- chain of streams with buffers
  - ready, tested implementation
  - a simple code
  - re-usable

Streams - better example - code

```
String from = "SOURCE-PATH";
String to = "DESTINATION-PATH";
InputStream in = new FileInputStream(from);
OutputStream out = new FileOutputStream(to);
in = new BufferedInputStream(in);
out = new BufferedOutputStream(out);
int data;
while ((data = in.read()) != -1) {
  out.write(data);
}
in.close();
out.close();
```



Streams - better example- problems

- better, but still a large number of requests
- would do better to transfer data portions
  - buffers allow you to work on arrays
  - streams can also work on arrays
- then why use buffers?
  - you have influence on your code
  - sometimes you need to pass the stream somewhere
  - you do not know how it is used there
  - even if on arrays, they may be too short
  - creating a buffer you have impact on its size!

Streams - even better

```
String from = "SOURCE-PATH";
String to = "DESTINATION-PATH";
InputStream in = new FileInputStream(from);
OutputStream out = new FileOutputStream(to);
// efficient but DANGEROUS
byte[] buffer = new byte[in.available()];
in.read(buffer);
out.write(buffer);
in.close();
out.close();
```

Streams - even, even better

```
final static int BUFFER_SIZE = 1024 * 1024;
// ...
String from = "SOURCE-PATH";
String to = "DESTINATION-PATH";
InputStream in = new FileInputStream(from);
OutputStream out = new FileOutputStream(to);
byte[] buffer = new byte[BUFFER_SIZE];
int read;
while ((read = in.read(buffer)) != -1) {
  out.write(buffer, 0, read);
}
in.close();
out.close();
```

Streams - can it be done better?

- previous solution is already secure
- efficiently moves data, but ...
  - buffer is created for each call of the code
  - heavy burden on the GC
  - with multiple threads may run out of memory
- solution
  - central buffer - static variable
  - synchronized access to the buffer
    - for the entire operation - efficient, but threads can be starved
    - for each iteration - no starvation, but less efficient

Streams - more, more...

```
final static int SIZE = 100 * 1024;
private static byte[] buffer = new byte[SIZE];
// ...
String from = "SOURCE-PATH";
String to = "DESTINATION-PATH";
InputStream in = new FileInputStream(from);
OutputStream out = new FileOutputStream(to);
int read;
synchronized(buffer) {
  while ((read = in.read(buffer)) != -1) {
    out.write(buffer, 0, read);
  }
}
in.close();
out.close();
```

Streams - comparision

- Copying JPEG file - 370 KB:

Strategy	Time [ms]
Clean streams	10 800
buffered streams	130
own buffer	33
central buffer	22

ATTENTION!
No information about hardware and JVM - data must be taken qualitatively

Free access

files containing records of known size need not be read through streams

- place of record in the file can be calculated based on the order and size
- you need a free file access
- *RandomAccessFile*
  - *seek()* - to indicate the position of the read / write
  - can move forward and backward
  - records can have different size
  - there must be a way to specify the beginning and size of the record
  - defined access method by constructor - read *r*, read and write *rw*
  - *length()* - length of the file
  - *getFilePointer()* - the current file position
  - reading / writing methods

Free access - example

```
String path = "SOME-PATH";
RandomAccessFile rf = new RandomAccessFile(path, "rw");

//write 10 numbers
for (int i = 0; i < 10; i++) {
  rf.writeDouble(i * Math.PI);
}

//read 5-th number
rf.seek(4 * Double.SIZE / 8);
double result = rf.readDouble();
```

New input-output

- streams, even after optimization are slow
- Java 1.4 introduced a new I/O library
    - *java.nio.**
    - significant increase in I/O speed
    - buffers and channels - structure closer to the one used by OS
    - *channel* - source/destination of the data
    - *buffer* - data transporter
    - no direct operations on the channel
    - all I/O to do on channel through buffer
- old library used "underneath" the new
    - already faster than previous implementations
    - additional layer slows
    - you can "go one level down" - *getChannel()*

FileChannel

- channel supporting files
- modes
  - reading - taken from *FileInputStream* or *RandomAccessFile* (r)
  - writing - taken from *FileOutputStream*
  - reading and writing - taken from *RandomAccessFile* (rw)
- features
  - *read*(ByteBuffer) - read record
  - *write*(ByteBuffer) - write record
  - *position*(long) - moving through the file
  - *position*() - get the current position
  - *transferTo/From*(..) - rewriting data between channels
- overloaded functions for read and write

ByteBuffer

- creation
  - *wrap(byte[])* - wraps existing array
  - *allocate(int)* - new buffer allocation
  - *allocateDirect(int)* - 'direct' allocation
    - more associated with the OS - may be beyond the heap!
    - theoretically the fastest I/O
    - virtually dependent on OS
    - longer time of creation and destruction - test before you use!
  - *flip()* - preparing to write to the channel
  - *clear()* - preparing to read from the channel
  - *put()* - insert data
  - *get()* - retrieve data

New input-output - example

```
final static int BUFFER_SIZE = 1024 * 1024;
// ...
String from = "SOURCE-PATH";
String to = "DESTINATION-PATH";
FileChannel in = new FileInputStream(from).getChannel();
FileChannel out = new FileInputStream(to).getChannel();
ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
while (in.read(buffer) != -1) {
  buffer.flip();
  out.write(buffer);
  buffer.clear();
}
in.close();
out.close();
```

New I/O - better example

```
String from = "SOURCE-PATH";
String to = "DESTINATION-PATH";
FileChannel in = new FileInputStream(from).getChannel();
FileChannel out = new FileInputStream(to).getChannel();
in.transferTo(0, in.size(), out);
// or
// out.transferFrom(in, 0, in.size());
in.close();
out.close();
```

Memory mapped files

- most efficient form of work with large files
- file treated as a very large array
- "pretends" that the file is entirely in memory
- *map(mode, position, size)*
  - FileChannel method creates the mapping
  - creates an object of class *MappedByteBuffer*
  - position and size allow you to map a specific part of the file
  - **you can map maximum 2GB!!!**
  - *FileChannel.MapMode* - available mapping modes
    - *PRIVATE* - private use (copy-on-write)
    - *READ_ONLY*
    - *READ_WRITE*

Memory mapped files - example

```
String file = "SOME-PATH";
int length = 0x8FFFFFF; // 128MB

MappedByteBuffer buf = new RandomAccessFile(file, "rw").
getChannel().map(FileChannel.MapMode.READ_WRITE, 0, length);

// write
for (int i = 0; i < length; i++) {
  buf.put((byte) 'x');
}

// read 6 chars from middle of file
for (int i = length / 2; i < length / 2 + 6; i++) {
  System.out.print((char) buf.get(i));
}
```

Comparison

- write 4 000 000 numbers (*int*)
- read all
- free access reading and writing of 200 000 numbers

Operation	Time [ms]	
	Old I/O with buffers	MappedByteBuffer
write	560	120
read	800	70
free accees R/W	5 320	20

ATTENTION:

Source: "Thinking in Java", Bruce Eckel

Serialization

- *ObjectInputStream* - write the object as a stream of bits (serialization)
- *ObjectOutputStream* - reading the bit stream and convert to object (deserialization)
- high-level streams - the need for the source/destination
  - can stream to / from a file
- or the web
  - RMI
  - EJB (RMI / IIOP)
- performance problems in network communication

Serialization - where do the problems come from?

- The class must implement *Serializable*
  - otherwise attempt to serialize ends with exception
  - interface has no methods - is only a marker
- Serialization defines *ObjectOutputStream*
  - ready and generic object write format
  - must be able to save the object of any class
  - hence the overhead and redundant information

Serialization - example

```
ObjectOutputStream out = new ObjectOutputStream(System.out);

Person person = new Person();
person.setFirstName("Jan");
person.setLastName("Kowalski");
person.setHeight(182);
person.setBirthday(new Date(70, 11, 10));

out.writeObject(person);
```

Serialization - result

- amount of data
  - 3 + 8 characters - name
  - one short number - height
  - one long number - date
- "clean" data: 11 + 2 + 8 = 21 bytes
- the result of serialization: 183 bytes!
- here, nearly nine times more! - trivial case...


```
¬íNULENQsrNUL
domain.Person{ďš”NAKŻ©uSTXNULEOTINULACKheightLNULBSbirthdaytNULDLELjava/util/Date;LNUL
firstNametNULDC2Ljava/lang/String;LNULBSlastNameqNUL~NULSTXxpNULNULNUL¶srNULSOjava.util.DatehjSOHKYtEM
ETXNULNULxpwBSNULNULNULACKć.U€xtNULETXJantNULBSKowalski
```

Optimization of the protocol

- transient - specify which attributes are not persistent
  - small profit - most must be persistent, often all
  - still overgrown serialization format...
- You can change the default serialization format
- create your own reading and writing logic
- can not change the serialization/deserialization class
- changes only in the serialized class
  - do not implement Serializable
  - instead you implement Externalizable
    - readExternal (ObjectInput in)
    - writeExternal (ObjectOutput out)
- You can achieve much better performance at the level of
  - amount of data being written to disk
  - network communications and other I/O

Externalization - example

```
public void writeExternal(ObjectOutput out) throws
IOException {
  out.writeUTF(firstName);
  out.writeUTF(lastName);
  out.writeShort(height);
  out.writeLong(birthday.getTime());
}

public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
  firstName = in.readUTF();
  lastName = in.readUTF();
  height = in.readShort();
  birthday = new Date(in.readLong());
}
```

Externalization - result

- "clean" data: 11 + 2 + 8 = 21 bytes
- result of externalization: 60 bytes
  - of which 35 is the header that identifies the class and the end
  - data takes only 25 bytes
    - 21 (pure) + 2 * 2 bytes string length (*writeUTF*)
- comparison with serialization (excluding headers)
  - skip header: 183 - 35 = 148 bytes
  - results comparison: 148/25 = 5.92
  - the trivial example of serialization six times worse

¬íNULENQsrNULDC1domain.ext.Person‹¦ESC=
®„DC4FFNULNULxpwEMNULETXJanNULBSKowalskiNUL¶NULNULNULACKć.U€x

Conclusion

- What are differences between streams and channels?
- How to map files into memory?
- How can serialization be optimized?