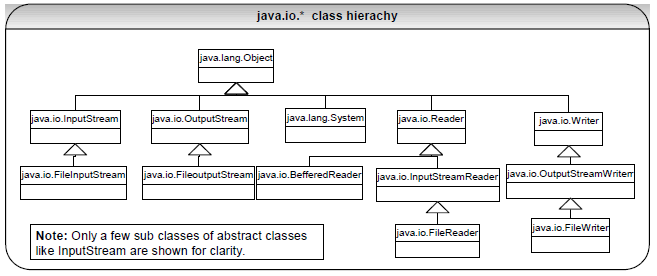
Explain the Java I/O streaming concept and the use of the decorator design pattern in Java I/O?



Java input and output is defined in terms of an abstract concept called a “**stream**”, which is a sequence of data.

There are 2 kinds of streams.

􀂃 Byte streams (8 bit bytes) 􀃆 Abstract classes are: **InputStream** and **OutputStream**

􀂃 Character streams (16 bit UNICODE) 􀃆 Abstract classes are: **Reader** and **Writer**

**Design pattern:** *java.io.\** classes use the **decorator design pattern**. The decorator design pattern **attaches**

**responsibilities to objects at runtime**. Decorators are more flexible than inheritance because the **inheritance**

**attaches responsibility to classes at compile time**. The *java.io.\** classes use the decorator pattern to construct

different combinations of behaviour at runtime based on some basic classes.

**Attaching responsibilities to classes at**

**compile time using subclassing.**

**Attaching responsibilities to objects at runtime using a decorator**

**design pattern.**

Inheritance (aka subclassing) attaches

responsibilities to classes at compile time.

When you extend a class, each individual

changes you make to child class will affect all

instances of the child classes. Defining many

classes using inheritance to have all possible

combinations is problematic and inflexible.

By attaching responsibilities to **objects at runtime**, you can apply changes

to each individual object you want to change.

File **file** = new File(“c:/temp”);

FileInputStream **fis** = new FileInputStream(**file**);

BufferedInputStream bis = new BufferedInputStream(**fis**);

Decorators decorate an object by enhancing or restricting functionality of an

object it decorates. The decorators add or restrict functionality to decorated

Java

27

objects either before or after forwarding the request. At runtime the

BufferedInputStream (bis), which is a **decorator** (aka a **wrapper** around

decorated object), forwards the method call to its **decorated** object

FileInputStream (fis). The ‘bis’ will apply the additional functionality of

buffering around the lower level file (i.e. fis) I/O.

**The New I/O (NIO): more scalable and better performance**

Java has long been not suited for developing programs that perform a lot of I/O operations. Furthermore,

commonly needed tasks such as file locking, non-blocking and asynchronous I/O operations and ability to map file

to memory were not available. Non-blocking I/O operations were achieved through work around such as

multithreading or using JNI. The **New I/O** API (aka **NIO**) in J2SE 1.4 has changed this situation.

A server’s ability to handle several client requests effectively depends on how it uses I/O streams. When a server

has to handle hundreds of clients simultaneously, it must be able to use I/O services concurrently. One way to

cater for this scenario in Java is to use threads but having almost one-to-one ratio of threads (100 clients will have

100 threads) is prone to enormous **thread overhead and can result in performance and scalability problems**

**due to consumption of memory stacks and CPU context switching**. To overcome this problem, a new set of

non-blocking I/O classes have been introduced to the Java platform in java.nio package. The non-blocking I/O

mechanism is built around *Selectors* and *Channels*. ***Channels***, ***Buffers*** and ***Selectors*** are the core of the NIO.

A **Channel** class represents a bi-directional communication channel (similar to *InputStrean* and *OutputStream*)

between datasources such as a socket, a file, or an application component, which is capable of performing one or

more I/O operations such as reading or writing. Channels can be non-blocking, which means, no I/O operation will

wait for data to be read or written to the network. The good thing about NIO channels is that they can be

asynchronously interrupted and closed. So if a thread is blocked in an I/O operation on a channel, another thread

can interrupt that blocked thread.

**Buffers** hold data. Channels can fill and drain *Buffers*. Buffers replace the need for you to do your own buffer

management using byte arrays. There are different types of Buffers like ByteBuffer, CharBuffer, DoubleBuffer, etc.

A **Selector** class is responsible for multiplexing (combining multiple streams into a single stream) by allowing a

single thread to service multiple channels. Each *Channel* registers events with a *Selector*. When events arrive

from clients, the Selector demultiplexes (separating a single stream into multiple streams) them and dispatches

the events to corresponding Channels. To achieve non-blocking I/O a *Channe*l class must work in conjunction with

a *Selector* class.

**Design pattern:** NIO uses a **reactor design pattern**, which demultiplexes events (separating single stream into

multiple streams) and dispatches them to registered object handlers. The reactor pattern is similar to an **observer**

**pattern** (aka publisher and subscriber design pattern), but an observer pattern handles only a single source of

events (i.e. a single publisher with multiple subscribers) where a reactor pattern handles multiple event sources

(i.e. multiple publishers with multiple subscribers). The intent of an observer pattern is to define a one-to-many

dependency so that when one object (i.e. the publisher) changes its state, all its dependents (i.e. all its

subscribers) are notified and updated correspondingly.

Another sought after functionality of NIO is its ability to map a file to memory. There is a specialized form of a

Buffer known as MappedByteBuffer, which represents a buffer of bytes mapped to a file. To map a file to MappedByteBuffer, you must first get a channel for a file. Once you get a channel then you map it to a buffer and

subsequently you can access it like any other ByteBuffer. Once you map an input file to a CharBuffer, you can do

pattern matching on the file contents. This is similar to running “grep” on a UNIX file system.

Another feature of NIO is its ability to lock and unlock files. Locks can be exclusive or shared and can be held on a

contiguous portion of a file. But file locks are subject to the control of the underlying operating system.

How can you improve Java I/O performance?

Java applications that utilise Input/Output are excellent candidates for performance tuning. Profiling of Java

applications that handle significant volumes of data will show significant time spent in I/O operations. This means

substantial gains can be had from I/O performance tuning. Therefore, I/O efficiency should be a high priority for

developers looking to optimally increase performance.

The basic rules for speeding up I/O performance are

􀂃 Minimise accessing the hard disk.

􀂃 Minimise accessing the underlying operating system.

􀂃 Minimise processing bytes and characters individually.

Let us look at some of the techniques to improve I/O performance. **CO**

􀂃 Use **buffering** to minimise disk access and underlying operating system. As shown below, with buffering

large chunks of a file are read from a disk and then accessed a byte or character at a time.

**Without buffering :** inefficient code

try{

File f = new File("myFile.txt");

FileInputStream fis = new FileInputStream(f);

int count = 0;

int b = ;

while((b = **fis.read()**) != -1){

if(b== '\n') {

count++;

}

}

// fis should be closed in a finally block.

fis.close() ;

}

catch(IOException io){}

**Note:** fis.read() is a native method call to the

underlying system.

**With Buffering:** yields better performance

try{

File f = new File("myFile.txt");

FileInputStream fis = new FileInputStream(f);

BufferedInputStream bis = new BufferedInputStream(fis);

int count = 0;

int b = ;

while((b = bis.read()) != -1){

if(b== '\n') {

count++;

}

}

//bis should be closed in a finally block.

bis.close() ;

}

catch(IOException io){}

**Note:** bis.read() takes the next byte from the input buffer and only

rarely access the underlying operating system.

Instead of reading a character or a byte at a time, the above code with buffering can be improved further by

reading one line at a time as shown below:

FileReader fr = new FileReader(f);

BufferedReader br = new BufferedReader(fr);

While (br.readLine() != null) count++;

By default the **System.out** is line buffered, which means that the output buffer is flushed when a new line

character is encountered. This is required for any interactivity between an input prompt and display of output.

The line buffering can be disabled for faster I/O operation as follows:

FileOutputStream fos = new FileOutputStream(file);

BufferedOutputStream bos = new BufferedOutputStream(fos, 1024);

PrintStream ps = new PrintStream(bos,false);

System.setOut(ps);

while (someConditionIsTrue)

System.out.println(“blah…blah…”);

}

It is recommended to use logging frameworks like **Log4J** or **apache commons logging**, which uses

buffering instead of using default behaviour of **System.out.println(…..)** for better performance. Frameworks

like Log4J are configurable, flexible, extensible and easy to use. Use the NIO package, if you are using JDK 1.4 or later, which uses performance-enhancing features like

buffers to hold data, memory mapping of files, non-blocking I/O operations etc.

􀂃 I/O performance can be improved by minimising the calls to the underlying operating systems. The Java

runtime itself cannot know the length of a file, querying the file system for isDirectory(), isFile(), exists() etc

must query the underlying operating system.

􀂃 Where applicable caching can be used to improve performance by reading in all the lines of a file into a Java

collection class like an ArrayList or a HashMap and subsequently access the data from an in-memory

collection instead of the disk.