**Problem with Traditional Java IO API**

A **stream-oriented I/O** system deals with data one or more bytes at a time. An input stream produces one byte of data, and an output stream consumes one byte of data. Important thing is that bytes are not cached anywhere. Furthermore, we cannot move forth and back in the data in a stream.

If you need to move forth and back in the data read from a stream, we must cache it in a buffer first. Java NIO's buffer oriented approach is slightly different. Data is read into a buffer from which it is later processed. You can move forth and back in the buffer as you need to. This gives you a bit more flexibility during processing. However, you also need to check if the buffer contains all the data you need in order to fully process it. And, you need to make sure that when reading more data into the buffer, you do not overwrite data in the buffer you have not yet processed.

**Java NIO**

Java NIO (New IO) is an alternative IO API for Java, meaning alternative to the standard [**Java IO**](http://tutorials.jenkov.com/java-io/index.html) and [**Java Networking**](http://tutorials.jenkov.com/java-networking/index.html) API's.

The **java.nio package was introduced in Java 1.4** and updated **in Java 1.7 (NIO.2)**with [enhanced file operations](https://www.baeldung.com/java-nio-2-file-api) and an [*ASynchronousSocketChannel*](https://www.baeldung.com/java-nio2-async-socket-channel).

It supports a buffer-oriented, channel based approach for I/O operations. It provides: Buffer – to read chunks of data at a time

**Java NIO fundamental components are given below:**

* **Charset Decoder** – for mapping raw bytes to/from readable characters
* **Channel** – for communicating with the outside world
* **Buffer** – are containers for data or charsets and their associated decoders and encoders translate between bytes and Unicode characters.
* **Channels and Buffers:** In the standard IO API you work with byte streams and character streams.
* [***Selector***](https://www.baeldung.com/java-nio-selector) – to enable multiplexing on a Selectable Channel and provide access to any Channels that are ready for I/O. Single thread can monitor the multiple channels for data.
* **Non-blocking mode** – to read whatever is ready meaning Java NIO enables you to do non-blocking IO. For instance, a thread can ask a channel to read data into a buffer. While the channel reads data into the buffer, the thread can do something else. Once data is read into the buffer, the thread can then continue processing it. The same is true for writing data to channels.

**What is Channels in java?**

Channels are used for data transfer between a buffer and an entity. Java NIO Channels are similar to streams with a few differences:

* You can both **read and write** to Channels. Streams (IO API) are typically one-way (read or write).
* Channels can be read and written asynchronously.
* Channels always read to, or write from, a Buffer or other channel.

As mentioned above, you can read data from a channel into a buffer, and write data from a buffer into a channel. Java NIO Buffers are used when interacting with NIO Channels. Here is an illustration of that:

|  |
| --- |
| Java NIO: Channels and Buffers |
| **Java NIO: Channels read data into Buffers, and Buffers write data into Channels** |

Below are the most important Channel implementations in Java NIO:

1. FileChannel
2. DatagramChannel
3. SocketChannel
4. ServerSocketChannel

The **FileChannel** reads data from and to files.

The **DatagramChannel** can read and write data over the network via UDP.

The **SocketChannel** can read and write data over the network via TCP.

The **ServerSocketChannel** allows you to listen for incoming TCP connections, like a web server does. For each incoming connection a **SocketChannel** is created.

## Java NIO FileChannel

## The Java NIO FileChannel is a channel that is connected to a file. Using a file channel you can read data from a file, and write data to a file. The Java NIO FileChannel class is an alternative to  [standard Java IO API](http://tutorials.jenkov.com/java-io/file.html). FileChannel perform is faster than standard I/O when we read a large file.

**Opening a FileChannel**

In order to read a file using *FileChannel*, we must open it. We cannot open a FileChannel directly. You need to obtain a FileChannel via an InputStream, OutputStream, File class, Pathclass or RandomAccessFile. Here is how you open a FileChannel via a **FileInputStream**:

getchanne() method: In order to read data from file we uses file channel. Object of file channel can be created only by calling the getChannel() method on file object as we can't create file object directly

**FileInputStream aFile = new FileInputStream ("data/nio-data.txt", "rw");**

**FileChannel inChannel = aFile.getChannel();**

**Reading Data from a FileChannel**

To read data from a FileChannel We can use the read() methods. Here is an example:

**ByteBuffer buf = ByteBuffer.allocate(48);**

**int bytesRead = inChannel.read(buf);**

First a Buffer is allocated. The data read from the FileChannel is read into the Buffer.

Second the FileChannel.read() method is called. This method reads data from the FileChannel into the Buffer. The int returned by the read() method tells how many bytes were written into the Buffer. If -1 is returned, the end-of-file is reached.

**Writing Data to a FileChannel**

Writing data to a FileChannel is done using the FileChannel.write() method, which takes a Buffer as parameter. Here is an example:

String newData = "New String to write to file..." + System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);

buf.clear();

buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {

**channel.write(buf);**

}

Notice how the FileChannel.write() method is called inside a while-loop. There is no guarantee of how many bytes the write() method writes to the FileChannel. Therefore we repeat the write() call until the Buffer has no further bytes to write.

**Closing a FileChannel**

When you are done using a FileChannel you must close it. Here is how that is done:

channel.close();

**FileChannel Size**

The size() method of the FileChannel object returns the file size of the file the channel is connected to. Here is a simple example:

long fileSize = channel.size();

**FileChannel Truncate**

You can truncate a file by calling the FileChannel.truncate() method. When you truncate a file, you cut it off at a given length. Here is an example:

channel.truncate(1024);

This example truncates the file at 1024 bytes in length.

## Basic Buffer Usage

## Buffers provide a mechanism to store a fixed amount of primitive data elements in an in-memory container. In the NIO, all data is handled with buffers. When data is read, it is read directly into a buffer. When data is written, it is written into a buffer.

A buffer is essentially a block of memory into which you can write data. Using a Buffer to read and write data typically follows this little 4-step process:

1. Write data into the Buffer
2. Call ***buffer.flip()***
3. Read data out of the Buffer
4. Call ***buffer.clear()* or *buffer.compact()***

When you write data into a buffer, the buffer keeps track of how much data you have written. Once you need to read the data, you need to switch the buffer from writing mode into reading mode using the ***flip()*** method call. In reading mode the buffer lets you read all the data written into the buffer.

Once you have read all the data, you need to clear the buffer, to make it ready for writing again. You can do this in two ways: By calling ***clear()*** or by calling ***compact().*** The ***clear()*** method clears the whole buffer. The ***compact()*** method only clears the data which you have already read. Any unread data is moved to the beginning of the buffer, and data will now be written into the buffer after the unread data.

**flip(): T**he flip() method switches a Buffer from writing mode to reading mode. Calling flip() sets the position back to 0, and sets the limit to where position just was.

### Capacity

Being a memory block, a Buffer has a certain fixed size, also called its "capacity". You can only write capacity bytes, longs, chars etc. into the Buffer. Once the Buffer is full, you need to empty it (read the data, or clear it) before you can write more data into it.

### Position

When you write data into the Buffer, you do so at a certain position. Initially the position is 0. When a byte, long etc. has been written into the Buffer the position is advanced to point to the next cell in the buffer to insert data into. Position can maximally become capacity - 1.

When you read data from a Buffer you also do so from a given position. When you flip a Buffer from writing mode to reading mode, the position is reset back to 0. As you read data from the Buffer you do so from position, and position is advanced to next position to read.

### Limit

In write mode the limit of a Buffer is the limit of how much data you can write into the buffer. In write mode the limit is equal to the capacity of the Buffer.

When flipping the Buffer into read mode, limit means the limit of how much data you can read from the data. Therefore, when flipping a Buffer into read mode, limit is set to write position of the write mode. In other words, you can read as many bytes as were written (limit is set to the number of bytes written, which is marked by position).



**Buffer Types**

Java NIO comes with the following **Buffer** types:

* ByteBuffer
* MappedByteBuffer
* CharBuffer
* DoubleBuffer
* FloatBuffer
* IntBuffer
* LongBuffer
* ShortBuffer

**Selector**

* The Java NIO Selector is a component which can examine one or more [**Java NIO Channel**](http://tutorials.jenkov.com/java-nio/channels.html) instances, and determine which channels are ready for e.g. reading or writing. This way a single thread can manage multiple channels.

## Why Use a Selector?

* The advantage of using just a single thread to handle multiple channels is that you need fewer threads to handle the channels. Actually, you can use just one thread to handle all of your channels. Switching between threads is expensive for an operating system, and each thread takes up some resources (memory) in the operating system too.

## Example: Basic Channel and Buffer

Here is a basic example that uses a FileChannel to read some data into a Buffer:

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {  // **TODO** Auto-generated method stub  FileInputStream fromFile = **new** FileInputStream("toFile.txt");  FileChannel inChannel = fromFile.getChannel();  //ByteBuffer buf = ByteBuffer.allocate(48); // initialize buffer, has a capacity of 48 bytes  **long** fileSize = inChannel.size();  ByteBuffer buf = ByteBuffer.*allocate*((**int**) fileSize); // initialize buffer, has a capacity of 48 bytes  **int** bytesRead = inChannel.read(buf);//read into buffer.  **while** (bytesRead != -1) {  System.***out***.println("Read " + bytesRead);  buf.flip(); //make buffer ready for read  //the buf.flip() call. First you read into a Buffer. Then you flip it.  **while**(buf.hasRemaining()){  System.***out***.print((**char**) buf.get()); // read 1 byte at a time  }  buf.clear(); //make buffer ready for writing  bytesRead = inChannel.read(buf);  }  fromFile.close();  } |

# Example: Java NIO Channel to Channel Transfers

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {  // **TODO** Auto-generated method stub  //Input files  String[] inputFiles = **new** String[]{"inputFile1.txt","inputFile2.txt"};    //Files contents will be written in these files  String outputFile = "outputFile.txt";      //Get channel for output file    FileOutputStream fos = **new** FileOutputStream("outputFile.txt");  WritableByteChannel targetChannel = fos.getChannel();    **for** (**int** i = 0; i < inputFiles.length; i++)  {  //Get channel for input files  FileInputStream fis = **new** FileInputStream(inputFiles[i]);  FileChannel inputChannel = fis.getChannel();    //Transfer data from input channel to output channel  inputChannel.transferTo(0, inputChannel.size(), targetChannel);    //close the input channel  inputChannel.close();  fis.close();  }    //finally close the target channel  targetChannel.close();  fos.close();  } |

**Example: Multiple source channel to single output channel**

|  |
| --- |
| **public** **class** niowriteData {    **public** **static** **void** main(String[] args) **throws** IOException  {  String[] inputFiles = {"C:\\Users\\PSAdmin\\Downloads\\TestingFolder\\file1.txt",  "C:\\Users\\PSAdmin\\Downloads\\TestingFolder\\file2.txt"};    String outputFile = "nioOutput.txt";    // GEt Channel  FileOutputStream fos = **new** FileOutputStream(outputFile);  FileChannel targetChannel = fos.getChannel();      **for**(**int** i =0; i < inputFiles.length; i++)  {  // GEt channel for inputFiles  FileInputStream fis = **new** FileInputStream(inputFiles[i]);  FileChannel inputchannel = fis.getChannel();  **long** size = inputchannel.size();  ByteBuffer buf = ByteBuffer.*allocate*((**int**)size);  **int** inputBuffer = inputchannel.read(buf);  buf.flip();    **while**(buf.hasRemaining())  {  System.***out***.println(i);  targetChannel.write(buf);  //inputchannel.transferTo(0, inputchannel.size(), targetChannel);    }  buf.clear();  inputchannel.close();  //fis.close();  }  fos.close();    }  } |

**Main Differences between Java NIO and IO**

The table below summarizes the main differences between Java NIO and IO

|  |  |
| --- | --- |
| **IO** | **NIO** |
| Stream oriented | Buffer oriented |
| Blocking IO | Non blocking IO |
|  | Selectors |

**Conclusion**

NIO allows you to manage multiple channels using only a single (or fewer) threads, but the cost is that parsing the data might be somewhat more complicated than when reading data from a blocking stream using standard IO.

If you need to manage thousands of open connections simultaneously, which each only send a little data, for instance a chat server, implementing the server in NIO is probably an advantage. Similarly, if you need to keep a lot of open connections to other computers, e.g. in a P2P network, using a single thread to manage all of your outbound connections might be an advantage.

If you have fewer connections with very high bandwidth, sending a lot of data at a time, standard IO server implementation should be your choice.

Read below article for more details

https://dzone.com/articles/java-nio-vs-io

http://tutorials.jenkov.com/java-nio/nio-vs-io.html

<https://www.baeldung.com/java-filechannel>

<http://tutorials.jenkov.com/java-nio/buffers.html>

https://javapapers.com/java/java-nio-file-read-write-with-channels/