NIO PACKAGE (New IO):

A *channel* represents an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct I/O operations, for example reading or writing.

Java NIO (New IO) is an alternative IO API for Java (from Java 1.4), 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. Java NIO offers a different way of working with IO than the standard IO API's.

**Java NIO: Channels and Buffers**

In the standard IO API you work with byte streams and character streams. In NIO you work with channels and buffers. Data is ways read from a channel into a buffer, or written from a buffer to a channel.

**Java NIO: Non-blocking IO**

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.

**Java NIO: Selectors**

Java NIO contains the concept of "selectors". A selector is an object that can monitor multiple channels for events (like: connection opened, data arrived etc.). Thus, a single thread can monitor multiple channels for data.

Java NIO consist of the following core components:

* Channels
* Buffers
* Selectors

Java NIO has more classes and components than these, but the Channel, Buffer and Selector forms the core of the API, in my opinion. The rest of the components, like Pipe and FileLock are merely utility classes to be used in conjunction with the three core components.

**Channels and Buffers**

Typically, all IO in NIO starts with a Channel. A Channel is a bit like a stream. From the Channel data can be read into a Buffer. Data can also be written from a Buffer into a Channel. Here is an illustration of that:



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

There are several Channel and Buffer types. Here is a list of the primary Channel implementations in Java NIO:

* FileChannel
* DatagramChannel
* SocketChannel
* ServerSocketChannel

As you can see, these channels cover UDP + TCP network IO, and file IO.

There are a few interesting interfaces accompanying these classes too.

Here is a list of the core Buffer implementations in Java NIO:

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

These Buffer's cover the basic data types that you can send via IO: byte, short, int, long, float, double and characters.

Java NIO also has a MappedByteBuffer which is used in conjunction with memory mapped files. I'll leave thisBuffer out of this overview though.

**Selectors**

A Selector allows a single thread to handle multiple Channel's. This is handy if your application has many connections (Channels) open, but only has low traffic on each connection. For instance, in a chat server.

Here is an illustration of a thread using a Selector to handle 3 Channel's:

|  |
| --- |
| Java NIO: Selectors |
| **Java NIO: A Thread uses a Selector to handle 3 Channel's** |

To use a Selector you register the Channel's with it. Then you call it's select() method. This method will block until there is an event ready for one of the registered channels. Once the method returns, the thread can then process these events. Examples of events are incoming connection, data received etc.

Example of java.nio.channels.FileChannel

This class is used to read the File data as input, as a channel data.

RandomAccessFile 🡪 is a class which accepts a file name as input for reading and writing. In this class there is a Factory method named as getChannel().

Method Signature 🡪 FileChannel getChannel()

Program ::

import java.io.RandomAccessFile;

import java.io.IOException;

import java.nio.channels.FileChannel;

import java.nio.ByteBuffer;

class NIOChannelDemo

{

public static void main(String[] args) throws IOException

{

RandomAccessFile aFile = new RandomAccessFile("C://5pmcore/Demo33.java", "rw");

FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);

while (bytesRead != -1) {

//System.out.println("Read " + bytesRead);

buf.flip();

while(buf.hasRemaining()){

System.out.print((char) buf.get());

}

buf.clear();

bytesRead = inChannel.read(buf);

}

aFile.close();

System.out.println("Hello World!");

}

}

Java NIO Buffers are used when interacting with NIO Channels. As you know, data is read from channels into buffers, and written from buffers into channels.

A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

**Basic Buffer Usage**

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. Thecompact() 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.

**Buffer Capacity, Position and Limit**

A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

A Buffer has three properties you need to be familiar with, in order to understand how a Buffer works. These are:

* capacity
* position
* limit

The meaning of position and limit depends on whether the Buffer is in read or write mode. Capacity always means the same, no matter the buffer mode.

Here is an illustration of capacity, position and limit in write and read modes. The explanation follows in the sections after the illustration.

**Capacity**

Being a memory block, a Buffer has a certain fixed size, also called its "capacity". You can only writecapacity 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 fromposition, 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

As you can see, these Buffer types represent different data types. In other words, they let you work with the bytes in the buffer as char, short, int, long, float or double instead.

The MappedByteBuffer is a bit special, and will be covered in its own text.

**Allocating a Buffer**

To obtain a Buffer object you must first allocate it. Every Buffer class has an allocate() method that does this. Here is an example showing the allocation of a ByteBuffer, with a capacity of 48 bytes:

ByteBuffer buf = ByteBuffer.allocate(48);

Here is an example allocating a CharBuffer with space for 1024 characters:

CharBuffer buf = CharBuffer.allocate(1024);

**Writing Data to a Buffer**

You can write data into a Buffer in two ways:

1. Write data from a Channel into a Buffer
2. Write data into the Buffer yourself, via the buffer's put() methods.

Here is an example showing how a Channel can write data into a Buffer:

int bytesRead = inChannel.read(buf); //read into buffer.

Here is an example that writes data into a Buffer via the put() method:

buf.put(127);

There are many other versions of the put() method, allowing you to write data into the Buffer in many different ways. For instance, writing at specific positions, or writing an array of bytes into the buffer. See the JavaDoc for the concrete buffer implementation for more details.

**flip()**

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

In other words, position now marks the reading position, and limit marks how many bytes, chars etc. were written into the buffer - the limit of how many bytes, chars etc. that can be read.

**Reading Data from a Buffer**

There are two ways you can read data from a Buffer.

1. Read data from the buffer into a channel.
2. Read data from the buffer yourself, using one of the get() methods.

Here is an example of how you can read data from a buffer into a channel:

//read from buffer into channel.

int bytesWritten = inChannel.write(buf);

Here is an example that reads data from a Buffer using the get() method:

byte aByte = buf.get();

There are many other versions of the get() method, allowing you to read data from the Buffer in many different ways. For instance, reading at specific positions, or reading an array of bytes from the buffer. See the JavaDoc for the concrete buffer implementation for more details.

**rewind()**

The Buffer.rewind() sets the position back to 0, so you can reread all the data in the buffer. The limitremains untouched, thus still marking how many elements (bytes, chars etc.) that can be read from the Buffer.

**clear() and compact()**

Once you are done reading data out of the Buffer you have to make the Buffer ready for writing again. You can do so either by calling clear() or by calling compact().

If you call clear() the position is set back to 0 and the limit to capacity. In other words, the Buffer is cleared. The data in the Buffer is not cleared. Only the markers telling where you can write data into theBuffer are.

If there is any unread data in the Buffer when you call clear() that data will be "forgotten", meaning you no longer have any markers telling what data has been read, and what has not been read.

If there is still unread data in the Buffer, and you want to read it later, but you need to do some writing first, callcompact() instead of clear().

compact() copies all unread data to the beginning of the Buffer. Then it sets position to right after the last unread element. The limit property is still set to capacity, just like clear() does. Now the Buffer is ready for writing, but you will not overwrite the unread data.

**mark() and reset()**

You can mark a given position in a Buffer by calling the Buffer.mark() method. You can then later reset the position back to the marked position by calling the Buffer.reset() method. Here is an example:

buffer.mark();

//call buffer.get() a couple of times, e.g. during parsing.

buffer.reset(); //set position back to mark.