**Java NIO Tutorial [[[1]](#footnote-1)]**

Java NIO: Table of Contents

Here is the table of contents for this Java NIO tutorial:

1. [Java NIO Introduction](#JavaNIOIntroduction)
2. [Java NIO Overview](#JavaNIOOverview)
3. [Java NIO Channel](#JavaNIOChannel)
4. [Java NIO Buffer](#JavaNIOBuffer)
5. [Java NIO Scatter / Gather](#JavaNIOScatterGather)
6. [Java NIO Channel to Channel Transfer](#JavaNIOChanneltoChannelTransfers)
7. [Java NIO Selector](#JavaNIOSelector)
8. [Java NIO FileChannel](#JavaNIOFileChannel)
9. [Java NIO SocketChannel](#JavaNIOSocketChannel)
10. [Java NIO ServerSocketChannel](#JavaNIOServerSocketChannel)
11. [Java NIO DataGramChannel](#JavaNIODatagramChannel)
12. [Java NIO Pipe](#JavaNIOPipe)

**Java NIO Introduction**

Java NIO (New IO) is an alternative IO API for Java (from Java 1.4), meaning alternative to the standard Java IO 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 always read from a channel into a buffer, or written from a buffer to a channel.

Java NIO: Asynchronous IO

Java NIO enables you to do asynchronous 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 Overview**

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. Therefore, I'll focus on these three components in this NIO overview. The other components are explained in their own texts elsewhere in this tutorial.

**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:

|  |
| --- |
| 说明: Java NIO: Channels and Buffers |
| **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, but I'll keep them out of this Java NIO overview for simplicity's sake. They'll be explained where relevant, in other texts of this Java NIO tutorial.

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 this Buffer 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:

|  |
| --- |
| 说明: 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.

**Java NIO Channel**

Java NIO Channels are similar to streams with a few differences:

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

As mentioned above, you read data from a channel into a buffer, and write data from a buffer into a channel. Here is an illustration of that:

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

**Channel Implementations**

Here are the most important Channel implementations in Java NIO:

* FileChannel
* DatagramChannel
* SocketChannel
* 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.

**Basic Channel Example**

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

RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "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();

Notice the buf.flip() call. First you read into a Buffer. Then you flip it. Then you read out of it. I'll get into more detail about that in the next text about Buffer's.

**Java NIO Buffer**

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.

Here is a list of the topics covered for NIO Buffer's:

1. Basic Buffer Usage
2. Buffer Capacity, Position and Limit
3. Buffer Types
4. Allocating a Buffer
5. Writing Data to a Buffer
6. flip()
7. Reading Data from a Buffer
8. clear() and compact()
9. mark() and reset()
10. equals() and compareTo()

**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. 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.

Here is a simple Buffer usage example, with the write, flip, read and clear operations maked in bold:

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

FileChannel inChannel = aFile.getChannel();

//create buffer with capacity of 48 bytes

ByteBuffer buf = ByteBuffer.allocate(48);

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

while (bytesRead != -1) {

**buf.flip();** //make buffer ready for read

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);

}

aFile.close();

**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.

|  |
| --- |
| 说明: Java NIO: Buffer capacity, position and limit in write and read mode. |
| **Buffer capacity, position and limit in write and read mode.** |

**Capacity**

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

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 the position 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 limit remains 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 the Buffer 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, call compact() 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.

**equals() and compareTo()**

It is possible to compare two buffers using equals() and compareTo().

**equals()**

Two buffers are equal if:

1. They are of the same type (byte, char, int etc.)
2. They have the same amount of remaining bytes, chars etc. in the buffer.
3. All remaining bytes, chars etc. are equal.

As you can see, equals only compares part of the Buffer, not every single element inside it. In fact, it just compares the remaining elements in the Buffer.

**compareTo()**

The compareTo() method compares the remaining elements (bytes, chars etc.) of the two buffers, for use in e.g. sorting routines. A buffer is considered "smaller" than another buffer if:

1. The first element which is equal to the corresponding element in the other buffer, is smaller than that in the other buffer.
2. All elements are equal, but the first buffer runs out of elements before the second buffer does (it has fewer elements).

**Java NIO Scatter / Gather**

Java NIO comes with built-in scatter / gather support. Scatter / gather are concepts used in reading from, and writing to channels.

A scattering read from a channel is a read operation that reads data into more than one buffer. Thus, the channel "scatters" the data from the channel into multiple buffers.

A gathering write to a channel is a write operation that writes data from more than one buffer into a single channel. Thus, the channel "gathers" the data from multiple buffers into one channel.

Scatter / gather can be really useful in situations where you need to work with various parts of the transmitted data separately. For instance, if a message consists of a header and a body, you might keep the header and body in separate buffers. Doing so may make it easier for you to work with header and body separately.

**Scattering Reads**

A "scattering read" reads data from a single channel into multiple buffers. Here is an illustration of that principle:

Here is an illustration of the Scatter principle:

|  |
| --- |
| 说明: Java NIO: Scattering Read |
| **Java NIO: Scattering Read** |

Here is a code example that shows how to perform a scattering read:

ByteBuffer header = ByteBuffer.allocate(128);

ByteBuffer body = ByteBuffer.allocate(1024);

ByteBuffer[] bufferArray = { header, body };

channel.read(buffers);

Notice how the buffers are first inserted into an array, then the array passed as parameter to the channel.read() method. The read() method then writes data from the channel in the sequence the buffers occur in the array. Once a buffer is full, the channel moves on to fill the next buffer.

The fact that scattering reads fill up one buffer before moving on to the next, means that it is not suited for dynamically sized message parts. In other words, if you have a header and a body, and the header is fixed size (e.g. 128 bytes), then a scattering read works fine.

**Gathering Writes**

A "gathering write" writes data from multiple buffers into a single channel. Here is an illustration of that principle:

|  |
| --- |
| 说明: Java NIO: Gathering Write |
| **Java NIO: Gathering Write** |

Here is a code example that shows how to perform a gathering write:

ByteBuffer header = ByteBuffer.allocate(128);

ByteBuffer body = ByteBuffer.allocate(1024);

//write data into buffers

ByteBuffer[] bufferArray = { header, body };

channel.write(buffers);

The array of buffers are passed into the write() method, which writes the content of the buffers in the sequence they are encountered in the array. Only the data between position and limit of the buffers is written. Thus, if a buffer has a capacity of 128 bytes, but only contains 58 bytes, only 58 bytes are written from that buffer to the channel. Thus, a gathering write works fine with dynamically sized message parts, in contrast to scattering reads.

**Java NIO Channel to Channel Transfers**

In Java NIO you can transfer data directly from one channel to another, if one of the channels is a FileChannel. The FileChannel class has a transferTo() and a transferFrom() method which does this for you.

**transferFrom()**

The FileChannel.transferFrom() method transfers data from a source channel into the FileChannel. Here is a simple example:

RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");

FileChannel fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");

FileChannel toChannel = toFile.getChannel();

long position = 0;

long count = fromChannel.size();

**toChannel.transferFrom(position, count, fromChannel);**

The two first parameters, position and count, tell where in the destination file to start writing (position), and how many bytes to transfer maximally (count). If the source channel has fewer than count bytes, less is transfered.

Additionally, some SocketChannel implementations may transfer only the data the SocketChannel has ready in its internal buffer here and now - even if the SocketChannel may later have more data available. Thus, it may not transfer the entire data requested (count) from the SocketChannel into FileChannel.

**transferTo()**

The transferTo() method transfer from a FileChannel into some other channel. Here is a simple example:

RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");

FileChannel fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");

FileChannel toChannel = toFile.getChannel();

long position = 0;

long count = fromChannel.size();

**fromChannel.transferTo(position, count, toChannel);**

Notice how similar the example is to the previous. The only real difference is the which FileChannel object the method is called on. The rest is the same.

The issue with SocketChannel is also present with the transferTo() method. The SocketChannel implementation may only transfer bytes from the FileChannel until the send buffer is full, and then stop.

**Java NIO Selector**

A Selector is a Java NIO component which can examine one or more NIO Channel's, and determine which channels are ready for e.g. reading or writing. This way a single thread can manage multiple channels, and thus multiple network connections.

Here is a list of the topics covered in this text:

1. Why Use a Selector?
2. Creating a Selector
3. Registering Channels with a Selector
4. SelectionKey's
5. Selecting Channels via a Selector
6. wakeUp()
7. close()
8. Full Example

**Why Use a Selector?**

The advantage of using just a single thread to handle multiple channels is that you need less 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. Therefore, the less threads you use, the better.

Keep in mind though, that modern operating systems and CPU's become better and better at multitasking, so the overheads of multithreading becomes smaller over time. In fact, if a CPU has multiple cores, you might be wasting CPU power by **not** multitasking. Anyways, that design discussion belongs in a different text. It suffices to say here, that you can handle multiple channels with a single thread, using a Selector.

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

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

**Creating a Selector**

You create a Selector by calling the Selector.open() method, like this:

Selector selector = Selector.open();

**Registering Channels with the Selector**

In order to use a Channel with a Selector you must register the Channel with the Selector. This is done using the SelectableChannel.register() method, like this:

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP\_READ);

The Channel must be in non-blocking mode to be used with a Selector. This means that you cannot use FileChannel's with a Selector since FileChannel's cannot be switched into non-blocking mode. Socket channels will work fine though.

Notice the second parameter of the register() method. This is an "interest set", meaning what events you are interested in listening for in the Channel, via the Selector. There are four different events you can listen for:

1. Connect
2. Accept
3. Read
4. Write

A channel that "fires an event" is also said to be "ready" for that event. So, a channel that has connected successfully to another server is "connect ready". A server socket channel which accepts an incoming connection is "accept" ready. A channel that has data ready to be read is "read" ready. A channel that is ready for you to write data to it, is "write" ready.

These four events are represented by the four SelectionKey constants:

1. SelectionKey.OP\_CONNECT
2. SelectionKey.OP\_ACCEPT
3. SelectionKey.OP\_READ
4. SelectionKey.OP\_WRITE

If you are interested in more than one event, OR the constants together, like this:

int interestSet = SelectionKey.OP\_READ | SelectionKey.OP\_WRITE;

I'll return to the interest set a bit further down in this text.

**SelectionKey's**

As you saw in the previous section, when you register a Channel with a Selector the register() method returns a SelectionKey objects. This SelectionKey object contains a few interesting properties:

* The interest set
* The ready set
* The Channel
* The Selector
* An attached object (optional)

I'll describe these properties below.

**Interest Set**

The interest set is the set of events you are interested in "selecting", as described in the section "Registering Channels with the Selector". You can read and write that interest set via the SelectionKey like this:

int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept = interestSet & SelectionKey.OP\_ACCEPT;

boolean isInterestedInConnect = interestSet & SelectionKey.OP\_CONNECT;

boolean isInterestedInRead = interestSet & SelectionKey.OP\_READ;

boolean isInterestedInWrite = interestSet & SelectionKey.OP\_WRITE;

As you can see, you can AND the interest set with the given SelectionKey constant to find out if a certain event is in the interest set.

**Ready Set**

The ready set is the set of operations the channel is ready for. You will primarily be accessing the ready set after a selection. Selection is explained in a later section. You access the ready set like this:

int readySet = selectionKey.readyOps();

You can test in the same way as with the interest set, what events / operations the channel is ready for. But, you can also use these four methods instead, which all return a boolean:

selectionKey.isAcceptable();

selectionKey.isConnectable();

selectionKey.isReadable();

selectionKey.isWritable();

**Channel + Selector**

Accessing the channel + selector from the SelectionKey is trivial. Here is how it's done:

Channel channel = selectionKey.channel();

Selector selector = selectionKey.selector();

**Attaching Objects**

You can attach an object to a SelectionKey this is a handy way of recognizing a given channel, or attaching further information to the channel. For instance, you may attach the Buffer you are using with the channel, or an object containing more aggregate data. Here is how you attach objects:

selectionKey.attach(theObject);

Object attachedObj = selectionKey.attachment();

You can also attach an object already while registering the Channel with the Selector, in the register() method. Here is how that looks:

SelectionKey key = channel.register(selector, SelectionKey.OP\_READ, theObject);

**Selecting Channels via a Selector**

Once you have register one or more channels with a Selector you can call one of the select() methods. These methods return the channels that are "ready" for the events you are interested in (connect, accept, read or write). In other words, if you are interested in channels that are ready for reading, you will receive the channels that are ready for reading from the select() methods.

Here are the select() methods:

* int select()
* int select(long timeout)
* int selectNow()

**select()** blocks until at least one channel is ready for the events you registered for.

**select(long timeout)** does the same as select() except it blocks for a maximum of timeout milliseconds (the parameter).

**selectNow()** doesn't block at all. It returns immediately with whatever channels are ready.

The int returned by the select() methods tells how many channels are ready. That is, how many channels that became ready since last time you called select(). If you call select() and it returns 1 because one channel has become ready, and you call select() one more time, and one more channel has become ready, it will return 1 again. If you have done nothing with the first channel that was ready, you now have 2 ready channels, but only one channel had become ready between each select() call.

**selectedKeys()**

Once you have called one of the select() methods and its return value has indicated that one or more channels are ready, you can access the ready channels via the "selected key set", by calling the selectors selectedKeys() method. Here is how that looks:

Set<SelectionKey> selectedKeys = selector.selectedKeys();

When you register a channel with a Selector the Channel.register() method returns a SelectionKey object. This key represents that channels registration with that selector. It is these keys you can access via the selectedKeySet() method. From the SelectionKey.

You can iterate this selected key set to access the ready channels. Here is how that looks:

Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

SelectionKey key = keyIterator.next();

if(key.isAcceptable()) {

// a connection was accepted by a ServerSocketChannel.

} else if (key.isConnectable()) {

// a connection was established with a remote server.

} else if (key.isReadable()) {

// a channel is ready for reading

} else if (key.isWritable()) {

// a channel is ready for writing

}

keyIterator.remove();

}

This loop iterates the keys in the selected key set. For each key it tests the key to determine what the channel referenced by the key is ready for.

Notice the keyIterator.remove() call at the end of each iteration. The Selector does not remove the SelectionKey instances from the selected key set itself. You have to do this, when you are done processing the channel. The next time the channel becomes "ready" the Selector will add it to the selected key set again.

The channel returned by the SelectionKey.channel() method should be cast to the channel you need to work with, e.g a ServerSocketChannel or SocketChannel etc.

**wakeUp()**

A thread that has called the select() method which is blocked, can be made to leave the select() method, even if no channels are yet ready. This is done by having a different thread call the Selector.wakeup() method on the Selector which the first thread has called select() on. The thread waiting inside select() will then return immediately.

If a different thread calls wakeup() and no thread is currently blocked inside select(), the next thread that calls select() will "wake up" immediately.

**close()**

When you are finished with the Selector you call its close() method. This closes the Selector and invalidates all SelectionKey instances registered with this Selector. The channels themselves are not closed.

**Full Example**

Here is a full example which opens a Selector, registers a channel with it (the channel instantiation is left out), and keeps monitoring the Selector for "readiness" of the four events (accept, connect, read, write).

Selector selector = Selector.open();

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP\_READ);

while(true) {

int readyChannels = selector.select();

if(readyChannels == 0) continue;

Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

SelectionKey key = keyIterator.next();

if(key.isAcceptable()) {

// a connection was accepted by a ServerSocketChannel.

} else if (key.isConnectable()) {

// a connection was established with a remote server.

} else if (key.isReadable()) {

// a channel is ready for reading

} else if (key.isWritable()) {

// a channel is ready for writing

}

keyIterator.remove();

}

}

**Java NIO FileChannel**

A 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.

A FileChannel cannot be set into non-blocking mode. It always runs in blocking mode.

**Opening a FileChannel**

Before you can use a FileChannel you must open it. You cannot open a FileChannel directly. You need to obtain a FileChannel via an InputStream, OutputStream, or a RandomAccessFile. Here is how you open a FileChannel via a RandomAccessFile:

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

FileChannel inChannel = aFile.getChannel();

**Reading Data from a FileChannel**

To read data from a FileChannel you call one of 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 witten 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 Position**

When reading or writing to a FileChannel you do so at a specific position. You can obtain the current position of the FileChannel object by calling the position() method.

You can also set the position of the FileChannel by calling the position(long pos) method.

Here are two examples:

long pos channel.position();

channel.position(pos +123);

If you set the position after the end of the file, and try to read from the channel, you will get -1 - the end-of-file marker.

If you set the position after the end of the file, and write to the channel, the file will be expanded to fit the position and written data. This may result in a "file hole", where the physical file on the disk has gaps in the written data.

**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.

**FileChannel Force**

The FileChannel.force() method flushes all unwritten data from the channel to the disk. An operating system may cache data in memory for performance reasons, so you are not guaranteed that data written to the channel is actually written to disk, until you call the force() method.

The force() method takes a boolean as parameter, telling whether the file meta data (permission etc.) should be flushed too.

Here is an example which flushes both data and meta data:

channel.force(true);

**Java NIO Socket Channel**

A Java NIO SocketChannel is a channel that is connected to a TCP network socket. There are two ways a SocketChannel can be created:

1. You open a SocketChannel and connect to a server somewhere on the internet.
2. A SocketChannel can be created when an incoming connection arrives at a ServerSocketChannel.

**Opening a SocketChannel**

Here is how you open a SocketChannel:

SocketChannel socketChannel = SocketChannel.open();

socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));

**Closing a SocketChannel**

You close a SocketChannel after use by calling the SocketChannel.close() method. Here is how that is done:

socketChannel.close();

**Reading from a SocketChannel**

To read data from a SocketChannel you call one of the read() methods. Here is an example:

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = socketChannel.read(buf);

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

Second the SocketChannel.read() method is called. This method reads data from the SocketChannel into the Buffer. The int returned by the read() method tells how many bytes were witten into the Buffer. If -1 is returned, the end-of-stream is reached (the connection is closed).

**Writing to a SocketChannel**

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

**Non-blocking Mode**

You can set a SocketChannel into non-blocking mode. When you do so, you can call connect(), read() and write() in asynchronous mode.

**connect()**

If the SocketChannel is in non-blocking mode, and you call connect(), the method may return before a connection is established. To determine whether the connection is established, you can call the finishConnect() method, like this:

socketChannel.configureBlocking(false);

socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));

while(! socketChannel.finishConnect() ){

//wait, or do something else...

}

**write()**

In non-blocking mode the write() method may return without having written anything. Therefore you need to call the write() method in a loop. But, since this is already being done in the previous write examples, no need to do anything differently here.

**read()**

In non-blocking mode the read() method may return without having read any data at all. Therefore you need to pay attention to the returned int, which tells how many bytes were read.

**Non-blocking Mode with Selectors**

The non-blocking mode of SocketChannel's works much better with Selector's. By registering one or more SocketChannel's with a Selector, you can ask the Selector for channels that are ready for reading, writing etc. How to use Selector's with SocketChannel's is explained in more detail in a later text in this tutorial.

**Java NIO ServerSocketChannel**

A Java NIO ServerSocketChannel is a channel that can listen for incoming TCP connections, just like a ServerSocket in standard Java IO. The ServerSocketChannel class is located in the java.nio.channels package.

Here is an example:

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.socket().bind(new InetSocketAddress(9999));

while(true){

SocketChannel socketChannel = serverSocketChannel.accept();

//do something with socketChannel...

}

**Opening a ServerSocketChannel**

You open a ServerSocketChannel by calling the ServerSocketChannel.open() method. Here is how that looks:

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

**Closing a ServerSocketChannel**

Closing a ServerSocketChannel is done by calling the ServerSocketChannel.close() method. Here is how that looks:

serverSocketChannel.close();

**Listening for Incoming Connections**

Listening for incoming connections is done by calling the ServerSocketChannel.accept() method. When the accept() method returns, it returns a SocketChannel with an incoming connection. Thus, the accept() method blocks until an incoming connection arrives.

Since you are typically not interested in listening just for a single connection, you call the accept() inside a while-loop. Here is how that looks:

while(true){

SocketChannel socketChannel = serverSocketChannel.accept();

//do something with socketChannel...

}

Of course you would use some other stop-criteria than true inside the while-loop.

**Non-blocking Mode**

A ServerSocketChannel can be set into non-blocking mode. In non-blocking mode the accept() method returns immediately, and may thus return null, if no incoming connection had arrived. Therefore you will have to check if the returned SocketChannel is null. Here is an example:

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.socket().bind(new InetSocketAddress(9999));

**serverSocketChannel.configureBlocking(false);**

while(true){

SocketChannel socketChannel = serverSocketChannel.accept();

**if(socketChannel != null){**

**//do something with socketChannel...**

**}**

}

**Java NIO DatagramChannel**

A Java NIO DatagramChannel is a channel that can send and receive UDP packets. Since UDP is a connection-less network protocol, you cannot just by default read and write to a DatagramChannel like you do from other channels. Instead you send and receive packets of data.

**Opening a DatagramChannel**

Here is how you open a DatagramChannel:

DatagramChannel channel = DatagramChannel.open();

channel.socket().bind(new InetSocketAddress(9999));

This example opens a DatagramChannel which can receive packets on UDP port 9999.

**Receiving Data**

You receive data from a DatagramChannel by calling its receive() method, like this:

ByteBuffer buf = ByteBuffer.allocate(48);

buf.clear();

**channel.receive(buf);**

The receive() method will copy the content of a received packet of data into the given Buffer. If the received packet contains more data than the Buffer can contain, the remaining data is discarded silently.

**Sending Data**

You can send data via a DatagramChannel by calling its send() method, like this:

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

ByteBuffer buf = ByteBuffer.allocate(48);

buf.clear();

buf.put(newData.getBytes());

buf.flip();

**int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com", 80));**

This example sends the string to the "jenkov.com" server on UDP port 80. Nothing is listening on that port though, so nothing will happen. You will not be notified of whether the send packet was received or not, since UDP does not make any guarantees about delivery of data.

**Connecting to a Specific Address**

It is possible to "connect" a DatagramChannel to a specific address on the network. Since UDP is connection-less, this way of connecting to an address does not create a real connection, like with a TCP channel. Rather, it locks your DatagramChannel so you can only send and receive data packets from one specific address.

Here is an example:

channel.connect(new InetSocketAddress("jenkov.com", 80));

When connected you can also use the read() and write() method, as if you were using a traditional channel. You just don't have any guarantees about delivery of the sent data. Here are a few examples:

int bytesRead = channel.read(buf);

int bytesWritten = channel.write(buf);

**Java NIO Pipe**

A Java NIO Pipe is a one-way data connection between two threads. A Pipe has a source channel and a sink channel. You write data to the sink channel. This data can then be read from the source channel.

Here is an illustration of the Pipe principle:

|  |
| --- |
| 说明: Java NIO: Pipe Internals |
| **Java NIO: Pipe Internals** |

**Creating a Pipe**

You open a Pipe by calling the Pipe.open() method. Here is how that looks:

Pipe pipe = Pipe.open();

**Writing to a Pipe**

To write to a Pipe you need to access the sink channel. Here is how that is done:

Pipe.SinkChannel sinkChannel = pipe.sink();

You write to a SinkChannel by calling it's write() method, like this:

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()) {

**sinkChannel.write(buf);**

}

**Reading from a Pipe**

To read from a Pipe you need to access the source channel. Here is how that is done:

Pipe.SourceChannel sourceChannel = pipe.source();

To read from the source channel you call its read() method like this:

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);

The int returned by the read() method tells how many bytes were read into the buffer.

1. Author：Jakob Jenkov Original link: <http://tutorials.jenkov.com/java-nio/index.html> Visit time: 2011-05-10 [↑](#footnote-ref-1)