# http://tutorials.jenkov.com/java-nio/overview.html

1. **Java NIO Tutorial**

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 we 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: Non-blocking IO**

It enables us 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 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. See the menu at the top corner of this page.

## 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 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 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.

# 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:

|  |
| --- |
| 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.

## 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.

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.

|  |
| --- |
| 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 write capacitybytes, 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 the positionback 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 the Bufferare.

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.

## 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:

|  |
| --- |
| 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(bufferArray);

Notice how the buffers are first inserted into an array, then the array passed as parameter to thechannel.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:

|  |
| --- |
| 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(bufferArray);

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(fromChannel, position, count);**

The 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 SocketChannelimplementation 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.

## 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:

|  |
| --- |
| 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 useFileChannel'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 reaturn 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 timeoutmilliseconds (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 selectorsselectedKeys() 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 SelectionKeyobject. This key represents that channels registration with that selector. It is these keys you can access via theselectedKeySet() 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 theSelectionKey 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 callsselect() 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 Selector 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();

}

}

**[Next:](http://tutorials.jenkov.com/java-nio/file-channel.html)**[Java NIO FileChannel](http://tutorials.jenkov.com/java-nio/file-channel.html)

# Java NIO SocketChannel

A Java NIO SocketChannel is a channel that is connected to a TCP network socket. It is Java NIO's equivalent of[**Java Networking's Sockets**](http://tutorials.jenkov.com/java-networking/sockets.html). 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**](http://tutorials.jenkov.com/java-nio/server-socket-channel.html).

## 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 aBuffer 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 theBuffer 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() andwrite() 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 thefinishConnect() 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 moreSocketChannel'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 SocketChannel

A Java NIO SocketChannel is a channel that is connected to a TCP network socket. It is Java NIO's equivalent of[**Java Networking's Sockets**](http://tutorials.jenkov.com/java-networking/sockets.html). 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**](http://tutorials.jenkov.com/java-nio/server-socket-channel.html).

## 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 aBuffer 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 theBuffer 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() andwrite() 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 thefinishConnect() 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 moreSocketChannel'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**](http://tutorials.jenkov.com/java-networking/server-sockets.html) in standard Java Networking. The ServerSocketChannel class is located in thejava.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

   
By [Jakob Jenkov](http://jakob.jenkov.com/)

 Connect with me:

Rate article:

Share article:

Table of Contents

* [Opening a DatagramChannel](http://tutorials.jenkov.com/java-nio/datagram-channel.html#opening-a-datagramchannel)
* [Receiving Data](http://tutorials.jenkov.com/java-nio/datagram-channel.html#receiving-data)
* [Sending Data](http://tutorials.jenkov.com/java-nio/datagram-channel.html#sending-data)
* [Connecting to a Specific Address](http://tutorials.jenkov.com/java-nio/datagram-channel.html#connecting-to-a-specific-address)

Last updated: 2014-06-23

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);

**[Next:](http://tutorials.jenkov.com/java-nio/pipe.html)**[Java NIO Pipe](http://tutorials.jenkov.com/java-nio/pipe.html)

|  |  |  |
| --- | --- | --- |
| **Connect with me:** |  | **Newsletter** - Get all my free tips! |
|  |  | Top of Form         Bottom of Form |
|  |  |  |

# Java NIO Pipe

   
By [Jakob Jenkov](http://jakob.jenkov.com/)

 Connect with me:

Rate article:

Share article:

Table of Contents

* [Creating a Pipe](http://tutorials.jenkov.com/java-nio/pipe.html#creating-a-pipe)
* [Writing to a Pipe](http://tutorials.jenkov.com/java-nio/pipe.html#writing-to-a-pipe)
* [Reading from a Pipe](http://tutorials.jenkov.com/java-nio/pipe.html#reading-from-a-pipe)

Last updated: 2014-06-23

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:

|  |
| --- |
| 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.

# Java NIO vs. IO

   
By [Jakob Jenkov](http://jakob.jenkov.com/)

 Connect with me:

Rate article:

Share article:

Table of Contents

* [Main Differences Betwen Java NIO and IO](http://tutorials.jenkov.com/java-nio/nio-vs-io.html#main-differences-between-java-nio-and-io)
* [Stream Oriented vs. Buffer Oriented](http://tutorials.jenkov.com/java-nio/nio-vs-io.html#stream-oriented-vs-buffer-oriented)
* [Blocking vs. Non-blocking IO](http://tutorials.jenkov.com/java-nio/nio-vs-io.html#blocking-vs-non-blocking)
* [Selectors](http://tutorials.jenkov.com/java-nio/nio-vs-io.html#selectors)
* [How NIO and IO Influences Application Design](http://tutorials.jenkov.com/java-nio/nio-vs-io.html#how-nio-and-io-influences-application-design)
  + [The API Calls](http://tutorials.jenkov.com/java-nio/nio-vs-io.html#the-api-calls)
  + [The Processing of Data](http://tutorials.jenkov.com/java-nio/nio-vs-io.html#the-processing-of-data)
* [Summary](http://tutorials.jenkov.com/java-nio/nio-vs-io.html#summary)

Last updated: 2014-06-23

When studying both the Java NIO and IO API's, a question quickly pops into mind:

When should I use IO and when should I use NIO?

In this text I will try to shed some light on the differences between Java NIO and IO, their use cases, and how they affect the design of your code.

## Main Differences Betwen Java NIO and IO

The table below summarizes the main differences between Java NIO and IO. I will get into more detail about each difference in the sections following the table.

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

## Stream Oriented vs. Buffer Oriented

The first big difference between Java NIO and IO is that IO is stream oriented, where NIO is buffer oriented. So, what does that mean?

Java IO being stream oriented means that you read one or more bytes at a time, from a stream. What you do with the read bytes is up to you. They are not cached anywhere. Furthermore, you 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, you will need to 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.

## Blocking vs. Non-blocking IO

Java IO's various streams are blocking. That means, that when a thread invokes a read() or write(), that thread is blocked until there is some data to read, or the data is fully written. The thread can do nothing else in the meantime.

Java NIO's non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available, or nothing at all, if no data is currently available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else.

The same is true for non-blocking writing. A thread can request that some data be written to a channel, but not wait for it to be fully written. The thread can then go on and do something else in the mean time.

What threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output.

## Selectors

Java NIO's selectors allow a single thread to monitor multiple channels of input. You can register multiple channels with a selector, then use a single thread to "select" the channels that have input available for processing, or select the channels that are ready for writing. This selector mechanism makes it easy for a single thread to manage multiple channels.

## How NIO and IO Influences Application Design

Whether you choose NIO or IO as your IO toolkit may impact the following aspects of your application design:

1. The API calls to the NIO or IO classes.
2. The processing of data.
3. The number of thread used to process the data.

### The API Calls

Of course the API calls when using NIO look different than when using IO. This is no surprise. Rather than just read the data byte for byte from e.g. an InputStream, the data must first be read into a buffer, and then be processed from there.

### The Processing of Data

The processing of the data is also affected when using a pure NIO design, vs. an IO design.

In an IO design you read the data byte for byte from an InputStream or a Reader. Imagine you were processing a stream of line based textual data. For instance:

Name: Anna

Age: 25

Email: anna@mailserver.com

Phone: 1234567890

This stream of text lines could be processed like this:

InputStream input = ... ; // get the InputStream from the client socket

BufferedReader reader = new BufferedReader(new InputStreamReader(input));

String nameLine = reader.readLine();

String ageLine = reader.readLine();

String emailLine = reader.readLine();

String phoneLine = reader.readLine();

Notice how the processing state is determined by how far the program has executed. In other words, once the firstreader.readLine() method returns, you know for sure that a full line of text has been read. The readLine()blocks until a full line is read, that's why. You also know that this line contains the name. Similarly, when the secondreadLine() call returns, you know that this line contains the age etc.

As you can see, the program progresses only when there is new data to read, and for each step you know what that data is. Once the executing thread have progressed past reading a certain piece of data in the code, the thread is not going backwards in the data (mostly not). This principle is also illustrated in this diagram:

|  |
| --- |
| Java IO: Reading data from a blocking stream. |
| **Java IO: Reading data from a blocking stream.** |

A NIO implementation would look different. Here is a simplified example:

ByteBuffer buffer = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buffer);

Notice the second line which reads bytes from the channel into the ByteBuffer. When that method call returns you don't know if all the data you need is inside the buffer. All you know is that the buffer contains some bytes. This makes processing somewhat harder.

Imagine if, after the first read(buffer) call, that all what was read into the buffer was half a line. For instance, "Name: An". Can you process that data? Not really. You need to wait until at leas a full line of data has been into the buffer, before it makes sense to process any of the data at all.

So how do you know if the buffer contains enough data for it to make sense to be processed? Well, you don't. The only way to find out, is to look at the data in the buffer. The result is, that you may have to inspect the data in the buffer several times before you know if all the data is inthere. This is both inefficient, and can become messy in terms of program design. For instance:

ByteBuffer buffer = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buffer);

while(! bufferFull(bytesRead) ) {

bytesRead = inChannel.read(buffer);

}

The bufferFull() method has to keep track of how much data is read into the buffer, and return either true orfalse, depending on whether the buffer is full. In other words, if the buffer is ready for processing, it is considered full.

The bufferFull() method scans through the buffer, but must leave the buffer in the same state as before thebufferFull() method was called. If not, the next data read into the buffer might not be read in at the correct location. This is not impossible, but it is yet another issue to watch out for.

If the buffer is full, it can be processed. If it is not full, you might be able to partially process whatever data is there, if that makes sense in your particular case. In many cases it doesn't.

The is-data-in-buffer-ready loop is illustrated in this diagram:

|  |
| --- |
| Java NIO: Reading data from a channel until all needed data is in buffer. |
| **Java NIO: Reading data from a channel until all needed data is in buffer.** |

## Summary

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

If you need to manage thousands of open connections simultanously, 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. This one thread, multiple connections design is illustrated in this diagram:

|  |
| --- |
| Java NIO: A single thread managing multiple connections. |
| **Java NIO: A single thread managing multiple connections.** |

If you have fewer connections with very high bandwidth, sending a lot of data at a time, perhaps a classic IO server implementation might be the best fit. This diagram illustrates a classic IO server design:

|  |
| --- |
| Java IO: A classic IO server design - one connection handled by one thread. |
| **Java IO: A classic IO server design - one connection handled by one thread.** |

# Java NIO Path

   
By [Jakob Jenkov](http://jakob.jenkov.com/)

 Connect with me:

Rate article:

Share article:

Table of Contents

* [Creating a Path Instance](http://tutorials.jenkov.com/java-nio/path.html#creating-a-path-instance)
  + [Creating an Absolute Path](http://tutorials.jenkov.com/java-nio/path.html#creating-an-absolute-path)
  + [Creating a Relative Path](http://tutorials.jenkov.com/java-nio/path.html#creating-a-relative-path)
* [Path.normalize()](http://tutorials.jenkov.com/java-nio/path.html#path-normalize)

Last updated: 2015-03-12

The Java Path interface is part of the Java NIO 2 update which Java NIO received in Java 6 and Java 7. The JavaPath interface was added to Java NIO in Java 7. The Path interface is located in the java.nio.file package, so the fully qualified name of the Java Path interface is java.nio.file.Path.

A Java Path instance represents a *path* in the file system. A path can point to either a file or a directory. A path can be absolute or relative. An absolute path contains the full path from the root of the file system down to the file or directory it points to. A relative path contains the path to the file or directory relative to some other path. Relative paths may sound a bit confusing. Don't worry. I will explain relative paths in more detail later in this Java NIO Path tutorial.

Do not confuse a *file system path* with the path environment variable in some operating systems. Thejava.nio.file.Path interface has nothing to do with the path environment variable.

In many ways the java.nio.file.Path interface is similar to the [**java.io.File**](http://tutorials.jenkov.com/java-io/file.html) class, but there are some minor differences. In many cases though, you can replace the use of the File class with use of the Path interface.

## Creating a Path Instance

In order to use a java.nio.file.Path instance you must create a Path instance. You create a Path instance using a static method in the Paths class (java.nio.file.Paths) named Paths.get(). Here is a JavaPaths.get() example:

import java.nio.file.Path;

import java.nio.file.Paths;

public class PathExample {

public static void main(String[] args) {

Path path = Paths.get("c:\\data\\myfile.txt");

}

}

Notice the two import statements at the top of the example. To use the Path interface and the Paths class we must first import them.

Second, notice the Paths.get("c:\\data\\myfile.txt") method call. It is the call to the Paths.get()method that creates the Path instance. The Paths.get() method is a factory method for Path instances, in other words.

### Creating an Absolute Path

Creating an absolute path is done by calling the Paths.get() factory method with the absolute file as parameter. Here is an example of creating a Path instance representing an absolute path:

Path path = Paths.get("c:\\data\\myfile.txt");

The absolute path is c:\data\myfile.txt. The double \ characters are necessary in Java strings, since the \is an escape character, meaning the following character tells what character is really to be located at this place in the string. By writing \\ you tell the Java compiler to write a single \ character into the string.

The above path is a Windows file system path. On a Unix system (Linux, MacOS, FreeBSD etc.) the above absolute path could look like this:

Path path = Paths.get("/home/jakobjenkov/myfile.txt");

The absolute path is now /home/jakobjenkov/myfile.txt .

If you used this kind of path on a Windows machine (a path starting with /) the path would be interpreted as relative to the current drive. For instance, the path

/home/jakobjenkov/myfile.txt

could be interpreted as being located on the C drive. Then the path would correspond to this full path:

C:/home/jakobjenkov/myfile.txt

### Creating a Relative Path

A relative path is a path that points from one path (the base path) to a directory or file. The full path (the absolute path) of a relative path is derived by combining the base path with the relative path.

The Java NIO Path class can also be used to work with relative paths. You create a relative path using thePaths.get(basePath, relativePath) method. Here are two relative path examples in Java:

Path projects = Paths.get("d:\\data", "projects");

Path file = Paths.get("d:\\data", "projects\\a-project\\myfile.txt");

The first example creates a Java Path instance which points to the path (directory) d:\data\projects. The second example creates a Path instance which points to the path (file) d:\data\projects\a-project\myfile.txt .

When working with relative paths there are two special codes you can use inside the path string. These codes are:

* .
* ..

The . code means "current directory". For instance, if you create a relative path like this:

Path currentDir = Paths.get(".");

System.out.println(currentDir.toAbsolutePath());

Then the absolute path the Java Path instance corresponds to will be the directory in which the application executing the above code is executed.

If the . is used in the middle of a path string it just means the same directory as the path was pointing to at that point. Here is an Path example illustrating that:

Path currentDir = Paths.get("d:\\data\\projects\.\a-project");

This path will correspond to the path:

d:\data\projects\a-project

The .. code means "parent directory" or "one directory up". Here is a Path Java example illustrating that:

Path parentDir = Paths.get("..");

The Path instance created by this example would correspond to the parent directory of the directory from which the application running this code was started.

If you use the .. code in the middle of a path string it will correspond to changing one directory up at that point in the path string. For instance:

String path = "d:\\data\\projects\\a-project\\..\\another-project";

Path parentDir2 = Paths.get(path);

The Java Path instance created by this example will correspond to this absolute path:

d:\data\projects\another-project

The .. code after the a-project directory changes directory up the the parent directory projects and then the path references down into the another-project directory from there.

The . and .. codes also work in combination with the two-string Paths.get() method. Here are two JavaPaths.get() examples showing simple examples of that:

Path path1 = Paths.get("d:\\data\\projects", ".\\a-project");

Path path2 = Paths.get("d:\\data\\projects\\a-project",

"..\\another-project");

There are more ways that the Java NIO Path class can be used to work with relative paths. You will learn more about that later in this tutorial.

## Path.normalize()

The normalize() method of the Path interface can normalize a path. Normalizing means that it removes all the. and .. codes in the middle of the path string, and resolves what path the path string refers to. Here is a JavaPath.normalize() example:

String originalPath =

"d:\\data\\projects\\a-project\\..\\another-project";

Path path1 = Paths.get(originalPath);

System.out.println("path1 = " + path1);

Path path2 = path1.normalize();

System.out.println("path2 = " + path2);

This Path example first creates a path string with a .. code in the middle. Then the example creates a Pathinstance from this path string, and prints that Path instance out (actually it prints Path.toString()).

The example then calls normalize() on the created Path instance, which returns a new Path instance. This new, normalized Path instance is then also printed out.

Here is the output printed from the above example:

path1 = d:\data\projects\a-project\..\another-project

path2 = d:\data\projects\another-project

As you can see, the normalized path does not contain the a-project\.. part, as this is redundant. The removed part adds nothing to the final absolute path.

# Java NIO Files

   
By [Jakob Jenkov](http://jakob.jenkov.com/)

 Connect with me:

Rate article:

Share article:

Table of Contents

* [Files.exists()](http://tutorials.jenkov.com/java-nio/files.html#files-exists)
* [Files.createDirectory()](http://tutorials.jenkov.com/java-nio/files.html#files-create-directory)
* [Files.copy()](http://tutorials.jenkov.com/java-nio/files.html#files-copy)
  + [Overwriting Existing Files](http://tutorials.jenkov.com/java-nio/files.html#overwriting-existing-files)
* [Files.move()](http://tutorials.jenkov.com/java-nio/files.html#files-move)
* [Files.delete()](http://tutorials.jenkov.com/java-nio/files.html#files-delete)
* [Files.walkFileTree()](http://tutorials.jenkov.com/java-nio/files.html#files-walkfiletree)
  + [Searching For Files](http://tutorials.jenkov.com/java-nio/files.html#searching-for-files)
  + [Deleting Directories Recursively](http://tutorials.jenkov.com/java-nio/files.html#deleting-directories-recursively)
* [Additional Methods in the Files Class](http://tutorials.jenkov.com/java-nio/files.html)

Last updated: 2015-04-15

The Java NIO Files class (java.nio.file.Files) provides several methods for manipulating files in the file system. This Java NIO Files tutorial will cover the most commonly used of these methods. The Files class contains many methods, so check the JavaDoc too, if you need a method that is not described here. The Filesclass just might have a method for it still.

The java.nio.file.Files class works with [**java.nio.file.Path**](http://tutorials.jenkov.com/java-nio/path.html) instances, so you need to understand the Path class before you can work with the Files class.

## Files.exists()

The Files.exists() method checks if a given Path exists in the file system.

It is possible to create Path instances that do not exist in the file system. For instance, if you plan to create a new directory, you would first create the corresponding Path instance, and then create the directory.

Since Path instances may or may not point to paths that exist in the file system, you can use theFiles.exists() method to determine if they do (in case you need to check that).

Here is a Java Files.exists() example:

Path path = Paths.get("data/logging.properties");

boolean pathExists =

Files.exists(path,

new LinkOption[]{ LinkOption.NOFOLLOW\_LINKS});

This example first creates a Path instance pointing to the path we want to check if exists or not. Second, the example calls the Files.exists() method with the Path instance as the first parameter.

Notice the second parameter of the Files.exists() method. This parameter is an array of options that influence how the Files.exists() determines if the path exists or not. In this example above the array contains the LinkOption.NOFOLLOW\_LINKS which means that the Files.exists() method should not follow symbolic links in the file system to determine if the path exists.

## Files.createDirectory()

The Files.createDirectory() method creates a new directory from a Path instance. Here is a JavaFiles.createDirectory() example:

Path path = Paths.get("data/subdir");

try {

Path newDir = Files.createDirectory(path);

} catch(FileAlreadyExistsException e){

// the directory already exists.

} catch (IOException e) {

//something else went wrong

e.printStackTrace();

}

The first line creates the Path instance that represents the directory to create. Inside the try-catch block theFiles.createDirectory() method is called with the path as parameter. If creating the directory succeeds, aPath instance is returned which points to the newly created path.

If the directory already exists, a java.nio.file.FileAlreadyExistsException will be thrown. If something else goes wrong, an IOException may get thrown. For instance, if the parent directory of the desired, new directory does not exist, an IOException may get thrown. The parent directory is the directory in which you want to create the new directory. Thus, it means the parent directory of the new directory.

## Files.copy()

The Files.copy() method copies a file from one path to another. Here is a Java NIO Files.copy() example:

Path sourcePath = Paths.get("data/logging.properties");

Path destinationPath = Paths.get("data/logging-copy.properties");

try {

Files.copy(sourcePath, destinationPath);

} catch(FileAlreadyExistsException e) {

//destination file already exists

} catch (IOException e) {

//something else went wrong

e.printStackTrace();

}

First the example creates a source and destination Path instance. Then the example calls Files.copy(), passing the two Path instances as parameters. This will result in the file referenced by the source path to be copied to the file referenced by the destination path.

If the destination file already exists, a java.nio.file.FileAlreadyExistsException is thrown. If something else goes wrong, an IOException will be thrown. For instance, if the directory to copy the file to does not exist, an IOException will be thrown.

### Overwriting Existing Files

It is possible to force the Files.copy() to overwrite an existing file. Here an example showing how to overwrite an existing file using Files.copy():

Path sourcePath = Paths.get("data/logging.properties");

Path destinationPath = Paths.get("data/logging-copy.properties");

try {

Files.copy(sourcePath, destinationPath,

StandardCopyOption.REPLACE\_EXISTING);

} catch(FileAlreadyExistsException e) {

//destination file already exists

} catch (IOException e) {

//something else went wrong

e.printStackTrace();

}

Notice the third parameter to the Files.copy() method. This parameter instructs the copy() method to overwrite an existing file if the destination file already exists.

## Files.move()

The Java NIO Files class also contains a function for moving files from one path to another. Moving a file is the same as renaming it, except moving a file can both move it to a different directory and change its name in the same operation. Yes, the java.io.File class could also do that with its renameTo() method, but now you have the file move functionality in the java.nio.file.Files class too.

Here is a Java Files.move() example:

Path sourcePath = Paths.get("data/logging-copy.properties");

Path destinationPath = Paths.get("data/subdir/logging-moved.properties");

try {

Files.move(sourcePath, destinationPath,

StandardCopyOption.REPLACE\_EXISTING);

} catch (IOException e) {

//moving file failed.

e.printStackTrace();

}

First the source path and destination path are created. The source path points to the file to move, and the destination path points to where the file should be moved to. Then the Files.move() method is called. This results in the file being moved.

Notice the third parameter passed to Files.move() . This parameter tells the Files.move() method to overwrite any existing file at the destination path. This parameter is actually optional.

The Files.move() method may throw an IOException if moving the file fails. For instance, if a file already exists at the destination path, and you have left out the StandardCopyOption.REPLACE\_EXISTING option, or if the file to move does not exist etc.

## Files.delete()

The Files.delete() method can delete a file or directory. Here is a Java Files.delete() example:

Path path = Paths.get("data/subdir/logging-moved.properties");

try {

Files.delete(path);

} catch (IOException e) {

//deleting file failed

e.printStackTrace();

}

First the Path pointing to the file to delete is created. Second the Files.delete() method is called. If theFiles.delete() fails to delete the file for some reason (e.g. the file or directory does not exist), anIOException is thrown.

## Files.walkFileTree()

The Files.walkFileTree() method contains functionality for traversing a directory tree recursively. ThewalkFileTree() method takes a Path instance and a FileVisitor as parameters. The Path instance points to the directory you want to traverse. The FileVisitor is called during traversion.

Before I explain how the traversal works, here is first the FileVisitor interface:

public interface FileVisitor {

public FileVisitResult preVisitDirectory(

Path dir, BasicFileAttributes attrs) throws IOException;

public FileVisitResult visitFile(

Path file, BasicFileAttributes attrs) throws IOException;

public FileVisitResult visitFileFailed(

Path file, IOException exc) throws IOException;

public FileVisitResult postVisitDirectory(

Path dir, IOException exc) throws IOException {

}

You have to implement the FileVisitor interface yourself, and pass an instance of your implementation to thewalkFileTree() method. Each method of your FileVisitor implementation will get called at different times during the directory traversal. If you do not need to hook into all of these methods, you can extend theSimpleFileVisitor class, which contains default implementations of all methods in the FileVisitorinterface.

Here is a walkFileTree() example:

Files.walkFileTree(path, new FileVisitor<Path>() {

@Override

public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws IOException {

System.out.println("pre visit dir:" + dir);

return FileVisitResult.CONTINUE;

}

@Override

public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {

System.out.println("visit file: " + file);

return FileVisitResult.CONTINUE;

}

@Override

public FileVisitResult visitFileFailed(Path file, IOException exc) throws IOException {

System.out.println("visit file failed: " + file);

return FileVisitResult.CONTINUE;

}

@Override

public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {

System.out.println("post visit directory: " + dir);

return FileVisitResult.CONTINUE;

}

});

Each of the methods in the FileVisitor implementation gets called at different times during traversal:

The preVisitDirectory() method is called just before visiting any directory. The postVisitDirectory()method is called just after visiting a directory.

The visitFile() mehtod is called for every file visited during the file walk. It is not called for directories - only files. The visitFileFailed() method is called in case visiting a file fails. For instance, if you do not have the right permissions, or something else goes wrong.

Each of the four methods return a FileVisitResult enum instance. The FileVisitResult enum contains the following four options:

* CONTINUE
* TERMINATE
* SKIP\_SIBLINGS
* SKIP\_SUBTREE

By returning one of these values the called method can decide how the file walk should continue.

CONTINUE means that the file walk should continue as normal.

TERMINATE means that the file walk should terminate now.

SKIP\_SIBLINGS means that the file walk should continue but without visiting any siblings of this file or directory.

SKIP\_SUBTREE means that the file walk should continue but without visiting the entries in this directory. This value only has a function if returned from preVisitDirectory(). If returned from any other methods it will be interpreted as a CONTINUE.

### Searching For Files

Here is a walkFileTree() that extends SimpleFileVisitor to look for a file named README.txt :

Path rootPath = Paths.get("data");

String fileToFind = File.separator + "README.txt";

try {

Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {

@Override

public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {

String fileString = file.toAbsolutePath().toString();

//System.out.println("pathString = " + fileString);

if(fileString.endsWith(fileToFind)){

System.out.println("file found at path: " + file.toAbsolutePath());

return FileVisitResult.TERMINATE;

}

return FileVisitResult.CONTINUE;

}

});

} catch(IOException e){

e.printStackTrace();

}

### Deleting Directories Recursively

The Files.walkFileTree() can also be used to delete a directory with all files and subdirectories inside it. The Files.delete() method will only delete a directory if it is empty. By walking through all directories and deleting all files (inside visitFile()) in each directory, and afterwards delete the directory itself (insidepostVisitDirectory()) you can delete a directory with all subdirectories and files. Here is a recursive directory deletion example:

Path rootPath = Paths.get("data/to-delete");

try {

Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {

@Override

public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {

System.out.println("delete file: " + file.toString());

Files.delete(file);

return FileVisitResult.CONTINUE;

}

@Override

public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {

Files.delete(dir);

System.out.println("delete dir: " + dir.toString());

return FileVisitResult.CONTINUE;

}

});

} catch(IOException e){

e.printStackTrace();

}

## Additional Methods in the Files Class

The java.nio.file.Files class contains many other useful functions, like functions for creating symbolic links, determining the file size, setting file permissions etc. Check out the JavaDoc for the java.nio.file.Filesclass for more information about these methods.

# Java NIO AsynchronousFileChannel

   
By [Jakob Jenkov](http://jakob.jenkov.com/)

 Connect with me:

Rate article:

Share article:

Table of Contents

* [Creating an AsynchronousFileChannel](http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html#creating-an-asynchronousfilechannel)
* [Reading Data](http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html#reading-data)
* [Reading Data Via a Future](http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html#reading-data-via-a-future)
* [Reading Data Via a CompletionHandler](http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html#reading-data-via-a-completionhandler)
* [Writing Data](http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html#writing-data)
* [Writing Data Via a Future](http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html#writing-data-via-a-future)
* [Writing Data Via a CompletionHandler](http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html#writing-data-via-a-completion-handler)

Last updated: 2015-05-07

In Java 7 the AsynchronousFileChannel was added to Java NIO. The AsynchronousFileChannelmakes it possible to read data from, and write data to files asynchronously. This tutorial will explain how to use theAsynchronousFileChannel.

## Creating an AsynchronousFileChannel

You create an AsynchronousFileChannel via its static method open(). Here is an example of creating anAsynchronousFileChannel:

Path path = Paths.get("data/test.xml");

AsynchronousFileChannel fileChannel =

AsynchronousFileChannel.open(path, StandardOpenOption.READ);

The first parameter to the open() method is a [**Path**](http://tutorials.jenkov.com/java-nio/path.html) instance pointing to the file theAsynchronousFileChannel is to be associated with.

The second parameter is one or more open options which tell the AsynchronousFileChannel what operations is to be performed on the underlying file. In this example we used the StandardOpenOption.READ which means that the file will be opened for reading.

## Reading Data

You can read data from an AsynchronousFileChannel in two ways. Each way to read data call one of theread() methods of the AsynchronousFileChannel. Both methods of reading data will be covered in the following sections.

## Reading Data Via a Future

The first way to read data from an AsynchronousFileChannel is to call the read() method that returns aFuture. Here is how calling that read() method looks:

Future<Integer> operation = fileChannel.read(buffer, 0);

This version of the read() method takes ByteBuffer as first parameter. The data read from theAsynchronousFileChannel is read into this ByteBuffer. The second parameter is the byte position in the file to start reading from.

The read() method return immediately, even if the read operation has not finished. You can check the when the read operation is finished by calling the isDone() method of the Future instance returned by the read()method.

Here is a longer example showing how to use this version of the read() method:

AsynchronousFileChannel fileChannel =

AsynchronousFileChannel.open(path, StandardOpenOption.READ);

ByteBuffer buffer = ByteBuffer.allocate(1024);

long position = 0;

Future<Integer> operation = fileChannel.read(buffer, position);

while(!operation.isDone());

buffer.flip();

byte[] data = new byte[buffer.limit()];

buffer.get(data);

System.out.println(new String(data));

buffer.clear();

This example creates an AsynchronousFileChannel and then creates a ByteBuffer which is passed to theread() method as parameter, along with a position of 0. After calling read() the example loops until theisDone() method of the returned Future returns true. Of course, this is not a very efficient use of the CPU - but somehow you need to wait until the read operation has completed.

Once the read operation has completed the data read into the ByteBuffer and then into a String and printed toSystem.out .

## Reading Data Via a CompletionHandler

The second method of reading data from an AsynchronousFileChannel is to call the read() method version that takes a CompletionHandler as a parameter. Here is how you call this read() method:

fileChannel.read(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {

@Override

public void completed(Integer result, ByteBuffer attachment) {

System.out.println("result = " + result);

attachment.flip();

byte[] data = new byte[attachment.limit()];

attachment.get(data);

System.out.println(new String(data));

attachment.clear();

}

@Override

public void failed(Throwable exc, ByteBuffer attachment) {

}

});

Once the read operation finishes the CompletionHandler's completed() method will be called. As parameters to the completed() method are passed an Integer telling how many bytes were read, and the "attachment" which was passed to the read() method. The "attachment" is the third parameter to the read()method. In this case it was the ByteBuffer into which the data is also read. You can choose freely what object to attach.

If the read operation fails, the failed() method of the CompletionHandler will get called instead.

## Writing Data

Just like with reading, you can write data to an AsynchronousFileChannel in two ways. Each way to write data call one of the write() methods of the AsynchronousFileChannel. Both methods of writing data will be covered in the following sections.

## Writing Data Via a Future

The AsynchronousFileChannel also enables you to write data asynchronously. Here is a full JavaAsynchronousFileChannel write example:

Path path = Paths.get("data/test-write.txt");

AsynchronousFileChannel fileChannel =

AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);

long position = 0;

buffer.put("test data".getBytes());

buffer.flip();

Future<Integer> operation = fileChannel.write(buffer, position);

buffer.clear();

while(!operation.isDone());

System.out.println("Write done");

First an AsynchronousFileChannel is opened in write mode. Then a ByteBuffer is created and some data written into it. Then the data in the ByteBuffer is written to the file. Finally the example checks the returnedFuture to see when the write operation has completed.

Note, that the file must already exist before this code will work. If the file does not exist the write() method will throw a java.nio.file.NoSuchFileException .

You can make sure that the file the Path points to exists with the following code:

if(!Files.exists(path)){

Files.createFile(path);

}

## Writing Data Via a CompletionHandler

You can also write data to the AsynchronousFileChannel with a CompletionHandler to tell you when the write is complete instead of a Future. Here is an example of writing data to the AsynchronousFileChannelwith a CompletionHandler:

Path path = Paths.get("data/test-write.txt");

if(!Files.exists(path)){

Files.createFile(path);

}

AsynchronousFileChannel fileChannel =

AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);

long position = 0;

buffer.put("test data".getBytes());

buffer.flip();

fileChannel.write(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {

@Override

public void completed(Integer result, ByteBuffer attachment) {

System.out.println("bytes written: " + result);

}

@Override

public void failed(Throwable exc, ByteBuffer attachment) {

System.out.println("Write failed");

exc.printStackTrace();

}

});

The CompletionHandler's completed() method will get called when the write operation completes. If the write fails for some reason, the failed() method will get called instead.

Notice how the ByteBuffer is used as attachment - the object which is passed on to theCompletionHandler's methods.