JAVA NIO

Veerle Ongenae

Java NIO

2

- Java New IO
- Alternatief
  - Java IO
  - Java Networking

Industrieel Ingenieur Informatica, UGent

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

Java NIO: Channels and Buffers

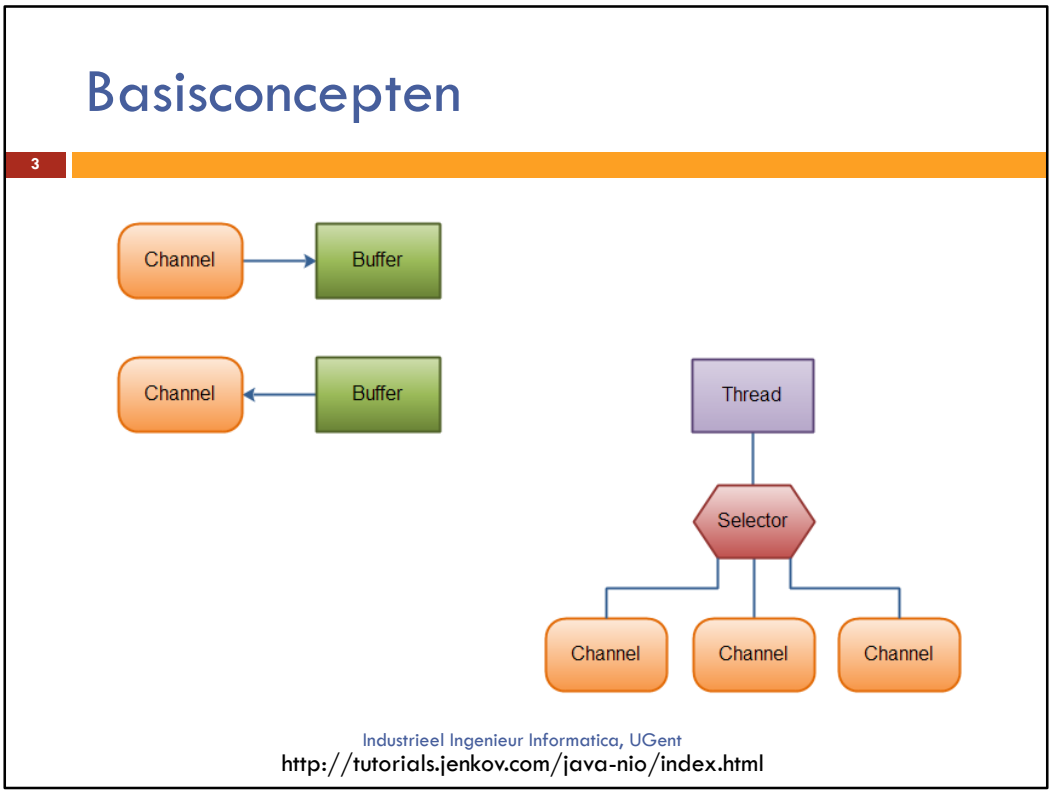
In the standard IO API you work with byte streams and character streams. In NIO you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.

Java NIO: Non-blocking IO

Java NIO enables you to do non-blocking IO. For instance, a thread can ask a channel to read data into a buffer. While the channel reads data into the buffer, the thread can do something else. Once data is read into the buffer, the thread can then continue processing it. The same is true for writing data to channels.

Java NIO: Selectors

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



Java NIO consist of the following core components:

- Channels
- Buffers
- Selectors

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

Channels and Buffers

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

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.

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.
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.

Channel

4

```
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();
```


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 before, you read data from a channel into a buffer, and write data from a buffer into a channel.

Channel Implementations

Here are the most important Channel implementations in Java NIO:

- **FileChannel** reads data from and to files.
- **DatagramChannel** can read and write data over the network via UDP.
- **SocketChannel** can read and write data over the network via TCP.
- **ServerSocketChannel** allows you to listen for incoming TCP connections, like a web server does. For each incoming connection a SocketChannel is created.

The **Basic Channel Example**

Notice the buf.flip() call.

First you read into a Buffer. Then you flip it. Then you read out of it.

Instances of **RandomAccessFile** support both reading and writing to a random access file. A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the *file pointer*; input

operations read bytes starting at the file pointer and advance the file pointer past the bytes read. If the random access file is created in read/write mode, then output operations are also available; output operations write bytes starting at the file pointer and advance the file pointer past the bytes written. Output operations that write past the current end of the implied array cause the array to be extended. The file pointer can be read by the getFilePointer method and set by the seek method.

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:
- Write data into the Buffer
- Call buffer.flip()
- Read data out of the Buffer
- 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.

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:
- Write data from a Channel into a Buffer
- 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.

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

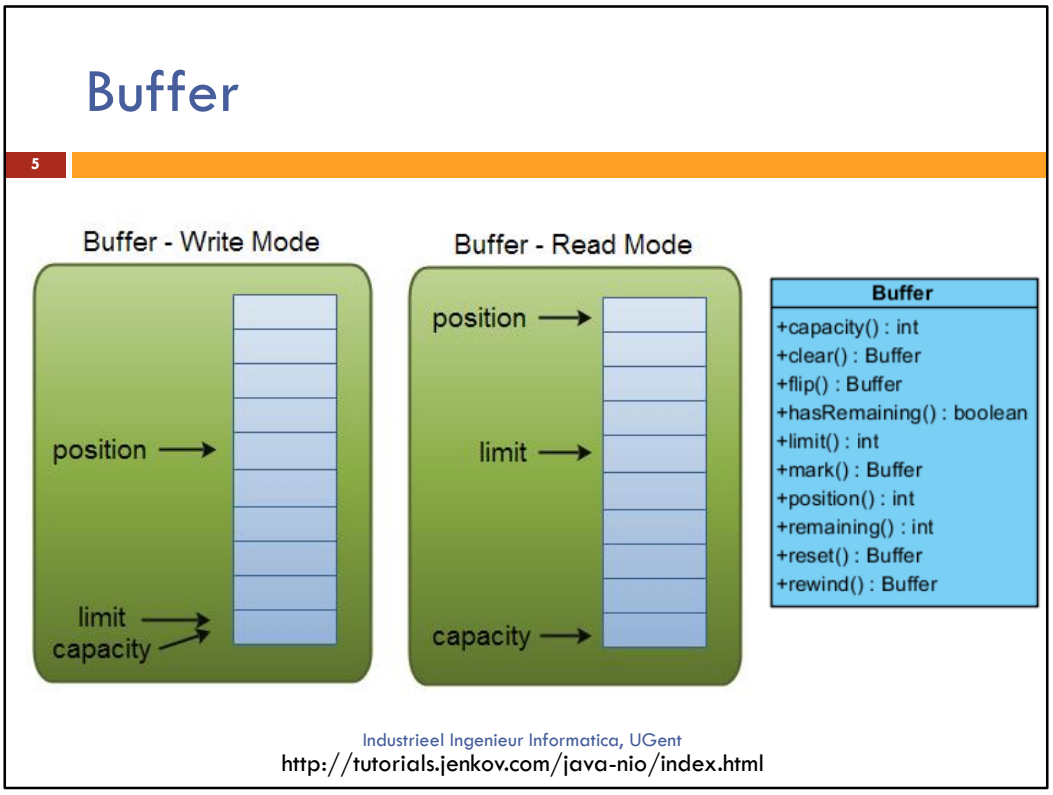
Here is an example of how you can read data from a buffer into a channel:

```
//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.



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.

Capacity

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

Position

When you write data into the Buffer, you do so at a certain position. Initially the position is 0. When a byte, long etc. has been written into the Buffer the position is

advanced to point to the next cell in the buffer to insert data into. Position can maximally become capacity - 1.
When you read data from a Buffer you also do so from a given position. When you flip a Buffer from writing mode to reading mode, the position is reset back to 0. As you read data from the Buffer you do so from position, and position is advanced to next position to read.

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

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.

MappedByteBuffer: a direct byte buffer whose content is a memory-mapped region of a file.

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:

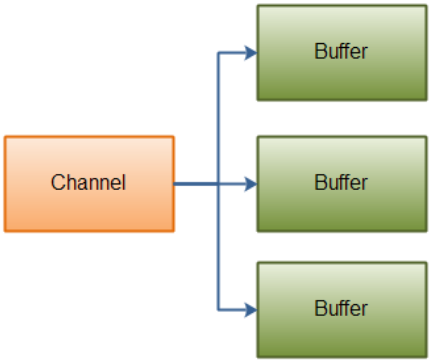
```
buffer.mark();
//call buffer.get() a couple of times, e.g. during parsing.
buffer.reset(); //set position back to mark.
```

equals() and compareTo()
It is possible to compare two buffers using equals() and compareTo().
equals()
Two buffers are equal if:
- They are of the same type (byte, char, int etc.)
- They have the same amount of remaining bytes, chars etc. in the buffer.
- 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:
- The first element which is equal to the corresponding element in the other buffer, is smaller than that in the other buffer.
- All elements are equal, but the first buffer runs out of elements before the second buffer does (it has fewer elements).

Scattering reads

6




```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);

ByteBuffer[] bufferArray = { header, body };

channel.read(bufferArray);
```

Industrieel Ingenieur Informatica, UGent
http://tutorials.jenkov.com/java-nio/index.html

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.

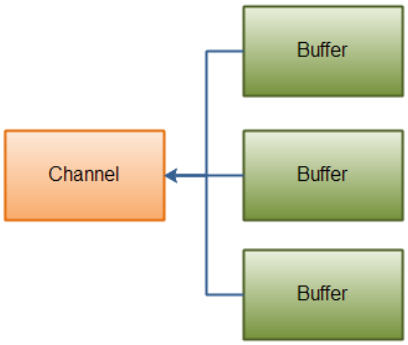
Scattering Reads
A "scattering read" reads data from a single channel into multiple buffers.

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

7




```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);

//write data into buffers

ByteBuffer[] bufferArray = { header, body };

channel.write(bufferArray);
```

Industrieel Ingenieur Informatica, UGent
http://tutorials.jenkov.com/java-nio/index.html

A "gathering write" writes data from multiple buffers into a single channel.

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.

Channel to channel transfer

8

```
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);
```


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.

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.

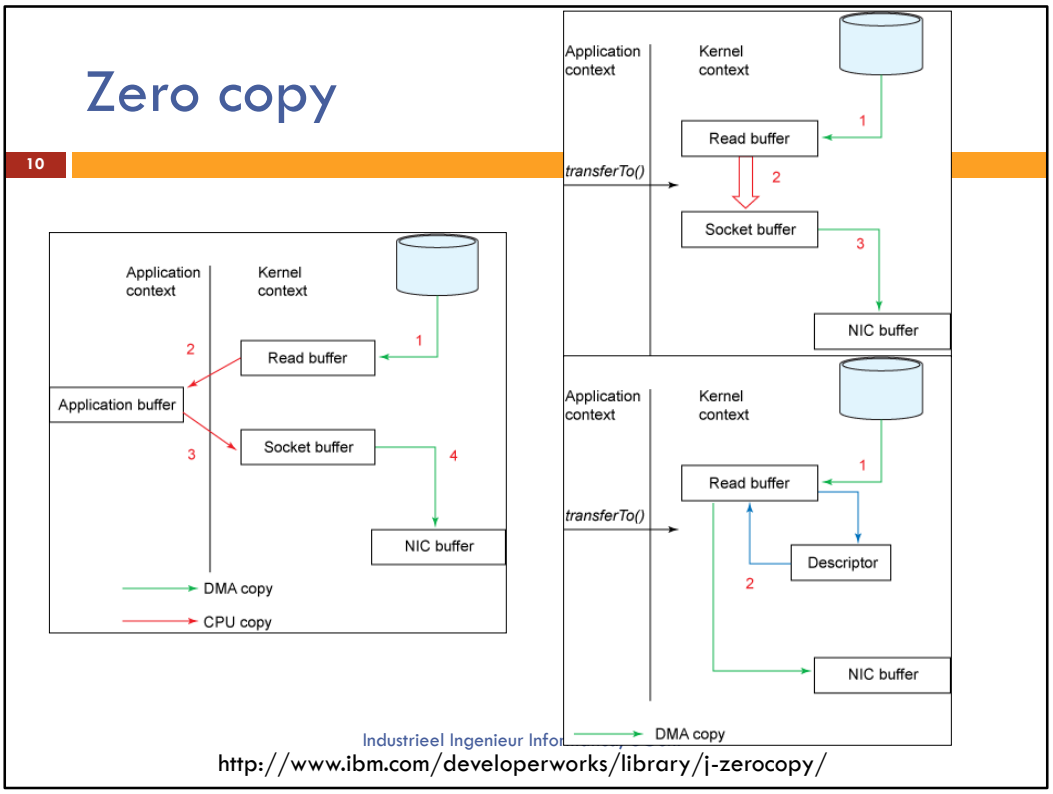
Channel to channel transfer

9

```
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);
```


transferTo()
The transferTo() method transfer from a FileChannel into some other channel.

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.



Many Web applications serve a significant amount of static content, which amounts to reading data off of a disk and writing the exact same data back to the response socket. This activity might appear to require relatively little CPU activity, but it's somewhat inefficient: the kernel reads the data off of disk and pushes it across the kernel-user boundary to the application, and then the application pushes it back across the kernel-user boundary to be written out to the socket. In effect, the application serves as an inefficient intermediary that gets the data from the disk file to the socket.

Each time data traverses the user-kernel boundary, it must be copied, which consumes CPU cycles and memory bandwidth. Fortunately, you can eliminate these copies through a technique called — appropriately enough — *zero copy*. Applications that use zero copy request that the kernel copy the data directly from the disk file to the socket, without going through the application. Zero copy greatly improves application performance and reduces the number of context switches between kernel and user mode.

The Java class libraries support zero copy on Linux and UNIX systems through the transferTo() method in java.nio.channels.FileChannel. You can use the transferTo() method to transfer bytes directly from the channel on which it is invoked to another writable byte channel, without requiring data to flow through the application. This article first demonstrates the overhead incurred by simple file transfer done through traditional copy semantics, then shows how the zero-copy technique using transferTo() achieves better performance.

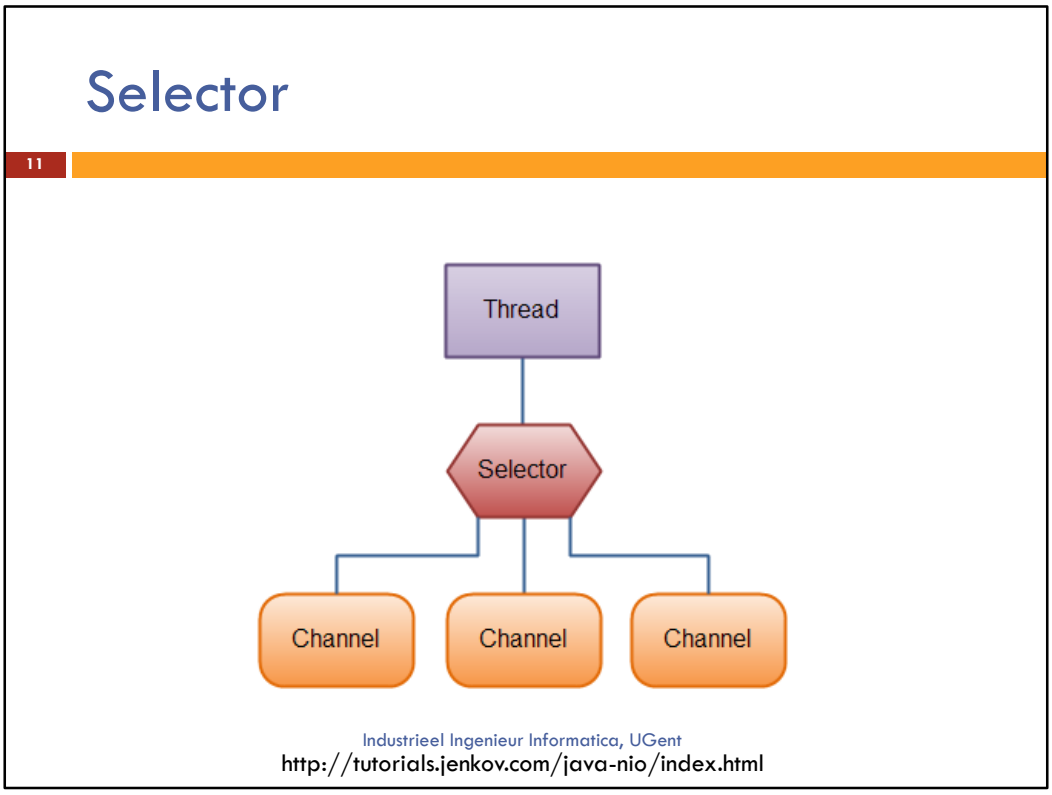
Date transfer: The traditional approach
Consider the scenario of reading from a file and transferring the data to another program over the network. (This scenario describes the behavior of many server applications, including Web applications serving static content, FTP servers, mail servers, and so on.) (figuur 1)

Use of the intermediate kernel buffer (rather than a direct transfer of the data into the user buffer) might seem inefficient. But intermediate kernel buffers were introduced into the process to improve performance. Using the intermediate buffer on the read side allows the kernel buffer to act as a "readahead cache" when the application hasn't asked for as much data as the kernel buffer holds. This significantly improves performance when the requested data amount is less than the kernel buffer size. The intermediate buffer on the write side allows the write to complete asynchronously.
Unfortunately, this approach itself can become a performance bottleneck if the size of the data requested is considerably larger than the kernel buffer size. The data gets copied multiple times among the disk, kernel buffer, and user buffer before it is finally delivered to the application.
Zero copy improves performance by eliminating these redundant data copies.

Data transfer: The zero-copy approach
If you re-examine the traditional scenario, you'll notice that the second and third data copies are not actually required. The application does nothing other than cache the data and transfer it back to the socket buffer. Instead, the data could be transferred directly from the read buffer to the socket buffer. The transferTo() method lets you do exactly this. (figuur 2)

This is an improvement: we've reduced the number of context switches from four to two and reduced the number of data copies from four to three (only one of which involves the CPU). But this does not yet get us to our goal of zero copy. We can further reduce the data duplication done by the kernel if the underlying network interface card supports *gather operations*. In Linux kernels 2.4 and later, the socket buffer descriptor was modified to accommodate this requirement. This approach not only reduces multiple context switches but also eliminates the duplicated data copies that require CPU involvement. The user-side usage still remains the same, but the intrinsics have changed:

- The transferTo() method causes the file contents to be copied into a kernel buffer by the DMA (direct memory access) engine.
- No data is copied into the socket buffer. Instead, only descriptors with information about the location and length of the data are appended to the socket buffer. The DMA engine passes data directly from the kernel buffer to the protocol engine, thus eliminating the remaining final CPU copy.



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.

Selector – aanmaak en registratie

12

```
// aanmaak selector
Selector selector = Selector.open();

// registratie één of meerdere kanalen
channel.configureBlocking(false);
SelectionKey key
  = channel.register(selector, SelectionKey.OP_READ);
```


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:

- Connect
- Accept
- Read
- 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:

- SelectionKey.OP_CONNECT
- SelectionKey.OP_ACCEPT
- SelectionKey.OP_READ
- 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;
```

Selector - gebruik

13

```
while(true) {
  // wachten
  int readyChannels = selector.select();
  if(readyChannels == 0) continue;

  for (SelectionKey key : selector.selectedKeys()) {
    .. // kanaal klaar om te …
  }
  selector.selectedKeys().clear();
}
```

Industrieel Ingenieur Informatica, UGent

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()** blocks until at least one channel is ready for the events you registered for.
- **int select(long timeout)** does the same as select() except it blocks for a maximum of timeout milliseconds (the parameter).
- **int 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.

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.

You can iterate this selected key set to access the ready channels.
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 `selector.selectedKeys().clear();` call after the 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.

Selector - communicatie

14

```
if(key.isAcceptable()) {
    // a connection was accepted by a ServerSocketChannel.
    Channel channel = key.channel(); …
} 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
}
…
```

Industrieel Ingenieur Informatica, UGent

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)

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. 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);
```

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.

SocketChannel: aanmaak - lezen

15

```
// aanmaken
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(
  new InetSocketAddress("http://jenkov.com", 80));

// lezen
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = socketChannel.read(buf);
```


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. There are two ways a SocketChannel can be created:

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

Reading from a SocketChannel

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).

SocketChannel: schrijven - sluiten

16

```
// buffer met info
String newData = "New String to write to file...";
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
// data schrijven
buf.flip();
while(buf.hasRemaining()) {
    socketChannel.write(buf);
}
// afsluiten
socketChannel.close();
```


Writing to a SocketChannel
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.

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.

In non-blocking mode the write() method may return without having written anything.

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.

The non-blocking mode of SocketChannel's works best 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.

ServerSocketChannel: aanmaak

17

```
ServerSocketChannel serverSocketChannel =
  ServerSocketChannel.open();

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

while(true){
  SocketChannel socketChannel = serverSocketChannel.accept();

  //do something with socketChannel...
}

serverSocketChannel.close();
```


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.

Voorbeeld

- ChatServerProgram.java
- ChatServer.java

Java IO versus java NIO

19

IO	NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
	Selectors

Industrieel Ingenieur Informatica, UGent

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?

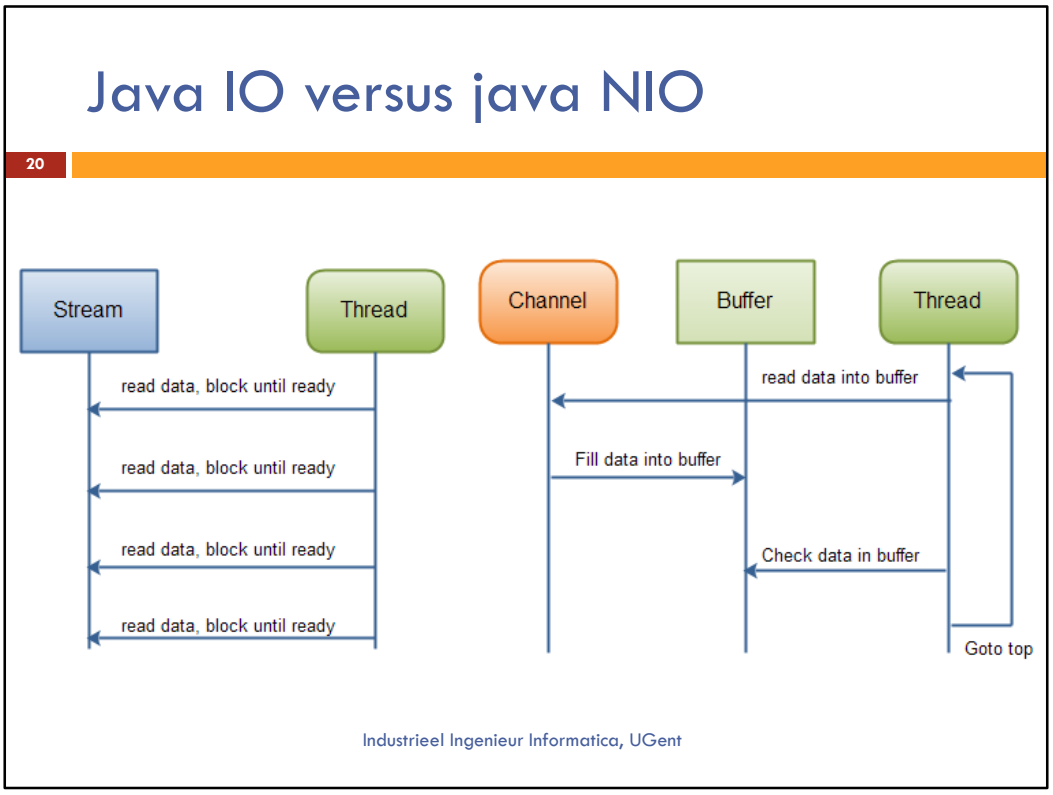
Main Differences Between Java NIO and IO
The table above summarizes the main differences between Java NIO and IO.

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:

- The API calls to the NIO or IO classes.
- The processing of data.
- 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 first reader.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 second readLine() 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).

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:

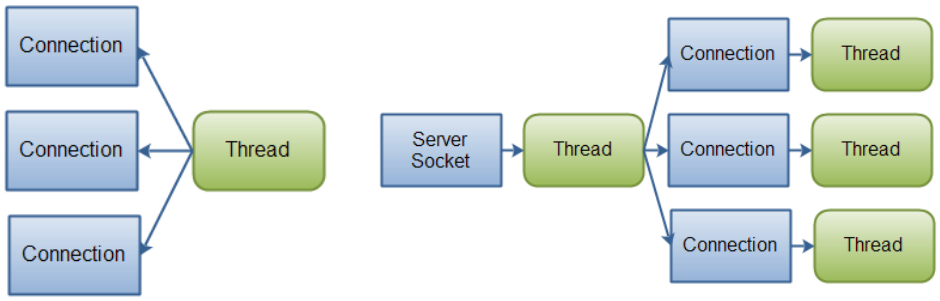
```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(! bufferFull(bytesRead) ) {
  bytesRead = inChannel.read(buffer);
}
```

The bufferFull() method has to keep track of how much data is read into the buffer, and return either true or false, 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 the bufferFull() 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.

Java IO versus java NIO

21



Industrieel Ingenieur Informatica, UGent

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

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

The primary contribution of NIO and NIO.2 to the Java platform is to improve performance in one of the core areas of Java application development: input/output processing. Neither package is particularly easy to work with, nor are the New Input/Output APIs required for every Java I/O scenario. Used correctly, though, Java NIO and NIO.2 can slash the time required for some common I/O operations.

Java IO versus java NIO

22

- Change notifiers
- Selectors help multiplex
- Channels
- Memory mapping
- Character encoding and searching

http://www.javaworld.com/article/2078654/java-se/java-se-five-ways-to-maximize-java-nio-and-nio-2.html

Change notifiers (because everybody needs a listener)
Many enterprise-class applications need to take a specific action when:

- A file is uploaded into an FTP folder
- A configuration definition is changed
- A draft document is updated
- Another file-system event occurs

These are all examples of change notification or change response. In early versions of Java (and other languages), *polling* was typically the best way to detect change events. Polling is a particular kind of endless loop: check the file-system or other object, compare it to its last-known state, and, if there's no change, check back again after a brief interval, such as a hundred milliseconds or ten seconds. Continue the loop indefinitely.
NIO.2 gives us a better way to express change detection. NIO.2's Watcher class is considerably more straightforward and easy-to-use for change notification than the traditional I/O solution based on polling.

Selectors and asynchronous I/O: Selectors help multiplex
Newcomers to NIO sometimes associate it with "non-blocking input/output." NIO is more than non-blocking I/O but the error makes sense: basic I/O in Java is *blocking* -- meaning that it waits until it can complete an operation -- whereas non-blocking, or asynchronous, I/O is one of the most-used NIO facilities.
NIO's non-blocking I/O is *event-based*. This means that a *selector* (or callback or

listener) is defined for an I/O channel, then processing continues. When an event occurs on the selector -- when a line of input arrives, for instance -- the selector "wakes up" and executes. All of this is achieved *within a single thread*, which is a significant contrast to typical Java I/O. Unix and Unix-like operating systems have long had efficient implementations of selectors, so this sort of networking program is a model of good performance for a Java-coded networking program.

Channels: Promise and reality

In NIO, a *channel* can be any object that reads or writes. Its job is to abstract files and sockets. NIO channels support a consistent collection of methods, so it's possible to program without having special cases depending on whether stdout, a network connection, or some other channel is actually in use. Channels share this characteristic of the *streams* of Java's basic I/O. Streams provide blocking I/O; channels support asynchronous I/O.

While NIO is often promoted for its performance advantages, it's more precise to say it is highly *responsive*. In some cases NIO actually performs *worse* than basic Java I/O. For simple sequential reads and writes of small files, for instance, a straightforward streams implementation might be two or three times faster than the corresponding event-oriented channel-based coding. Also, *non*-multiplexed channels -- that is, channels in separate threads -- can be much slower than channels that register their selectors in a single thread.

The next time you need to define a programming problem in terms of dimensions pertinent to streams or channels, try asking the following questions:

- How many I/O objects must you read and write?
- Is there a natural sequence between different I/O objects or might they all need to happen simultaneously?
- Do your I/O objects only last for a short interval or are they likely to persist during the lifetime of your process?
- Is it more natural to do your I/O in a single thread or several distinct ones?
- Is network traffic likely to look the same as local I/O or do the two have different patterns?

This sort of analysis is good practice for deciding when to use streams or channels. Remember: NIO and NIO.2 don't replace basic I/O; they just supplement it.

Memory mapping -- where it counts

The most consistently dramatic performance improvement in the use of NIO involves memory mapping. *Memory mapping* is an OS-level service that makes segments of a file appear for programming purposes like areas of memory.

Memory mapping has a number of consequences and implications, more than I'll get into here. At a high level, it helps make I/O happen at the speed of memory access, rather than file access. The former is often two orders of magnitude faster than the latter.

In practical cases, memory mapping is interesting not only for the raw speed of I/O, but also because several different readers and writers can attach simultaneously to the same file image. This technique is powerful enough to be dangerous, but in the right hands it makes for exceedingly fast implementations.

Character encoding and searching

The final feature of NIO is charset, a package for converting between different character encodings. Even before NIO, Java built in much of the same functionality through the getBytes method. charset is welcome, though, because it is more flexible than getBytes and easier to implement at a low architectural level, which yields superior performance. This is particularly valuable for searches that must be sensitive to the encoding, collation, and other characteristics of languages other than English.