

# NIO

Path and Files

File Walk and WatchService

Channels and Buffers

Non-Blocking Channel Operations

Asynchronous Channels

Miscellaneous

# MOTIVATION

NIO und NIO.2 introduced **Channels** as alternative to Input/OutputStreams

## Channels

- provide elementary API (low-level API)
- support efficient reading from and writing to files and sockets
- support asynchronous operations

Package `java.nio.channels`

Note: Input/OutputStreams are now implemented using channels

# CONCEPTS

## Channels:

- Channels are created for data sources and sinks, i.e. files, sockets
- then support bi-directional reading and writing

Buffer: Channels work with buffers; the following buffers are supported:

- `ByteBuffer`
- `CharBuffer`
- `ShortBuffer`
- `IntBuffer`
- `LongBuffer`
- `FloatBuffer`
- `DoubleBuffer`

**Selectors:** Selectors allow listening to multiple channels and thus allow a thread to handle multiple channels simultaneously

Asynchronous Channels for asynchronous operations

Locking of files using channels

# CHANNELS

Channels are created for files or sockets, e.g., for files  
`Files.newByteChannel(Path path)`

```
try (  
    ByteChannel srcChnl = Files.newByteChannel(srcPath, StandardOpenOption.READ);  
    ByteChannel destChnl = Files.newByteChannel(destPath, StandardOpenOption.WRITE);  
) {
```

Further, there are a set of methods which create channels with different properties

Channel Zoo:

**Interfaces:**

`ByteChannel`, `ReadableByteChannel`, `WritableByteChannel`, `SeekableByteChannel`,  
`AsynchronousByteChannel`, `AsynchronousChannel`, ...

**Classes implementing various interfaces:**

`FileChannel`, `SocketChannel`, `ServerSocketChannel`, `AsynchronousFileChannel`,  
`AsynchronousSocketChannel`, `AsynchronousServerSocketChannel`,  
`DatagramChannel`, ...

# READING AND WRITING WITH CHANNELS

Reading and writing using buffers,  
i.e., channels read data into buffers and write data from buffers

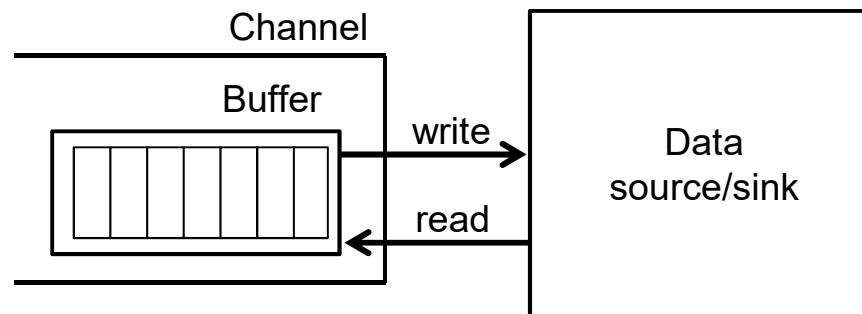
-1 for end of stream

read data put into buffer

```
int bytesRead = channel.read( buffer );
```

```
channel.write( buffer );
```

written data taken from buffer



Buffers are similar to arrays but with special API

Factory methods for creating buffers

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

# BUFFER API

Essential method for buffers:

- **put**: filling a buffer with data

```
public ByteBuffer put(byte[] src)
public ByteBuffer put(byte[] src, int offset, int length)
public ByteBuffer put(ByteBuffer src)
```

**get**: getting data from buffer

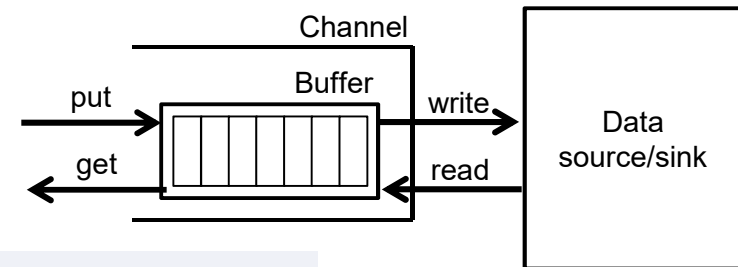
```
public byte get()
public ByteBuffer get(byte[] dst)
public ByteBuffer get(byte[] dst, int offset, int length)
public byte get(int index)
public char getChar()
public double getDouble()
...
```

- **clear**: deleting the content

```
public final Buffer clear()
```

- **flip** and **rewind**: resetting read and write cursors

```
public Buffer flip()
public Buffer rewind()
```



# BUFFER API

Properties:

- **capacity**: capacity of buffer

```
public int capacity()
```

- **position**: current reading or writing position

```
public int position()  
public Buffer position(int newPosition)
```

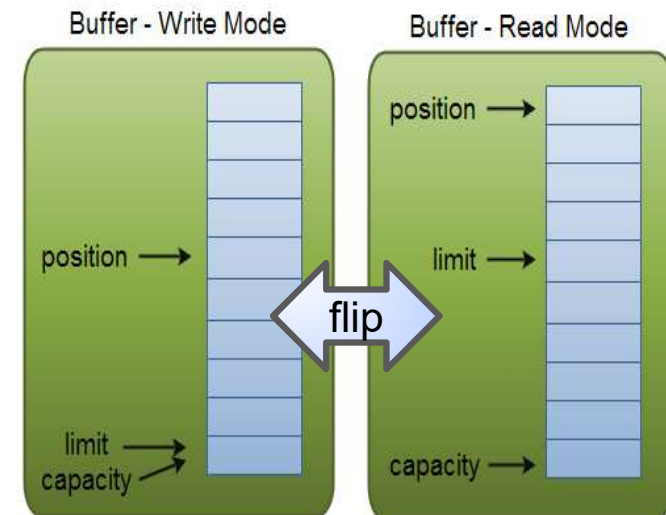
- **limit**: position how far a buffer can be read or written

```
public int limit()  
public Buffer limit(int newLimit)
```

# BUFFER BEHAVIOR

Interpretation of **position** and **limit** dependent on read/write mode of buffer

- Write Mode = putting data into buffer
  - **position**: next position for new element
  - **limit**: equal to capacity
- Read Mode = getting data from buffer
  - **position**: next position for accessing element
  - **limit**: to which position one can get data (exclusive)
- **flip**, **clear**, **rewind** set position und limit
  - **flip**: sets limit to current position and position to 0
  - **clear**: sets limit to capacity and position to 0
  - **rewind**: sets position to 0





# BUFFER BEHAVIOR

Example: Putting data into buffer and then reading buffer

```
ByteBuffer buffer = ByteBuffer.allocate(8);
```

```
byte[] bytes = new byte[2] { (byte) 2, (byte)3 };
```

- Putting data into buffer

sets buffer to write mode

```
buffer.clear();  
buffer.put((byte)1);  
buffer.put(bytes);  
buffer.put((byte)4);
```

position	limit	capacity
0	8	8
1	8	8
3	8	8
4	8	8

- Getting data from buffer

sets buffer to read mode

```
buffer.flip();  
byte b = buffer.get();  
buffer.get(bytes);  
b = buffer.get();
```

position	limit	capacity
0	4	8
1	4	8
3	4	8
4	4	8

# EXAMPLE: COPYING FILE

```
try (  
    ByteChannel srcChnl = Files.newByteChannel(srcPath, StandardOpenOption.READ);  
    ByteChannel destChnl = Files.newByteChannel(destPath, StandardOpenOption.WRITE,  
                                                StandardOpenOption.CREATE);  
    ) {  
        ByteBuffer buffer = ByteBuffer.allocate(16);  
  
        int nRead = srcChnl.read(buffer);  
        while (nRead >= 0) {  
            buffer.flip();  
            destChnl.write(buffer);  
            buffer.clear();  
            nRead = srcChnl.read(buffer);  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

New buffer

Read data and fill buffer

Flip for getting read data

Writing data from buffer

Clearing buffer for new data

Read new data and fill buffer

# CHARACTER ENCODING

## Class `Charset` for character encoding

- Package `java.nio.charset`

## Conversion between `ByteBuffer` and `CharBuffer`

- Creating a `Charset` object by static method `forName` with name of encoding

```
Charset cset = Charset.forName("UTF-8");
```

- Encoding

```
String strg = "...";  
ByteBuffer bbuf = cset.encode(strg);
```

```
String strg = "...";  
CharBuffer cbuf = CharBuffer.allocate(32);  
cbuf.append(strg);
```

```
cbuf.flip();  
ByteBuffer bbuf = cset.encode(cbuf);
```

Switch mode!

- Decoding

```
CharBuffer cbuf = cset.decode(bbuf);  
String strg = cbuf.toString();
```

# READER AND WRITER

## Using CharBuffers and Charset with Reader and Writer

- Creating Reader and Writer with character encoding

```
Charset cset = Charset.forName("UTF-8");  
Writer out = new OutputStreamWriter(new FileOutputStream("srcfile.txt"), cset);  
  
Reader in = new InputStreamReader(new FileInputStream("destfile"), cset);
```

- Reading into and writing from CharBuffers

```
CharBuffer cbuf = CharBuffer.allocate(64);  
int nRead = reader.read(cbuf);  
  
cbuf.flip();  
writer.append(cbuf);
```

### Example: File copy

```
Charset cset = Charset.forName("UTF-8");  
try ( Reader reader = new InputStreamReader(new FileInputStream("srcfile.txt"), cset);  
      Writer writer = new OutputStreamWriter(new FileOutputStream("destfile.txt"), cset); ) {  
    CharBuffer cbuf = CharBuffer.allocate(64);  
    int nRead = reader.read(cbuf);  
    while (nRead >= 0) {  
        cbuf.flip();  
        writer.append(cbuf);  
        cbuf.clear();  
        nRead = reader.read(cbuf);  
    }  
} catch (IOException e) { ... }
```

# SOCKETS AND CHANNELS

Special classes and methods for working with channels and sockets

## ServerSocketChannel

- for creating connection at server

```
ServerSocketChannel server = ServerSocketChannel.open();  
server.bind(new InetSocketAddress(port));  
SocketChannel channel = server.accept();
```

Open server channel  
bind it to address  
try to connect to clients

## SocketChannel

- for client/server communication

```
SocketChannel channel = SocketChannel.open();  
channel.connect(new InetSocketAddress(SERVER_IP, PORT));
```

Open channel  
bind it to address and  
connect

- bidirectional communication

```
channel.write( buffer );  
channel.read( buffer );
```

Writing to channel from buffer  
Reading from channel into buffer

- closing channel required

```
channel.close();
```

# NIO

Path and Files

File Walk and WatchService

Channels and Buffers

Non-Blocking Channel Operations

Asynchronous Channels

Miscellaneous

# NON-BLOCKING CHANNEL OPERATIONS (1/2)

Channels support non-blocking operations

- Operations executed asynchronously
  - Channels have to be configured to be non-blocking

```
channel.configureBlocking(false);
```

- then read, write, accept do not block
- but trigger events

## Selector for working with events

- Creating and opening **Selector**

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

- Registering channel operations at **Selector**
  - SelectionKey** serves as access to registration

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

SelectionKey.OP\_READ  
SelectionKey.OP\_WRITE  
SelectionKey.OP\_OPEN  
SelectionKey.OP\_ACCEPT

Multiple channels can be registered with same Selector  
→ then the Selector can handle multiple (many) channels

# NON-BLOCKING CHANNEL OPERATIONS (2/2)

## Handling events

- Getting events from **Selector**

```
int n = selector.select();  
int n = selector.select(1000);
```

blocking wait until events are available!!

with timeout in ms

- Accessing set of events

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

- Iteration and handling events

```
Iterator<SelectionKey> keyIterator = keys.iterator();  
while (keyIterator.hasNext()) {  
    SelectionKey key = keyIterator.next();  
    if (key.isAcceptable()) {  
        ...  
    } else if (key.isReadable()) {  
        ...  
    } else if (key.isWritable()) {  
        ...  
    } ...  
    keyIterator.remove();  
}
```

removal of keys required!!



# EXAMPLE: LISTENING TO MULTIPLE SOCKETS

## Handling inputs from multiple socket connections

- Server allows multiple client connections

```
ServerSocketChannel server = null;
try {
    server = ServerSocketChannel.open();
    server.socket().bind(new InetSocketAddress(port));
    server.configureBlocking(true);
    while (!stopServer) {
        SocketChannel channel = server.accept();
```

- Channel is set into non-blocking Mode
- and registered at **Selector** for read operations

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
} catch (ClosedByInterruptException ie) {
    ...
} catch (IOException e1) {
    ...
} finally {
    try {
        server.close();
    } catch (IOException e) { }
}
```

# EXAMPLE: LISTENING TO MULTIPLE SOCKETS

- Inputs of all channels can be handled in one loop (and with a single thread)

```
while (! stopServer) {  
    try {  
        int n = selector.select(1000);  
        Set<SelectionKey> keys = selector.selectedKeys();  
        Iterator<SelectionKey> keyIt = keys.iterator();  
        while (keyIt.hasNext()) {  
            SelectionKey key = keyIt.next();  
            if (key.isReadable()) {  
                SocketChannel chnl = (SocketChannel)key.channel();  
                buffer.clear();  
                chnl.read(buffer);  
                buffer.flip();  
                byte[] data = new byte[buffer.limit()];  
                buffer.get(data);  
                System.out.println(Arrays.toString(data));  
            }  
            keyIt.remove();  
        }  
    } catch (IOException e) {  
    }  
}
```

Blocks until event (with timeout)

Reaction to input

Access of triggering channel

Read from channel

Output on console

# ATTACHMENTS TO SELECTIONKEYS

## SelectionKeys allow attachments

→ can be used for forwarding important information in events

**Note:** Very useful when handling multiple client channels

- Adding attachment at registration

```
Object attachment = ... ;  
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);  
key.attach( attachment );
```

- Accessing attachment when getting event

```
int n = selector.select(500);  
Set<SelectionKey> keys = selector.selectedKeys();  
Iterator<SelectionKey> keyIt = keys.iterator();  
while (keyIt.hasNext()) {  
    SelectionKey key = keyIt.next();  
  
    Object attachment = key.attachment();
```

**Note:** Get channel where event occurred with `key.channel()`

```
SocketChannel chnl = (SocketChannel)key.channel();
```

# NIO

Path and Files

File Walk and WatchService

Channels and Buffers

Non-Blocking Channel Operations

Asynchronous Channels

Miscellaneous

# ASYNCHRONOUS CHANNELS

Asynchronous channels support asynchronous event processing

- either with **Futures**
- or with **CompletionHandler**

analogous  
AsynchronousSocketChannel,  
AsynchronousServerSocketChannel

Approach:

- Open an **AsynchronousChannels**, e.g., an **AsynchronousFileChannel**

```
AsynchronousFileChannel fileChannel =  
    AsynchronousFileChannel.open(path, StandardOpenOption.READ);
```

- Asynchronous read with **Future**

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

- and getting event and data from **Future**

```
int nRead = future.get();  
// processing data  
buffer.flip();  
byte[] data = new byte[buffer.limit()];  
buffer.get(data);  
...
```

blocks!

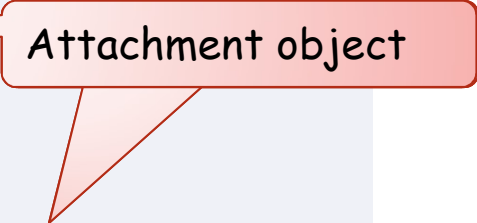
# COMPLETIONHANDLER

Use `CompletionHandler` to implement callback

- Provide `CompletionHandler` with operations
- After termination of operation method `completed` or `failed` is called
- Further, using `attachment` allows forwarding object from call to handler

Note: `CompletionHandler` is generic in result and attachment object

```
fileChannel.read(buffer, position, attachment,  
    new CompletionHandler<Integer, Object>() {  
  
    @Override  
    public void completed(Integer result, Object attachment) {  
        buffer.flip();  
        byte[] data = new byte[buffer.limit()];  
        buffer.get(data);  
  
        ...  
    }  
    @Override  
    public void failed(Throwable exc, Object attachment) {  
        // handle failed read operation  
    }  
});
```



# NIO

Path and Files

File Walk and WatchService

Channels and Buffers

Non-Blocking Channel Operations

Asynchronous Channels

Miscellaneous

# LOCKS

## Locking of files using channels

```
FileLock lock = fileChannel.lock( );
```

waits for file lock

```
FileLock lock = fileChannel.tryLock( );
```

returns null if lock not available

## Lock for releasing file lock

```
lock.release();
```