Basis for efficient IO

Each io operation adds overhead, longstanding best practice is to perform IO in larger rather than smaller blocks, this means use some kind of buffering.

Also each data copy operation slows things down. Ideally read directly into the end buffer (typically requires OS support). Avoid secondary copy operations.

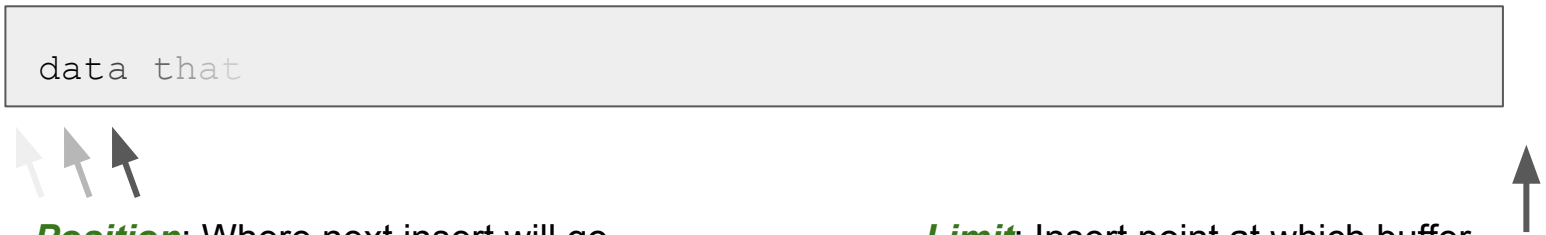
Java NIO provides for very low-level (non-object oriented) buffers that are efficient in use

Filling an NIO buffer

IO *read* operations *write* into a buffer

A newly created buffer is ready to receive data

Capacity: How much data the buffer holds when full



Position: Where next insert will go. Initially 0, increments during filling.

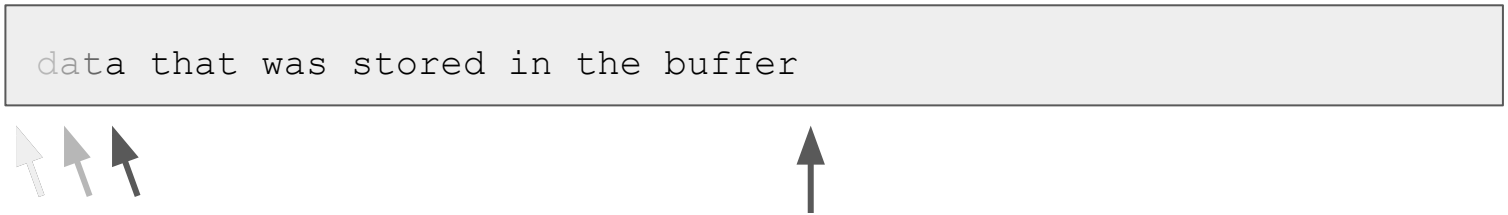
Limit: Insert point at which buffer becomes full. Same as capacity for filling operations.

Emptying an NIO buffer

IO *write* operations *read* from a buffer.

After filling, call buffer.flip() to prepare for emptying.

Capacity: How much data the buffer holds when full



Position: Where next data item will be extracted from.
Initially 0 after the flip() increments as data is extracted from the buffer.

Limit: Index point of first "non-data".
The flip() operation sets this to the Position prior to the flip operation

Character data encoding conversion

Text data might need to be converted between encodings.

Java uses UTF-16 (version 10.0 at Java SE 11)

External encoding depends on platform and data source, but is commonly UTF-8

A CharSet represents a text encoding, and provides encoder and decoder objects that can convert to or from that external encoding

A CharSetDecoder takes data from a ByteBuffer and places the UTF-16 equivalent into a CharBuffer. CharSetEncoder does the reverse.

Decoders empty a source, and store in a target, buffer. Encoders vice-versa

Non-blocking network IO

A Selector allows registering for many "interesting" events in a network IO system, including: a client has connected, a socket has readable data, a socket is ready to accept data for writing.

The selector blocks a thread until at least one event has occurred.

When the thread unblocks, it can determine which event(s) occurred, and obtain the necessary supporting information to process it/them.

Setting up a non-blocking network server

Initialize a Selector object:

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

Initialize a ServerSocketChannel:

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

Bind the socket to the local port:

```
ssc.bind(new InetSocketAddress(8000));
```

Configure the socket as non-blocking:

```
ssc.configureBlocking(false);
```

Register for accept events:

```
ssc.register(sel, SelectionKey.OP_ACCEPT);
```

Loop body for non-blocking network server

Await an interesting event:

```
sel.select();
```

A "key" is an object indicating an active event. Get the currently active keys:

```
Iterator<SelectionKey> keys = sel.selectedKeys().iterator();
```

Iterate over each key and process it:

```
while (keys.hasNext()) {
  int readyOperations = key.readOps();
```

Identify which operations are ready

Bit flags indicate what operations are ready:

```
if (readyOperations & SelectionKey.OP_ACCEPT) != 0)
 // process a new connection
```

Operations are:
`SelectionKey.OP_READ` the port has read data
`SelectionKey.OP_WRITE` the port is ready to write data
`SelectionKey.OP_CONNECT` the (client) port has connected to the server

Processing an operation

Extract the active socket from the key.
`key.channel()`

For a server accept operation cast to `ServerSocketChannel`, for regular socket, cast to `SocketChannel`.
`((ServerSocketChannel)(key.channel())` or
`((SocketChannel)(key.channel())`

Then process the resulting channel as appropriate.

Processing accept operations

When a ServerSocketChannel accepts:

1) Extract the newly connected `SocketChannel`
   ```
   SocketChannel sc = ssc.accept();
   ```
2) Configure the `SocketChannel` in non-blocking mode
   ```
   sc.configureBlocking(false);
   ```
3) Register with the `Selector` for read (and/or write) operations
   ```
   sc.register(selector, SelectionKey.OP_READ);
   ```

Processing data-read operations

To process a socket that has data available, call `read` to move data into a buffer:

```
sc.read(byteBuffer);
```

1) If text data has been completely read, use a `CharSetDecoder` to convert it and put the result in a `CharBuffer`:
   ```
   decoder.decode(byteBuffer, charBuffer, true);
   ```
2) Call `flip()` on the `CharBuffer` before extracting the converted text
3) Process the data according to the business requirements.
4) When ready to send a response register with the selector:
   ```
   sc.register(sel, SelectionKey.OP_WRITE, dataToBeWritten);
   ```

Processing ready to write operations

To process a ready-to-send socket, get `dataToBeWritten` from the key:

```
String message = (String)key.attachment();
```

1) If conversion is needed, use the appropriate encoder, then `flip()` the resulting `DataBuffer`
2) Write the data to the `SocketChannel`:

```
sc.write(byteBuffer);
```

NIO utility functions

Package `java.nio.file` contains classes `Files`, `Paths`, and the interface `Path` (among others). These provide utility methods for path and file operations.

Objects implementing `Path` describe paths and directories on a file system, allowing operations to be performed on those file system elements. Since Java 11, static factory methods `of(String first, String … rest)` and `of(URI u)` create `Path` objects.

Prior to Java 11, the class `Paths` provides platform agnostic factory methods for building `Path` implementations

The `Files` class provides static utilities for working with files, permissions, directories and attributes, walking trees, and more.