

UNIT 9

Nonblocking I/O

(3 Hours) — Network Programming (CAC355)

Please note

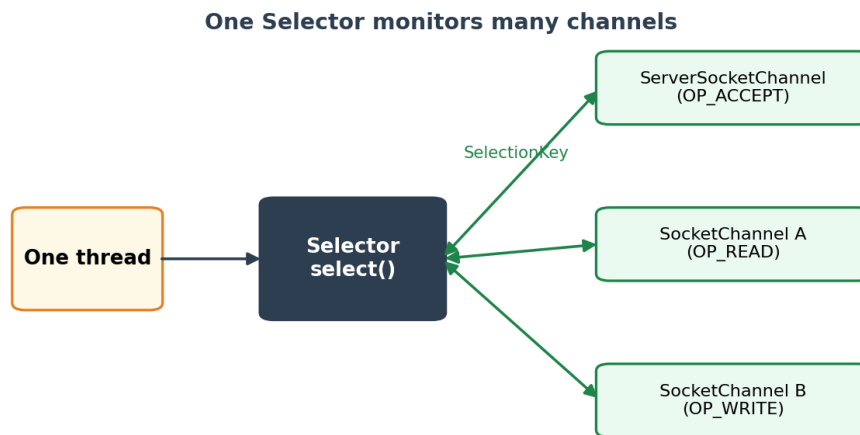
These notes may contain errors. They were generated automatically, so please verify important facts, API details and code against your textbook, the official Java documentation and your teacher's material before relying on them (especially for exams).

9.0 Why Nonblocking I/O?

In the classic **java.io** model each socket read or write **blocks** the calling thread until data is available. A server therefore needs one thread per client, which becomes expensive when thousands of clients are connected but only a few are active at any moment. **Nonblocking I/O (NIO)**, introduced in Java 1.4 in the package **java.nio** and **java.nio.channels**, lets a **single thread** watch many connections and act only on those that are ready.

Feature	Stream I/O (java.io / java.net)	NIO (java.nio)
Data unit	Byte or character streams	Buffers (blocks of data)
Connection object	Socket, ServerSocket	SocketChannel, ServerSocketChannel
Blocking behaviour	Always blocking	Blocking or nonblocking (configurable)
Threads for N clients	Usually N threads	One thread plus a Selector
Direction	Separate input and output streams	One channel is both readable and writable

NIO is built on three core abstractions: **Buffers** (hold the data), **Channels** (connect to the I/O source) and **Selectors** (tell us which channels are ready).



Channels are registered with the selector; `select()` reports which are ready

Figure 9.1: A single thread uses a Selector to serve many channels.

9.1 An Example Client and Server

The typical nonblocking design registers the server channel with a selector, then loops: call `select()`, iterate over the ready keys, and handle each according to its ready operation (accept, read or write).

9.1.1 Nonblocking Echo Server

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.Iterator;

public class NioEchoServer {
    public static void main(String[] args) throws IOException {
        Selector selector = Selector.open();
        ServerSocketChannel server = ServerSocketChannel.open();
        server.bind(new InetSocketAddress(9000));
        server.configureBlocking(false); // must be nonblocking
        server.register(selector, SelectionKey.OP_ACCEPT);

        while (true) {
            selector.select(); // blocks until something is ready
            Iterator<SelectionKey> it = selector.selectedKeys().iterator();
            while (it.hasNext()) {
                SelectionKey key = it.next();
                it.remove(); // IMPORTANT: remove handled key

                if (key.isAcceptable()) {
                    SocketChannel client = server.accept();
                    if (client == null) continue;
                    client.configureBlocking(false);
                    client.register(selector, SelectionKey.OP_READ,
                                   ByteBuffer.allocate(256));
                } else if (key.isReadable()) {
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer buf = (ByteBuffer) key.attachment();
                    int n = client.read(buf);
                    if (n == -1) { client.close(); continue; } // client closed
                    buf.flip();
                    client.write(buf); // echo back
                    buf.clear();
                }
            }
        }
    }
}
```

Listing 9.1: Echo server with one thread and a Selector

Note: This simple version assumes the write completes in one call. A production server would register `OP_WRITE` and keep any unsent bytes in the buffer (see Section 9.4).

9.1.2 Nonblocking Client

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;

public class NioClient {
    public static void main(String[] args) throws IOException {
        SocketChannel ch = SocketChannel.open();
        ch.configureBlocking(false);
        ch.connect(new InetSocketAddress("localhost", 9000));
        while (!ch.finishConnect()) { /* do other work or wait */ }

        ch.write(ByteBuffer.wrap("Hello NIO".getBytes(StandardCharsets.UTF_8)));

        ByteBuffer buf = ByteBuffer.allocate(256);
        while (ch.read(buf) == 0) { /* nothing yet */ }
        buf.flip();
        System.out.println(StandardCharsets.UTF_8.decode(buf));
        ch.close();
    }
}
```

Listing 9.2: Simple nonblocking client (busy-waits for clarity)

9.2 Buffers

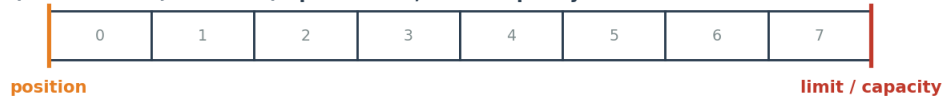
A **Buffer** is a fixed-size container for data of one primitive type. There is one subclass for each primitive type (**ByteBuffer**, **CharBuffer**, **ShortBuffer**, **IntBuffer**, **LongBuffer**, **FloatBuffer**, **DoubleBuffer**); there is no **BooleanBuffer**. **ByteBuffer** is the most important for networking because channels read and write bytes.

9.2.1 The Four Key Properties

Property	Meaning
capacity	Maximum number of elements; fixed when the buffer is created, never changes.
limit	Index of the first element that must not be read or written.
position	Index of the next element to read or write.
mark	A remembered position that <code>reset()</code> can return to (optional).

The invariant is always: $0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$.

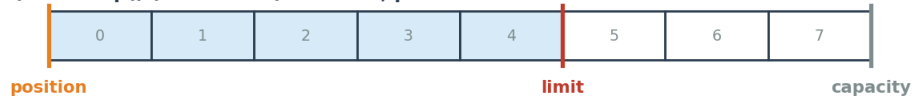
1) New buffer (allocate 8): position=0, limit=capacity=8



2) After put() of 5 bytes (write mode): position=5



3) After flip() (read mode): limit=5, position=0



`flip()` = `limit←position, position←0` | `clear()` resets for writing | `compact()` keeps unread bytes

Figure 9.2: position, limit and capacity before and after flip().

9.2.2 Creating Buffers

- **allocate(n)** – creates a new buffer with n elements on the Java heap, e.g. `ByteBuffer.allocate(1024)`.
- **allocateDirect(n)** – (ByteBuffer only) memory outside the heap; faster for I/O but costlier to create.
- **wrap(array)** – a buffer backed by an existing array; changes to one show up in the other.

```
ByteBuffer b1 = ByteBuffer.allocate(8);
ByteBuffer b2 = ByteBuffer.allocateDirect(1024);
byte[] data = "Hello".getBytes();
ByteBuffer b3 = ByteBuffer.wrap(data);           // backed by data[]
```

9.2.3 Filling and Draining

Filling means writing into the buffer with `put()`; **draining** means reading out with `get()`. Relative `put/get` use and advance the position; absolute versions take an index and leave the position unchanged. To switch from filling to draining call **flip()**.

Method	Effect
<code>put(x) / get()</code>	Relative: use position, then <code>position++</code> .
<code>put(i, x) / get(i)</code>	Absolute: use index i, position unchanged.
<code>flip()</code>	<code>limit = position; position = 0</code> ; mark discarded. Prepares for reading.
<code>clear()</code>	<code>position = 0; limit = capacity</code> . Prepares for writing (data is not erased).

Method	Effect
<code>rewind()</code>	<code>position = 0</code> ; limit unchanged. Re-read the same data.
<code>remaining()</code> / <code>hasRemaining()</code>	<code>limit - position</code> / whether <code>remaining() > 0</code> .

Example 9.1 – Fill, flip, drain

```
ByteBuffer buf = ByteBuffer.allocate(8); buf.put((byte)'N').put((byte)'I').put((byte)'O');
```

What are position and limit now, and after `buf.flip()`?

Solution

After three puts: `position = 3`, `limit = 8`, `capacity = 8`.

After `flip()`: `position = 0`, `limit = 3`. Three `get()` calls return N, I, O; then `hasRemaining()` is false.

Forgetting `flip()` is the most common NIO bug: `get()` would read the empty bytes after position 3.

9.2.4 Bulk Methods

Moving one element at a time is slow. Bulk methods transfer arrays or other buffers at once: **`put(byte[])`**, **`put(byte[], offset, length)`**, **`put(ByteBuffer)`**, **`get(byte[])`** and **`get(byte[], offset, length)`**. A bulk `get()` throws **`BufferUnderflowException`** if fewer bytes remain than requested, and a bulk `put()` throws **`BufferOverflowException`** if there is not enough room.

9.2.5 Data Conversion

`ByteBuffer` provides typed methods that convert between bytes and primitives: **`putInt`**, **`getInt`**, **`putLong`**, **`getLong`**, **`putDouble`**, **`getDouble`**, **`putChar`**, etc. The conversion respects the buffer's **byte order** (big-endian by default, matching network order); it can be changed with **`order(ByteOrder.LITTLE_ENDIAN)`**. Text is converted with a **`Charset`** using `encode()` and `decode()`.

```
ByteBuffer bb = ByteBuffer.allocate(12);
bb.putInt(42).putDouble(3.14);
bb.flip();
int i = bb.getInt();           // 42
double x = bb.getDouble();     // 3.14

ByteBuffer enc = StandardCharsets.UTF_8.encode("Hello"); // String -> bytes
CharBuffer dec = StandardCharsets.UTF_8.decode(enc);     // bytes -> chars
```

9.2.6 View Buffers

A **view buffer** is a different-typed window onto the same bytes of a `ByteBuffer`, created by `asIntBuffer()`, `asCharBuffer()`, `asLongBuffer()`, etc. It has its own position, limit and mark, but writes through the view change the underlying bytes (and vice versa). Its capacity is the number of whole elements that fit in the remaining bytes.

9.2.7 Compacting Buffers

When a read is only partly processed, the unread bytes must be kept while new data is added. `compact()` copies the bytes between position and limit to the start of the buffer, sets position to the number of bytes copied, and sets limit to capacity, so the buffer is ready for further `put()/read()` calls.

```
buf.flip();           // switch to reading
// ... consume some bytes with get() ...
buf.compact();        // move unread bytes to the front, ready to fill again
channel.read(buf);    // append new data after them
```

9.2.8 Duplicating and Slicing Buffers

Method	Result
<code>duplicate()</code>	New buffer sharing the same content; independent position, limit and mark (initially copied).
<code>slice()</code>	New buffer sharing a subsequence from the current position to the limit; its position is 0 and capacity = <code>remaining()</code> .
<code>asReadOnlyBuffer()</code>	Shares content but any attempt to modify throws <code>ReadOnlyBufferException</code> .

Note: Content is shared, not copied. Changing a byte through one buffer is visible through the others.

9.2.9 Marking and Resetting

`mark()` records the current position; a later `reset()` sets position back to that mark (throwing `InvalidMarkException` if no mark is set). Calling `flip()`, `rewind()` or `clear()` discards the mark.

9.2.10 Object Methods

Buffers override `equals()`, `hashCode()`, `compareTo()` and `toString()`. Two buffers are equal if they hold the same element type and the same *remaining* elements (from position to limit); capacity and position themselves are ignored. `toString()` prints the class, position, limit and capacity, e.g. `java.nio.HeapByteBuffer[pos=0 lim=3 cap=8]`. Because buffers are mutable, using them as hash-map keys is risky.

9.3 Channels

A **Channel** represents an open connection to an entity such as a file or a socket. Unlike streams, a channel is typically **bidirectional**, and data always moves between a channel and a **Buffer**. Channels can be put into **nonblocking mode** (`SelectableChannel` subclasses only), where `read/write/accept` return immediately.

Channel class	Purpose
<code>SocketChannel</code>	TCP client connection.
<code>ServerSocketChannel</code>	Listens for incoming TCP connections.
<code>DatagramChannel</code>	UDP (see Unit 10).
<code>FileChannel</code>	File I/O (not selectable).
<code>Asynchronous*Channel</code>	Completion-based I/O (Java 7+).

9.3.1 SocketChannel

- **`SocketChannel.open()`** creates an unconnected channel; **`open(SocketAddress)`** opens and connects (blocking).
- **`connect(addr)`** – in nonblocking mode returns false immediately; call **`finishConnect()`** until it returns true. **`isConnectionPending()`** tells whether a connect is in progress.
- **`read(ByteBuffer)`** returns the number of bytes read, **0** if none available (nonblocking), or **-1** at end of stream.
- **`write(ByteBuffer)`** returns bytes written; in nonblocking mode it may write fewer than remaining(), so loop or use `OP_WRITE`.
- Scattering read **`read(ByteBuffer[])`** and gathering write **`write(ByteBuffer[])`** move data to/from several buffers in one call.

9.3.2 ServerSocketChannel

Create with **`ServerSocketChannel.open()`** then **`bind(SocketAddress)`** (Java 7+; older code used `socket().bind()`). **`accept()`** returns a `SocketChannel`; in nonblocking mode it returns **null** when no connection is pending. The channel only supports `OP_ACCEPT` in a selector.

9.3.3 The Channels Class

java.nio.channels.Channels is a utility class that bridges the stream and channel worlds. **newChannel(InputStream/OutputStream)** wraps a stream as a channel; **newInputStream(ReadableByteChannel)**, **newOutputStream(WritableByteChannel)**, **newReader(...)** and **newWriter(...)** do the reverse. Streams obtained from a channel are blocking and, for a selectable channel, require it to be in blocking mode.

9.3.4 Asynchronous Channels

Java 7 (NIO.2) added **AsynchronousSocketChannel**, **AsynchronousServerSocketChannel** and **AsynchronousFileChannel**. Operations return immediately and report completion in one of two ways:

- **Future** style – e.g. `Future<Integer> f = channel.read(buf);` later call `f.get()`.
- **CompletionHandler** style – pass a handler with `completed(result, attachment)` and `failed(exc, attachment)` callbacks, which run on a thread pool (an **AsynchronousChannelGroup**).

```
AsynchronousServerSocketChannel srv =
    AsynchronousServerSocketChannel.open().bind(new InetSocketAddress(9000));
srv.accept(null, new CompletionHandler<AsynchronousSocketChannel, Void>() {
    public void completed(AsynchronousSocketChannel ch, Void att) {
        srv.accept(null, this);           // accept next client
        // ... ch.read(...) / ch.write(...) with more handlers ...
    }
    public void failed(Throwable t, Void att) { t.printStackTrace(); }
});
```

9.3.5 Socket Options

Since Java 7, channels support options through **setOption(SocketOption, value)** and **getOption()**, using constants in **StandardSocketOptions**: *SO_RCVBUF*, *SO_SNDBUF*, *SO_KEEPALIVE*, *SO_REUSEADDR*, *SO_LINGER*, *TCP_NODELAY*, *IP_TOS* (and *SO_BROADCAST*, *IP_MULTICAST_** for datagram channels). **supportedOptions()** lists those a channel supports.

```
SocketChannel ch = SocketChannel.open();
ch.setOption(StandardSocketOptions.TCP_NODELAY, true);
ch.setOption(StandardSocketOptions.SO_RCVBUF, 64 * 1024);
```

9.4 Readiness Selection

Readiness selection lets one thread learn which of many registered channels can proceed without blocking. The channel must first be set to nonblocking mode, then registered with a Selector using `channel.register(selector, ops[, attachment])`, which returns a **SelectionKey**.

9.4.1 The Selector Class

Method	Description
<code>Selector.open()</code>	Creates a selector.
<code>select()</code>	Blocks until at least one registered channel is ready (or <code>wakeup()/interrupt()</code>); returns number of updated keys.
<code>select(timeout)</code>	Same but waits at most <code>timeout</code> ms.
<code>selectNow()</code>	Non-blocking: returns immediately, possibly 0.
<code>selectedKeys()</code>	Set of keys whose channels are ready. You must remove keys you handle.
<code>keys()</code>	All registered keys (not just ready ones).
<code>wakeup()</code>	Makes a blocked <code>select()</code> return immediately.
<code>close()</code>	Closes the selector and cancels all its keys.

9.4.2 The SelectionKey Class

A **SelectionKey** represents the registration of one channel with one selector. It records the **interest set** (operations we care about), the **ready set** (operations that can proceed), the channel, the selector and an optional **attachment** object (commonly the per-connection buffer or state).

Constant	Value meaning	Valid for
<code>OP_ACCEPT</code>	A new connection can be accepted	<code>ServerSocketChannel</code>
<code>OP_CONNECT</code>	A connection attempt finished	<code>SocketChannel</code>
<code>OP_READ</code>	Data can be read	<code>SocketChannel</code> , <code>DatagramChannel</code>
<code>OP_WRITE</code>	Data can be written	<code>SocketChannel</code> , <code>DatagramChannel</code>

- **isAcceptable()**, **isConnectable()**, **isReadable()**, **isWritable()** test the ready set.
- **interestOps(int)** changes what we are interested in (e.g. add `OP_WRITE` only while output is pending, otherwise `select()` would spin because sockets are almost always writable).
- **attach(Object)** / **attachment()** store and retrieve per-connection data.
- **cancel()** deregisters the key; closing the channel also cancels it.

- Combine operations with bitwise OR: `OP_READ | OP_WRITE`.

Example 9.2 – Why must we call `it.remove()`?

Inside the loop over `selectedKeys()`, what happens if we forget `it.remove()`?

Solution

The selector never removes keys from the selected-key set itself. A key left in the set is seen as ready again on the next pass, even if the channel has no data, so the server would repeatedly process stale events (and `read()` may return 0 or `accept()` null).

Key Takeaways

- ✓ NIO = Buffers + Channels + Selectors; one thread can serve many connections.
- ✓ Buffer state is tracked by position, limit and capacity; `flip()` switches from writing to reading, `clear()/compact()` back again.
- ✓ `read()` returns `-1` at end of stream and 0 when nothing is available in nonblocking mode.
- ✓ Channels must be nonblocking before registering with a Selector.
- ✓ Always remove handled keys from `selectedKeys()`; use `interestOps` for `OP_WRITE` only when needed.
- ✓ Asynchronous channels (Java 7) use `Future` or `CompletionHandler` instead of a selector loop.

— End of Unit 9 —