Chapter 2

Introduction to NIO: New I/O

Advanced Topics in Java

Khalid Azim Mughal
khalid@ii.uib.no
http://www.ii.uib.no/~khalid/atij/

Version date: 2004-09-01

Overview

- Buffers
  - Byte Buffers
- Channels
  - File Channels
  - Sever Socket Channels
  - Socket Channels
- Selectors
  - Multiplexing I/O

Problem with Streams and Blocked I/O

- Wastage of CPU cycles because of blocked I/O
- Proliferation of "under-the-hood" objects that can impact garbage collection
- Proliferation of threads that can incur performance hit

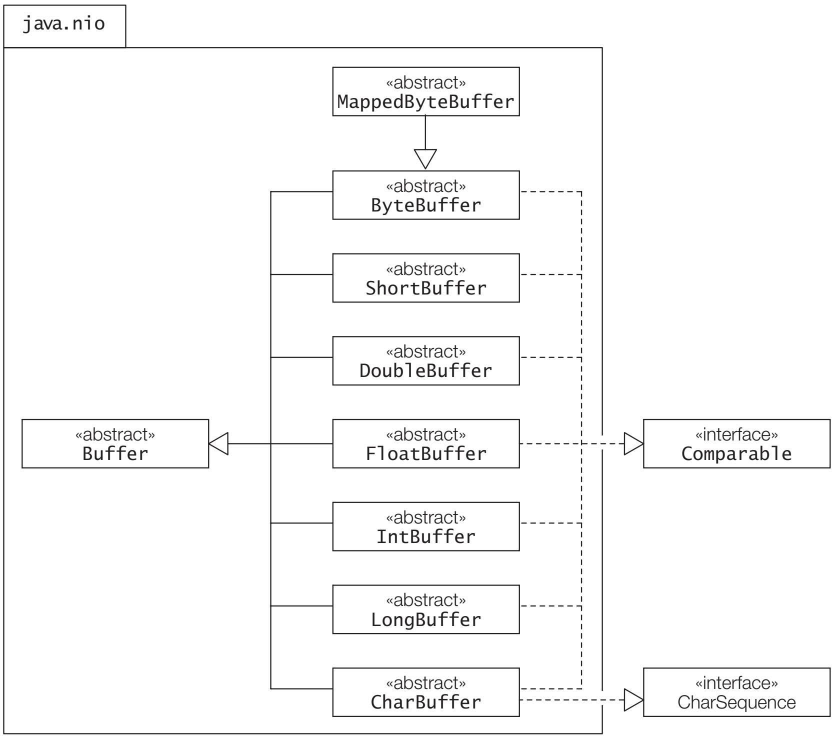
NEW I/O

- Provide support for high-performance and scalable I/O for files and sockets.
- Key features:
  - Handle data *chunk-wise* using a *buffer*-oriented model and not *byte-wise* using the *byte-stream*-oriented model.
  - Utilize operative system-level buffers and calls to do I/O

Buffers

- A *buffer* is a non-resizable, in-memory container of data values of a particular primitive type.
- All buffers are type specific, apart from the `ByteBuffer` which allows reading and writing of the other six primitive types.
- Note that channels only use *byte buffers* as source or destination for data.

Buffer Hierarchy: `java.nio`

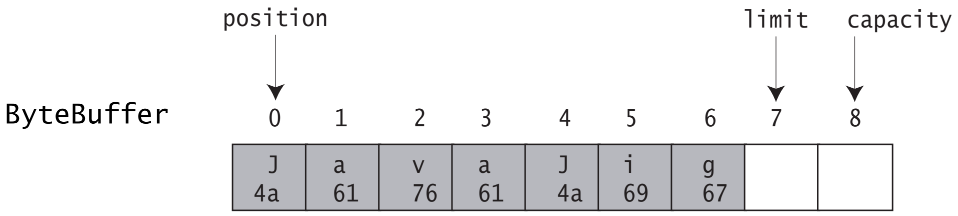


Buffer Attributes

- `capacity`: Maximum number of elements the buffer can accommodate.
- `limit`: Index of the first element that should *not* be read or written to, i.e. number of live data values in the buffer.
- `position`: Index of the next element where the data value should be read from or written to.
- `mark`: Remembers the index indicated by `position` when the method `mark()` is called. The `reset()` method sets `position` back to `mark`.

The following relationship always holds:

`0 <= mark <= position <= limit <= capacity`



`mark` *is undefined unless set by the mark() method*

Creating Byte Buffers

- By *wrapping* an existing array in a buffer, which allocates space and copies the array contents.

```
ByteBuffer buffer = ByteBuffer.wrap(byteArray);
```

- By *allocation*, which only allocates space for the buffer's content.

```
ByteBuffer buffer1 = ByteBuffer.allocate(9);
```

Filling and Draining Byte Buffers

- A write operation *fills* data in the buffer, and a read operation *drains* data from the buffer.
- A *relative* read/write operation updates `position`, whereas an *absolute* read/write operation does not.
- Note that most operations return a reference to the buffer so that method calls on the buffer can be *chained*.
- Reading and writing *single bytes*:

```
byte get()                          // Relative read
byte get(int index)                 // Absolute read
ByteBuffer put(byte b)              // Relative write
ByteBuffer put(byte b, int index)   // Absolute write
```

- Reading and writing *contiguous sequences of bytes* (*relative bulk moves*):

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

- The method `remaining()` returns the number of elements between the current position and the limit.

Flipping a Byte Buffer

- A buffer which is filled, must be *flipped*, before it is drained.
- The method `flip()` prepares the buffer for this purpose.
- Flipping sets the limit to current position, and the position to 0.

Clearing a Byte Buffer

- A buffer that has been drained, can be *refilled* by first *clearing* the buffer.
- The method `clear()` resets the buffer for this purpose.
- Clearing sets the limit to capacity, and the position to 0.

Direct and Nondirect Byte Buffers

- A *direct* buffer is allocated contiguous memory and accessed using native access methods.
- A *nondirect* buffer is managed entirely by the JVM.
- Direct byte buffers are recommended when interacting with channels.

Example: Using Byte Buffers

```
import java.nio.*;

public class Buffers {

    public static void main(String[] args) {

      // Create from array.
      byte[] byteArray = {(byte)'J', (byte)'a', (byte)'v', (byte)'a',
                          (byte)'J', (byte)'i', (byte)'g'};
      ByteBuffer buffer = ByteBuffer.wrap(byteArray);
      System.out.println("After wrapping:");
      printBufferInfo(buffer);

      // Create by allocation.
      ByteBuffer buffer1 = ByteBuffer.allocate(9);
      System.out.println("After allocation:");
      printBufferInfo(buffer1);
```

```
   // Filling the buffer
   fillBuffer(buffer1);

   // Flip before draining
   buffer1.flip();
   System.out.println("After flipping:");
   printBufferInfo(buffer1);

   // Drain the buffer
   drainBuffer(buffer1);
  }
  public static void printBufferInfo(ByteBuffer buffer) {
    System.out.print("Position: " + buffer.position());
    System.out.print("\tLimit: "    + buffer.limit());
    System.out.println("\tCapacity: " + buffer.capacity());
  }
  public static void fillBuffer(ByteBuffer buffer) {
    buffer.put((byte)'J').put((byte)'a').put((byte)'v').
        put((byte)'a').put((byte)'J').put((byte)'i').
        put((byte)'g');
    System.out.println("After filling:");
    printBufferInfo(buffer);
  }
```

```
  public static void drainBuffer(ByteBuffer buffer) {
    System.out.print("Contents: ");
    int count = buffer.remaining();
    for (int i = 0; i < count; i++)
      System.out.print((char) buffer.get());
    System.out.println();
    System.out.println("After draining:");
    printBufferInfo(buffer);
  }
}
```

Output from running the program:

```
After wrapping:
Position: 0 Limit: 7  Capacity: 7
After allocation:
Position: 0 Limit: 9  Capacity: 9
After filling:
Position: 7 Limit: 9  Capacity: 9
After flipping:
Position: 0 Limit: 7  Capacity: 9
Contents: JavaJig
After draining:
Position: 7 Limit: 7  Capacity: 9
```

Encoding Character Buffers and Decoding Byte Buffers

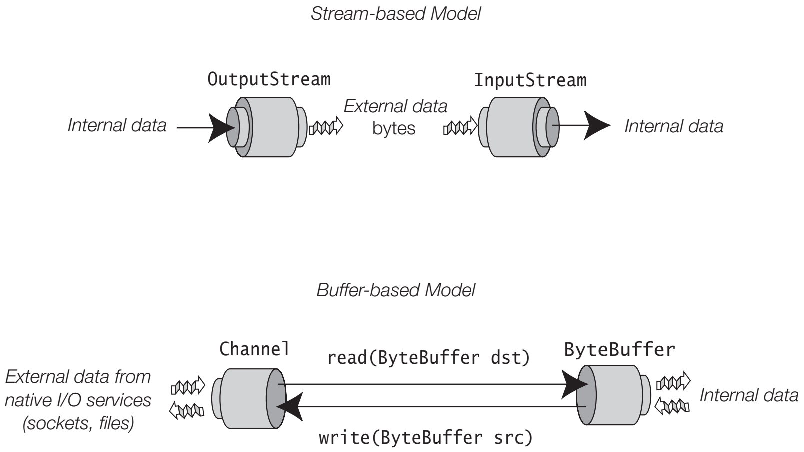
- Example: `EncodingDecoding.java`

```
// Create a byte and a character buffer.
ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
CharBuffer charBuffer = CharBuffer.allocate(1024);

// Setup a decoder and an encoder -- assume ASCII charset.
Charset charset = Charset.forName("US-ASCII");
CharsetDecoder decoder = charset.newDecoder();
CharsetEncoder encoder = charset.newEncoder();

// ENCODING
// Fill the char buffer.
charBuffer.put((new Date().toString()+ "\r\n"));
charBuffer.flip();

// Encode the char buffer.
encoder.encode(charBuffer, buffer, false);
```

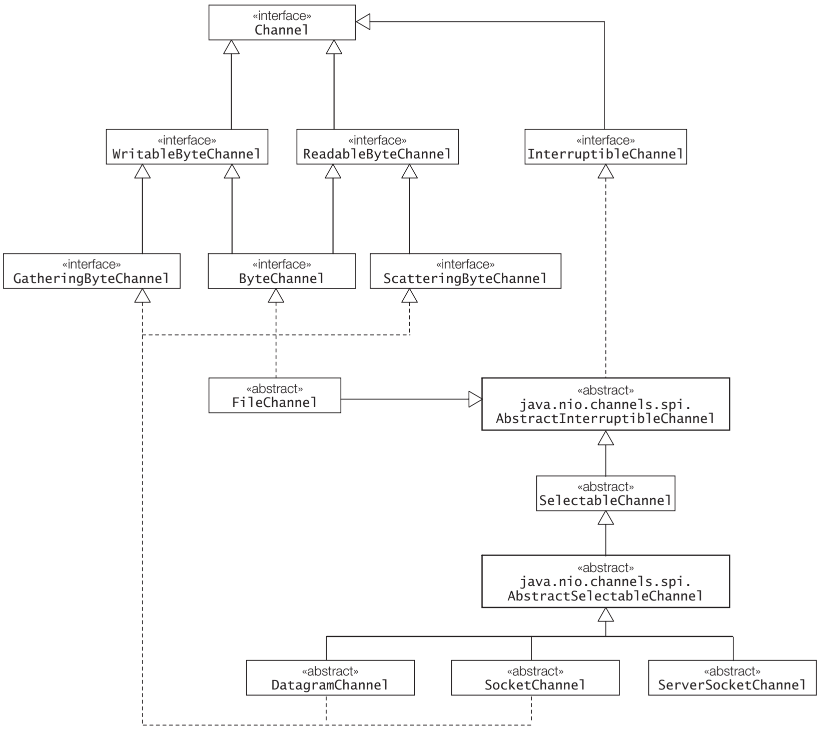
```
// DECODING
// Prepare to decode the byte buffer
buffer.flip();
charBuffer.clear();

// Decode the byte buffer.
decoder.decode(buffer, charBuffer, false);

// Write the charbuffer to console
charBuffer.flip();
System.out.print(charBuffer);
```

Channels

- A *channel* is a pipeline for transporting data efficiently between byte buffers and I/O services (files and sockets).
- Channels can operate in *blocking* and *nonblocking mode.*
  - A *blocking* operation does not return until it completes or is timed-out or is interrupted, i.e. the invoking thread can be put to sleep.
  - A *nonblocking* operation returns immediately with a status indicating either that it completed or that nothing was done, i.e. the invoking thread is never put to sleep.



Channels Hierarchy: `java.nio.channels`


«interface» Channel
«interface» WritableByteChannel
«interface» ReadableByteChannel
«interface» InterruptibleChannel
«interface» GatheringByteChannel
«interface» ByteChannel
«interface» ScatteringByteChannel
«abstract» FileChannel
«abstract» java.nio.channels.spi. AbstractInterruptibleChannel
«abstract» SelectableChannel
«abstract» java.nio.channels.spi. AbstractSelectableChannel
«abstract» DatagramChannel
«abstract» SocketChannel
«abstract» ServerSocketChannel

File Channels

- A *file channel* is a pipeline for transporting data efficiently between byte buffers and files.
  - A file channel only operates in *blocking mode*.
- The classes `FileInputStream`, `FileOutputStream`, and `RandomAccessFile` all provide a `FileChannel` which can be used to transfer data between a byte buffer and a file.
  - The stream classes provide the `getChannel()` method to obtain the file channel.
- Given that `channel` is a `FileChannel` and `buffer` is a `ByteBuffer`.
  - Read data *from* the channel *to* the buffer using the `read()` method:

    ```
    int byteCount = channel.read(buffer); // Return value >= 0 or -1 (EOF)
    ```

  - Write data *to* the channel *from* the buffer using the `write()` method:

    ```
    int byteCount = channel.write(buffer); // Return value >= 0
    ```

  These methods can do a partial transfer, meaning that it might to necessary to call them repeated to complete the transfer.
- A file channel should be closed when no longer needed (`close()` method). This also closes the underlying I/O service.

Example: Copy a File

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class FileCopy {
  public static void main (String[] argv) throws IOException {
    ...
    // Get the file names from the command line.
    String srcFileName = argv[0];
    String dstFileName = argv[1];

    // Open files for I/O.
    FileInputStream  srcFile = new FileInputStream(srcFileName);
    FileOutputStream dstFile = new FileOutputStream(dstFileName);

    // Obtain file channels to the files.
    FileChannel source      = srcFile.getChannel();
    FileChannel destination = dstFile.getChannel();

    // Do the copying.
    copy(source, destination);
```

```
    // Close the channels.
    source.close();
    destination.close();
  }

  private static void copy(FileChannel source, FileChannel destination)
                           throws IOException {
    // Create a direct byte buffer to read source file contents.
    ByteBuffer buffer = ByteBuffer.allocateDirect(16 * 1024);
    // Read from source file into the byte buffer using the source file channel.
    while (source.read(buffer) != -1) { // EOF?
      // Prepare to drain the buffer
      buffer.flip();
      // Drain the buffer using the destination file channel
      while (buffer.hasRemaining()) {
        destination.write(buffer);
      }
      // Clear the buffer for reuse
      buffer.clear();
    }
  }
}
```

Socket Channels

- A *socket channel* is a pipeline for transporting data efficiently between byte buffers and socket connections.
  - A socket channel can operate both in *blocking* and *nonblocking mode*.
- Socket channels are *selectable*.
  - They can be used in conjunction with *selectors* to provide *multiplexed I/O* (discussed later).

Socket Channel Class	Peer Socket Class	Channel Functionality
`DatagramChannel`	`DatagramSocket`	Read and write to byte buffers
`SocketChannel`	`Socket`	Read and write to byte buffers
`ServerSocketChannel`	`ServerSocket`	Only handle connections

- A socket channel can be obtained from a socket using the `getChannel()` method.
- A socket can be obtained from a socket channel using the `socket()` method.

Example: Server Socket Channel

- Source file: `ServerUsingChannel.java`.
- The server does nonblocking handling of clients.
- It sends a canned message to the current client before handling a new connection.

```
...
public static final String GREETING =
      "Hello, Earthling. I have bad news for you.\r\n";
...
ByteBuffer buffer = ByteBuffer.wrap(GREETING.getBytes());
```

- Procedure for setting up a nonblocking server socket channel:

```
// Obtain a server socket channel.
ServerSocketChannel ssc = ServerSocketChannel.open();

// Bind to port. (Have to do this via peer socket.)
ServerSocket ss = ssc.socket();
InetSocketAddress addr = new InetSocketAddress(port);
ss.bind(addr);

// Make the server nonblocking.
ssc.configureBlocking(false);
```

- Handling connections with a nonblocking server socket channel:

```
while (true) {
  System.out.println("Checking for a connection");
  // Check if there is a client.
  SocketChannel sc = ssc.accept(); // Nonblocking
  if (sc != null) { // Any connection?
    System.out.println("Incoming connection from: "
      + sc.socket().getRemoteSocketAddress());

    // Rewind to reuse the buffer content.
    buffer.rewind();
    // Write to the channel.
    while (buffer.hasRemaining()) {
      sc.write(buffer);
    }
    // Close the channel.
    sc.close();

  } else { // No connection, take a nap.
    Thread.sleep(2000);
  }
}
```

Example: Client-side Socket Channel

- Source file: `ClientUsingChannel.java`.
- The client makes a nonblocking connection to the server to retrieve a canned message.

```
// Create a nonblocking socket channel
SocketChannel sc = SocketChannel.open();
sc.configureBlocking(false);

System.out.println("Making connection");
InetSocketAddress addr = new InetSocketAddress(remoteHost, remotePort);
sc.connect(addr);    // Nonblocking
while (!sc.finishConnect()) { // Concurrent connection
  System.out.println("I am waiting ...");
}
System.out.println("Connection established");
// Read data from channel.
readFromChannel(sc);

// Close the channel
sc.close();
```

- Retrieving the canned message from the server:

```
private static void readFromChannel(SocketChannel sc)
                    throws IOException {
  // Create a buffer.
  ByteBuffer buffer = ByteBuffer.allocate(10*1024);

  // Read data from the channel and write to console.
  while (sc.read(buffer) != -1) {
    buffer.flip();
    int count = buffer.remaining();
    for(int i = 0; i < count; i++)
        System.out.print((char) buffer.get());
    buffer.clear();
  }
}
```

Instead of writing byte-wise to the console, we can write in bulk using a channel which is connected to the console.
See the source code file `ClientUsingChannel.java` for an example.

Multiplexing I/O using Selectors

- Multiplexing makes it possible for a single thread to efficiently manage many I/O channels.
- *Readiness selection* enables multiplexing I/O, and involves using the following classes:
  - `Selector:` A selector monitors selectable channels that are registered with it.
  - `SelectableChannel`: A channel that a selector can monitor for I/O activity. All socket channels are *selectable*.
  - `SelectionKey`: A selection key encapsulates the relationship between a specific selectable channel and a specific selector.
- Programming for readiness selection:

1. Register selectable channels with a selector for specific I/O activity.
2. Invoke the `select()` method on the selector.
3. Each invocation results in a set of selected keys.
4. Iterate through the selected key set, handing the I/O activity on the channel identified by a key, and removing the key from the set afterwards.
5. Repeat steps 2, 3, and 4 as necessary.

Example: Multiplexing Server

- Implementing a simple server which listens for incoming connections (See source file `MultiplexingServer.java`).
- A single `Selector` object is used to listen to the server socket for accepting new connections.
- All client socket channels are registered with the selector to monitor incoming data.
- The method `doMultiplexing()` method in class `MultiplexingServer` implements readiness selection.
  - Create and bind a nonblocking server socket channel to a given port:

```
// Allocate an unbound server socket channel
ServerSocketChannel serverChannel = ServerSocketChannel.open();

// Get the associated ServerSocket to bind it with
ServerSocket serverSocket = serverChannel.socket();

// Set the port which the server channel will listen to
serverSocket.bind(new InetSocketAddress(port));

// Set nonblocking mode for the listening socket
serverChannel.configureBlocking(false);
```

– Create a selector and register the server socket channel with the selector to accept connections:

```
// Create a new Selector for use.
Selector selector = Selector.open();

// Register the ServerSocketChannel with the selector to
// accept connections.
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

– *Polling loop* to repeatedly invoke the `select()` method on the selector:

```
while (true) {
    System.out.println("Listening on port " + port);
    // Monitor registered channels
    int n = selector.select();
    if (n == 0) {
        continue;   // Continue to loop if no I/O activity.
    }
    // Handle I/O activity given by the selected key set.
    ...
}
```

– Iterate over the selected key set resulting from the `select()` method invocation:

```
// Get an iterator over the set of selected keys.
Iterator it = selector.selectedKeys().iterator();

// Look at each key in the selected set
while (it.hasNext()) {
    SelectionKey key = (SelectionKey) it.next();
    // Is a new connection coming in?
    if (key.isAcceptable()) {
        System.out.println("Setting up new connection");
        ServerSocketChannel server =
            (ServerSocketChannel) key.channel();
        SocketChannel channel = server.accept();

        // Set the new channel nonblocking.
        channel.configureBlocking(false);

        // Register it with the selector.
        channel.register(selector, SelectionKey.OP_READ);
        System.out.println("New connection established" +
                           " and registered");
    }
```

```
        // Is there data to read on this channel?
        if (key.isReadable()) {
            replyClient((SocketChannel) key.channel());
        }

        // Remember to remove key from selected set.
        it.remove();
    }
```

– The `replyClient()` method echoes back the data from the client:

```
// Read from the socket channel.
buffer.clear();
int count = socketChannel.read(buffer);
if (count <= 0) {
    // Close channel on EOF or if there is no data,
    // which also invalidates the key.
    socketChannel.close();
    return;
}
// Echo back to client.
buffer.flip();
socketChannel.write(buffer);
```

Example: Multiplexing Client

- Implementing a simple client which uses a `Selector` object to listen for data from the server socket (See source file `ClientForMultiplexServer.java`).
- The method `goToWork()` method in class `ClientForMultiplexServer` implements readiness selection for the client.
  - Note that the client send an initial message (the current time) to the server.

```
public void goToWork() throws IOException {
  // Create a nonblocking socket channel and connect to server.
  SocketChannel sc = SocketChannel.open();
  sc.configureBlocking(false);
  InetSocketAddress addr = new InetSocketAddress(remoteHost,
                                                 remotePort);
  sc.connect(addr);    // Nonblocking
  while (!sc.finishConnect()) {
    System.out.println("I am waiting ...");
  }

  // Send initial message to server.
  buffer.put((new Date().toString()+ "\r\n").getBytes());
  buffer.flip();
  sc.write(buffer);
```

```
    // Create a new selector for use.
    Selector selector = Selector.open();

    // Register the socket channel with the selector.
    sc.register(selector, SelectionKey.OP_READ);

    // Polling loop
    while (true) {
        System.out.println("Listening for server on port " +
                           remotePort);

        // Monitor the registered channel.
        int n = selector.select();
        if (n == 0) {
            continue;   // Continue to loop if no I/O activity.
        }
```

```
        // Get an iterator over the set of selected keys.
        Iterator it = selector.selectedKeys().iterator();

        // Look at each key in the selected set.
        while (it.hasNext()) {
            // Get key from the selected set.
            SelectionKey key = (SelectionKey) it.next();

            // Remove key from selected set.
            it.remove();

            // Get the socket channel from the key.
            SocketChannel keyChannel = (SocketChannel) key.channel();

            // Is there data to read on this channel?
            if (key.isReadable()) {
                replyServer(keyChannel);
            }
        }
    }
  }
```

- The method `replyServer()` method in class `ClientForMultiplexServer` reads data from the server and sends a reply (current time).

```
private void replyServer(SocketChannel socketChannel) throws IOException {
  System.out.println("Replying on " + socketChannel);
  // Read from server.
  buffer.clear();
  int count = socketChannel.read(buffer);
  if (count <= 0) {
      // Close channel on EOF or if there is no data,
      // which also invalidates the key.
      socketChannel.close();
      return;
  }
  // Print on console.
  buffer.flip();
  outConsole.write(buffer);
  // Send new message.
  buffer.clear();
  buffer.put((new Date().toString()+ "\r\n").getBytes());
  buffer.flip();
  socketChannel.write(buffer);
}
```

Running the Multiplexing Server and Client

- Output from the server:

```
>java MultiplexingServer
Listening on port 1234
Setting up new connection
New connection established and registered
Listening on port 1234
Reading on java.nio.channels.SocketChannel[connected local=/127.0.0.1:1234
remote=/127.0.0.1:2704]
Listening on port 1234
...
```

- Output from the client:

```
>java ClientForMultiplexServer
Listening for server on port 1234
Replying on java.nio.channels.SocketChannel[connected local=/127.0.0.1:2730
remote=localhost/127.0.0.1:1234]
Sat Nov 08 17:15:15 CET 2003
Listening for server on port 1234
...
Replying on java.nio.channels.SocketChannel[connected local=/127.0.0.1:2730
remote=localhost/127.0.0.1:1234]
Sat Nov 08 17:15:52 CET 2003
```