JAVA NIO PROGRAMMING COOKBOOK

Hot Recipes for the Java NIO Library

JAVA CODE GEEKS

Java Code Geeks

Java 2 Java Developers Resource Center

Java NIO Programming Cookbook

Contents

1 Java Nio Tutorial for Beginners 1

1.1 Introduction . . . 1

1.2 Java NIO . . . 2

1.2.1 Buffers . . . 2

1.2.1.1 Example usage of a ByteBuffer . . . 3

1.2.2 Channels . . . 4

1.2.2.1 Example usage of a FileChannel . . . 4

1.2.3 Charsets . . . 4

1.2.4 Selectors . . . 5

1.3 Summary . . . 7

1.4 Working with example source code . . . 7

1.5 Download the source code . . . 7

2 Java Nio SSL Example 8

2.1 Definition of Secure Sockets Layer Protocol (SSL) . . . 8

2.2 The SSL Engine API . . . 8

2.2.1 Lifecycle . . . 8

2.2.2 SSL Handshake . . . 11

2.3 Nio SSL Example . . . 11

2.3.1 Main class . . . 11

2.3.2 NioSSLProvider class . . . 14

2.3.3 SSLProvider class . . . 16

2.4 Download Java Source Code . . . 18

3 Java Nio Socket Example 19

3.1 Standard Java sockets . . . 19

3.2 Nonblocking SocketChannel with java.nio . . . 20

3.3 Example . . . 20

3.3.1 The Server code . . . 20

3.3.2 The Client code . . . 24

3.3.3 The output . . . 25

3.4 Download Java Source Code . . . 25

4 Java Nio Write File Example 26

4.1 Introduction to the NIO API . . . 26
4.2 Creating a NIO Path . . . 26
4.3 Writing files with the NIO API . . . 26
4.3.1 Using NIO API with write() . . . 27
4.3.2 Using NIO API with newBufferedWriter() . . . 27
4.3.3 Using NIO API to copy a file with an Outputstream . . . 27
4.4 Summary . . . 27
4.5 Download The Source Code . . . 28

5 Java Nio Heartbeat Example 29

5.1 Introduction . . . 29
5.2 Technologies used . . . 30
5.3 Overview . . . 30
5.3.1 DatagramChannel . . . 31
5.3.2 MulticastChannel . . . 31
5.4 Multicaster . . . 31
5.5 Subscriber . . . 33
5.6 Running the program . . . 35
5.7 Summary . . . 36
5.8 Download the source code . . . 37

6 Java Nio Large File Transfer Tutorial 38

6.1 Introduction . . . 38
6.2 Technologies used . . . 38
6.3 FileChannel . . . 38
6.4 Background . . . 39
6.5 Program . . . 39
6.5.1 Local copy . . . 39
6.5.2 Remote copy . . . 40
6.5.2.1 Asynchronous large file transfer . . . 43
6.6 Running the program . . . 47
6.7 Summary . . . 48
6.8 Download the source code . . . 48

7 Java Nio Asynchronous Channels Tutorial 49

7.1 Introduction 49
7.2 Technologies used 50
7.3 API Interaction 50
7.4 AsynchronousChannel 50
7.5 AsynchronousByteChannel 51
7.6 AsynchronousFileChannel 51
7.6.1 AsynchronousFileChannel Exceptions 52
7.7 AsynchronousServerSocketChannel 55
7.7.1 AsynchronousServerSocketChannel Exceptions 55
7.8 AsynchronousSocketChannel 56
7.8.1 AsynchronousSocketChannel Exceptions 56
7.9 Summary 58
7.10 Download the source code 58

8 Java Nio Echo Server Tutorial 59

8.1 Introduction 59
8.2 Technologies used 59
8.3 Overview 59
8.4 The EchoServer 59
8.4.1 ChannelWriter 60
8.4.2 Client 60
8.4.3 Server 61
8.5 Example code 64
8.6 Summary 64
8.7 Download the source code 65

9 Java Nio ByteBuffer Example 66

9.1 Introduction 66
9.2 Technologies used 67
9.3 Overview 67
9.4 Test cases 67
9.5 Summary 72
9.6 Download the source code 73

Copyright (c) Exelixis Media P.C., 2017

All rights reserved. Without limiting the rights under
copyright reserved above, no part of this publication
may be reproduced, stored or introduced into a retrieval system, or
transmitted, in any form or by any means (electronic, mechanical,
photocopying, recording or otherwise), without the prior written
permission of the copyright owner.

Preface

java.nio (NIO stands for non-blocking I/O) is a collection of Java programming language APIs that offer features for intensive I/O operations. It was introduced with the J2SE 1.4 release of Java by Sun Microsystems to complement an existing standard I/O.

The APIs of NIO were designed to provide access to the low-level I/O operations of modern operating systems. Although the APIs are themselves relatively high-level, the intent is to facilitate an implementation that can directly use the most efficient operations of the underlying platform. (Source: https://en.wikipedia.org/wiki/New_I/O_(Java))

In this book, we provide a series of tutorials on Java NIO examples that will help you kick-start your own projects.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at https://www.javacodegeeks.com/

Chapter 1

Java Nio Tutorial for Beginners

This article is a beginners tutorial on Java NIO (New IO). We will take a high level look at this API which provides an alternative to Java IO. The Java NIO API can be viewed here. The example code demonstrates use of the core abstractions in this topic.

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or linux)

1.1 Introduction

Since Java 1.4 the Java NIO API has provided an alternate method of dealing with IO operations. Why did we need an alternate method for doing IO? As time progresses new problem sets arrive and new approaches to solving these problems are thought of. To understand the need for an alternate means of IO handling one should probably understand the core differences between the two approaches.

IO	NIO
Core differences:	**Core differences:**
Stream oriented processing	Uses buffers
Blocking in processing	Non blocking in processing
Good for:	**Good for:**
High data volume with low simultaneous open file descriptor counts (eg: less client connections with more data chunks per connection)	Less data volume with high simultaneous open file descriptor counts (eg: More connections with smaller / infrequent "chunks" of data)

Figure 1.1: Differences Between IO and NIO

NIO puts us in a position to make more judicious use of server / machine resources. By bringing us closer to the metal with an intelligent selection of abstractions we are able to better apply finite server resources to meet the increasing demands of modern day scale.

1.2 Java NIO

A quick glance at the summary of the Java NIO API reveals to us the core abstractions one should be familiar with when working with Java NIO. These are:

- Buffers : A container to hold data for the purposes of reading and or writing.
- Channels : An abstraction for dealing with an open connection to some component that is performing some kind of IO operation at a hardware level.
- Charsets : Contains charsets, decoders and encoders for translating between bytes and unicode.
- Selectors : A means to work with multiple channels via one abstraction.

1.2.1 Buffers

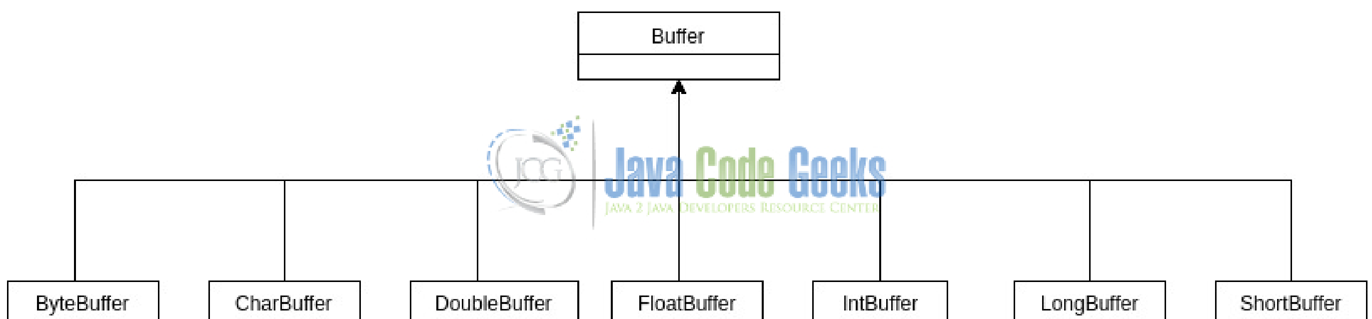

Figure 1.2: Shows the Buffer class hierarchy

A Buffer is a container for a fixed size of data of a specific primitive type (char, byte, int, long, float etc). A Buffer has content, a position, a limit and capacity. It can flip, rewind, mark and reset its position reinforcing the core differences between NIO and IO (buffer vs stream).

- Capacity = number of elements it contains.
- Limit = index of element that must not be read or written.
- Position = index of next element to read or write.
- Flip = invert position and limit when toggling the IO operation on a Buffer. (eg: write out to console what we just read from a Channel into the Buffer).
- Rewind = sets position to 0 and leaves limit unchanged in order to re-read the Buffer.
- Mark = bookmarks a position in the Buffer.
- Reset = resets the position to the previous mark.

What does all that mean? Well basically we put content into a Buffer (either read it from a Channel or put it directly into the Buffer with the intent to write it to a Channel).

We then advance the cursor through the content of the Buffer as we read or write. We flip a Buffer to change our IO operation on the Buffer (ie: go from reading to writing).

The capacity represents the total capacity the Buffer can hold with regard to content. The actual metric used for measurement depends on the type of the Buffer. (eg: CharBuffer capacity measured in characters and ByteBuffer capacity measured in Bytes).

1.2.1.1 Example usage of a ByteBuffer

`Reading from Channel into ByteBuffer`

```
...
final ByteBuffer buffer = createBuffer();
while (fileChannel.read(buffer) != -1) {
    contents.append(new String(buffer.array()));
    buffer.clear();
}
...
private ByteBuffer createBuffer() {
    return ByteBuffer.allocate(BYTE_BUFFER_LENGTH);
}
...
```

- line 2: A ByteBuffer is created with a defined capacity. (BYTE_BUFFER_LENGTH)
- line 3: Data is read from the specified FileChannel into the ByteBuffer.
- line 4: The ByteBuffer's current contents are added to the StringBuilder. This is done via convenience method `array()` as a result of the way the ByteBuffer was created in the example (via `allocate()`).
- line 5: The ByteBuffer is cleared to prepare for reading more data from the channel, this will set the position cursor back to 0 and allow contents to be read from the FileChannel back into the ByteBuffer repeating the process until no more data is available.

Alternate method for reading from Channel into ByteBuffer

```
...
buffer.flip();
if (buffer.hasRemaining()) {
    byte [] src = new byte[buffer.limit()];
    buffer.get(src);
    contents.append(new String(src));
}
....
```

- line 2: Invert the position and limit of the Buffer to retrieve what has been read from the Channel.
- line 3: Ensure there is something to read, ie: The difference between limit and position is > 0.
- line 4: Create a byte array to be the size of the data in the Buffer.
- line 5: Retrieve the contents of the Buffer into the byte array.
- line 6: Create a String array from the contents of the byte array.

It is important to also note that the instantiation of a new String to hold the bytes implicitly uses the default Charset to decode the bytes from their byte values to their corresponding unicode characters. If the default Charset was not what we were looking for, then instantiating a new String with the appropriate Charset would be required.

1.2.2 Channels

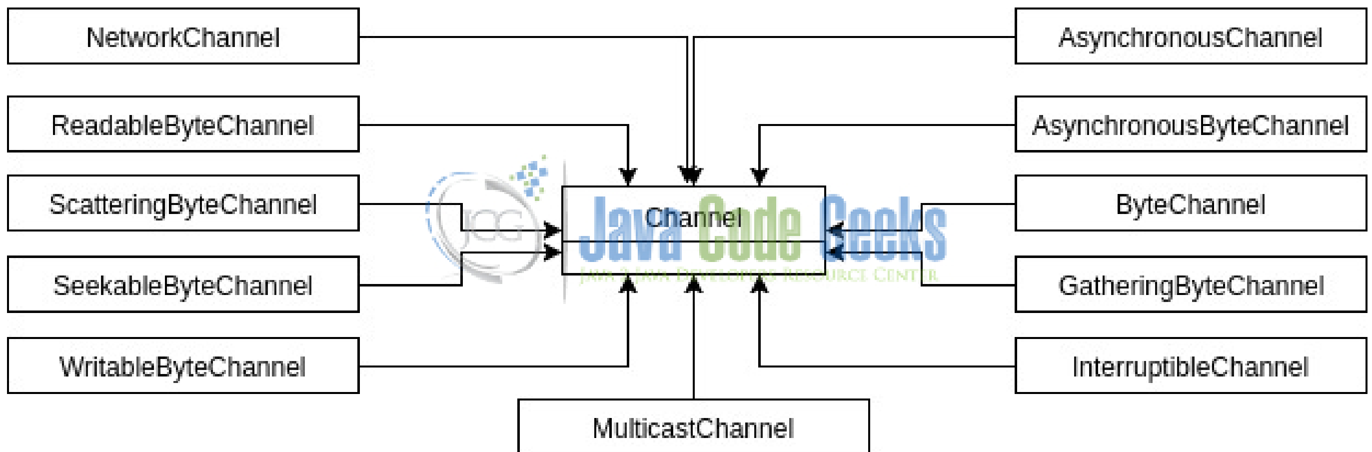

Figure 1.3: Interface hierarchy of Channel

A Channel is a proxy (open connection proxy) to a component that is responsible for native IO (file or network socket). By acting as a proxy to some native IO component we are able to write and / or read from a Channel. Some Channel implementations allow us to put them into non-blocking mode allowing read and write operations to be non-blocking. The same Channel can be used for both reading and writing.

A Channel is open upon creation and remains that way until it is closed.

1.2.2.1 Example usage of a FileChannel

Creating a FileChannel

```
...

final File file = new File(FileChannelReadExample.class.getClassLoader().getResource(path). ↩
    getFile());
return fileOperation == FileOperation.READ ? new FileInputStream(file).getChannel() :
                        new FileOutputStream(file).getChannel();
...
```

- line 3: Create a File Object
- line 4: Depending on the type of File operation (read or write) we create the necessary Stream and get the Channel from the Stream.

1.2.3 Charsets

A Charset is a mapping between 16 bit unicode characters and bytes. Charsets work with decoders and encoders which facilitate the adaption from bytes to characters and vice versa.

- Encoding: The process of transforming a sequence of characters into bytes
- Decoding: The process of transforming bytes into character buffers.

Charset provides other utility methods for looking up a Charset by name, creating coders (encoder or decoders) and getting the default Charset. Typically when one works with ByteBuffer and String as is the case in the example, the default Charset is what we would normally use if we do not explicitly specify one. This would suffice most of the time.

Charset usage

```
...
final Charset defaultCharset = Charset.defaultCharset();
final String text = "Lorem ipsum";

final ByteBuffer bufferA = ByteBuffer.wrap(text.getBytes());
final ByteBuffer bufferB = defaultCharset.encode(text);

final String a = new String(bufferA.array());
final CharBuffer charBufferB = defaultCharset.decode(bufferB);

System.out.println(a);
System.out.println(new String(charBufferB.array()));
...
```

- line 2: The default Charset is retrieved.
- line 5: The sample text is wrapped in a ByteBuffer. The default Charset is used implicitly when encoding the characters into bytes.
- line 6: The sample text is encoded explicitly using the default Charset encoder.
- line 8: A String is created using the default Charset decoder implicitly .
- line 9: A Character Buffer (ultimately a String) is created using the default Charset decoder explicitly.

1.2.4 Selectors

Selectors as the name implies, select from multiple SelectableChannel types and notify our program when IO has happened on one of those channels. It is important to note that during the registration process (registering a SelectableChannel with a Selector) we declare the IO events we are interested in, termed the "interest set" This can be:

- Connect
- Accept
- Read
- Write

With this proxy in place and the added benefit of setting those SelectableChannel types into non-blocking mode we are able to multiplex over said channels in a very efficient way, typically with very few threads, even as little as one.

Selector usage with SelectableChannel

```
try (final Selector selector = Selector.open();
     final ServerSocketChannel serverSocket = ServerSocketChannel.open();) {
    final InetSocketAddress hostAddress =
          new InetSocketAddress(Constants.HOST, Constants.PORT);
    serverSocket.bind(hostAddress);
    serverSocket.configureBlocking(false);
    serverSocket.register(selector, serverSocket.validOps(), null);

    while (true) {
```

```
        final int numSelectedKeys = selector.select();
        if (numSelectedKeys > 0) {
            handleSelectionKeys(selector.selectedKeys(), serverSocket);
        }
    }
}
```

- line 1: We create a Selector using the systems default SelectorProvider.
- line 2: We create a ServerSocketChannel which is a SelectableChannel.
- line 6: We configure the ServerSocketChannel for non-blocking mode.
- line 7: We then register the ServerSocketChannel with the Selector, we receive a SelectionKey from the registration although I discard it, having no use for it. The `serverSocket.validOps()` call will return an operation set that is supported by the Channel, which in this case is only the "Accept Connection" event. The returned SelectionKey contains an "interest set" which indicates the set of IO events the Selector must monitor the Channel for.
- line 10: We call `select()` on the Selector which is blocking until some IO occurs on any of the SelectableChannel instances that are registered with it. It will return the number of keys which are ready for IO activity.

The following code snippet demonstrates iterating through all the SelectionKey instances that indicate IO "ready" events from Channel instances managed by the single Selector. We are only interested in "Accept" and Readable" events. For every new connection accepted an "Accept" event is signaled and we can act on it. Likewise with a "read" ready event we can read incoming data. It is important to remove the SelectionKey from the set after handling it, as the Selector does not do this and you will continue to process that stale event.

`Working with SelectionKeys`

```
final Iterator<SelectionKey> selectionKeyIterator = selectionKeys.iterator();
while (selectionKeyIterator.hasNext()) {
    final SelectionKey key = selectionKeyIterator.next();

    if (key.isAcceptable()) {
        acceptClientSocket(key, serverSocket);
    } else if (key.isReadable()) {
        readRequest(key);
    } else {
        System.out.println("Invalid selection key");
    }

    selectionKeyIterator.remove();
}
```

- line 13: Remember to remove the SelectionKey from the selected set as the Selector does not do this for us, if we don’t do it, we will continue to process stale events.

The following code snippet demonstrates the use of registration of a SocketChannel with the same Selector that manages the ServerSocketChannel. Here, however, the interest set is only for IO "read" events.

`Registering a Channel with a Selector`

```
final SocketChannel client = serverSocket.accept();
client.configureBlocking(false);
client.register(key.selector(), SelectionKey.OP_READ);
```

1.3 Summary

In this beginners tutorial we understood some of the differences between IO and NIO and reasons for NIO's existence and applicability. We have also covered the 4 main abstractions when working with NIO. Those are:

- Buffers
- Channels
- Selectors
- Charsets

We have seen how they can be used and how they work in tandem with each other. With this tutorial in hand, you understand the basics of creating Channels and using them with Buffers. How to interact with Buffers and the rich API it provides for traversing buffer content. We have also learnt how to register Channels with Selectors and interact with the Selector via its SelectionKey abstraction.

1.4 Working with example source code

The source code contains the following examples:

- Charset example.
- FileChannel example. This example reads from a classpath resource file `src/main/resources/file/input.txt` and writes a String literal to a classpath resource `src/main/resources/file/output.txt.` Be sure to check the folder `target/classes/file` when wanting to view the output of the write example.
- Client Server example. Start the server first, then start the client. The client will attempt 10 connections to the server and write the same text 10 times to the server which will simply write the contents to console.

1.5 Download the source code

This was a Java Nio Tutorial for Beginners Example.

Download

You can download the full source code of this example here: **Java NIO tutorial for beginners**

Chapter 2

Java Nio SSL Example

This is an example of a non-blocking I/O provided by `java.nio` using SSL handshake.

2.1 Definition of Secure Sockets Layer Protocol (SSL)

SSL is the secure communication protocol of choice for a large part of the Internet community. There are many applications of SSL in existence, since it is capable of securing any transmission over TCP. Secure HTTP, or HTTPS, is a familiar application of SSL in e-commerce or password transactions. Along with this popularity comes demands to use it with different I/O and threading models in order to satisfy the applications' performance, scalability, footprint, and other requirements. There are demands to use it with blocking and non-blocking I/O channels, asynchronous I/O, input and output streams and byte buffers.The main point of the protocol is to provide privacy and reliability between two communicating applications. The following fundamental characteristics provide connection security:

- Privacy - connection using encryption
- Identity authentication - identification using certificates
- Reliability - dependable maintenance of a secure connection through message integrity

Many developers may be wondering how to use SSL with Java NIO. With the traditional blocking sockets API, security is a simple issue: just set up an `SSLContext` instance with the appropriate key material, use it to create instances of `SSLSocketFactory` or `SSLServerSocketFactory` and finally use these factories to create instances of `SSLServerSocket` or `SSLSocket` . In Java 1.6, a new abstraction was introduced to allow applications to use the SSL/TLS protocols in a transport independent way, and thus freeing applications to choose transport and computing models that best meet their needs. Not only does this new abstraction allow applications to use non-blocking I/O channels and other I/O models, it also accommodates different threading models.

2.2 The SSL Engine API

The new abstraction is therefore an advanced API having as core class the `javax.net.ssl.SSLEngine` . It encapsulates an SSL/TLS state machine and operates on inbound and outbound byte buffers supplied by the user of the SSLEngine.

2.2.1 Lifecycle

The SSLEngine must first go through the handshake, where the server and the client negotiate the cipher suite and the session keys. This phase typically involves the exchange of several messages. After completing the handshake, the application can start sending and receiving application data. This is the main state of the engine and will typically last until the connection is CLOSED (see image below). In some situations, one of the peers may ask for a renegotiation of the session parameters, either

to generate new session keys or to change the cipher suite. This forces a re-handshake. When one of the peers is done with the connection, it should initiate a graceful shutdown, as specified in the SSL/TLS protocol. This involves exchanging a couple of closure messages between the client and the server to terminate the logical session before physically closing the socket.

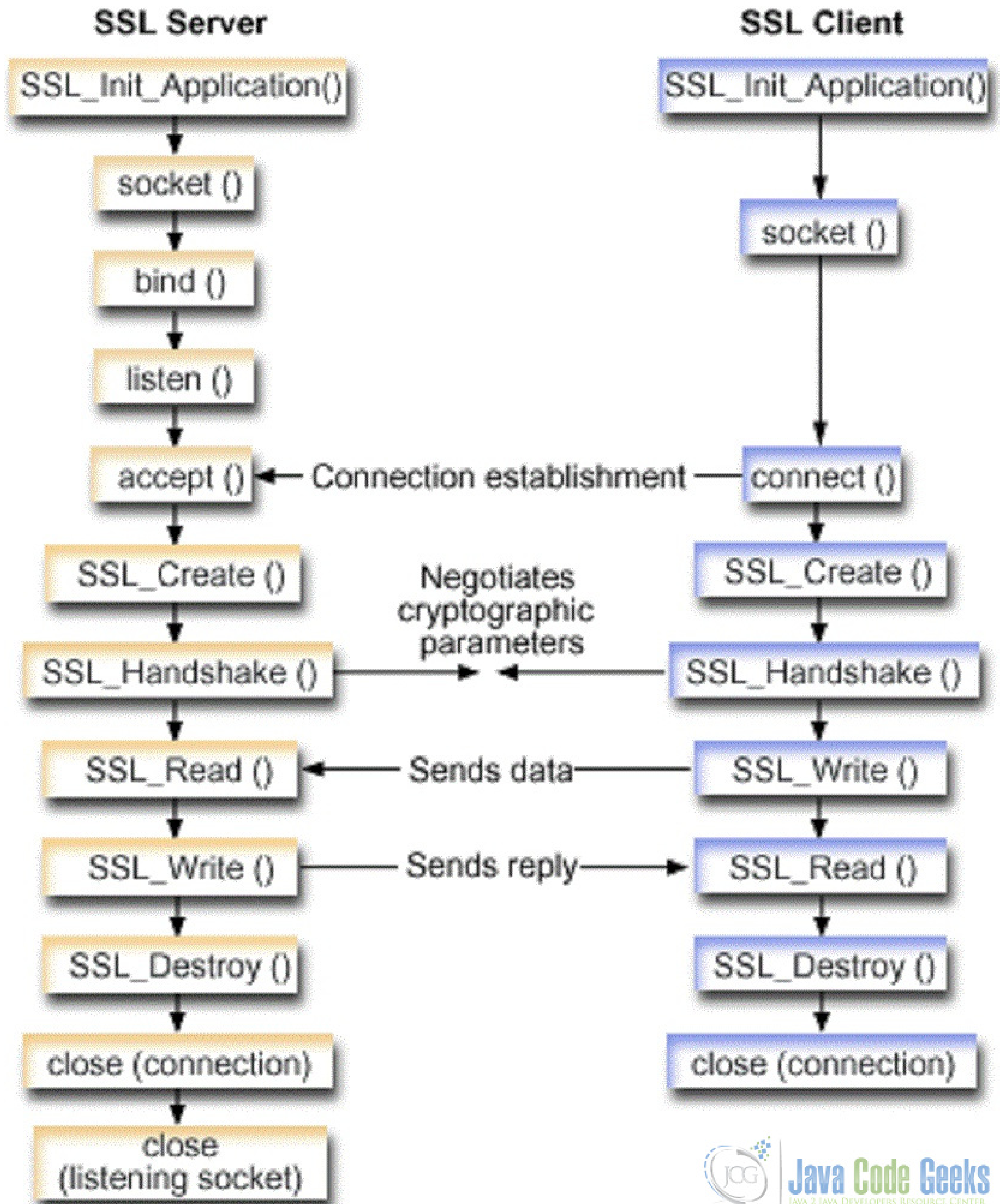

Figure 2.1: SSL Lifecycle

2.2.2 SSL Handshake

The two main SSLEngine methods `wrap()` and `unwrap()` are responsible for generating and consuming network data respectively. Depending on the state of the SSLEngine, this data might be handshake or application data. Each SSLEngine has several phases during its lifetime. Before application data can be sent/received, the SSL/TLS protocol requires a handshake to establish cryptographic parameters. This handshake requires a series of back-and-forth steps by the SSLEngine. The SSL Process can provide more details about the handshake itself. During the initial handshaking, `wrap()` and `unwrap()` generate and consume handshake data, and the application is responsible for transporting the data. This sequence is repeated until the handshake is finished. Each SSLEngine operation generates a `SSLEngineResult` , of which the `SSLEngineResult.HandshakeStatus` field is used to determine what operation needs to occur next to move the handshake along. Below is an example of the handshake process:

client	SSL/TLS message	HSStatus
wrap()	ClientHello	NEED_UNWRAP
unwrap()	ServerHello/Cert/ServerHelloDone	NEED_WRAP
wrap()	ClientKeyExchange	NEED_WRAP
wrap()	ChangeCipherSpec	NEED_WRAP
wrap()	Finished	NEED_UNWRAP
unwrap()	ChangeCipherSpec	NEED_UNWRAP
unwrap()	Finished	FINISHED

Figure 2.2: Typical SSL Handshake

2.3 Nio SSL Example

The following example creates a connection to https://www.amazon.com/ and displays the decrypted HTTP response.

2.3.1 Main class

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.concurrent.Executor;
```

```
import java.util.concurrent.Executors;

import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLEngine;
import javax.net.ssl.SSLSession;

public class NioSSLExample
{
   public static void main(String[] args) throws Exception
   {
      InetSocketAddress address = new InetSocketAddress("www.amazon.com", 443);
      Selector selector = Selector.open();
      SocketChannel channel = SocketChannel.open();
      channel.connect(address);
      channel.configureBlocking(false);
      int ops = SelectionKey.OP_CONNECT | SelectionKey.OP_READ;

      SelectionKey key =  channel.register(selector, ops);

      // create the worker threads
      final Executor ioWorker = Executors.newSingleThreadExecutor();
      final Executor taskWorkers = Executors.newFixedThreadPool(2);

      // create the SSLEngine
      final SSLEngine engine = SSLContext.getDefault().createSSLEngine();
      engine.setUseClientMode(true);
      engine.beginHandshake();
      final int ioBufferSize = 32 * 1024;
      final NioSSLProvider ssl = new NioSSLProvider(key, engine, ioBufferSize, ioWorker,  ↩
          taskWorkers)
      {
         @Override
         public void onFailure(Exception ex)
         {
            System.out.println("handshake failure");
            ex.printStackTrace();
         }

         @Override
         public void onSuccess()
         {
            System.out.println("handshake success");
            SSLSession session = engine.getSession();
            try
            {
               System.out.println("local principal: " + session.getLocalPrincipal());
               System.out.println("remote principal: " + session.getPeerPrincipal());
               System.out.println("cipher: " + session.getCipherSuite());
            }
            catch (Exception exc)
            {
               exc.printStackTrace();
            }

            //HTTP request
            StringBuilder http = new StringBuilder();
            http.append("GET / HTTP/1.0\\r\\n");
            http.append("Connection: close\\r\\n");
            http.append("\\r\\n");
            byte[] data = http.toString().getBytes();
            ByteBuffer send = ByteBuffer.wrap(data);
            this.sendAsync(send);
```

```
         }

         @Override
         public void onInput(ByteBuffer decrypted)
         {
            // HTTP response
            byte[] dst = new byte[decrypted.remaining()];
            decrypted.get(dst);
            String response = new String(dst);
            System.out.print(response);
            System.out.flush();
         }

         @Override
         public void onClosed()
         {
            System.out.println("ssl session closed");
         }
      };

      // NIO selector
      while (true)
      {
         key.selector().select();
         Iterator keys = key.selector().selectedKeys().iterator();
         while (keys.hasNext())
         {
            keys.next();
            keys.remove();
            ssl.processInput();
         }
      }
   }
}
```

From the above code:

- In the `main()` method on lines 18-25, a `Selector` is created and a `SocketChannel` is registered having a selection key interested in socket-connect and socket-read operations for the connection to the amazon url:

```
      InetSocketAddress address = new InetSocketAddress("www.amazon.com", 443);
      Selector selector = Selector.open();
      SocketChannel channel = SocketChannel.open();
      channel.connect(address);
      channel.configureBlocking(false);
      int ops = SelectionKey.OP_CONNECT | SelectionKey.OP_READ;
      SelectionKey key =  channel.register(selector, ops);
```

- On lines 28-29, an `ioWorker` thread is created for executing the `SSLProvider` runnable and also a `ThreadPool` containing 2 threads for executing the delegated runnable task for the SSL Engine.
- On lines 32-34, the `SSLEngine` is initiated in client mode and with initial handshaking:

```
      final SSLEngine engine = SSLContext.getDefault().createSSLEngine();
      engine.setUseClientMode(true);
      engine.beginHandshake();
```

- On lines 36-59, the `NioSSLProvider` object is instantiated. This is responsible for writing and reading from the `ByteChannel` and also as the entry point for the SSL Handshaking. Upon successful negotiation with the amazon server, the local and remote principals are printed and also the name of the SSL cipher suite which is used for all connections in the session.

- The HTTP request is sent from the client after successful handshake on lines 62-67:

```
StringBuilder http = new StringBuilder();
http.append("GET / HTTP/1.0\\r\\n");
http.append("Connection: close\\r\\n");
http.append("\\r\\n");
byte[] data = http.toString().getBytes();
ByteBuffer send = ByteBuffer.wrap(data);
```

- On line 72, the `onInput` method is called whenever the SSL Engine completed an operation with `javax.net.ssl.SSLEngineResult.Status.OK` . The partial decrypted response is printed each time:

```
public void onInput(ByteBuffer decrypted)
{
   // HTTP response
   byte[] dst = new byte[decrypted.remaining()];
   decrypted.get(dst);
   String response = new String(dst);
   System.out.print(response);
   System.out.flush();
}
```

- Finally, the nio `Selector` loop is started on line 90 by processing the selection keys which remain valid until the channel is closed.

2.3.2 NioSSLProvider class

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.SelectionKey;
import java.nio.channels.WritableByteChannel;
import java.util.concurrent.Executor;

import javax.net.ssl.SSLEngine;

public abstract class NioSSLProvider extends SSLProvider
{
   private final ByteBuffer buffer = ByteBuffer.allocate(32 * 1024);
   private final SelectionKey key;

   public NioSSLProvider(SelectionKey key, SSLEngine engine, int bufferSize, Executor  ←
       ioWorker, Executor taskWorkers)
   {
      super(engine, bufferSize, ioWorker, taskWorkers);
      this.key = key;
   }

   @Override
   public void onOutput(ByteBuffer encrypted)
   {
      try
      {
         ((WritableByteChannel) this.key.channel()).write(encrypted);
      }
      catch (IOException exc)
      {
```

```
            throw new IllegalStateException(exc);
        }
    }

    public boolean processInput()
    {
            buffer.clear();
        int bytes;
        try
        {
            bytes = ((ReadableByteChannel) this.key.channel()).read(buffer);
        }
        catch (IOException ex)
        {
            bytes = -1;
        }
        if (bytes == -1) {
            return false;
        }
        buffer.flip();
        ByteBuffer copy = ByteBuffer.allocate(bytes);
        copy.put(buffer);
        copy.flip();
        this.notify(copy);
        return true;
    }
}
```

From the above code:

- A sequence of bytes is read from the channel on line 40:

```
bytes = ((ReadableByteChannel) this.key.channel()).read(buffer);
```

and a new byte buffer is allocated on line 50:

```
ByteBuffer copy = ByteBuffer.allocate(bytes);
```

- The `notify` method is called on line 53, which triggers the ssl handshake procedure and via the helper method `isHandShaking` on line 1 of the SSLProvider class, the wrap/unwrap sequence starts.
- If the `wrap()` helper method from the SSLProvider class is called, then the buffered data are encoded into SSL/TLS network data:

```
wrapResult = engine.wrap(clientWrap, serverWrap);
```

and if the return value of the SSLEngine operation is OK then the `onOutput()` method on line 22 is called in order to write the encrypted response from the server into the `ByteChannel`:

```
((WritableByteChannel) this.key.channel()).write(encrypted);
```

- If the `unwrap()` helper method from the SSLProvider class is called, then an attempt to decode the SSL network data from the server is made on line 95 of the SSLProvider class:

```
unwrapResult = engine.unwrap(clientUnwrap, serverUnwrap);
```

and if the return value of the SSLEngine operation is OK, the decrypted message from the server is printed.

2.3.3 SSLProvider class

For simplicity, we present the basic helper methods of this class:

```
private synchronized boolean isHandShaking()
   {
      switch (engine.getHandshakeStatus())
      {
         case NOT_HANDSHAKING:
            boolean occupied = false;
            {
               if (clientWrap.position() > 0)
                   occupied |= this.wrap();
               if (clientUnwrap.position() > 0)
                   occupied |= this.unwrap();
            }
            return occupied;

         case NEED_WRAP:
            if (!this.wrap())
               return false;
            break;

         case NEED_UNWRAP:
            if (!this.unwrap())
               return false;
            break;

         case NEED_TASK:
            final Runnable sslTask = engine.getDelegatedTask();
            Runnable wrappedTask = new Runnable()
            {
               @Override
               public void run()
               {
                  sslTask.run();
                  ioWorker.execute(SSLProvider.this);
               }
            };
            taskWorkers.execute(wrappedTask);
            return false;

         case FINISHED:
            throw new IllegalStateException("FINISHED");
      }

      return true;
   }

   private boolean wrap()
   {
      SSLEngineResult wrapResult;

      try
      {
         clientWrap.flip();
         wrapResult = engine.wrap(clientWrap, serverWrap);
         clientWrap.compact();
      }
      catch (SSLException exc)
      {
         this.onFailure(exc);
```

```
            return false;
        }

        switch (wrapResult.getStatus())
        {
            case OK:
                if (serverWrap.position() > 0)
                {
                    serverWrap.flip();
                    this.onOutput(serverWrap);
                    serverWrap.compact();
                }
                break;

            case BUFFER_UNDERFLOW:
                // try again later
                break;

            case BUFFER_OVERFLOW:
                throw new IllegalStateException("failed to wrap");

            case CLOSED:
                this.onClosed();
                return false;
        }

        return true;
    }

    private boolean unwrap()
    {
        SSLEngineResult unwrapResult;

        try
        {
            clientUnwrap.flip();
            unwrapResult = engine.unwrap(clientUnwrap, serverUnwrap);
            clientUnwrap.compact();
        }
        catch (SSLException ex)
        {
            this.onFailure(ex);
            return false;
        }

        switch (unwrapResult.getStatus())
        {
            case OK:
                if (serverUnwrap.position() > 0)
                {
                    serverUnwrap.flip();
                    this.onInput(serverUnwrap);
                    serverUnwrap.compact();
                }
                break;

            case CLOSED:
                this.onClosed();
                return false;

            case BUFFER_OVERFLOW:
                throw new IllegalStateException("failed to unwrap");
```

```
            case BUFFER_UNDERFLOW:
                return false;
        }

        if (unwrapResult.getHandshakeStatus() == HandshakeStatus.FINISHED)
        {
                this.onSuccess();
                return false;
        }

        return true;
    }
```

2.4 Download Java Source Code

This was an example of SSL handshake with `java.nio`

Download

You can download the full source code of this example here: **NioSSLExample**

Chapter 3

Java Nio Socket Example

This article introduces the `SocketChannel` class and its basic usage. This class is defined in the java.nio package.

3.1 Standard Java sockets

Socket programming involves two systems communicating with one another. In implementations prior to NIO, Java TCP client socket code is handled by the java.net.Socket class. A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes, `Socket` and `ServerSocket` , that implement the client side of the connection and the server side of the connection, respectively. The below image illustrates the nature of this communication:

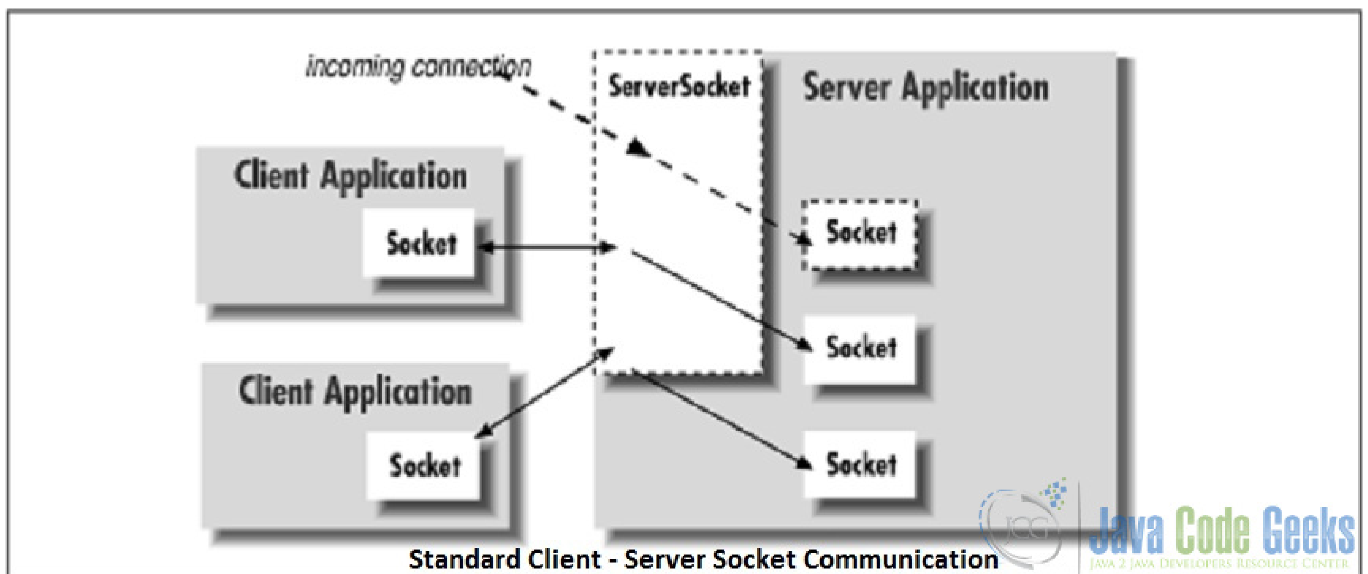

Figure 3.1: SocketExample

A socket is basically a blocking input/output device. It makes the thread that is using it to block on reads and potentially also block on writes if the underlying buffer is full. Therefore, different threads are required if the server has many open sockets. From a simplistic perspective, the process of a blocking socket communication is as follows:

- Create a `ServerSocket` , specifying a port to listen on.

- Invoke the ServerSocket's `accept()` method to listen on the configured port for a client connection.
- When a client connects to the server, the `accept()` method returns a `Socket` through which the server can communicate with the client: an `InputStream` is obtained to read from the client and an `OutputStream` to write to the client.

3.2 Nonblocking SocketChannel with java.nio

With the standard java sockets, if the server needed to be scalable, the socket had to be passed to another thread for processing so that the server could continue listening for additional connections, meaning call the ServerSocket's `accept()` method again to listen for another connection.

A `SocketChannel` on the other hand is a non-blocking way to read from sockets, so that you can have one thread communicate with multiple open connections at once. With socket channel we describe the communication channel between client and server. It is identified by the server IP address and the port number. Data passes through the socket channel by buffer items. A selector monitors the recorded socket channels and serializes the requests, which the server has to satisfy. The Keys describe the objects used by the selector to sort the requests. Each key represents a single client sub-request and contains information to identify the client and the type of the request. With non-blocking I/O, someone can program networked applications to handle multiple simultaneous connections without having to manage multiple thread collection, while also taking advantage of the new server scalability that is built in to java.nio. The below image illustrates this procedure:

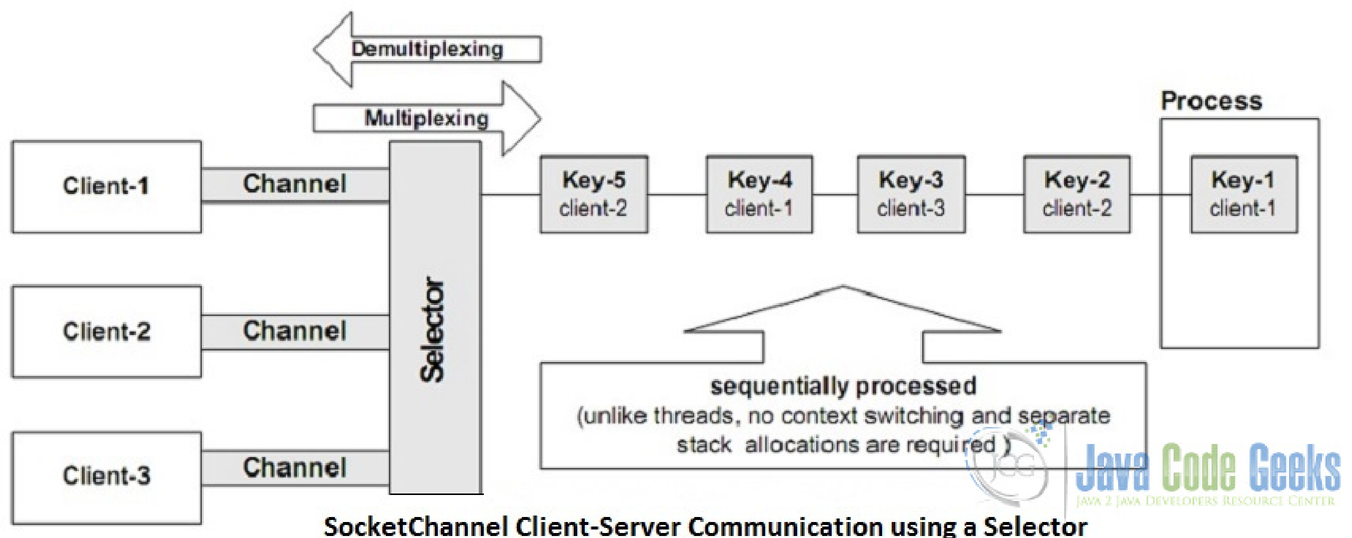

Figure 3.2: SocketChannel

3.3 Example

The following example shows the use of `SocketChannel` for creating a simple echo server, meaning it echoes back any message it receives.

3.3.1 The Server code

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
```

```
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.*;

public class SocketServerExample {
        private Selector selector;
    private Map<SocketChannel,List> dataMapper;
    private InetSocketAddress listenAddress;

    public static void main(String[] args) throws Exception {
        Runnable server = new Runnable() {
                        @Override
                        public void run() {
                                 try {
                                        new SocketServerExample("localhost", 8090). ↩
                                            startServer();
                                } catch (IOException e) {
                                        e.printStackTrace();
                                }

                        }
                };

                Runnable client = new Runnable() {
                        @Override
                        public void run() {
                                 try {
                                         new SocketClientExample().startClient();
                                } catch (IOException e) {
                                        e.printStackTrace();
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }

                        }
                };
       new Thread(server).start();
       new Thread(client, "client-A").start();
       new Thread(client, "client-B").start();
    }

    public SocketServerExample(String address, int port) throws IOException {
        listenAddress = new InetSocketAddress(address, port);
        dataMapper = new HashMap<SocketChannel,List>();
    }

    // create server channel
    private void startServer() throws IOException {
        this.selector = Selector.open();
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.configureBlocking(false);

        // retrieve server socket and bind to port
        serverChannel.socket().bind(listenAddress);
        serverChannel.register(this.selector, SelectionKey.OP_ACCEPT);

        System.out.println("Server started...");

        while (true) {
            // wait for events
            this.selector.select();
```

```
            //work on selected keys
            Iterator keys = this.selector.selectedKeys().iterator();
            while (keys.hasNext()) {
                SelectionKey key = (SelectionKey) keys.next();

                // this is necessary to prevent the same key from coming up
                // again the next time around.
                keys.remove();

                if (!key.isValid()) {
                    continue;
                }

                if (key.isAcceptable()) {
                    this.accept(key);
                }
                else if (key.isReadable()) {
                    this.read(key);
                }
            }
        }
    }

    //accept a connection made to this channel's socket
    private void accept(SelectionKey key) throws IOException {
        ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
        SocketChannel channel = serverChannel.accept();
        channel.configureBlocking(false);
        Socket socket = channel.socket();
        SocketAddress remoteAddr = socket.getRemoteSocketAddress();
        System.out.println("Connected to: " + remoteAddr);

        // register channel with selector for further IO
        dataMapper.put(channel, new ArrayList());
        channel.register(this.selector, SelectionKey.OP_READ);
    }

    //read from the socket channel
    private void read(SelectionKey key) throws IOException {
        SocketChannel channel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        int numRead = -1;
        numRead = channel.read(buffer);

        if (numRead == -1) {
            this.dataMapper.remove(channel);
            Socket socket = channel.socket();
            SocketAddress remoteAddr = socket.getRemoteSocketAddress();
            System.out.println("Connection closed by client: " + remoteAddr);
            channel.close();
            key.cancel();
            return;
        }

        byte[] data = new byte[numRead];
        System.arraycopy(buffer.array(), 0, data, 0, numRead);
        System.out.println("Got: " + new String(data));
    }
}
```

From the above code:

- In the `main()` method on lines 43-45, one thread for creating the `ServerSocketChannel` is started and two client threads responsible for starting the clients which will create a `SocketChannel` for sending messsages to the server.

```
new Thread(server).start();
new Thread(client, "client-A").start();
new Thread(client, "client-B").start();
```

- In the `startServer()` method on line 54, the server `SocketChannel` is created as nonBlocking, the server socket is retrieved and bound to the specified port:

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
// retrieve server socket and bind to port
serverChannel.socket().bind(listenAddress);
```

Finally, the `register` method associates the selector to the socket channel.

```
serverChannel.register(this.selector, SelectionKey.OP_ACCEPT);
```

The second parameter represents the type of the registration. In this case, we use `OP_ACCEPT` , which means the selector merely reports that a client attempts a connection to the server. Other possible options are: `OP_CONNECT` , which will be used by the client; `OP_READ` ; and `OP_WRITE` . After that, the `select` method is used on line 67, which blocks the execution and waits for events recorded on the selector in an infinite loop.

```
this.selector.select();
```

- The selector waits for events and creates the keys. According to the key-types, an opportune operation is performed. There are four possible types for a key:
  - Acceptable: the associated client requests a connection.
  - Connectable: the server accepted the connection.
  - Readable: the server can read.
  - Writeable: the server can write.
- If an acceptable key is found, the `accept(SelectionKey key)` on line 93 is invoked in order to create a channel which accepts this connection, creates a standard java socket on line 97 and register the channel with the selector:

```
ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
SocketChannel channel = serverChannel.accept();
channel.configureBlocking(false);
Socket socket = channel.socket();
SocketAddress remoteAddr = socket.getRemoteSocketAddress();
```

- After receiving a readable key from the client, the `read(SelectionKey key)` is called on line 107 which reads from the socket channel. A byte buffer is allocated for reading from the channel

```
numRead = channel.read(buffer);
```

and the client's transmitted data are echoed on System.out:

```
System.out.println("Got: " + new String(data));
```

3.3.2 The Client code

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

public class SocketClientExample {

    public void startClient()
            throws IOException, InterruptedException {

        InetSocketAddress hostAddress = new InetSocketAddress("localhost", 8090);
        SocketChannel client = SocketChannel.open(hostAddress);

        System.out.println("Client... started");

        String threadName = Thread.currentThread().getName();

        // Send messages to server
        String [] messages = new String []
                        {threadName + ": test1",threadName + ": test2",threadName + ": ←
                            test3"};

        for (int i = 0; i < messages.length; i++) {
            byte [] message = new String(messages [i]).getBytes();
            ByteBuffer buffer = ByteBuffer.wrap(message);
            client.write(buffer);
            System.out.println(messages [i]);
            buffer.clear();
            Thread.sleep(5000);
        }
        client.close();
    }
}
```

- In the above client code, each client thread creates a socket channel on the server's host address on line 12:

```
SocketChannel client = SocketChannel.open(hostAddress);
```

- On line 19, a String array is created to be transmitted to the server using the previously created socket. The data contain also each thread's name for distinguishing the sender:

```
String threadName = Thread.currentThread().getName();
// Send messages to server
String [] messages = new String []
{threadName + ": test1",threadName + ": test2",threadName + ": test3"};
```

- For each string message a buffer is created on line 24:

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

and each message is written to the channel from the given buffer on line 25:

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

3.3.3 The output

```
Server started...
Client... started
Client... started
client-A: test1
client-B: test1
Connected to: /127.0.0.1:51468
Got: client-B: test1
Connected to: /127.0.0.1:51467
Got: client-A: test1
client-A: test2
client-B: test2
Got: client-B: test2
Got: client-A: test2
client-A: test3
client-B: test3
Got: client-B: test3
Got: client-A: test3
Connection closed by client: /127.0.0.1:51468
Connection closed by client: /127.0.0.1:51467
```

3.4 Download Java Source Code

This was an example of `java.nio.SocketChannel`

Download

You can download the full source code of this example here: **SocketExampleNio.zip**

Chapter 4

Java Nio Write File Example

With this example we are going to demonstrate how to use the Non-blocking I/O API, or `NIO.2` API (NIO API) for short, to write data to a file. The examples in this article are compiled and run in a Mac OS unix environment.

Please note that Java SE 8 is required to run the code in this article.

4.1 Introduction to the NIO API

The `NIO.2` API was introduced in Java 7 as a replacement for the `java.io.File` class. It provides a flexible, and intuitive API for use with files.

4.2 Creating a NIO Path

In order to write a file to the file system we must first create a Path for the destination of the file. A `Path` object is a hierarchical representation of the path on a system to the file or directory. The `java.nio.file.Path` interface is the primary entry point for working with the `NIO 2` API.

The easiest way to create a Path Object is to use the `java.nio.files.Paths` factory class. The class has a static `get()` method which can be used to obtain a reference to a file or directory. The method accepts either a string, or a sequence of strings(which it will join to form a path) as parameters. A `java.nio.file.Path` , like a `File` , may refer to either an absolute or relative path within the file system. This is displayed in the following examples:

```
Path p1 = Paths.get( "cats/fluffy.jpg" );
Path p2 = Paths.get("/home/project");
Path p3 = Paths.get("/animals", "dogs", "labradors");
```

In the above:

- p1 creates a relative reference to a file in the current working directory.
- p2 creates a reference to an absolute directory in a Unix based system.
- p3 creates a reference to the absolute directory /animals/dogs/labradors

4.3 Writing files with the NIO API

Once we have a Path Object we are able to execute most of the operations that were previously possible with `java.io.File` .

4.3.1 Using NIO API with write()

The `Files.write()` method is an option when writing strings to file. It is cleanly and concisely expressed. It is not a good idea to use a method like this to write larger files, due to its reliance on byte arrays. As shown below there are more efficient ways of handling these.

```
Path path = Paths.get("src/main/resources/question.txt");

String question = "To be or not to be?";

Files.write(path, question.getBytes());
```

4.3.2 Using NIO API with newBufferedWriter()

The `NIO.2` API has methods for writing files using java.io streams. The Files.newBufferedWriter(Path,Charset) writes to the file at the specified Path location, using the user defined Charset for character encoding. This BufferedWriter method is preferential due to its efficient performance especially when completing a large amount of write operations. Buffered operations have this effect as they aren't required to call the operating system's write method for every single byte, reducing on costly I/O operations. Here's an example:

```
Path path = Paths.get("src/main/resources/shakespeare.txt");
try(BufferedWriter writer = Files.newBufferedWriter(path, Charset.forName("UTF-8"))){
writer.write("To be, or not to be. That is the question.");
}catch(IOException ex){
ex.printStackTrace();
}
```

This method will create a new file at the given path, or overwrite it if it already exists.

4.3.3 Using NIO API to copy a file with an Outputstream

In this example we use the NIO API in conjunction with an output stream to copy a file and its contents from one file to a new instance with a new name. We achieve this by using the Files.copy() method, which accepts (Path source, Outputstream os) as its parameters.

```
Path oldFile = Paths.get("src/main/resources/", "oldFile.txt");
Path newFile = Paths.get("src/main/resources/", "newFile.txt");
   try (OutputStream os = new FileOutputStream(newFile.toFile())) {

      Files.copy(oldFile, os);

    } catch (IOException ex) {
    ex.printStackTrace();
    }
```

The code above is an example of the try with resources syntax which was introduced in Java 7 and made it easier handle exceptions while correctly closing streams and other resources which are used in a try-catch block. FileOutputStream is used to handle binary data.

4.4 Summary

In this article we've introduced you a couple ways to use the NIO API to write a file to your file system. Along the way we've touched upon the Path object.

4.5 Download The Source Code

This was an example of writing to a file with Java NIO2 API.

Download

You can download the full source code of this example here: **Java Nio Write File Example**

Chapter 5

Java Nio Heartbeat Example

This article is a tutorial on implementing a simple Java NIO Heartbeat. This example will take the form of "n" number of "Broadcast" mode processes which will multicast data via UDP to "n" number of "Subscribe" processes that have expressed interest in receiving said traffic.

5.1 Introduction

This article builds on three earlier articles on the subject of Java NIO, namely Java Nio Tutorial for Beginners, Java Nio Asynchronous Channels Tutorial and "Java Nio EchoServer". Before getting stuck into the "meat" of our example, it is best to get some background into the topic itself. According to Wikipedia a "heartbeat" in computer systems is a periodic signal generated by hardware or software to indicate normal operation or to synchronize parts of a system. So true to the name it is indeed a measure of life of individual components in a distributed computer system and one can deduce then by it's absence, presence and frequency, the state of the system in which it occurs.

In the same breath, when one talks about "heartbeats" in computer systems the term "UDP" often comes up and with good reason. It is the protocol of choice when implementing "hearbeat" type solutions, weather it be cluster membership negotiations or life signing (heartbeats). The low latency of this "connection-less" protocol also plays to the nature of "heartbeating" in distributed systems.

Important to note that unlike TCP, UDP makes no guarantee on delivery of packets, the low latency of UDP stems from this not having to guarantee delivery via the typical SYN ACK (3 way handshake etc).

We go one step further in this example and we multicast the traffic out to interested parties. Now why would we do this and what other choices are there? Typically the following choices would present themselves:

- Unicast: From one machine to another. One-to-One
- Broadcast: From one machine to all possible machines. One-to-All (within the broadcast domain - ie: behind a router or in a private network)
- Multicast: From one machine to multiple machines that have stated interest in receiving said traffic. This can traverse the broadcast domain and extend past a router.

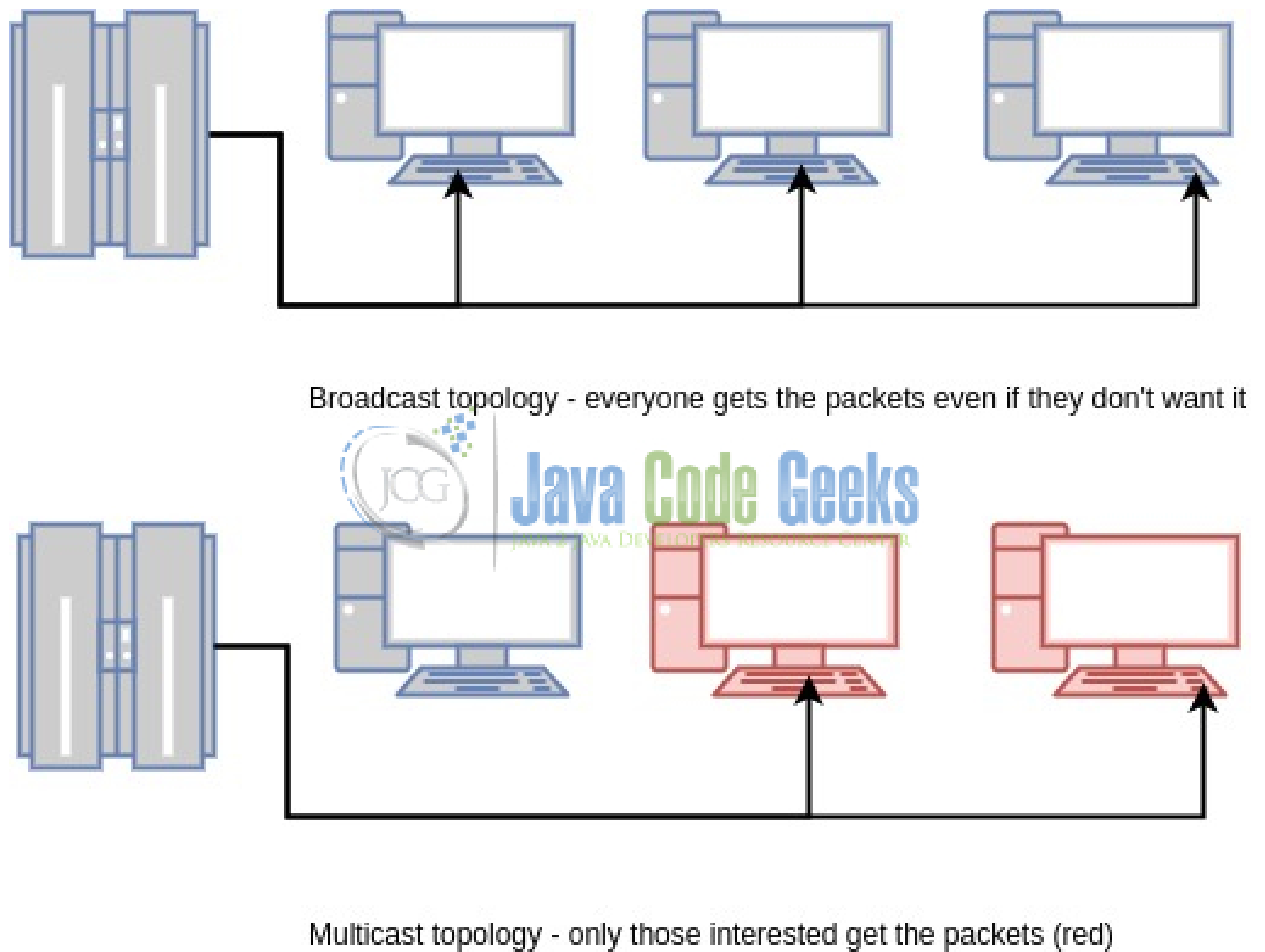

Figure 5.1: Topology of Broadcast vs Multicast

5.2 Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or Linux will do fine)

5.3 Overview

The abstractions of use to us when wanting to effect UDP in Java Nio would be the DatagramChannel, which also happens to be a SelectableChannel priming it for use by a Selector in a very Thread efficient manner. It also happens to implement MulticastChannel which supports Internet Protocol (IP) multicasting.

5.3.1 DatagramChannel

A DatagramChannel is opened by one of the static `open(...)` methods of the class itself. One of the `open(...)` methods are of particular interest to us and that is:

DatagramChannel open for multicast

```
public static DatagramChannel open(ProtocolFamily family) throws IOException
```

The ProtocolFamily is required when attempting to multicast with this Channel and should correspond to the IP type of the multicast group that this Channel will join. eg: IPV4 `StandardProtocolFamily.INET` A DatagramChannel need not be connected to use the `send(...)` and `receive(...)` methods of this class, conversely so the `read(...)` and `write(...)` methods do.

5.3.2 MulticastChannel

This Channel supports (IP) multicasting. Of particular interest to us is this part of it's API:

DatagramChannel configuration

```
...
channel.setOption(StandardSocketOptions.IP_MULTICAST_IF, NetworkInterface);
channel.join(InetAddress, this.multicastNetworkInterface);
...
```

line 2: the NetworkInterface is the interface through which we will send / receive UDP multicast traffic line 3: we ensure we join the multicast group (express interest in receiving traffic to this group) by way of passing a InetAddress (the multicast IP) and a NetworkInterface (the interface through which we will receive said multicast traffic). Multicast IP ranges range from 224.0.0.0 to 239.255.255.255 typically.

A MulticastChannel can join "n" number of multicast groups and can join a group on different network interfaces.

5.4 Multicaster

Multicaster

```
final class Multicaster implements ScheduledChannelOperation {

    private final String id;
    private final ScheduledExecutorService scheduler;
    private final NetworkInterface networkInterface;
    private final InetSocketAddress multicastGroup;

    Multicaster(final String id, final String ip, final String interfaceName, final int ↩
        port, final int poolSize) {
        if (StringUtils.isEmpty(id) || StringUtils.isEmpty(ip) || StringUtils.isEmpty( ↩
            interfaceName)) {
            throw new IllegalArgumentException("required id, ip and interfaceName");
        }

        this.id = id;
        this.scheduler = Executors.newScheduledThreadPool(poolSize);
        this.multicastGroup = new InetSocketAddress(ip, port);

        try {
            this.networkInterface = NetworkInterface.getByName(interfaceName);
        } catch (SocketException e) {
```

```
            throw new RuntimeException("unable to start broadcaster", e);
        }
    }

    @Override
    public ScheduledExecutorService getService() {
        return this.scheduler;
    }

    void run(final CountDownLatch endLatch) {
        assert !Objects.isNull(endLatch);

        try (DatagramChannel channel = DatagramChannel.open()) {

            initChannel(channel);
            doSchedule(channel);

            endLatch.await();
        } catch (IOException | InterruptedException e) {
            throw new RuntimeException("unable to run broadcaster", e);
        } finally {
            this.scheduler.shutdownNow();
        }
    }

    private void doSchedule(final DatagramChannel channel) {
        assert !Objects.isNull(channel);

        doSchedule(channel, new Runnable() {
            public void run() {
                System.out.println(String.format("Multicasting for %s", Multicaster.this.id ↩
                    ));

                try {
                    Multicaster.this.doBroadcast(channel);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }, 0L, Constants.Schedule.PULSE_DELAY_IN_MILLISECONDS, TimeUnit.MILLISECONDS);
    }

    private void initChannel(final DatagramChannel channel) throws IOException {
        assert !Objects.isNull(channel);

        channel.bind(null);
        channel.setOption(StandardSocketOptions.IP_MULTICAST_IF, this.networkInterface);
    }

    private void doBroadcast(final DatagramChannel channel) throws IOException {
        assert !Objects.isNull(channel);

        Pulse.broadcast(this.id, this.multicastGroup, channel);
    }
}
```

- line 14: we create a ScheduledExecutorService with the purposes of scheduling the multicast heartbeat pulse to the multicast group
- line 15: we create a InetSocketAddress which will be the multicast group to which we will send our heartbeats
- line 18: we create a NetworkInterface which will encapsulate the interface through which our multicast heartbeats will travel

- line 34: we initialize our DatagramChannel
- line 35: we schedule our heartbeat thread
- line 48-58: represents the schedule task that is run, this is quite simply a `send(...)` operation on the DatagramChannel to the InetSocketAddress which represents our multicast group
- line 64: allow any socket address to be bound to the socket - does not matter
- line 65: ensure we set the NetworkInterface to be used for the multicast heartbeats that are sent. We don't set the TTL for the multicast, although you could if you want.

5.5 Subscriber

Subscriber

```
final class Subscriber implements ScheduledChannelOperation {

    private final String id;
    private final ScheduledExecutorService scheduler;
    private final NetworkInterface networkInterface;
    private final InetSocketAddress hostAddress;
    private final InetAddress group;
    private final ConcurrentMap<String, Pulse> pulses;

    Subscriber(final String id, final String ip, final String interfaceName, final int port ↩
        , final int poolSize) {
        if (StringUtils.isEmpty(id) && StringUtils.isEmpty(ip) || StringUtils.isEmpty( ↩
            interfaceName)) {
            throw new IllegalArgumentException("required id, ip and interfaceName");
        }

        this.id = id;
        this.scheduler = Executors.newScheduledThreadPool(poolSize);
        this.hostAddress = new InetSocketAddress(port);
        this.pulses = new ConcurrentHashMap<>();

        try {
            this.networkInterface = NetworkInterface.getByName(interfaceName);
            this.group = InetAddress.getByName(ip);
        } catch (SocketException | UnknownHostException e) {
            throw new RuntimeException("unable to start broadcaster", e);
        }
    }

    @Override
    public ScheduledExecutorService getService() {
        return this.scheduler;
    }

    void run() {
        try (final DatagramChannel channel = DatagramChannel.open(StandardProtocolFamily. ↩
            INET); final Selector selector = Selector.open()) {

            System.out.printf("Starting subscriber %s", id);
            initChannel(channel, selector);
            doSchedule(channel);

            while (!Thread.currentThread().isInterrupted()) {
                if (selector.isOpen()) {
```

```
                    final int numKeys = selector.select();
                    if (numKeys > 0) {
                        handleKeys(channel, selector.selectedKeys());
                    }
                } else {
                    Thread.currentThread().interrupt();
                }
            }
        } catch (IOException e) {
            throw new RuntimeException("unable to run subscriber", e);
        } finally {
            this.scheduler.shutdownNow();
        }
    }

    private void initChannel(final DatagramChannel channel, final Selector selector) throws ←
         IOException {
        assert !Objects.isNull(channel) && Objects.isNull(selector);

        channel.configureBlocking(false);
        channel.setOption(StandardSocketOptions.SO_REUSEADDR, true);
        channel.bind(this.hostAddress);
        channel.setOption(StandardSocketOptions.IP_MULTICAST_IF, this.networkInterface);
        channel.join(this.group, this.networkInterface);
        channel.register(selector, SelectionKey.OP_READ);
    }

    private void handleKeys(final DatagramChannel channel, final Set<SelectionKey> keys)  ←
        throws IOException {
        assert !Objects.isNull(keys) && !Objects.isNull(channel);

        final Iterator<SelectionKey> iterator = keys.iterator();
        while (iterator.hasNext()) {

            final SelectionKey key = iterator.next();
            try {
                if (key.isValid() && key.isReadable()) {
                    Pulse.read(channel).ifPresent((pulse) -> {
                        this.pulses.put(pulse.getId(), pulse);
                    });
                } else {
                    throw new UnsupportedOperationException("key not valid.");
                }
            } finally {
                iterator.remove();
            }
        }
    }

    private void doSchedule(final DatagramChannel channel) {
        assert !Objects.isNull(channel);

        doSchedule(channel, new Runnable() {
            public void run() {
                Subscriber.this.pulses.forEach((id, pulse) -> {
                    if (pulse.isDead(Constants.Schedule. ←
                        DOWNTIME_TOLERANCE_DEAD_SERVICE_IN_MILLISECONDS)) {
                        System.out.println(String.format("FATAL   : %s removed", id));
                        Subscriber.this.pulses.remove(id);
                    } else if (!pulse.isValid(Constants.Schedule. ←
                        DOWNTIME_TOLERANCE_IN_MILLISECONDS)) {
                        System.out.println(String.format("WARNING : %s is down", id));
```

```
                    } else {
                        System.out.println(String.format("OK      : %s is up", id));
                    }
                });
            }
        }, 0L, Constants.Schedule.PULSE_DELAY_IN_MILLISECONDS, TimeUnit.MILLISECONDS);
    }
}
```

- line 16: we create a ScheduledExecutorService to schedule the polling of the heartbeat pulses we have received thus far via UDP multicast
- line 17: we create a InetSocketAddress for the specified port and instantiate it for the "localhost"
- line 21: we create NetworkInterface for the specified interface name, this will be the interface through which the `Subscriber` will receive UDP multicast heartbeat pulses
- line 22: we create a InetAddress representing the multicast group from which we will receive multicast messages
- line 34: we open the DatagramChannel but also specify the `ProtocolFamily` and this should correspond to the address type of the multicast group this Channel will be joining.
- line 37-38: we initialize the Channel and schedule the polling of heartbeat pulses
- line 40-49: while the current thread is still running we utilize the `Selector` and await incoming UDP multicast heartbeats in a non-blocking way.
- line 63-64: we set the multicast interface and join the multicast group using the multicast interface
- line 77-79: we read a Pulse from the UDP multicast packet.

5.6 Running the program

The example project is a maven project and must be built into a "fat" or "uber" jar by issuing the following command `mvn clean install package` . The resulting artifact can be found in the "target" folder located in the project root folder. The project can be run in two modes, one being "MULTICAST" and the other being "SUBSCRIBE". Obviously the "MULTICAST" mode will publish packets (heartbeats) to the multicast group and the "SUBSCRIBE" mode will receive said heartbeats.

The beauty of the example is that you can spin up as many "MULTICAST" processes as you wish (ensure you give them all unique id's) and as many "SUBSCRIBE" processes as you wish (ensure you give them also unique id's). This can be done in random order meaning "MULTICAST" or "SUBSCRIBE" in any order. Simply put, as soon as heartbeats arrive, the subscribers will know about it and begin reporting as shown below:

Figure 5.2: Running 2 multicast processes and 2 subscribe processes

On top are the two "MULTICAST" processes and on the bottom are the two "SUBSCRIBE" processes. Notice how the subscribers report the other processes being "up" until I terminate one of them ("A" - top left) and then after a configurable tolerance is exceeded, the last packet / pulse is reported as too old, the subscribers notify us that "A is down". After a short while A is removed and considered dead. We then bring A back up and you see immediately a heartbeat is received and the subscribers begin reporting "A is up" along with "B", which never went down.

To run this example you need a network interface, ie: you need to be plugged into a network point as the network interface name is required. Although I have been successful in sending packets via the wireless interface as the "BROADCASTER", to the multicast group, and receiving them in the "SUBSCRIBER" via a LAN interface, I would highly recommend using a LAN (cable) interface for both MULTICAST and BROADCAST processes.The interface will serve as the NetworkInterface on which the multicast group is joined.

The following arguments are required to run the program:

- -i *Unique id of the process* : eg: A or B or S1
- -m *Mode* : can be one of MULICAST or SUBSCRIBE
- -n *NetworkInterface name* : be sure to run `ifconfig` (linux) or `ipconfig` (windows) to find the interface name. eg: eth0
- Running a MULTICAST process: `java -jar heartbeat-0.0.1-SNAPSHOT.jar -m MULTICAST -i A -n eth0`
- Running a SUBSCRIBE process: `java -jar heartbeat-0.0.1-SNAPSHOT.jar -m SUBSCRIBE -i s1 -n eth0`

5.7 Summary

In this example we demonstrated how to build a simple heartbeat application using UDP and Java NIO. We took it a bit further and leveraged multicasting as a method to publish our heartbeat signals across the network to interested parties. These interested

parties can change state, notify persons or even try to resurrect down / faulty services when they become aware of problems via the heartbeat signals.

5.8 Download the source code

This was a Java Nio Heartbeat tutorial

Download

You can download the full source code of this example here: **Java Nio Heartbeat tutorial**

Chapter 6

Java Nio Large File Transfer Tutorial

This article is a tutorial on transferring a large file using Java Nio. It will take shape via two examples demonstrating a simple local file transfer from one location on hard disk to another and then via sockets from one remote location to another remote location.

6.1 Introduction

This tutorial will make use of the FileChannel abstraction for both remote and local copy. Augmenting the remote copy process will be a simple set of abstractions (ServerSocketChannel & SocketChannel) that facilitate the transfer of bytes over the wire. Finally we wrap things up with an asynchronous implementation of large file transfer. The tutorial will be driven by unit tests that can run from command line using maven or from within your IDE.

6.2 Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or Linux will do fine)

6.3 FileChannel

A FileChannel is a type of Channel used for writing, reading, mapping and manipulating a File. In addition to the familiar Channel (read, write and close) operations, this Channel has a few specific operations:

- Has the concept of an absolute position in the File which does not affect the Channels current position.
- Parts or regions of a File can be mapped directly into memory and work from memory, very useful when dealing with large files.
- Writes can be forced to the underlying storage device, ensuring write persistence.
- Bytes can be transferred from one ReadableByteChannel / WritableByteChannel instance to another ReadableByteChannel / WritableByteChannel, which FileChannel implements. This yields tremendous IO performance advantages that some Operating systems are optimized for.

- A part or region of a File may be locked by a process to guard against access by other processes.

FileChannels are thread safe. Only one IO operation that involves the FileChannels position can be in flight at any given point in time, blocking others. The view or snapshot of a File via a FileChannel is consistent with other views of the same File within the same process. However, the same cannot be said for other processes. A file channel can be created in the following ways:

- `... FileChannel.open(...)`
- `... FileInputStream(...).getChannel()`
- `... FileOutputStream(...).getChannel()`
- `... RandomAccessFile(...).getChannel()`

Using one of the stream interfaces to obtain a FileChannel will yield a Channel that allows either read, write or append privileges and this is directly attributed to the type of Stream (FileInputStream or FileOutputStream) that was used to get the Channel. Append mode is a configuration artifact of a FileOutputStream constructor.

6.4 Background

The sample program for this example will demonstrate the following:

- Local transfer of a file (same machine)
- Remote transfer of a file (potentially remote different processes, although in the unit tests we spin up different threads for client and server)
- Remote transfer of a file asynchronously

Particularly with large files the advantages of asynchronous non blocking handling of file transfer cannot be stressed enough. Large files tying up connection handling threads soon starve a server of resources to handle additional requests possibly for more large file transfers.

6.5 Program

The code sample can be split into local and remote domains and within remote we further specialize an asynchronous implementation of file transfer, at least on the receipt side which is arguably the more interesting part.

6.5.1 Local copy

`FileCopy`

```
final class FileCopy

    private FileCop() {
        throw new IllegalStateException(Constants.INSTANTIATION_NOT_ALLOWED);
    }

    public static void copy(final String src, final String target) throws IOException {
        if (StringUtils.isEmpty(src) || StringUtils.isEmpty(target)) {
            throw new IllegalArgumentException("src and target required");
        }

        final String fileName = getFileName(src);
```

```
        try (FileChannel from = (FileChannel.open(Paths.get(src), StandardOpenOption.READ)) ↩
            ;
                FileChannel to = (FileChannel.open(Paths.get(target + "/" + fileName),  ↩
                    StandardOpenOption.CREATE_NEW, StandardOpenOption.WRITE))) {
            transfer(from, to, 0l, from.size());
        }
    }

    private static String getFileName(final String src) {
        assert StringUtils.isNotEmpty(src);

        final File file = new File(src);
        if (file.isFile()) {
            return file.getName();
        } else {
            throw new RuntimeException("src is not a valid file");
        }
    }

    private static void transfer(final FileChannel from, final FileChannel to, long  ↩
        position, long size) throws IOException {
        assert !Objects.isNull(from) && !Objects.isNull(to);

        while (position < size) {
            position += from.transferTo(position, Constants.TRANSFER_MAX_SIZE, to);
        }
    }
}
```

- line 14: we `open` the `from` Channel with the `StandardOpenOption.READ` meaning that this Channel will only be read from. The path is provided.
- line 15: the `to` Channel is opened with the intention to write and create, the path is provided.
- line 31-37: the two Channels are provided (from & to) along with the `position` (initially where to start reading from) and the `size` indicating the amount of bytes to transfer in total. A loop is started where attempts are made to transfer up to `Constants.TRANSFER_MAX_SIZE` in bytes from the `from` Channel to the `to` Channel. After each iteration the amount of bytes transferred is added to the `position` which then advances the cursor for the next transfer attempt.

6.5.2 Remote copy

`FileReader`

```
final class FileReader {

    private final FileChannel channel;
    private final FileSender sender;

    FileReader(final FileSender sender, final String path) throws IOException {
        if (Objects.isNull(sender) || StringUtils.isEmpty(path)) {
            throw new IllegalArgumentException("sender and path required");
        }

        this.sender = sender;
        this.channel = FileChannel.open(Paths.get(path), StandardOpenOption.READ);
    }

    void read() throws IOException {
        try {
```

```
            transfer();
        } finally {
            close();
        }
    }

    void close() throws IOException {
        this.sender.close();
        this.channel.close();
    }

    private void transfer() throws IOException {
        this.sender.transfer(this.channel, 0l, this.channel.size());
    }
}
```

- line 12: the FileChannel is opened with the intent to read `StandardOpenOption.READ` , the `path` is provided to the File.
- line 15-21: we ensure we transfer the contents of the FileChannel entirely and the close the Channel.
- line 23-26: we close the `sender` resources and then close the FileChannel
- line 29: we call `transfer(...)` on the `sender` to transfer all the bytes from the FileChannel

FileSender

```
final class FileSender {

    private final InetSocketAddress hostAddress;
    private SocketChannel client;

    FileSender(final int port) throws IOException {
        this.hostAddress = new InetSocketAddress(port);
        this.client = SocketChannel.open(this.hostAddress);
    }

    void transfer(final FileChannel channel, long position, long size) throws IOException {
        assert !Objects.isNull(channel);

        while (position < size) {
            position += channel.transferTo(position, Constants.TRANSFER_MAX_SIZE, this. ↩
                client);
        }
    }

    SocketChannel getChannel() {
        return this.client;
    }

    void close() throws IOException {
        this.client.close();
    }
}
```

line 11-17: we provide the FileChannel, `position` and `size` of the bytes to transfer from the given `channel` . A loop is started where attempts are made to transfer up to `Constants.TRANSFER_MAX_SIZE` in bytes from the provided Channel to the SocketChannel `client` . After each iteration the amount of bytes transferred is added to the `position` which then advances the cursor for the next transfer attempt.

`FileReceiver`

```
final class FileReceiver {

    private final int port;
    private final FileWriter fileWriter;
    private final long size;

    FileReceiver(final int port, final FileWriter fileWriter, final long size) {
        this.port = port;
        this.fileWriter = fileWriter;
        this.size = size;
    }

    void receive() throws IOException {
        SocketChannel channel = null;

        try (final ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()) {
            init(serverSocketChannel);

            channel = serverSocketChannel.accept();

            doTransfer(channel);
        } finally {
            if (!Objects.isNull(channel)) {
                channel.close();
            }

            this.fileWriter.close();
        }
    }

    private void doTransfer(final SocketChannel channel) throws IOException {
        assert !Objects.isNull(channel);

        this.fileWriter.transfer(channel, this.size);
    }

    private void init(final ServerSocketChannel serverSocketChannel) throws IOException {
        assert !Objects.isNull(serverSocketChannel);

        serverSocketChannel.bind(new InetSocketAddress(this.port));
    }
}
```

The `FileReceiver` is a mini server that listens for incoming connections on the `localhost` and upon connection, accepts it and initiates a transfer of bytes from the accepted Channel via the `FileWriter` abstraction to the encapsulated FileChannel within the `FileWriter`. The `FileReceiver` is only responsible for receiving the bytes via socket and then delegates transferring them to the `FileWriter`.

`FileWriter`

```
final class FileWriter {

    private final FileChannel channel;

    FileWriter(final String path) throws IOException {
        if (StringUtils.isEmpty(path)) {
            throw new IllegalArgumentException("path required");
        }

        this.channel = FileChannel.open(Paths.get(path), StandardOpenOption.WRITE, ↩
            StandardOpenOption.CREATE_NEW);
```

```
    }

    void transfer(final SocketChannel channel, final long bytes) throws IOException {
        assert !Objects.isNull(channel);

        long position = 0l;
        while (position < bytes) {
            position += this.channel.transferFrom(channel, position, Constants. ↩
                TRANSFER_MAX_SIZE);
        }
    }

    int write(final ByteBuffer buffer, long position) throws IOException {
        assert !Objects.isNull(buffer);

        int bytesWritten = 0;
        while(buffer.hasRemaining()) {
            bytesWritten += this.channel.write(buffer, position + bytesWritten);
        }

        return bytesWritten;
    }

    void close() throws IOException {
        this.channel.close();
    }
}
```

The `FileWriter` is simply charged with transferring the bytes from a SocketChannel to it's encapsulated FileChannel. As before, the transfer process is a loop which attempts to transfer up to `Constants.TRANSFER_MAX_SIZE` bytes with each iteration.

6.5.2.1 Asynchronous large file transfer

The following code snippets demonstrate transferring a large file from one remote location to another via an asynchronous receiver `FileReceiverAsync`.

`OnComplete`

```
@FunctionalInterface
public interface OnComplete {

    void onComplete(FileWriterProxy fileWriter);
}
```

The `OnComplete` interface represents a callback abstraction that we pass to our `FileReceiverAsync` implementation with the purposes of executing this once a file has been successfully and thoroughly transferred. We pass a `FileWriterProxy` to the `onComplete(...)` and this can server as context when executing said method.

`FileWriterProxy`

```
final class FileWriterProxy {

    private final FileWriter fileWriter;
    private final AtomicLong position;
    private final long size;
    private final String fileName;

    FileWriterProxy(final String path, final FileMetaData metaData) throws IOException {
```

```
        assert !Objects.isNull(metaData) && StringUtils.isNotEmpty(path);

        this.fileWriter = new FileWriter(path + "/" + metaData.getFileName());
        this.position = new AtomicLong(0l);
        this.size = metaData.getSize();
        this.fileName = metaData.getFileName();
    }

    String getFileName() {
        return this.fileName;
    }

    FileWriter getFileWriter() {
        return this.fileWriter;
    }

    AtomicLong getPosition() {
        return this.position;
    }

    boolean done() {
        return this.position.get() == this.size;
    }
}
```

The `FileWriterProxy` represents a proxy abstraction that wraps a `FileWriter` and encapsulates `FileMetaData` . All of this is needed when determining what to name the file, where to write the file and what the file size is so that we know when the file transfer is complete. During transfer negotiation this meta information is compiled via a custom protocol we implement before actual file transfer takes place.

`FileReceiverAsync`

```
final class FileReceiverAsync {

    private final AsynchronousServerSocketChannel server;
    private final AsynchronousChannelGroup group;
    private final String path;
    private final OnComplete onFileComplete;

    FileReceiverAsync(final int port, final int poolSize, final String path, final  ←
        OnComplete onFileComplete) {
        assert !Objects.isNull(path);

        this.path = path;
        this.onFileComplete = onFileComplete;

        try {
            this.group = AsynchronousChannelGroup.withThreadPool(Executors. ←
                newFixedThreadPool(poolSize));
            this.server = AsynchronousServerSocketChannel.open(this.group).bind(new  ←
                InetSocketAddress(port));
        } catch (IOException e) {
            throw new IllegalStateException("unable to start FileReceiver", e);
        }
    }

    void start() {
        accept();
    }

    void stop(long wait) {
```

```
        try {
            this.group.shutdown();
            this.group.awaitTermination(wait, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
            throw new RuntimeException("unable to stop FileReceiver", e);
        }
    }

    private void read(final AsynchronousSocketChannel channel, final FileWriterProxy proxy) ↩
         {
        assert !Objects.isNull(channel) && !Objects.isNull(proxy);

        final ByteBuffer buffer = ByteBuffer.allocate(Constants.BUFFER_SIZE);
        channel.read(buffer, proxy, new CompletionHandler<Integer, FileWriterProxy>() {

            @Override
            public void completed(final Integer result, final FileWriterProxy attachment) {
                if (result >= 0) {
                    if (result > 0) {
                        writeToFile(channel, buffer, attachment);
                    }

                    buffer.clear();
                    channel.read(buffer, attachment, this);
                } else if (result < 0 || attachment.done()) {
                    onComplete(attachment);
                    close(channel, attachment);
                }
            }

            @Override
            public void failed(final Throwable exc, final FileWriterProxy attachment) {
                throw new RuntimeException("unable to read data", exc);
            }
        });
    }

    private void onComplete(final FileWriterProxy proxy) {
        assert !Objects.isNull(proxy);

        this.onFileComplete.onComplete(proxy);
    }

    private void meta(final AsynchronousSocketChannel channel) {
        assert !Objects.isNull(channel);

        final ByteBuffer buffer = ByteBuffer.allocate(Constants.BUFFER_SIZE);
        channel.read(buffer, new StringBuffer(), new CompletionHandler<Integer,  ↩
            StringBuffer>() {

            @Override
            public void completed(final Integer result, final StringBuffer attachment) {
                if (result < 0) {
                    close(channel, null);
                } else {

                    if (result > 0) {
                        attachment.append(new String(buffer.array()).trim());
                    }

                    if (attachment.toString().contains(Constants.END_MESSAGE_MARKER)) {
```

```
                        final FileMetaData metaData = FileMetaData.from(attachment.toString ↩
                            ());
                        FileWriterProxy fileWriterProxy;

                        try {
                            fileWriterProxy = new FileWriterProxy(FileReceiverAsync.this. ↩
                                path, metaData);
                            confirm(channel, fileWriterProxy);
                        } catch (IOException e) {
                            close(channel, null);
                            throw new RuntimeException("unable to create file writer proxy" ↩
                                , e);
                        }
                    } else {
                        buffer.clear();
                        channel.read(buffer, attachment, this);
                    }
                }
            }

            @Override
            public void failed(final Throwable exc, final StringBuffer attachment) {
                close(channel, null);
                throw new RuntimeException("unable to read meta data", exc);
            }
        });
    }

    private void confirm(final AsynchronousSocketChannel channel, final FileWriterProxy  ↩
        proxy) {
        assert !Objects.isNull(channel) && !Objects.isNull(proxy);

        final ByteBuffer buffer = ByteBuffer.wrap(Constants.CONFIRMATION.getBytes());
        channel.write(buffer, null, new CompletionHandler<Integer, Void>() {

            @Override
            public void completed(final Integer result, final Void attachment) {
                while (buffer.hasRemaining()) {
                    channel.write(buffer, null, this);
                }

                read(channel, proxy);
            }

            @Override
            public void failed(final Throwable exc, final Void attachment) {
                close(channel, null);
                throw new RuntimeException("unable to confirm", exc);
            }

        });
    }

    private void accept() {
        this.server.accept(null, new CompletionHandler() {
            public void completed(final AsynchronousSocketChannel channel, final Void  ↩
                attachment) {

                // Delegate off to another thread for the next connection.
                accept();

                // Delegate off to another thread to handle this connection.
```

```
                meta(channel);
            }

            public void failed(final Throwable exc, final Void attachment) {
                throw new RuntimeException("unable to accept new connection", exc);
            }
        });
    }

    private void writeToFile(final AsynchronousSocketChannel channel, final ByteBuffer ↩
        buffer, final FileWriterProxy proxy) {
        assert !Objects.isNull(buffer) && !Objects.isNull(proxy) && !Objects.isNull(channel ↩
            );

        try {
            buffer.flip();

            final long bytesWritten = proxy.getFileWriter().write(buffer, proxy.getPosition ↩
                ().get());
            proxy.getPosition().addAndGet(bytesWritten);
        } catch (IOException e) {
            close(channel, proxy);
            throw new RuntimeException("unable to write bytes to file", e);
        }
    }

    private void close(final AsynchronousSocketChannel channel, final FileWriterProxy proxy ↩
        ) {
        assert !Objects.isNull(channel);

        try {

            if (!Objects.isNull(proxy)) {
                proxy.getFileWriter().close();
            }
            channel.close();
        } catch (IOException e) {
            throw new RuntimeException("unable to close channel and FileWriter", e);
        }
    }
```

The `FileReceiverAsync` abstraction builds upon the idiomatic use of AsynchronousChannels demonstrated in this tutorial.

6.6 Running the program

The program can be run from within the IDE, using the normal JUnit Runner or from the command line using maven. Ensure that the test resources (large source files and target directories exist). Running tests from command line

```
mvn clean install
```

You can edit these in the `AbstractTest` and `FileCopyAsyncTest` classes. Fair warning the `FileCopyAsyncTest` can run for a while as it is designed to copy two large files asynchronously, and the test case waits on a CountDownLatch without a max wait time specified.

I ran the tests using the "spring-tool-suite-3.8.1.RELEASE-e4.6-linux-gtk-x86_64.tar.gz" file downloaded from the SpringSource website. This file is approximately 483mb large and below are my test elapsed times. (using a very old laptop).

```
Test elapsed time
```

```
Running com.javacodegeeks.nio.large_file_transfer.remote.FileCopyTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.459 sec - in com. ↩
    javacodegeeks.nio.large_file_transfer.remote.FileCopyTest
Running com.javacodegeeks.nio.large_file_transfer.remote.FileCopyAsyncTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 26.423 sec - in com. ↩
    javacodegeeks.nio.large_file_transfer.remote.FileCopyAsyncTest
Running com.javacodegeeks.nio.large_file_transfer.local.FileCopyTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.562 sec - in com. ↩
    javacodegeeks.nio.large_file_transfer.local.FileCopyTest
```

6.7 Summary

In this tutorial, we demonstrated how to transfer a large file from one point to another. This was showcased via a local copy and a remote transfer via sockets. We went one step further and demonstrated transferring a large file from one remote location to another via an asynchronous receiving node.

6.8 Download the source code

This was a Java NIO Large File Transfer tutorial

Download

You can download the full source code of this example here: **Java Nio Large File Transfer**

Chapter 7

Java Nio Asynchronous Channels Tutorial

This article is a tutorial on the Asynchronous Channels API which was released as part of Java 7. The API can be viewed here. The example code will demonstrate use of the core abstractions of this API and will capture the essence of using the API.

7.1 Introduction

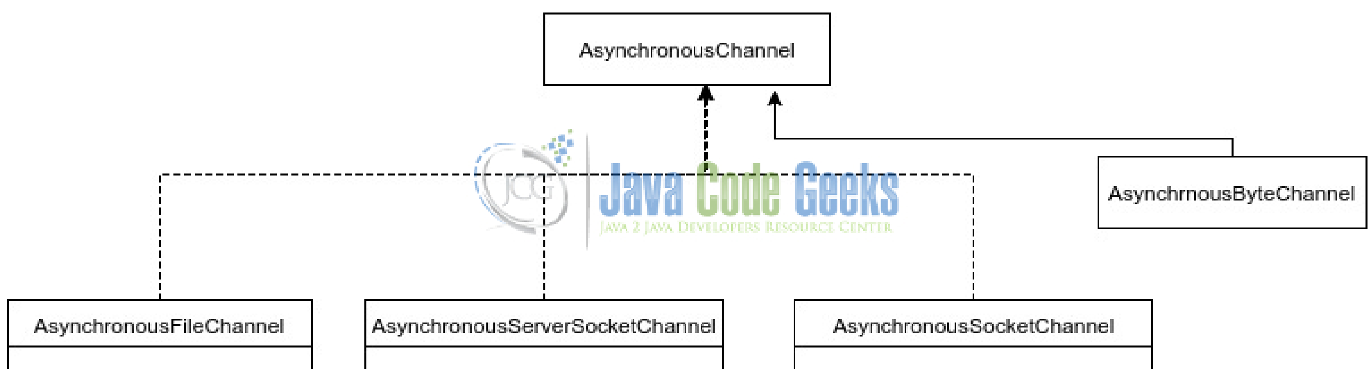

Figure 7.1: Core abstractions of Asynchronous Channels API

The Asynchronous Channels API's supplemented the core Java NIO API's with additional functionality in the Java 7 release. Coined NIO.2 the supplement provided many utilities for NIO usage but the crown jewel was the AsynchronousChannel API's.

A common phrase thrown around when discussing Java NIO is "non-blocking" but now one gets to add the word "asynchronous" as well. This can lead to a wonderful ice breaker in the form of "non-blocking asynchronous IO".

What a mouthful and even I had difficulty thoroughly digesting and understanding that, but I feel it important to understand what that phrase means and how it relates to the AsynchronousChannel API's.

- Asynchronous IO is where an interface or API allows us to provide call back code, to be executed when a particular IO operation completes. This is where the AsynchronousChannel class and much of it's hierarchy come into play.
- Non blocking IO is where an IO operation will return immediately either with data, an error or no data. ie: When reading from a non-blocking channel, either the number of bytes read is returned or `-1` meaning nothing more to read or an exception is thrown if some invalid state is encountered. Java NIO in JDK 1.4 introduced us to the Selector which was an abstraction that allowed us to leverage non-blocking IO.

AsynchronousChannel instances proxy IO operations and provide a means for notifying the program when said operations complete.

7.2 Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or Linux will do fine)

7.3 API Interaction

When interacting (reading, writing or connecting) with the AsynchronousChannel API the results of these interactions result in "Future" results or "Complete" results.

- Future results are encapsulated in the Future API. This facilitates a "pending" result which can later be retrieved or acted on by leveraging the Future API.
- Complete results are "hooked" into by supplying a CompletionHandler implementation to the method call (read, write or connect).

7.4 AsynchronousChannel

The AsynchronousChannel is a specialization of the Channel interface that enhances IO operations (read, write, connect or close) with asynchronous abilities. Calling `read()` or `write()` or `connect()` on the AsynchronousChannel produces a different result and provides a different method signature to that of the conventional NIO Channel implementations. This varies by way of:

- Returning a Future from a read, write or connect invocation
- Allowing a CompletionHandler implementation to be injected at method invocation to facilitate call back style processing when the IO event completes normally or via error.
- All methods being asynchronous return immediately and delegate processing of the IO operation to the kernel, with the instruction of being notified when the IO operation completes, either by way of the CompletionHandler implementation being invoked or the Future getting it's result.

Calling `close()` simply closes the Channel asynchronously and ensures any outstanding IO operations terminate via an AsynchronousCloseException. Typically AsynchronousChannel implementations are associated with an explicit Thread pool by way of the AsynchronousChannelGroup implementation which effectively manages all Channel instances associated with it and provides Thread resources for all Channel instances it manages to handle their IO operations. An AsynchronousChannel implementation is associated with the AsynchronousChannelGroup at construction time via the following:

- AsynchronousSocketChannel: `AsynchronousSocketChannel.open(group)`
- AsynchronousServerSocketChannel: `AsynchronousServerSocketChannel.open(group)`

What follows now are simple snippets of CompletionHandler and Future usage of the AsynchronousChannel API.

`CompletionHandler example`

```
channel.connect(remoteAddress, context, new CompletionHandler<Void, String>() {
    @Override
    public void completed(final Void result, final Object attachment) {...
    }

    @Override
    public void failed(final Throwable exc, final Object attachment) {...
    }
});
```

- line 1: `connect(...)` is called on the AsynchronousChannel (AsynchronousSocketChannel) implementation. A remote address to connect to is supplied, a context specific object `context` is supplied and a callback CompletionHandler implementation is supplied. The context specific object represents a method to propagate context to the CompletionHandler implementation, particularly if the CompletionHandler implementation is used in a stateless fashion, ie: shared. This "context" manifests itself as the `attachment` object in the CompletionHandler implementation. An example of propagating context could be when trying to assemble a complete client request that was spread across multiple Channel `read(...)` invocations.
- line 3: this method is called upon normal completion of the IO operation (read, write, connect). In the case of `connect(...)` the first argument to the method signature is `Void` whereas with `read(...)` and `write(...)` the first argument is the number of bytes read or written from the completed IO operation. The `attachment` argument is the manifestation of the `context` argument from line 1 and can be used to establish "context" in a stateless CompletionHandler implementation.
- line 7: this method is called upon abnormal (erroneous) completion of an IO operation (read, write, connect). In all IO operations (read, write, connect) the method signature is the same providing us with the reason for failure in the form of a Throwable instance and of course the `context` argument.

Future write example using AsynchronousFileChannel

```
final Future result = channel.write(buffer, filePosition);
```

- line 1: this method is called with a Buffer implementation and a position in the file to write from. The implementation will start writing from the given `position` and continue writing bytes until the `buffer` is written out to file. The Future return value encapsulates the pending result of how many bytes were written to the file.

7.5 AsynchronousByteChannel

The AsynchronousByteChannel is a specialization of the AsynchronousChannel that reads and write bytes. It is implemented concretely by AsynchronousSocketChannel.

7.6 AsynchronousFileChannel

The AsynchronousFileChannel class is an asynchronous channel for reading, writing, and manipulating a file via ByteBuffers. Creating an AsynchronousFileChannel instance can be done via the two static `open(...)` methods:

`AsynchronousFileChannel open method#1`

```
public static AsynchronousFileChannel open(Path file, OpenOption... options);
```

`AsynchronousFileChannel open method#2`

```
public static AsynchronousFileChannel open(Path file, Set<? extends OpenOption> options, ↩
    ExecutorService executor, FileAttribute<?>... attrs);
```

OpenOption, more specifically StandardOpenOption enumerates the various modes / options the File is manipulated with, eg: `OPEN, READ, WRITE` etc and will naturally have an effect on what can be done with the file. Interestingly enough the Channel does not allow for an AsynchronousChannelGroup at construction but rather an ExecutorService to allow for explicit thread resource usage as opposed to a default thread group.

The AsynchronousFileChannel provides methods for locking files, truncating files, and retrieving file sizes. Read and write actions expect a ByteBuffer and a position, position being the location in the file to start reading or writing from, illustrating one of the main differences between the FileChannel class. The position being required for multithreaded use. This type of Channel is safe for multithreaded use and multiple IO (read and write) operations can be outstanding at the same time but their order of execution is undetermined, be aware of this!

FileLocks, another feature of AsynchronousFileChannels, are as the name implies but can vary by type of lock and operating system support.

- shared lock - meaning the lock can be shared provided the lock granularity is "shared". Also the Channel has to be opened in `READ` mode otherwise a NonReadableChannelException will be thrown.
- exclusive lock - only one lock is held. Also the Channel must be opened in `write` mode otherwise a NonWritableChannelException will be thrown.

FileLocks can also lock the entire file or regions of the file based on position. eg: Locking a file from position 10 would imply locking the file from the 10th byte through to the end of the file.

7.6.1 AsynchronousFileChannel Exceptions

- OverlappingFileLockException: When a lock is already held for the File in question. Remember lock type will have an effect on this exception happening or not.
- NonReadableChannelException: When the Channel is not opened for reading.
- NonWritableChannelException: When the Channel is not opened for writing.
- AsynchronousCloseException: All pending asynchronous IO operations terminate with this when the Channel has been closed.
- ClosedChannelException: When the Channel is closed and you try to initiate an IO operation.

The following code snippets demonstrate use of the AsynchronousFileChannel via the Future API for reading, writing and locking. The samples are driven from unit tests all of which can be sourced from the download for this article.

`AsynchronousFileChannel read sample`

```
public String read(final String path) {
...
        try (AsynchronousFileChannel channel = AsynchronousFileChannel.open(pathToFile,  ←
            StandardOpenOption.READ)) {
                result = read(channel, ByteBuffer.allocate(Constants.BUFFER_SIZE), new  ←
                    StringBuilder(), START_POS);
        } catch (IOException e) {
                throw new RuntimeException(UNABLE_TO_READ_CONTENTS, e);
        }
...
}

private String read(final AsynchronousFileChannel channel, final ByteBuffer buffer, final  ←
    StringBuilder contents, final long filePosition) {
        assert !Objects.isNull(channel) && !Objects.isNull(buffer) && !Objects.isNull( ←
            contents);

        final Future<Integer> result = channel.read(buffer, filePosition);
        try {
```

```
                final int bytesRead = result.get();
                if (bytesRead != -1) {
                        contents.append(new String(buffer.array()).trim());

                        buffer.clear();
                        return read(channel, buffer, contents, filePosition + bytesRead);
                } else {
                        return contents.toString();
                }
        } catch (InterruptedException | ExecutionException e) {
                throw new RuntimeException(UNABLE_TO_READ_CONTENTS, e);
        }
}
```

- line 3-4: creates the AsynchronousFileChannel and calls the recursive `read` method with a newly constructed ByteBuffer.
- line 11: the method signature takes the position to continue reading from in each recursive routine.
- line 14: gets the result of the read, the number of bytes, blocks until the result is available.
- line 18: appends the contents of what was read from the ByteBuffer to the StringBuilder.
- line 20-21: clears the ByteBuffer prior to the next invocation and calls the method recursively again.

AsynchronousFileChannel write sample

```
public void write(final String path, final String contents) {
        final Path pathToFile = Paths.get(path);

        try (AsynchronousFileChannel channel = AsynchronousFileChannel.open(pathToFile,  ↩
            StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {
                final ByteBuffer buffer = ByteBuffer.wrap(contents.getBytes());

                write(channel, buffer, START_POS);
        } catch (IOException e) {
                throw new RuntimeException(UNABLE_TO_WRITE_CONTENTS, e);
        }
}

private void write(final AsynchronousFileChannel channel, final ByteBuffer buffer, final  ↩
    long filePosition) {
        assert !Objects.isNull(channel) && !Objects.isNull(buffer);

        final Future<Integer> result = channel.write(buffer, filePosition);
        try {
                final int bytesWritten = result.get();
                while (buffer.hasRemaining()) {
                        buffer.compact();
                        write(channel, buffer, bytesWritten + filePosition);
                }
        } catch (InterruptedException | ExecutionException e) {
                throw new RuntimeException(UNABLE_TO_WRITE_CONTENTS, e);
        }
}
```

- line 2: gets the Path object to the file.
- line 4-5: creates the AsynchronousFileChannel (ensures the file is created if not already via options) and also creates the ByteBuffer for the contents to write.
- line 7: calls write with the position of the file to start writing from.

- line 16: gets the result of the write, the number of bytes written.
- line 18-21: loops while their are still bytes in the ByteBuffer and writes it out to file.

AsynchronousFileChannel lock sample

```
@Test
public void testExclusiveLock() throws IOException, InterruptedException,  ↩
    ExecutionException {
        try (AsynchronousFileChannel channel = AsynchronousFileChannel.open(Paths.get(this. ↩
            filePath), StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {
             final FileLock lock = channel.lock().get();

             assertTrue("Lock is not exclusive", !lock.isShared());
        }
}

@Test
public void testSharedLock() throws IOException, InterruptedException, ExecutionException {
        try (AsynchronousFileChannel channel = AsynchronousFileChannel.open(Paths.get(this. ↩
            filePath), StandardOpenOption.READ, StandardOpenOption.CREATE)) {
             final FileLock lock = channel.lock(0, 0L, true).get();

             assertTrue("Lock is exclusive", lock.isShared());
        }
}

@Test(expected = OverlappingFileLockException.class)
public void testOverlappingLock() {
        final CountDownLatch innerThreadLatch = new CountDownLatch(1);
        final CountDownLatch testThreadLatch = new CountDownLatch(1);

        try (AsynchronousFileChannel channel = AsynchronousFileChannel.open(Paths.get(this. ↩
            filePath), StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {

                new Thread() {
                        public void run() {
                                try {
                                        channel.lock().get();
                                        innerThreadLatch.countDown();
                                        testThreadLatch.await();
                                } catch (OverlappingFileLockException | ExecutionException  ↩
                                    | InterruptedException e) {
                                        throw new RuntimeException("Unable to get lock on  ↩
                                            file for overlapping lock test", e);
                                }
                        }
                }.start();

                innerThreadLatch.await();
                channel.lock().get();
        } catch (InterruptedException | ExecutionException | IOException e) {
                throw new RuntimeException(e);
        } finally {
                testThreadLatch.countDown();
        }
}
```

- line 3: create the AsynchronousFileChannel ensuring we create the file if it does not already exist.
- line 4,6,13,15: obtains a FileLock in either shared or exclusive modes and validates that state.

- The final test, although not highlighted, is a test to prove an overlapping lock exception where two threads compete for the same lock. Latches are used to ensure co-ordination between their competitive spirits. The takeaway from this last test is that inside the same JVM process all Threads share the same locks, therefore trying to aquire an already held lock (exclusive) will result in an OverlappingFileLockException. Using file locks to synchronize thread access to file regions will not work, however in concert with normal thread synchronization and file locks one can achive co-ordinated access to files between threads and processes.

7.7 AsynchronousServerSocketChannel

The AsynchronousServerSocketChannel is a Channel for accepting new socket connections. An AsynchronousServerSocketChannel can be created via the two static `open(...)` methods:

`AsynchronousServerSocketChannel open method #1`

```
public static AsynchronousServerSocketChannel open(AsynchronousChannelGroup group) throws ↩
    IOException
```

`AsynchronousServerSocketChannel open method #2`

```
public static AsynchronousServerSocketChannel open() throws IOException
```

The AsynchronousChannelGroup is an abstraction that provides the AsynchronousServerSocketChannel with it's thread pool to handle it's IO operations asynchronously. The AsynchronousServerSocketChannel also implements the NetworkChannel interface which provides the ability to set channel SocketOption (more specifically StandardSocketOptions) values and to bind to SocketAddress values.

7.7.1 AsynchronousServerSocketChannel Exceptions

- AsynchronousCloseException: All outstanding IO operations on the Channel terminate with said Exception when the Channel has been closed.
- ClosedChannelException: Any new IO operations submitted after the Channel has been closed.
- NotYetBoundException: if `accept()` is called on a Channel instance that is not yet bound.
- ShutDownChannelGroupException: if the AsynchronousChannelGroup is already shutdown and a new IO operation is commenced.
- AcceptPendingException: if a thread calls `accept()` while another `accept()` call is still busy.

AsynchronousServerSocketChannel creation

```
...
private final AsynchronousServerSocketChannel server;
private final AsynchronousChannelGroup group;
...
public Server(final int port, final int poolSize, final String echo) {
        try {
                this.group = AsynchronousChannelGroup.withThreadPool(Executors. ↩
                    newFixedThreadPool(poolSize));
                this.server = AsynchronousServerSocketChannel.open(this.group).bind(new ↩
                    InetSocketAddress(port));
...
```

- line 7-8: The AsynchronousServerSocketChannel is created with a AsynchronousChannelGroup supplied and a specified `poolSize`.

AsynchronousServerSocketChannel accepting connection with CompletionHandler

```
...
this.server.accept(requestKey, new CompletionHandler<AsynchronousSocketChannel, String>() {
        public void completed(final AsynchronousSocketChannel channel, final String ↩
            attachment) {

                // Delegate off to another thread for the next connection.
                accept(IdGenerator.generate());

                // Delegate off to another thread to handle this connection.
                Server.this.read(channel, attachment);
        }

        public void failed(final Throwable exc, final String attachment) {
                System.out.println(String.format("Server: Failed to accept connection in ↩
                    thread %s", Thread.currentThread().getName()));
                exc.printStackTrace();
        }
});
```

- line 2-3: `accept()` is called and a `requestKey` and a CompletionHandler is supplied to handle the incoming connection. The `requestKey` is a unique String generated for the purposes of establishing context in the multithreaded / asynchronous Channel. The `attachment` in the `completed(...)` method call represents context and is actually the `requestKey` being ushered into the CompletionHandler from the earlier `accept()` call.
- line 6: We are non-blocking and it is important to delegate off as soon as possible to handle the next incoming connection, a unique key is generated (requestKey) which will later become the `attachment` (context) for the CompletionHandler.
- line 9: We handle the current connection by calling `read(...)` which will take the `attachment` for context and ultimately create a new CompletionHandler for the purposes of reading the client request.
- line 12: If the IO operation fails, this method is called with the context and the reason for failure.

7.8 AsynchronousSocketChannel

The AsynchronousSocketChannel is an asynchronous Channel for connected sockets. Such a Channel has the ability to connect to a remote address, read and write asynchronously, with the Future and CompletionHandler abstractions being provided as a means for manipulating the outcomes of said IO operations. As per the AsynchronousServerSocketChannel, the Asynchronous-SocketChannel also implements the NetworkChannel interface which provides the ability to set channel SocketOption (more specifically StandardSocketOptions) values and to bind to SocketAddress values.

An AsynchronousSocketChannel can be opened via the two static `open(...)` methods: AsynchronousSocketChannel open method #1

```
public static AsynchronousSocketChannel open(AsynchronousChannelGroup group) throws ↩
    IOException
```

`AsynchronousSocketChannel open method #2`

```
public static AsynchronousSocketChannel open() throws IOException
```

7.8.1 AsynchronousSocketChannel Exceptions

- AsynchronousCloseException: All pending asynchronous IO operations terminate with this when the Channel has been closed.
- ClosedChannelException: When the Channel is closed and you try to initiate an IO operation.

- NotYetConnectedException: When an IO operation is attempted on a Channel that is not yet connected.
- ReadPendingException: When a read is attempted before a previous read operation has completed.
- WritePendingException: When a write is attempted before a previous write operation has completed.
- ConnectionPendingException: If a connect operation is already in progress for the given Channel.
- AlreadyConnectedException: if a connect is attempted on a Channel that is already connected.

AsynchronousSocketChannel creation and connection

```
...
for (int i = 0; i < this.numConnections; i++) {
        AsynchronousSocketChannel client;
        try {
                client = AsynchronousSocketChannel.open(this.group);
                connect(client, IdGenerator.generate());
        } catch (IOException e) {
                throw new RuntimeException("Client: Unable to start clients", e);
        }
}
...
private void connect(final AsynchronousSocketChannel channel, final String requestId) {
        channel.connect(this.remoteAddress, requestId, new CompletionHandler<Void, String ↩
            >() {

            @Override
            public void completed(final Void result, final String attachment) {
                System.out.println(String.format("Client: Connect Completed in thread %s",  ↩
                    Thread.currentThread().getName()));
                updateMessageCache(attachment, StringUtils.EMPTY, Client.this.messageCache) ↩
                    ;

                write(channel, attachment);
            }

            @Override
            public void failed(final Throwable exc, final String attachment) {
                System.out.println(String.format("Client: Connect Failed in thread %s",  ↩
                    Thread.currentThread().getName()));
                exc.printStackTrace();

                Client.this.latch.countDown();
                closeChannel(channel);
            }
        });
}
...
private void write(final AsynchronousSocketChannel channel, final String requestId) {
        assert !Objects.isNull(channel);

        final ByteBuffer contents = create(Constants.BUFFER_SIZE);
        contents.put(requestId.getBytes());
        contents.put(Constants.END_MESSAGE_MARKER.getBytes());
        contents.flip();

        channel.write(contents, requestId, new CompletionHandler<Integer, String>() {

            @Override
            public void completed(final Integer result, final String attachment) {
                System.out.println(String.format("Client: Write Completed in thread %s",  ↩
                    Thread.currentThread().getName()));
```

```
                read(channel, attachment);
            }
```

- line 5: the AsynchronousSocketChannel is created supplying an AsynchronousChannelGroup upon creation for threading purposes.
- line 6: a connection is attempted for the Channel supplying a unique String value as context for the connection.
- line 12-13: `connect(...)` is called and in particular the Channel's ' `connect(...)` is invoked passing a `remoteAddress` `requestId` and a CompletionHandler to handle the outcome of the IO operation. The `requestId` is the context variable and manifests itself as the `attachment` in the CompletionHandler.
- line 20: `write(...)` is called passing the Channel upon which the connection was established and the context (attachment). So effectively upon connection competion we commence an IO operation and as this is a client in a client server program the first call of action is to write a request to the Server.
- line 29: we close the Channel upon failure to connect.
- line 42: `write(...)` is called on the Channel supplying a ByteBuffer as source, a context variable (requestId) and a CompletionHandler.

7.9 Summary

In this tutorial we have covered the main abstractions in the asynchronous channels API, specifically focusing on the types of AsnchronousChannel implementations, what they are and how to use them.

We have seen under what circumstances behavior could become exceptional (Exceptions) and how to manipulate the outcome of IO operations on said Channels via "pending" and complete results.

7.10 Download the source code

This was a Java NIO Asynchronous Channels tutorial

Download

You can download the full source code of this example here: **Java NIO Asynchronous Channels**

Chapter 8

Java Nio Echo Server Tutorial

This article is a tutorial on implementing a simple Java NIO "echo server". This example will take the form of a rather simple client server application whereby a client or many clients will connect to a running server and post message(s) to the server which will in turn be "echoed" back to the respective clients.

8.1 Introduction

This article builds on two earlier articles on the subject of Java NIO, namely "Java Nio Tutorial for Beginners" and "Java Nio Asynchronous Channels Tutorial" where we implement a simple "echo server" using some of the abstractions and techniques discussed in the earlier articles.

8.2 Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or Linux will do fine)

8.3 Overview

A server process is started with a port property specified at runtime. This server process listens for incoming connections from potential client processes. Once an inbound connection from a client is detected the server process is notified of this and the connection is accepted. The client is then able to send a message to the server. Upon receipt of this message the server is once again notified and the server begins to read the incoming request, which when complete is subsequently sent back on the same connection to the client, hence the "echo".

8.4 The EchoServer

What follows are the code snippets of the all the abstractions used in this EchoServer implementation.

8.4.1 ChannelWriter

ChannelWriter

```
public interface ChannelWriter {

    default void doWrite(final ByteBuffer buffer, final SocketChannel channel) throws ←
        IOException {
        if (Objects.isNull(buffer) || Objects.isNull(channel)) {
            throw new IllegalArgumentException("Required buffer and channel.");
        }

        while (buffer.hasRemaining()) {
            channel.write(buffer);
        }
    }
}
```

- line 8: we ensure their is still bytes remaining between the current position and the limit
- line 9: we attempt to write the remaining bytes in the ByteBuffer to the Channel

8.4.2 Client

Client

```
public final class Client implements ChannelWriter {

    private final InetSocketAddress hostAddress;

    public static void main(final String[] args) {
        if (args.length < 2) {
            throw new IllegalArgumentException("Expecting two arguments in order (1) port ←
                (2) message eg: 9999 \\"Hello world\\".");
        }

        new Client(Integer.valueOf(args[0])).start(args[1]);
    }

    private Client(final int port) {
        this.hostAddress = new InetSocketAddress(port);
    }

    private void start(final String message) {
        assert StringUtils.isNotEmpty(message);

        try (SocketChannel client = SocketChannel.open(this.hostAddress)) {

            final ByteBuffer buffer = ByteBuffer.wrap((message + Constants. ←
                END_MESSAGE_MARKER).trim().getBytes());

            doWrite(buffer, client);

            buffer.flip();

            final StringBuilder echo = new StringBuilder();
            doRead(echo, buffer, client);
```

```
            System.out.println(String.format("Message :\\t %s \\nEcho    :\\t %s", message, ↩
                echo.toString().replace(Constants.END_MESSAGE_MARKER, StringUtils.EMPTY)));
        } catch (IOException e) {
            throw new RuntimeException("Unable to communicate with server.", e);
        }
    }

    private void doRead(final StringBuilder data, final ByteBuffer buffer, final  ↩
        SocketChannel channel) throws IOException {
        assert !Objects.isNull(data) && !Objects.isNull(buffer) && !Objects.isNull(channel) ↩
            ;

        while (channel.read(buffer) != -1) {
            data.append(new String(buffer.array()).trim());
            buffer.clear();
        }
    }
}
```

- line 20: using `try(...)` (with resources) we open a SocketChannel to the configured InetSocketAddress
- line 22: we create a ByteBuffer wrapping the contents of the specified `message`
- line 24: we call `write(...)` passing the ByteBuffer and the SocketChannel
- line 26: flipping the ByteBuffer to initialize the position and limit for reading
- line 29: call `read(...)` passing the StringBuilder (for placing the read contents into), the ByteBuffer and the SocketChannel
- line 37-44: we ensure we read everything from the Server

8.4.3 Server

`Server`

```
public final class Server implements ChannelWriter {

    private static final int BUFFER_SIZE = 1024;

    private final int port;
    private final Map<SocketChannel, StringBuilder> session;

    public static void main(final String[] args) {
        if (args.length < 1) {
            throw new IllegalArgumentException("Expecting one argument (1) port.");
        }

        new Server(Integer.valueOf(args[0])).start();
    }

    private Server(final int port) {
        this.port = port;
        this.session = new HashMap<>();
    }

    private void start() {
        try (Selector selector = Selector.open(); ServerSocketChannel channel =  ↩
            ServerSocketChannel.open()) {
            initChannel(channel, selector);
```

```
            while (!Thread.currentThread().isInterrupted()) {
                if (selector.isOpen()) {
                    final int numKeys = selector.select();
                    if (numKeys > 0) {
                        handleKeys(channel, selector.selectedKeys());
                    }
                } else {
                    Thread.currentThread().interrupt();
                }
            }
        } catch (IOException e) {
            throw new RuntimeException("Unable to start server.", e);
        } finally {
            this.session.clear();
        }
    }

    private void initChannel(final ServerSocketChannel channel, final Selector selector)  ←
        throws IOException {
        assert !Objects.isNull(channel) && !Objects.isNull(selector);

        channel.socket().setReuseAddress(true);
        channel.configureBlocking(false);
        channel.socket().bind(new InetSocketAddress(this.port));
        channel.register(selector, SelectionKey.OP_ACCEPT);
    }

    private void handleKeys(final ServerSocketChannel channel, final Set<SelectionKey> keys ←
        ) throws IOException {
        assert !Objects.isNull(keys) && !Objects.isNull(channel);

        final Iterator<SelectionKey> iterator = keys.iterator();
        while (iterator.hasNext()) {

            final SelectionKey key = iterator.next();
            try {
                if (key.isValid()) {
                    if (key.isAcceptable()) {
                        doAccept(channel, key);
                    } else if (key.isReadable()) {
                        doRead(key);
                    } else {
                        throw new UnsupportedOperationException("Key not supported by  ←
                            server.");
                    }
                } else {
                    throw new UnsupportedOperationException("Key not valid.");
                }
            } finally {
                if (mustEcho(key)) {
                    doEcho(key);
                    cleanUp(key);
                }

                iterator.remove();
            }
        }
    }

    private void doAccept(final ServerSocketChannel channel, final SelectionKey key) throws ←
         IOException {
        assert !Objects.isNull(key) && !Objects.isNull(channel);
```

```
        final SocketChannel client = channel.accept();
        client.configureBlocking(false);
        client.register(key.selector(), SelectionKey.OP_READ);

        // Create a session for the incoming connection
        this.session.put(client, new StringBuilder());
    }

    private void doRead(final SelectionKey key) throws IOException {
        assert !Objects.isNull(key);

        final SocketChannel client = (SocketChannel) key.channel();
        final ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);

        final int bytesRead = client.read(buffer);
        if (bytesRead > 0) {
            this.session.get(client).append(new String(buffer.array()).trim());
        } else if (bytesRead < 0) {
            if (mustEcho(key)) {
                doEcho(key);
            }

            cleanUp(key);
        }
    }

    private void doEcho(final SelectionKey key) throws IOException {
        assert !Objects.isNull(key);

        final ByteBuffer buffer = ByteBuffer.wrap(this.session.get(key.channel()).toString ↩
            ().trim().getBytes());

        doWrite(buffer, (SocketChannel) key.channel());
    }

    private boolean mustEcho(final SelectionKey key) {
        assert !Objects.isNull(key);

        return (key.channel() instanceof SocketChannel) && this.session.get((SocketChannel) ↩
             key.channel()).toString().contains(Constants.END_MESSAGE_MARKER);
    }

    private void cleanUp(final SelectionKey key) throws IOException {
        assert !Objects.isNull(key);

        this.session.remove((SocketChannel) key.channel());

        key.channel().close();
        key.cancel();
    }
}
```

- line 22: using `try(...)` (with resources) we open ServerSocketChannel and a Selector. The Selector will allow the Server to multiplex over *n* number of SelectableChannel instances (ie: connections)
- line 23: we initialize the ServerSocketChannel and register it with the Selector. We also express interest in the `SelectionKey.OP_ACCEPT` IO operation meaning that the ServerSocketChannel will only be interested accepting connections
- line 26: check that the Selector is still open

- line 27: call `select()` on the Selector, this is a blocking call and will only return when their are SelectionKey instances (expressing IO events)
- line 29: handle the Set of SelectionKey instances from the `select()` call for the given ServerSocketChannel
- line 45: allows binding to the port even if a previous connection on that same port is still in a `TIME_WAIT` state
- line 46: ensure our Channel is in non-blocking mode for use by our Selector
- line 47: bind at the address
- line 48: register the Channel with the Selector
- line 59: whilst processing the keys ensure the SelectionKey is valid
- line 61: accept a new connection
- line 63: read from the connection
- line 71-76: ensure that after every IO event is handled we check if we must echo back to the client and if necessary cleanup (close) recources etc. Ensure we remove the SelectionKey from the Set of SelectionKey instances otherwise we will continue to process stale events
- line 84-89: for every incoming SocketChannel connection ensure we set blocking to false and express interest in `SelectionKey.OP_READ` IO events and create a new session
- line 99-100: if something was read - add it to the session buffer
- line 101-106: if the end of the stream has been reach, echo, if required to and clean up resources

8.5 Example code

The attached sample code is a maven project and can be compiled by executing the following: `mvn clean install` in the project folder, assuming all packages / programs are installed. Then navigate to the `target/classes` folder within the project folder and execute the following:

`Start Server`

```
java com.javacodegeeks.nio.echoserver.Server 9999
```

`Start Client`

```
java com.javacodegeeks.nio.echoserver.Client 9999 "Hello world!"
```

substituting the `9999` with any port number of your choosing and the `Hello world!` with any message of your choosing. If successful you should see the following output:

```
Message :        Hello world!
Echo    :        Hello world!
```

substituting "Hello world!" with whatever message you specified at runtime.

8.6 Summary

This example is demonstrated using the Selector class to multiplex over *n* number of SelectableChannels and echo back any messages received from said Channels. The Selector allowed our Server to handle the incoming IO events from said SelectableChannels provided they were `SelectionKey.OP_ACCEPT` or `SelectionKey.OP_READ` ready. It managed a Session per connected Channel and disposed of said Channel once the echo was complete.

8.7 Download the source code

This was a Java NIO EchoServer tutorial.

Download

You can download the full source code of this example here: **Java NIO EchoServer tutorial**

Chapter 9

Java Nio ByteBuffer Example

This article is a tutorial on demonstrating the usage of the Java Nio ByteBuffer. All examples are done in the form of unit tests to easily prove the expectations of the API.

9.1 Introduction

The ByteBuffer class is an abstract class which also happens to extend Buffer and implement Comparable. A Buffer is simply a linear finite sized container for data of a certain primitive type. It exhibits the following properties:

- capacity: the number of elements it contains
- limit: the index of where the data it contains ends
- position : the next element to be read or written

ByteBuffer has these properties but also displays a host of semantic properties of it's own. According to the ByteBuffer API the abstraction defines six categories of operations. They are:

- `get(...)` and `put(...)` operations that operate relatively (in terms of the current position) and absolutely (by supplying an index)
- bulk `get(...)` operation done relatively (in terms of the current position) which will get a number of bytes from the ByteBuffer and place it into the argument `array` supplied to the `get(...)` operation
- bulk `put(...)` operation done absolutely by supplying an `index` and the content to be inserted
- absolute and relative `get(...)` and `put(...)` operations that get and put data of a specific primitive type, making it convenient to work with a specific primitive type when interacting with the ByteBuffer
- creating a "view buffer' or view into the underlying ByteBuffer by proxying the underlying data with a Buffer of a specific primitive type
- compacting, duplicating and slicing a ByteBuffer

A ByteBuffer is implemented by the `HeapByteBuffer` and MappedByteBuffer abstractions. `HeapByteBuffer` further specializes into `HeapByteBufferR` (R being read-only), which will very conveniently throw a ReadOnlyBufferException and should you try to mutate it via it's API. The MappedByteBuffer is an abstract class which is implemented by `DirectByteBuffer` . All of the `HeapByteBuffer` implementations are allocated on the heap (obviously) and thus managed by the JVM.

9.2 Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or Linux will do fine)

9.3 Overview

A ByteBuffer is created via the the two static factory methods:

- `allocate(int)` this will allocate a `HeapByteBuffer` with the capacity specified by the `int` argument
- `allocateDirect(int)` this will allocate a `DirectByteBuffer` with the capacity specified by the `int` argument

The ByteBuffer class affords us the luxury of a fluent interface through much of it's API, meaning most operations will return a ByteBuffer result. This way we can obtain a ByteBuffer by also wrapping a `byte []`, slicing a piece of another ByteBuffer, duplicating an existing ByteBuffer and performing `get(...)` and `put(...)` operations against an existing ByteBuffer. I encourage you to review the ByteBuffer API to understand the semantics of it's API.

So why the distinction between direct and non-direct? It comes down to allowing the Operating System to access memory addresses contiguously for IO operations (hence being able to shove and extract data directly from the memory address) as opposed to leveraging the indirection imposed by the abstractions in the JVM for potentially non-contiguous memory spaces. Because the JVM cannot guarantee contiguous memory locations for `HeapByteBuffer` allocations the Operating System cannot natively shove and extract data into these types of ByteBuffers. So generally the rule of thumb is should you be doing a lot of IO, then the best approach is to allocate directly and re-use the ByteBuffer. Be warned `DirectByteBuffer` instances are not subject to the GC.

9.4 Test cases

To ensure determinism we have been explicit about the Charset in use, therefore any encoding of bytes or decoding of bytes will use the explicit `UTF-16BE` Charset.

Relative Get and Put operations Test cases

```
public class RelativeGetPutTest extends AbstractTest {

    @Test
    public void get() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));

        final byte a = buffer.get();
        final byte b = buffer.get();

        assertEquals("Buffer position invalid", 2, buffer.position());
        assertEquals("'H' not the first 2 bytes read", "H", new String(new byte[] { a, b }, ←
             BIG_ENDIAN_CHARSET));
    }

    @Test
    public void put() {
```

```
        final ByteBuffer buffer = ByteBuffer.allocate(24);

        buffer.put("H".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("e".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("l".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("l".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("o".getBytes(BIG_ENDIAN_CHARSET));

        buffer.put(" ".getBytes(BIG_ENDIAN_CHARSET));

        buffer.put("e".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("a".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("r".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("t".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("h".getBytes(BIG_ENDIAN_CHARSET));
        buffer.put("!".getBytes(BIG_ENDIAN_CHARSET));

        assertEquals("Buffer position invalid", 24, buffer.position());

        buffer.flip();
        assertEquals("Text data invalid", "Hello earth!", byteBufferToString(buffer));
    }

    @Test
    public void bulkGet() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));
        final byte[] output = new byte[10];

        buffer.get(output);

        assertEquals("Invalid bulk get data", "Hello", new String(output,  ←
            BIG_ENDIAN_CHARSET));

        assertEquals("Buffer position invalid", 10, buffer.position());
    }

    @Test
    public void bulkPut() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));
        final byte[] output = new String("earth.").getBytes(BIG_ENDIAN_CHARSET);

        buffer.position(12);

        buffer.put(output);

        assertEquals("Buffer position invalid", 24, buffer.position());
        buffer.flip();
        assertEquals("Text data invalid", "Hello earth.", byteBufferToString(buffer));
    }

    @Test
    public void getChar() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));

        buffer.mark();

        final byte a = buffer.get();
        final byte b = buffer.get();
```

```
        buffer.reset();

        char value = buffer.getChar();

        assertEquals("Buffer position invalid", 2, buffer.position());

        assertEquals("'H' not the first 2 bytes read", "H", new String(new byte[] { a, b }, ↩
             BIG_ENDIAN_CHARSET));
        assertEquals("Value and byte array not equal", Character.toString(value), new  ↩
            String(new byte[] { a, b }, BIG_ENDIAN_CHARSET));
    }

    @Test
    public void putChar() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ↩
            BIG_ENDIAN_CHARSET));

        buffer.position(22);

        buffer.putChar('.');

        assertEquals("Buffer position invalid", 24, buffer.position());

        buffer.flip();
        assertEquals("Text data invalid", "Hello world.", byteBufferToString(buffer));
    }
}
```

The above suite of test cases demonstrate relative `get()` and `put()` operations. These have a direct effect on certain ByteBuffer attributes (position and data). In addition to being able to invoke these operations with `byte` arguments or receive `byte` arguments we also demonstrate usage of the `putChar()` and `getChar(...)` methods which conveniently act on the matching primitive type in question. Please consult the API for more of these convenience methods

Absolute Get and Put operations Test cases

```
public class AbsoluteGetPutTest extends AbstractTest {

    @Test
    public void get() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ↩
            BIG_ENDIAN_CHARSET));

        final byte a = buffer.get(0);
        final byte b = buffer.get(1);

        assertEquals("Buffer position invalid", 0, buffer.position());
        assertEquals("'H' not the first 2 bytes read", "H", new String(new byte[] { a, b }, ↩
             BIG_ENDIAN_CHARSET));
    }

    @Test
    public void put() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ↩
            BIG_ENDIAN_CHARSET));

        final byte[] period = ".".getBytes(BIG_ENDIAN_CHARSET);
        int idx = 22;
        for (byte elem : period) {
            buffer.put(idx++, elem);
        }

        assertEquals("Position must remian 0", 0, buffer.position());
```

```
        assertEquals("Text data invalid", "Hello world.", byteBufferToString(buffer));
    }

    @Test
    public void getChar() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));
        char value = buffer.getChar(22);

        assertEquals("Buffer position invalid", 0, buffer.position());
        assertEquals("Invalid final character", "!", Character.toString(value));
    }

    @Test
    public void putChar() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));
        buffer.putChar(22, '.');

        assertEquals("Buffer position invalid", 0, buffer.position());
        assertEquals("Text data invalid", "Hello world.", byteBufferToString(buffer));
    }
}
```

The above suite of test cases demonstrate usage of the absolute variants of the `get(...)` and `put(...)` operations. Interestingly enough, only the underlying data is effected (`put(...)`) as the position cursor is not mutated owing to the method signatures providing client code the ability to provide an index for the relevant operation. Again convenience methods which deal with the various primitive types are also provided and we demonstrate use of the `...Char(...)` variants thereof.

ViewBuffer Test cases

```
public class ViewBufferTest extends AbstractTest {

    @Test
    public void asCharacterBuffer() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));
        final CharBuffer charBuffer = buffer.asCharBuffer();

        assertEquals("Buffer position invalid", 0, buffer.position());
        assertEquals("CharBuffer position invalid", 0, charBuffer.position());
        assertEquals("Text data invalid", charBuffer.toString(), byteBufferToString(buffer) ←
            );
    }

    @Test
    public void asCharacterBufferSharedData() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));
        final CharBuffer charBuffer = buffer.asCharBuffer();

        assertEquals("Buffer position invalid", 0, buffer.position());
        assertEquals("CharBuffer position invalid", 0, charBuffer.position());

        final byte[] period = ".".getBytes(BIG_ENDIAN_CHARSET);
        int idx = 22;
        for (byte elem : period) {
            buffer.put(idx++, elem);
        }

        assertEquals("Text data invalid", "Hello world.", byteBufferToString(buffer));
        assertEquals("Text data invalid", charBuffer.toString(), byteBufferToString(buffer) ←
```

```
            );
    }
}
```

In addition to the various convenience `get(...)` and `put(...)` methods that deal with the various primitive types ByteBuffer provides us with an assortment of methods that provide primitive ByteBuffer views of the underlying data eg: `asCharBuffer()` demonstrates exposing a Character Buffer view of the underlying data.

Miscellaneous ByteBuffer Test cases

```
public class MiscBufferTest extends AbstractTest {

    @Test
    public void compact() {
        final ByteBuffer buffer = ByteBuffer.allocate(24);
        buffer.putChar('H');
        buffer.putChar('e');
        buffer.putChar('l');
        buffer.putChar('l');
        buffer.putChar('o');

        buffer.flip();
        buffer.position(4);
        buffer.compact();

        assertEquals("Buffer position invalid", 6, buffer.position());

        buffer.putChar('n');
        buffer.putChar('g');

        assertEquals("Buffer position invalid", 10, buffer.position());
        buffer.flip();
        assertEquals("Invalid text", "llong", byteBufferToString(buffer));
    }

    @Test
    public void testDuplicate() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));
        final ByteBuffer duplicate = buffer.duplicate();

        assertEquals("Invalid position", 0, duplicate.position());
        assertEquals("Invalid limit", buffer.limit(), duplicate.limit());
        assertEquals("Invalid capacity", buffer.capacity(), duplicate.capacity());

        buffer.putChar(22, '.');

        assertEquals("Text data invalid", "Hello world.", byteBufferToString(buffer));
        assertEquals("Text data invalid",  byteBufferToString(duplicate),  ←
            byteBufferToString(buffer));
    }

    @Test
    public void slice() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));
        buffer.position(12);

        final ByteBuffer sliced = buffer.slice();
        assertEquals("Text data invalid", "world!", byteBufferToString(sliced));
        assertEquals("Invalid position", 0, sliced.position());
        assertEquals("Invalid limit", buffer.remaining(), sliced.limit());
```

```
        assertEquals("Invalid capacity", buffer.remaining(), sliced.capacity());

        buffer.putChar(22, '.');
        assertEquals("Text data invalid", "world.", byteBufferToString(sliced));
    }

    @Test
    public void rewind() {
        final ByteBuffer buffer = ByteBuffer.wrap("Hello world!".getBytes( ←
            BIG_ENDIAN_CHARSET));

        final byte a = buffer.get();
        final byte b = buffer.get();

        assertEquals("Invalid position", 2, buffer.position());

        buffer.rewind();

        assertEquals("Invalid position", 0, buffer.position());
        assertSame("byte a not same", a, buffer.get());
        assertSame("byte a not same", b, buffer.get());
    }

    @Test
    public void compare() {
        final ByteBuffer a = ByteBuffer.wrap("Hello world!".getBytes(BIG_ENDIAN_CHARSET));
        final ByteBuffer b = ByteBuffer.wrap("Hello world!".getBytes(BIG_ENDIAN_CHARSET));

        assertTrue("a is not the same as b", a.compareTo(b) == 0);
    }

    @Test
    public void compareDiffPositions() {
        final ByteBuffer a = ByteBuffer.wrap("Hello world!".getBytes(BIG_ENDIAN_CHARSET));
        final ByteBuffer b = ByteBuffer.wrap("Hello world!".getBytes(BIG_ENDIAN_CHARSET));

        a.position(2);

        assertTrue("a is the same as b", a.compareTo(b) != 0);
    }
}
```

9.5 Summary

In this tutorial we learned a bit about ByteBuffers, we understood how to create one, the various types, why we have the different types and when to use them as well as the core semantic operations defined by the abstraction.

ByteBuffers are not thread safe and hence many operations on it, need to be guarded against to ensure that multiple threads do not corrupt the data or views thereon. Be wary of relative `get(...)` and `put(...)` operations as these do sneaky things like advancing the ByteBuffers position.

Wrapping, slicing and duplicating all point to the `byte []` they wrapped or the ByteBuffer they sliced / duplicated. Changes to the source input or the resulting ByteBuffers will effect each other. Luckily with `slice(...)` and `duplicate(...)` the position, mark and limit cursors are independent.

When toggling between reading data into a ByteBuffer and writing the contents from that same ByteBuffer it is important to `flip()` the ByteBuffer to ensure the `limit` is set to the current `position`, the current `position` is reset back to 0 and the `mark`, if defined, is discarded. This will ensure the ensuing write will be able to write what was just read. Partial writes in this context can be guarded against by calling `compact()` right before the next iteration of read and is very elegantly demonstrated in the API under compact.

When comparing ByteBuffers the positions matter, ie: you can have segments of a ByteBuffer that are identical and these compare favorably should the two ByteBuffers, in question, have the same position and limit (`bytesRemaining()`) during comparison.

For frequent high volume IO operations a `DirectByteBuffer` should yield better results and thus should be preferred.

Converting a `byte []` into a ByteBuffer can be accomplished by wrapping the `byte []` via the `wrap(...)` method. Converting back to a `byte []` is not always that straight forward. Using the convenient `array()` method on ByteBuffer only works if the ByteBuffer is backed by a `byte []`. This can be confirmed via the `hasArray()` method. A bulk `get(...)` into an applicably sized `byte []` is your safest bet, but be on the guard for sneaky side effects, ie: bumping the `position` cursor.

9.6 Download the source code

This was a Java Nio ByteBuffer tutorial

Download

You can download the full source code of this example here: **Java Nio ByteBuffer tutorial**