Unit-9 (Nonblocking IO)

- Reading from or writing to a network socket doesn't halt the program.
- The program can perform other tasks while waiting for I/O operations to complete.
- The program check or be notified when the I/O operation is ready to proceed or has finished.

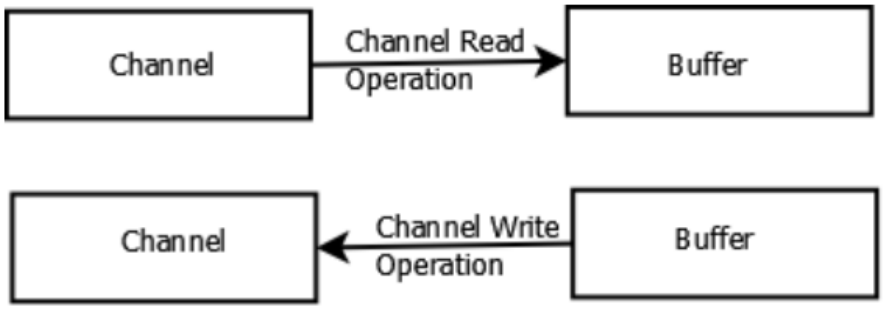
The **java.nio** package provides extensive support to build efficient network applications.

This package contains set of **classes and interfaces** for performing **non-blocking I/O** Operation.

Java NIO uses three core classes:

- **Buffer**: This holds information that is **read or written** to a channel. Buffers is used to store data temporarily as you perform read or write operations
- **Channel**: Channel interface represents **a source or destination of data**, such as file or network socket. Also provides methods for reading and writing to the channel.
- **Selector**: This is a mechanism to **handle multiple channels in a single thread**.

Conceptually, **buffers** and **channels** work together to process data. As shown in the next figure, data can be moved in either direction between a **buffer** and a **channel**:



The channel is connected to **some external data source**, while the buffer is used **internally to process the data**. There are several types of **channels and buffers**. A few of these are listed in the following tables.

Channel class	Purpose
`FileChannel`	This connects to a file
`DatagramChannel`	This supports datagram sockets
`SocketChannel`	This supports streaming sockets
`ServerSocketChannel`	This listens for socket requests
`NetworkChannel`	This supports a network socket
`AsynchronousSocketChannel`	This supports asynchronous streaming sockets

The table for buffers is as follows:

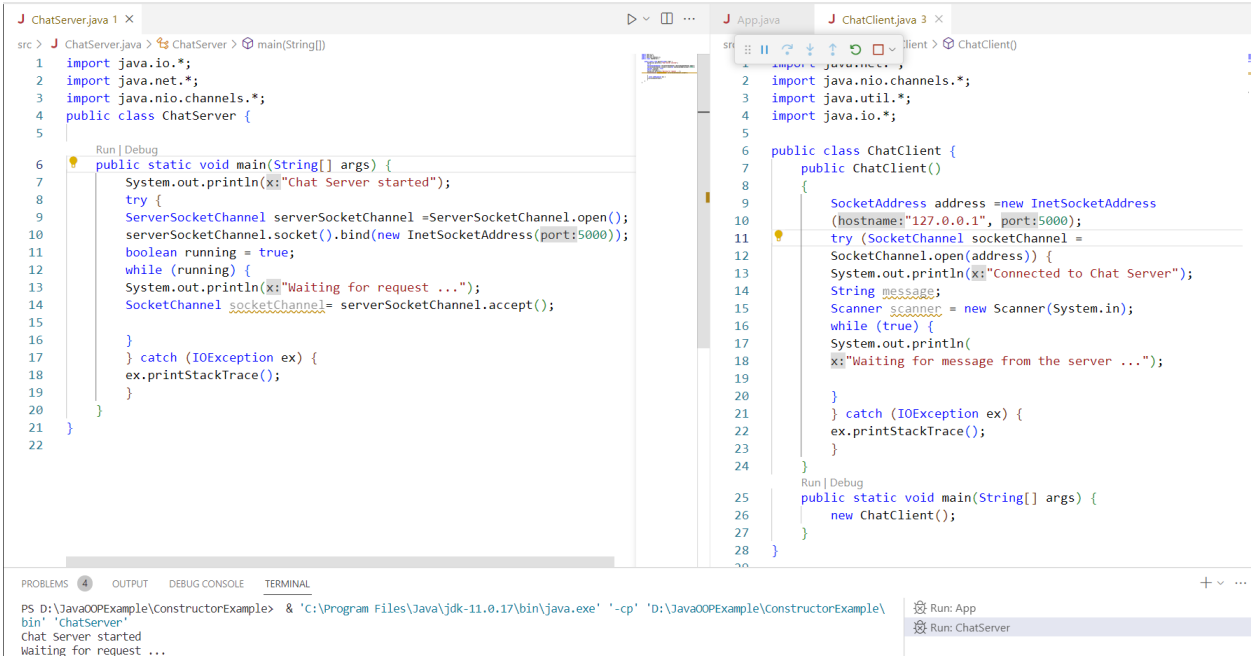
Buffer class	Data type supported
ByteBuffer	byte
CharBuffer	char
DoubleBuffer	double
FloatBuffer	float
IntBuffer	int
LongBuffer	long
ShortBuffer	short

Buffer:

- Buffers temporarily hold data while it's being transferred to or from channels.
- Each buffer has a fixed size or capacity when it's created.
- Buffers have fields and methods to manage and access the data they hold.
- Instead of directly using input or output streams, data is read or written to buffers.
- Data moves between channels and buffers, rather than streams, to manage I/O operations.

Lab: Write a simple chat client server application using nio

Simple Chat Client Server Application using NIO



Creating Buffer:

In Java, you can create a buffer for non-blocking I/O operations using the **java.nio.ByteBuffer** class. Here's an example of how to create a buffer for non-blocking I/O in Java:

App.java

src > App.java > App > main(String[])

```java
import java.nio.ByteBuffer;

public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        // Create a buffer with a capacity of 1024 bytes
        ByteBuffer buffer = ByteBuffer.allocate(capacity:1024);

        // Write data into the buffer
        String data = "Hello, Samriddhi College!";
        buffer.put(data.getBytes());

        // Prepare the buffer for reading
        buffer.flip();

        // Read data from the buffer
        byte[] readData = new byte[buffer.remaining()];
        buffer.get(readData);
        System.out.println("Read data: " + new String(readData));

        // Clear the buffer to make it ready for writing again
        buffer.clear();
    }
}
```

In Above example:

we create a **ByteBuffer** using the **ByteBuffer.allocate** method and specify the desired capacity of 1024 bytes. The buffer is then ready to be used for reading or writing data.

To write data into the buffer, we use the **put** method, which takes a byte array or other data types as input. Here string **Hello, Samriddhi College!** is converted to byte)

Next, we prepare the buffer for reading by calling the **flip** method. This sets the limit of the buffer to the current position and resets the position to zero,

To read the data from the buffer, we create a byte array with the remaining bytes in the buffer (**buffer.remaining()**) and use the **get** method to copy the data from the buffer into the array. We then convert the byte array back into a string and print it

Finally, we clear the buffer using the **clear** method to make it ready for writing again.

Filling and Draining:

Filling and draining a buffer refers to the process of **adding data to a buffer** (filling) and **retrieving data from a buffer** (draining). The **java.nio.ByteBuffer** class provides methods to facilitate these operations. **Filling** and **draining** a buffer can be useful when working with **I/O operations, network communication**, or **data processing scenarios**. Here's an example:

Bulk Method:

The **ByteBuffer** class provides **bulk methods** that offer convenience and performance benefits when working with **larger data sets**. The bulk methods includes **put()** and **get()** methods

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

public ByteBuffer put(byte[] array)
```

App.java

src > App.java > App > main(String[])

```java
import java.nio.ByteBuffer;

public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        // Create a byte array
        byte[] data = {1, 2, 3, 4, 5};
        // Create a buffer with a capacity of 5 bytes
        ByteBuffer buffer = ByteBuffer.allocate(capacity:5);
        // Put data from the byte array into the buffer
        buffer.put(data);
        // Prepare the buffer for reading
        buffer.flip();
        // Create a byte array for receiving the data from the buffer
        byte[] receivedData = new byte[buffer.remaining()];
        // Get data from the buffer into the byte array
        buffer.get(receivedData);
        // Print the received data
        for (byte b : receivedData) {
            System.out.print(b + " ");
        }
    }
}
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS D:\JavaOOPExample\ConstructorExample>  & 'C:\Program Files\Java\jdk-11.0.17\bin\java.exe'
1 2 3 4 5
PS D:\JavaOOPExample\ConstructorExample>
```

Data Conversion:

In Java, you can perform data conversion within a **ByteBuffer** using various methods provided by the **ByteBuffer** class. These methods allow you to convert between **different data types** and handle endianness (Many processors have instructions to convert a word in a register). Here are some commonly used data conversion methods in **ByteBuffer**:

- **putX() methods**: These methods allow you to store values of different data types in the buffer. For example:

  putInt(int value): Stores an integer value in the buffer.

putLong(long value): Stores a long value in the buffer.

putFloat(float value): Stores a float value in the buffer.

getX() methods: These methods allow you to retrieve values of different data types from the buffer. For example:

getInt(): Retrieves an integer value from the buffer.

getLong(): Retrieves a long value from the buffer.

getFloat(): Retrieves a float value from the buffer.

order() method: This method allows you to set the byte order (endianness) of the buffer. By default, the byte order is big-endian. You can use the order(ByteOrder) method to change the byte order. For example:

order(ByteOrder.BIG_ENDIAN): Sets the byte order to big-endian.(low to high)

highest value placed in lowest memory

order(ByteOrder.LITTLE_ENDIAN): Sets the byte order to little-endian.(high to low)

Example:

App.java

src > App.java > App > main(String[])

```java
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        // Create a buffer with a capacity of 8 bytes
        ByteBuffer buffer = ByteBuffer.allocate(capacity:100);
        // Set the byte order to little-endian
        buffer.order(ByteOrder.LITTLE_ENDIAN);
        // Put data of different types into the buffer
        buffer.putInt(value:42);
        buffer.putLong(value:123456L);
        buffer.putFloat(value:3.14f);
        // Prepare the buffer for reading
        buffer.flip();
        // Retrieve the data from the buffer
        int intValue = buffer.getInt();
        long longValue = buffer.getLong();
        float floatValue = buffer.getFloat();

        System.out.println("Int value: " + intValue);
        System.out.println("Long value: " + longValue);
        System.out.println("Float value: " + floatValue);
    }
}
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL

```
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS D:\JavaOOPExample\ConstructorExample>  & 'C:\Program Files\Java\jdk-11.0.17\bin\java.exe' '-cp' 'D:\Ja
Int value: 42
Long value: 123456
Float value: 3.14
PS D:\JavaOOPExample\ConstructorExample>
```

View Buffers

In Java, you can create **views of buffers** using the **slice(), duplicate(), and asReadOnlyBuffer()** methods provided by the **ByteBuffer** class. These methods allow you to create new buffers that share the underlying data with the **original buffer**, but have their own **position**, **limit**, and **capacity**. Creating views can be useful when you want to work with **subsets of data** that reference the same data.

```java
import java.nio.ByteBuffer;
public class App {
    public static void main(String[] args) throws Exception {
        // Create a buffer with a capacity of 10 bytes
        ByteBuffer originalBuffer = ByteBuffer.allocate(100);
```

```
        // Put some data into the original buffer
        originalBuffer.putInt(42);
        originalBuffer.putFloat(3.14f);
        originalBuffer.putDouble(123.456);

        // Create a view of the original buffer using slice()
        ByteBuffer sliceBuffer = originalBuffer.slice();

        // Create another view of the original buffer using duplicate()
        ByteBuffer duplicateBuffer = originalBuffer.duplicate();

        // Create a read-only view of the original buffer using
asReadOnlyBuffer()
        ByteBuffer readOnlyBuffer = originalBuffer.asReadOnlyBuffer();

        // Print the data in the views
        System.out.println("Slice Buffer:");
        printBufferContent(sliceBuffer);

        System.out.println("Duplicate Buffer:");
        printBufferContent(duplicateBuffer);

        System.out.println("Read-Only Buffer:");
        printBufferContent(readOnlyBuffer);
    }
     private static void printBufferContent(ByteBuffer buffer) {
        buffer.flip();
        while (buffer.hasRemaining()) {
            System.out.println(buffer.get());
        }
        System.out.println();
    }
}
```

Compacting Buffer

By compacting the buffer, you can **reuse the remaining space for new data** without needing to create a new buffer or clear the entire buffer. This can be particularly useful in scenarios where you want to avoid unnecessary **memory allocations** and efficiently manage your buffer resources.

Marking and Resetting Buffer:

In Java, marking and resetting are mechanisms provided by the **InputStream** class to manage

the position within a stream. These methods are particularly useful when you want to **read a portion of data** from the stream and then return to a specific position to read again.

public final Buffer mark()

public final Buffer reset()

The reset() method throws an InvalidMarkException, a runtime exception, if the mark is not set.

src > DayTimeServer.java > ...

```java
import java.io.*;
import java.net.*;
public class DayTimeServer {
    Run | Debug
    public static void main(String[] args) {
        String hostname = "example.com";
        int port = 80;
        try {
            Socket socket = new Socket(hostname, port);
            InputStream inputStream = socket.getInputStream();
            // Read data from the input stream
            byte[] buffer = new byte[1024];
            int bytesRead = inputStream.read(buffer);
            // Process the data
            String responseData = new String(buffer, offset:0, bytesRead);
            System.out.println("Received response: " + responseData);
            // Mark the current position
            inputStream.mark(readlimit:0);
            // Reset the stream back to the marked position
            inputStream.reset();
            // Read data again from the marked position
            bytesRead = inputStream.read(buffer);
            // Process the data again
            responseData = new String(buffer, offset:0, bytesRead);
            System.out.println("Received response again: " + responseData);
            // Close the socket and stream
            socket.close();
            inputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Object Method:

The buffer classes all provide the usual **equals(), hashCode(), and toString()** methods. They also implement Comparable, and therefore provide **compareTo()** methods.However, buffers are not **Serializable** or **Cloneable**.

```java
CharBuffer buffer1 = CharBuffer.wrap("12345678");
 CharBuffer buffer2 = CharBuffer.wrap("5678");
 buffer1.get();
 buffer1.get();
```

```
buffer1.get();
buffer1.get();
System.out.println(buffer1.equals(buffer2));
```

Channel

non-blocking I/O in Java can be achieved using the **java.nio.channels** package. The **Channel interface** and its implementations provide a way to perform non-blocking I/O operations for reading from and writing to network sockets.

In network programming there are only three really important channel classes,**SocketChannel**, **ServerSocketChannel**, and **DatagramChannel**.

SocketChannel:

SocketChannel is used for communication between the **client** and the **server**. It represents a channel for a TCP connection. It allows reading and writing data to and from the remote endpoint. SocketChannel is used by the client to connect to a server and exchange data. It is part of the **java.nio.channels** package and offers a **non-blocking** alternative to the traditional **java.net.Socket** class.

- **Connecting:**
  The SocketChannel class does not have any public constructors. Instead, you create a new SocketChannel object using one of the two static open() methods:
  public static SocketChannel open(SocketAddress remote) throws IOException
  public static SocketChannel open() throws IOException

  The first variant makes the connection. This method blocks (i.e., the method will not return until the connection is made or an exception is thrown). For example:
  SocketAddress address = new InetSocketAddress("www.cafeaulait.org", 80);
  SocketChannel channel = SocketChannel.open(address);

  The second version does not immediately connect. It creates an initially unconnected socket that must be connected later using the **connect()** method. For example:
  SocketChannel channel = SocketChannel.open();
  SocketAddress address = new InetSocketAddress("www.cafeaulait.org", 80);
  channel.connect(address);

  If the program wants to check whether the connection is complete, it can call these two methods:
  public abstract boolean isConnected()
  public abstract boolean isConnectionPending()

- **Reading**

```
ByteBuffer[] buffers = new ByteBuffer[2];
```

```
buffers[0] = ByteBuffer.allocate(1000);
buffers[1] = ByteBuffer.allocate(1000);
while (buffers[1].hasRemaining() && channel.read(buffers) != -1) ;
```

- **Writing**
  in Java, you can use the **write()** method along with a **ByteBuffer** containing the data to be sent.
  The basic write() method takes a single buffer as an argument:
  public abstract int write(ByteBuffer src) throws IOException

  Check the data is completely written
  while (buffer.hasRemaining() && channel.write(buffer) != -1) ;

  There are two methods to drain all the data:
  ```
  public final long write(ByteBuffer[] dsts) throws IOException
  public final long write(ByteBuffer[] dsts, int offset, int length)
  throws IOException
  ```
  The **first** variant drains all the buffers. The **second** method drains length buffers, starting with the one at offset.

- **Closing**
  Just as with regular sockets, you should close a channel when you're done with it to free up the port and any other resources it may be using:
  ```
  public void close() throws IOException
  ```

ServerSocketChannel:

ServerSocketChannel is used by the server to listen for incoming connections from clients. It represents a channel that listens for TCP connections.

It allows accepting connections from clients and creating separate **SocketChannel** instances for each client connection. **ServerSocketChannel** is used to accept incoming connections and handle multiple clients simultaneously.

- **Creating new ServerSocketChannel:**

```
 try {
//Create a ServerSocketChannel instance
ServerSocketChannel server = ServerSocketChannel.open();
//Configure the server socket channel to work in non-blocking mode
server.configureBlocking(false);
//Bind the server socket channel to a specific address and port
SocketAddress address = new InetSocketAddress(80);
server.bind(address);
} catch (IOException ex) {
```

```
System.err.println("Could not bind to port 80 because " + ex.getMessage());
}
```

- **Accepting Connection:**

```
while (true) {
    SocketChannel socketChannel = server.accept();
    // Handle the incoming connection
}
```

The **accept()** method blocks until an incoming connection is received. Once a connection is accepted, it returns a **SocketChannel** instance representing that connection. You can perform the necessary operations to handle the incoming connection within the loop.

The Channel Class:

The **Channels** class that can be used to convert streams to channels and vice versa. These methods provide a convenient way to bridge the gap between the **traditional stream-based I/O** and **the newer channel-based I/O**.

It has methods that convert from streams to channels

```
public static InputStream newInputStream(ReadableByteChannel ch)
public static OutputStream newOutputStream(WritableByteChannel ch)
public static ReadableByteChannel newChannel(InputStream in)
public static WritableByteChannel newChannel(OutputStream out)
```

Example:

```
// Convert InputStream to ReadableByteChannel
InputStream inputStream = new FileInputStream("input.txt");
ReadableByteChannel channel1 = Channels.newChannel(inputStream);

// Convert OutputStream to WritableByteChannel
OutputStream outputStream = new FileOutputStream("output.txt");
WritableByteChannel channel2 = Channels.newChannel(outputStream);

// Convert ReadableByteChannel to InputStream
InputStream convertedInputStream = Channels.newInputStream(channel1);

// Convert WritableByteChannel to OutputStream
OutputStream convertedOutputStream =
Channels.newOutputStream(channel2);
```

Methods that convert from channels to readers, and writers:

```
public static Reader newReader (ReadableByteChannel channel,
```

```
CharsetDecoder decoder, int minimumBufferCapacity)
public static Reader newReader (ReadableByteChannel ch, String encoding)
public static Writer newWriter (WritableByteChannel ch, String encoding)
```