UNIT-9: NONBLOCKING I/O (INPUT/OUTPUT)

- Nonblocking I/O (Input/Output) is a programming approach that allows an application to perform I/O operations **without blocking the execution of the program**. In the context of Java, nonblocking I/O is achieved using the NIO **(New I/O) package**, which provides an **alternative to the traditional blocking I/O operations** offered by the java.io package.
- In traditional blocking I/O, when an application performs an I/O operation (such as reading from a file or a network socket), the operation **blocks the execution of the program until the operation completes**. This means that **the program cannot continue executing other tasks until the I/O operation is finished**, which can lead **to inefficient resource utilization, especially in scenarios where multiple I/O operations** need to be **performed concurrently**.
- Nonblocking I/O, on the other hand, allows an application to **initiate an I/O operation and continue executing other tasks without waiting for the operation to complete**. It provides a mechanism for applications to **check the status of I/O operations and handle them asynchronously**. This enables more **efficient utilization of system resources** and the ability to handle **multiple I/O operations concurrently without blocking** the program's execution.

UNIT-9: NONBLOCKING I/O (INPUT/OUTPUT)

Here's an example scenario to illustrate the need for nonblocking I/O:

- Imagine you're building a server application that needs to handle **multiple client connections concurrently**. Each client connection requires reading data from the network and performing some processing on it before sending a response back. In a traditional blocking I/O approach, you would typically create a new thread for each client connection to handle the I/O operations.
- However, this approach can become **inefficient and resource-intensive** when dealing with a large number of client connections. Creating and managing threads for each connection can consume a significant amount of **memory and processing power**. Moreover, as the number of connections increases, the **overhead of thread creation, context switching, and synchronization** can impact the application's performance.
- Nonblocking I/O provides a solution to this problem by **allowing a single thread to handle multiple client connections concurrently**. It uses a **mechanism called a selector**, which can efficiently **monitor multiple I/O channels** for events and dispatch them to appropriate handlers. This enables the server application to **efficiently multiplex I/O operations from multiple clients without creating separate threads** for each connection.
- By using nonblocking I/O, the server application can achieve better scalability and responsiveness. It can efficiently handle a large number of client connections using fewer system resources, resulting in improved performance.
- Java provides **the java.nio package**, which includes classes and APIs for nonblocking I/O, **such as Selector, SocketChannel**, and **ServerSocketChannel**. These classes allow you to build applications that leverage nonblocking I/O capabilities and handle multiple I/O operations **concurrently in a single thread.**

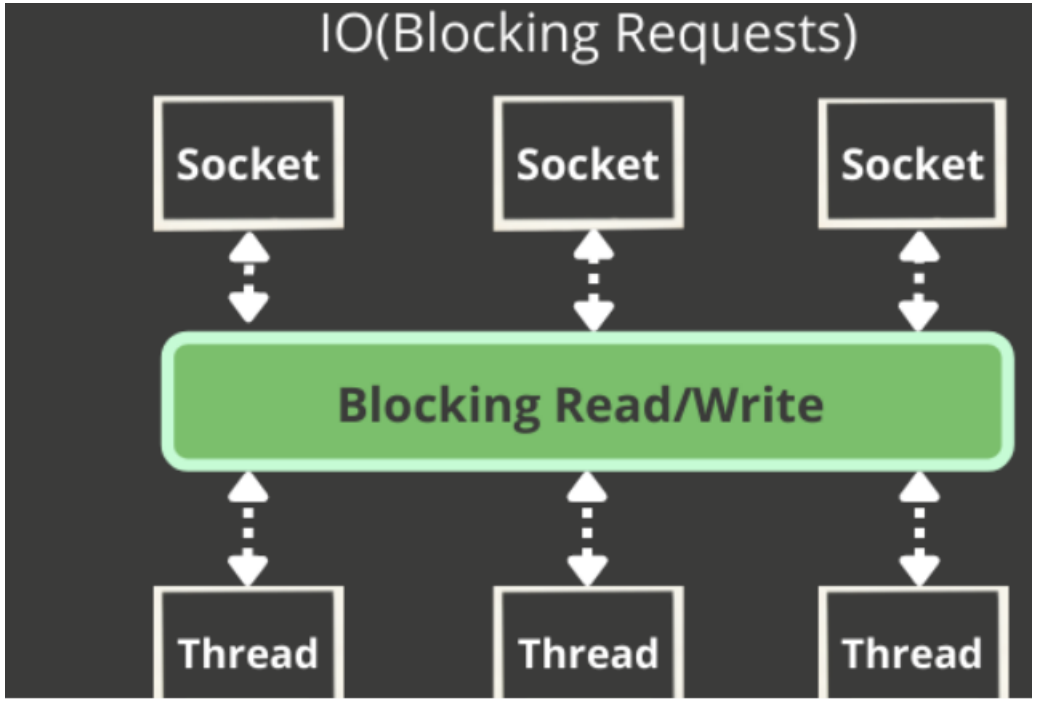
UNIT-9: NONBLOCKING I/O (INPUT/OUTPUT)

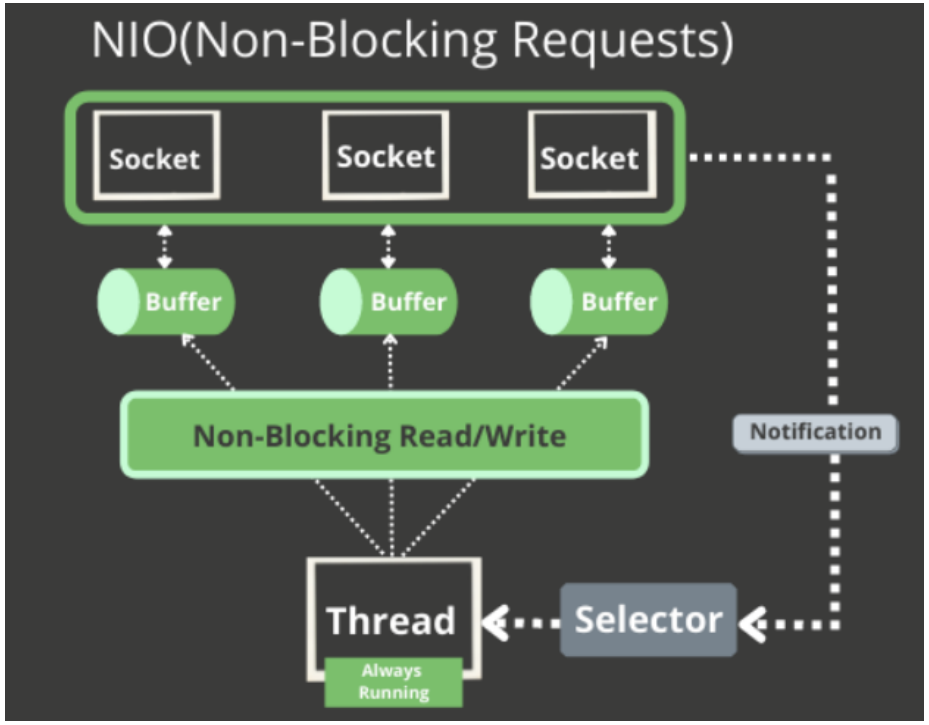
Nonblocking I/O is particularly useful in scenarios where an application needs to **handle multiple simultaneous connections** or perform I/O operations on multiple channels concurrently, such as in network servers, real-time systems, or applications that require **high scalability and responsiveness**.

By using nonblocking I/O, applications can **achieve better performance and responsiveness by effectively utilizing system resources and avoiding unnecessary blocking and waiting times**. It allows for more efficient handling of I/O operations and **enables applications to handle multiple tasks concurrently, improving overall system throughput and responsiveness.**

IO	NIO
It is based on the Blocking I/O operation	It is based on the Non-blocking I/O operation
It is Stream-oriented	It is Buffer-oriented
Channels are not available	Channels are available for Non-blocking I/O operation
Selectors are not available	Selectors are available for Non-blocking I/O operation

IO BLOCKING VS NON-BLOCKING





BUFFERS

Buffers in a non-blocking Java program refer to the **data structures used to temporarily store and manage data during asynchronous I/O operations**. They are an essential part of **non-blocking I/O programming** because they allow **for efficient handling of data between producers and consumers without blocking** the execution of the program.

In Java, **the java.nio** package provides a set of classes for **non-blocking I/O**, including the **ByteBuffer** class, which represents a buffer **for byte-oriented data**. Buffers facilitate the efficient transfer of data between I/O channels and the application by providing a structured way to hold and manipulate data.

ava NIO was created on three main components: **Buffers, Channels, and Selectors**.

Buffers: Buffer is **a block of memory used to temporarily store data** while it is being **moved from one place to another**.

Channels: Channel represents **a connection to objects that are capable of performing I/O operations**, such as files and sockets.

Selectors: They are used to **select the channels which are ready** for the I/O operation.

BUFFERS: CREATING, FILLING AND DRAINING

1. Import necessary Class.

import java.nio.ByteBuffer;

import java.nio.channels.SocketChannel;

2. Creating a buffer:

ByteBuffer buffer = ByteBuffer.allocate(capacity);

You create a buffer by calling the allocate method on the ByteBuffer class, specifying the desired capacity of the buffer. The capacity represents **the maximum amount of data the buffer can hold**.

3. Writing data to the buffer:

buffer.put(data); /filling

You can write data into the buffer using the put method. The **data can be a byte array or individual bytes**, depending on the context.

4.Preparing the buffer for reading:

buffer.flip(); //Draining

Before **reading data from the buffer**, you need to call the flip method, which **sets the limit to the current position and the position to zero**. It effectively marks the buffer as ready for reading.

5. Reading data from the buffer:

buffer.get(data);

To read data from the buffer, you can use the get method, which retrieves data from the buffer and stores it in the specified data byte array or individual byte variables.

These steps provide a **basic outline of creating and using buffers** in a non-blocking Java program. In practice, **buffers are often associated with non-blocking I/O channels, such as SocketChannel or DatagramChannel**, to facilitate efficient data transfer between the application and the underlying I/O operations.

BUFFERS: CREATING, FILLING AND DRAINING

6. **clear() method**:

The clear() method is used to **prepare the buffer for a new sequence of channel-read or relative put operations**. It **resets the position to 0 and the limit** to the capacity, effectively **clearing the buffer's contents**. The cleared buffer is ready to be written or filled with new data.

7. **wrap() method**:

The wrap() method creates **a new buffer that wraps an existing array**. It allows you to **wrap an array with a buffer without allocating new memory**. The resulting buffer will have its **position set to 0 and the limit set to the array's length**.

```
import java.nio.ByteBuffer;
public class BufferExample {
    public static void main(String[] args) {
        byte[] array = { 1, 2, 3, 4, 5 };
        ByteBuffer buffer = ByteBuffer.wrap(array);
        System.out.println(buffer.position()); // Output: 0
        System.out.println(buffer.limit()); // Output: 5
        System.out.println(buffer.get()); // Output: 1
    }
}
```

BUFFERS: CREATING, FILLING AND DRAINING

```java
import java.nio.ByteBuffer;
public class BufferNio {
    Run | Debug
    public static void main(String[] args) {
        // Create a buffer with a capacity of 10 bytes
        ByteBuffer buffer = ByteBuffer.allocate(capacity:10);

        // Write array data into the buffer
        byte[] dataArray = { 1, 2, 3, 4, 5 };
        buffer.put(dataArray);
        // Reset the buffer's position and limit for reading
        buffer.flip();
        // pullout data from buffer
        while (buffer.hasRemaining()) {
            System.out.println(buffer.get());
        }

    }
}
```

BUFFERS: BULK METHODS

The ByteBuffer class in Java provides a method put(byte[] data, int offset, int length) that allows you to put a specified portion of a byte array into the buffer. Here's how you can use it:

```
ByteBuffer buffer = ByteBuffer.allocate(10);

// Write a portion of a byte array into the buffer

byte[] dataArray = { 1, 2, 3, 4, 5 };

int offset = 1;  // Starting index in the array

int length = 3;  // Number of bytes to put into the buffer

buffer.put(dataArray, offset, length);
```

```
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)
```

DATA CONVERSION

```
public abstract char       getChar()
public abstract ByteBuffer putChar(char value)
public abstract char       getChar(int index)
public abstract ByteBuffer putChar(int index, char value)
public abstract short      getShort()
public abstract ByteBuffer putShort(short value)
public abstract short      getShort(int index)
public abstract ByteBuffer putShort(int index, short value)
public abstract int        getInt()
public abstract ByteBuffer putInt(int value)
public abstract int        getInt(int index)
public abstract ByteBuffer putInt(int index, int value)
```

```
public abstract long       getLong()
public abstract ByteBuffer putLong(long value)
public abstract long       getLong(int index)
public abstract ByteBuffer putLong(int index, long value)
public abstract float      getFloat()
public abstract ByteBuffer putFloat(float value)
public abstract float      getFloat(int index)
public abstract ByteBuffer putFloat(int index, float value)
public abstract double     getDouble()
public abstract ByteBuffer putDouble(double value)
public abstract double     getDouble(int index)
public abstract ByteBuffer putDouble(int index, double value)
```

VIEW BUFFERS

- View buffers are a feature provided by the **Java Buffer class** that allows you to **create views of the underlying buffer as different buffer types**. These views provide a convenient way to **interpret the data in the buffer according** to different data types, such as shorts, characters, integers, longs, floats, or doubles.
- The essential purpose of view buffers is to **provide flexibility in working with buffer data**. They allow you to **access and manipulate the same underlying data in different ways without the need for data conversion or duplication**. Here are a few reasons why view buffers are essential:

```
public abstract ShortBuffer   asShortBuffer()
public abstract CharBuffer    asCharBuffer()
public abstract IntBuffer     asIntBuffer()
public abstract LongBuffer    asLongBuffer()
public abstract FloatBuffer   asFloatBuffer()
public abstract DoubleBuffer  asDoubleBuffer()
```

```java
import java.nio.ByteBuffer;
import java.nio.IntBuffer;
public class ViewBuff {
    Run | Debug
    public static void main(String[] args) {
        // Create a byte buffer with 8 bytes
        ByteBuffer byteBuffer = ByteBuffer.allocate(capacity:18);
        // Put some data into the byte buffer
        byteBuffer.putInt(value:123);
        byteBuffer.putInt(value:456);
        byteBuffer.flip();
        // Create an int buffer view of the byte buffer
        IntBuffer intBuffer = byteBuffer.asIntBuffer();
        // Read and print the integers from the int buffer view
        while (intBuffer.hasRemaining()) {
            System.out.println(intBuffer.get());
        }
    }
}
```

COMPACTING BUFFERS

- **Compacting buffers** refers to a process in which the **data remaining in a buffer is moved to the beginning of the buffer**, **making room for new data to be written**. It is commonly used when working with byte buffers in Java's nio (New Input/Output) package.
- In Java's nio package, byte buffers provide a way to **efficiently read from and write to channels**. When data is written to a buffer, it occupies a certain amount of space, which is determined by the position and limit of the buffer. After the data is consumed, the buffer's position is advanced to the **end of the written data**, indicating that the **data has been processed**.
- When a **buffer becomes full or reaches its limit, and there is still more data to be written, compacting the buffer can be useful**.

```java
import java.nio.ByteBuffer;
public class BufferCompack {
    Run | Debug
    public static void main(String[] args) {
        // Create a byte buffer with a capacity of 8 bytes
        ByteBuffer buffer = ByteBuffer.allocate(capacity:3);
        // Write 6 bytes to the buffer
        buffer.put((byte) 1);
        buffer.put((byte) 2);
        buffer.put((byte) 3);
        // Compact the buffer to make room for new data
        buffer.compact();
        // Write 3 bytes to the buffer
        buffer.put((byte) 7);
        buffer.put((byte) 8);
        buffer.put((byte) 9);

        // Flip the buffer for reading
        buffer.flip();
        // Read and print the contents of the buffer
        while (buffer.hasRemaining()) {
            System.out.println(buffer.get());
        }
    }
}
```

DUPLICATING BUFFERS

- Duplicating buffers refers to **creating a new buffer** that **shares the same content as an existing buffer**. In Java's nio (New Input/Output) package, **the duplicate() method** is used to create a duplicate buffer.
- When a buffer is duplicated, **both the original buffer and the duplicate buffer maintain separate position, limit, and capacity values**. However, they **share the same underlying data**. **Any changes made to the data in one buffer will be reflected in the other buffer** as well.

```
import java.nio.ByteBuffer;
import java.util.Arrays;

public class DupBuffer {
    Run | Debug
    public static void main(String[] args) {
    // initialize buffer instance
    ByteBuffer sourceBuffer = ByteBuffer.allocate(capacity:5);
    // add values to buffer
    sourceBuffer.put((byte)2);
    sourceBuffer.put((byte)4);
    sourceBuffer.put((byte)10);

    // create a duplicate buffer
    ByteBuffer newBuffer = sourceBuffer.duplicate();
    // Print buffers
    System.out.println("The source buffer is: " + Arrays.toString(sourceBuffer.array()));
    System.out.println("The duplicate buffer is: " + Arrays.toString(newBuffer.array()));
    // add values to source buffer
    sourceBuffer.put((byte)5);
    sourceBuffer.put((byte)20);
    // Print buffers
    System.out.println("\nThe updated source buffer is: " + Arrays.toString(sourceBuffer.array()));
    System.out.println("The updated duplicate buffer is: " + Arrays.toString(newBuffer.array()));

  }
}
```

SLICING BUFFERS

- Slicing buffers refers to creating a new buffer that is **a "slice" or subset of an existing buffer**. In Java's nio (New Input/Output) package, the slice() method is used to create a slice buffer.
- When **a buffer is sliced**, the **new buffer represents a portion of the original buffer's content**. The slice buffer and the original buffer **share the same underlying data**, but they have **their own position, limit, and capacity values**. Changes made to the data in one buffer will be reflected in the other buffer.

```
public abstract ByteBuffer   slice()
public abstract IntBuffer    slice()
public abstract ShortBuffer  slice()
public abstract FloatBuffer  slice()
public abstract CharBuffer   slice()
public abstract DoubleBuffer slice()
```

```java
import java.nio.ByteBuffer;
import java.util.Arrays;
public class SliceBuffer {
    Run | Debug
    public static void main(String[] args) {
        // Create a byte buffer with a capacity of 8 bytes
        ByteBuffer buffer = ByteBuffer.allocate(capacity:3);
        // Write data to the buffer
        buffer.put((byte) 2);
        buffer.put((byte) 5);
        buffer.put((byte) 7);
        ByteBuffer sliceBuffer = buffer.slice();
        // Read and print the contents of the slice buffer
        System.out.println("The main buffer is: " + Arrays.toString(sliceBuffer.array()));
        System.out.println("The slic buffer is: " + Arrays.toString(buffer.array()));
    }
}
```

OBJECT METHOD

- In Java's nio (New Input/Output) package, the object methods equals(), hashCode(), and toString() are not specific to nio itself but are **general-purpose methods provided by the Object class**. These methods have the same behavior and purpose as described earlier, regardless of whether you're working with nio or other parts of Java.

```java
import java.nio.ByteBuffer;

public class ObjectMethod {
    Run | Debug
    public static void main(String[] args) {
        ByteBuffer buffer1 = ByteBuffer.allocate(capacity:4);
        ByteBuffer buffer2 = ByteBuffer.allocate(capacity:4);
        // Set data in buffers
        buffer1.putInt(value:123);
        buffer2.putInt(value:123);
        // Comparing buffers for equality
        boolean areEqual = buffer1.equals(buffer2);
        System.out.println("Buffers are equal: " + areEqual);
        // Hash codes of buffers
        int hashCode1 = buffer1.hashCode();
        int hashCode2 = buffer2.hashCode();
        System.out.println("Hash code of buffer1: " + hashCode1);
        System.out.println("Hash code of buffer2: " + hashCode2);
        // String representation of buffers
        String buffer1String = buffer1.toString();
        String buffer2String = buffer2.toString();
        System.out.println("Buffer1: " + buffer1String);
        System.out.println("Buffer2: " + buffer2String);
    }
}
```

THE MARK() AND RESET() METHODS

- In **the java.nio** package, the Buffer class provides the **mark() and reset()** methods that **allow you to mark a position in the buffer and then reset the position back to the marked position**.
- mark(): The **mark() method sets the buffer's mark at its current position**. This means that you can later reset the position back to the marked position using the reset() method. The mark is initially undefined and is cleared when you invoke **the clear(), compact(), or reset() methods**.
- reset(): The **reset() method sets the buffer's position back to the previously marked position**. If no mark is set, invoking reset() will throw a InvalidMarkException. When reset() is called, the buffer's position is updated to the marked position, allowing you to read or write data from that point onward.

```java
import java.nio.ByteBuffer;
public class MakResetBuff {
    Run | Debug
    public static void main(String[] args) {
        ByteBuffer buffer = ByteBuffer.allocate(capacity:10);
        buffer.put((byte) 1);
        buffer.put((byte) 2);
        buffer.put((byte) 3);
        System.out.println("Before Mark:" + buffer.toString());
        buffer.mark(); // Marking the position
        System.out.println("After Mark:" + buffer.toString());
        buffer.put((byte) 4);
        buffer.put((byte) 5);
        System.out.println("Before Reset:" + buffer.toString());
        buffer.reset(); // Resetting to the marked position
        System.out.println("After Reset:" + buffer.toString());
        buffer.put((byte) 6);
        buffer.put((byte) 7);
    }
}
```

THE CHANNELS CLASS

- **The Channels class** is a utility class in Java's NIO (New I/O) package that provides **static methods for working with channels**. Channels are a fundamental part of Java's **non-blocking I/O operations** and are used for **reading from and writing to I/O devices**, such as **sockets and files**.
- The Channels class provides methods for creating and working with different types of channels, including **SocketChannel and ServerSocketChannel.**
- **SocketChannel**: SocketChannel is a type of channel that represents a **two-way communication channel** for network sockets. It can be used for both reading from and writing to a socket. **SocketChannel** supports non-blocking I/O operations, which allows for **efficient handling of multiple connections in a single thread**.
- **ServerSocketChannel**: ServerSocketChannel is a type of channel that **listens for incoming connections from clients**. It represents a server-side socket that can **accept new connections. ServerSocketChannel** is used in server applications that need to handle **multiple client connections concurrently**.

```
import java.net.InetSocketAddress;
import java.nio.channels.SocketChannel;
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("example.com", 8080));
```

THE CHANNELS CLASS

- **ServerSocketChannel**: ServerSocketChannel is a type of channel that **listens for incoming connections from clients**. It represents a server-side socket that can **accept new connections. ServerSocketChannel** is used in server applications that need to handle **multiple client connections concurrently**.

```
import java.net.InetSocketAddress;
import java.nio.channels.SocketChannel;

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.bind(new InetSocketAddress(8080));
```

THE CHANNELS CLASS

The Channels class in Java NIO provides **several utility methods** for creating input and output streams from byte channels, as well as creating readers and writers from byte channels.

1. **public static InputStream newInputStream(ReadableByteChannel ch)**: This method creates a new InputStream from a **ReadableByteChannel**. It returns an input stream that reads bytes from the specified channel.
2. **public static OutputStream newOutputStream(WritableByteChannel ch):** This method creates a new OutputStream from a **WritableByteChannel**. It returns an output stream that writes bytes to the specified channel.
3. **public static ReadableByteChannel newChannel(InputStream in)**: This method creates a new **ReadableByteChannel** from an InputStream. It returns a byte channel that reads bytes from the specified input stream.
4. **public static WritableByteChannel newChannel(OutputStream out)**: This method creates a new **WritableByteChannel** from an OutputStream. It returns a byte channel that writes bytes to the specified output stream.

THE CHANNELS CLASS

1. **public static Reader newReader(ReadableByteChannel channel, CharsetDecoder decoder, int minimumBufferCapacity):** This method creates a new Reader from a ReadableByteChannel, a CharsetDecoder, and a minimum buffer capacity. It returns a reader that reads characters from the specified byte channel using the provided decoder.
2. **public static Reader newReader(ReadableByteChannel ch, String encoding):** This method creates a new Reader from a ReadableByteChannel and a character encoding. It returns a reader that reads characters from the specified byte channel using the specified character encoding.
3. **public static Writer newWriter(WritableByteChannel ch, String encoding):** This method creates a new Writer from a WritableByteChannel and a character encoding. It returns a writer that writes characters to the specified byte channel using the specified character encoding.
4. These methods provide convenient ways to bridge between byte channels and stream-oriented classes like InputStream, OutputStream, Reader, and Writer.

ASYNCHRONOUS CHANNELS

- Asynchronous channels in Java NIO provide **a non-blocking I/O mechanism** that allows you to perform I/O operations without **blocking the calling thread**. They are part of the **java.nio.channels** package and are **designed for asynchronous I/O operations**, where you can initiate an operation and **continue with other tasks while the operation completes in the background**.

The main asynchronous channel classes in Java NIO are:

1. **AsynchronousSocketChannel**: Represents an asynchronous socket channel used for non-blocking I/O operations on TCP/IP sockets.
2. **AsynchronousServerSocketChannel**: Represents an asynchronous server socket channel used for non-blocking I/O operations on server-side TCP/IP sockets.
3. These asynchronous channel classes provide several common methods for performing asynchronous I/O operations:
4. **open():** Creates a new asynchronous channel of the corresponding type.
5. **connect(SocketAddress, A attachment, CompletionHandler<Void, ? super A> handler):** Initiates a connection to a remote address asynchronously.
6. **read(ByteBuffer dst, A attachment, CompletionHandler<Integer, ? super A> handler)**: Initiates an asynchronous read operation into the given buffer.
7. **write(ByteBuffer src, A attachment, CompletionHandler<Integer, ? super A> handler):** Initiates an asynchronous write operation from the given buffer.
8. **accept(A attachment, CompletionHandler<AsynchronousSocketChannel, ? super A> handler)**: Accepts a connection from a client asynchronously, returning a new AsynchronousSocketChannel for communication with the client.

READINESS SELECTION

- Readiness selection, also known as I/O **multiplexing,** is a technique used in networking and operating system programming to **monitor multiple I/O sources (such as sockets, file descriptors, or devices**) for readiness to perform **I/O operations without blocking**.
- It allows you to efficiently handle I/O operations **on multiple resources using a single thread or process**.
- The readiness selection mechanism provides **a way to determine which I/O sources are ready for reading, writing, or have encountered an exceptional condition**.
- It allows you to avoid blocking on I/O operations and enables you to perform other tasks **or handle multiple connections simultaneously**.

In Java, the **Selector class** is part of **the java.nio.channels package** and is used for multiplexing I/O operations. It provides a mechanism for monitoring **multiple selectable channels** (such as sockets) and determining which channels are ready for I/O operations.

Here's an overview of the Selector class and its important methods:

1. Creating a Selector: To create a Selector instance, you can use the Selector.open() method, which returns a new Selector object.

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

2. Registering Channels: Before a channel can be monitored by a Selector, it needs to be **registered** with the Selector using the channel's **register() method**. This method returns a **SelectionKey** object that represents the **registration of the channel with the Selector**.

READINESS SELECTION

```
SelectableChannel channel ; // Obtain the channel you want to register
channel.configureBlocking(false); // Set channel to non-blocking mode
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

The register() method **takes two arguments**: the Selector instance and an interest set, represented by a combination of SelectionKey.OP_* constants. These constants specify the operations of interest, such as OP_READ for read operations or OP_WRITE for write operations.

3. Selecting Channels: The Selector class provides a few methods for selecting the channels that are ready for I/O operations. The most commonly used **method is select(),** which blocks until at least one channel is ready or until the method is interrupted.

- int **readyChannels** = selector.select();

- `SelectionKey.OP_ACCEPT`
- `SelectionKey.OP_CONNECT`
- `SelectionKey.OP_READ`
- `SelectionKey.OP_WRITE`

READINESS SELECTION

4. Working with Selected Keys: The selectedKeys() method returns a Set of SelectionKey objects that represent the channels ready for I/O operations. You can iterate over these keys to handle the I/O operations.

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();

for (SelectionKey key : selectedKeys) {

    if (key.isReadable()) {

        // Handle read operation

    }

    if (key.isWritable()) {

        // Handle write operation

    }

    // ...

}
```

READINESS SELECTION

5. Deregistering Channels: If you no longer want a channel to be monitored by the Selector, you can deregister it using the cancel() method of the corresponding SelectionKey.

```
SelectionKey key = channel.keyFor(selector);

key.cancel();
```

6. Closing the Selector: When you are done using the Selector, make sure to close it using the close() method. This releases the resources associated with the Selector.

```
selector.close();
```

NON BLOCKING CLIENT

```java
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;
public class Nbclient {
    private static final String SERVER_HOST = "localhost";
    private static final int SERVER_PORT = 8888;
    Run | Debug
    public static void main(String[] args) {
        try {
            SocketChannel clientChannel = SocketChannel.open();
            clientChannel.configureBlocking(block:false);
            clientChannel.connect(new InetSocketAddress(SERVER_HOST, SERVER_PORT));

            while (!clientChannel.finishConnect()) {
                // Wait for the connection to be established
            }
            System.out.println("Connected to server: " + clientChannel.getRemoteAddress());
            System.out.println(x:"Please enter text to communicate with server");
            Scanner scanner = new Scanner(System.in);

            while (true) {
                String message = scanner.nextLine();
                ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());
                clientChannel.write(buffer);
                buffer.clear();
                clientChannel.read(buffer);
                buffer.flip();
                byte[] data = new byte[buffer.limit()];
                buffer.get(data);
                String response = new String(data);
                System.out.println("Received from server: " + response);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

NON BLOCKING SERVER

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

public class Nbserver {
    private static final int PORT = 8888;

    Run | Debug
    public static void main(String[] args) {
        try {
            Selector selector = Selector.open();
            ServerSocketChannel serverChannel = ServerSocketChannel.open();
            serverChannel.bind(new InetSocketAddress(PORT));
            serverChannel.configureBlocking(block:false);
            serverChannel.register(selector, SelectionKey.OP_ACCEPT);
            System.out.println("Server started on port " + PORT);

            while (true) {
                selector.select();
                Set<SelectionKey> selectedKeys = selector.selectedKeys();
                Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
                while (keyIterator.hasNext()) {
                    SelectionKey key = keyIterator.next();
                    if (key.isAcceptable()) {
                        ServerSocketChannel server = (ServerSocketChannel) key.channel();
                        SocketChannel client = server.accept();
                        client.configureBlocking(block:false);
                        client.register(selector, SelectionKey.OP_READ);
                        System.out.println("New client connected: " + client.getRemoteAddress());
                    } else if (key.isReadable()) {
                        SocketChannel client = (SocketChannel) key.channel();
                        ByteBuffer buffer = ByteBuffer.allocate(capacity:1024);
                        int bytesRead = client.read(buffer);
                        if (bytesRead == -1) {
                            client.close();
                            System.out.println("Client disconnected: " + client.getRemoteAddress()
                        } else if (bytesRead > 0) {
                            buffer.flip();
                            byte[] data = new byte[buffer.limit()];
                            buffer.get(data);
                            String message = new String(data);
                            System.out.println("Received from client: " + message);
                            client.write(ByteBuffer.wrap(data));
                            buffer.clear();
                        }
                    }
                    keyIterator.remove();
                }
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```