

# Netty-Starter-With-Spring-Boot (사전 학습)

---

## Code Examples Available in a My GitHub Repository

### 1. bit와 byte

- [1- 1. 왜 bit와 byte를 알아야 할까요?](#)
- [1- 2. 비트\(bit\)란?](#)
- [1- 3. 비트의 특징](#)
- [1- 4. bit 연산 실습 예제](#)
- [1-5. 바이트 단위 데이터 처리 실습 예제](#)

### 2. Protocol

- [2-1. Protocol이 뭘까? \(TCP/IP, HTTP\)](#)
- [2.2 OSI \(Open Systems Interconnection Reference Model\) 7계층이란?](#)
- [2-2. TCP/IP \(Transmission Control Protocol/Internet Protocol\)](#)
- [2-3. HTTP\(Hypertext Transfer Protocol\)](#)
- [2-4. TCP/패킷과 세그먼트 차이가 뭘까?](#)

### 3. synchronous (동기) / asynchronous (비동기)

- [3-1. 동기 ? 비동기..?](#)
- [3-2. Sync\(동기 처리\)란?](#)
- [3-3. Async\(비동기 처리\)란?](#)

### 4. IO (블로킹) /NIO (논블로킹)

- [4-1. IO랑 NIO에 대해 알아야 하는 이유?](#)
- [4-2. IO\(Input/Output\)이란?](#)
- [4-3. NIO\(Non-blocking Input/Output\) 이란?](#)
- [4-4. Java IO에서 쓰는 Stream은 뭘까?](#)
- [4-5. Java NIO에서 쓰는 Channel은 뭘까?](#)
- [4-6. Stream / Channel 차이](#)
- [4.7 버퍼가 자꾸 나오네.. 뭘까요?](#)
- [4.8 IO Code Example](#)
- [4-9. NIO Code Example](#)

### 5. 이벤트 주도 모델

- [5-1. 개념](#)
- [5-2. 동작 방식](#)
- [5-3. 장점](#)
- [5-4. 단점](#)

## Code Examples Available in a My GitHub Repository

- 이 프로젝트의 코드는 GitHub에 공개되어 있습니다.

- **GitHub 저장소 주소:** <https://github.com/dev-vihaan-ji/netty-starter-with-spring-boot>
- **사용한 브랜치:** `day0`
- **docs:** 폴더에 문서가 저장되어 있으니 참고하시면 될 것 같습니다.

## 1. bit와 byte

### 1- 1. 왜 bit와 byte를 알아야 할까요?



#### 네트워크 통신 이해

- 네트워크 통신은 데이터를 비트와 바이트 단위로 전송합니다.
- 비트와 바이트에 대한 이해가 없다면 네트워크 통신의 기본 원리를 이해하기 어려울 수 있습니다.



#### 데이터 표현 이해

- Netty는 데이터를 버퍼(buffer)에 저장하고 처리합니다.
- 이때 비트와 바이트 단위로 데이터를 다루게 됩니다.
- 비트와 바이트에 대한 이해가 없다면 Netty에서 데이터를 효과적으로 처리하기 어려울 수 있습니다.



#### 성능 최적화

- 네트워크 통신에서 데이터 크기는 성능에 큰 영향을 미칩니다.
- 비트와 바이트에 대한 이해를 바탕으로 데이터 크기를 최소화하여 성능을 향상시킬 수 있습니다.



#### 디버깅 및 문제 해결

- 네트워크 통신 문제를 해결할 때 비트와 바이트 단위로 데이터를 분석해야 할 경우가 있습니다.
- 이때 비트와 바이트에 대한 이해가 필수적입니다.

### 1- 2. 비트(bit)란?



비트는 **Binary Digit**의 줄임말로 컴퓨터 과학에서 가장 작은 단위이며 0과 1의 값으로 구성(이진수)되어 있습니다.

이 bit는 컴퓨터에서 가장 기본이 되는 개념이며, 데이터를 표현하고 처리하는 데 사용됩니다.

## 1- 3. 비트의 특징

### 1. 이진 체계

- a. 이진 체계는 0과 1 두 개의 값을 사용하여 모든 데이터를 표현할 수 있는 방법(체계)입니다.
- b. 컴퓨터는 전기 신호(=전구)나 자기장(=자석) 등의 물리적인 수단을 사용하여 **bit**를 나타낼 수 있습니다.
- c. 만약 전구 스위치가 꺼져서 전류가 흐르지 않는 상태를 0으로 표현할 수 있을 것이고, 전구 스위치가 켜져서 전류가 흐르는 상태를 1로 표현할 수 있을 것입니다.

### 2. 데이터 표현

- a. **bit**를 사용하여 다양한 종류의 데이터를 표현할 수 있습니다.
- b. 예를 들면, 숫자, 문자, 이미지, 음성 등의 데이터를 이진 형태로 나타낼 수 있습니다.
- c. 컴퓨터에서 문자를 표현하기 위해서는 문자 인코딩 방식(ASCII, UTF-8)을 사용해서 **bit**로 매핑한 뒤 문자를 표현합니다.
- d. 예를 들면, 이진수 101을 인코딩 방식으로 변환하면 숫자 5가 됩니다.

### 3. 비트 묶음과 바이트

- a. **bit**는 컴퓨터에서 가장 작은 단위이며, 컴퓨터의 데이터를 나타내는 데 중요한 역할을 합니다.
- b. 여러 개의 **bit**를 조합해서 더 큰 데이터 단위를 표현할 수 있습니다.
- c. 8개의 **bit**를 묶어서 하나의 단위로 표현할 수 있는데, 이를 바이트(**byte**)라고 합니다.
- d. **byte**가 가장 일반적인 데이터 단위이지만, 다른 단위도 존재합니다.

### 4. 바이트(Byte)

- a. 컴퓨터는 일반적으로 8개 단위의 비트(**8 bit**)를 하나의 그룹으로 사용하는데 이를 바이트(**byte**)라 합니다. 즉 (**8 bit = 1 byte**)
- b. **byte**는 많은 컴퓨터 시스템에서 기본적인 데이터 단위로 사용된다. **byte**가 가장 일반적인 데이터 단위입니다.
- c. **byte**는  $256(2^8)$ 가지의 서로 다른 값을 나타낼 수 있으며, 문자, 숫자, 그래픽 등을 표현하는 데 사용됩니다.

d. 예를 들면, 파일 크기, 메모리 용량 등을 `byte` 단위로 표현할 수 있습니다.

## 5. 컴퓨터 구성

a. 컴퓨터의 CPU(중앙 처리 장치), 메모리(Memory), 저장 장치(Storage) 등은 모두 `bit`를 사용하여 데이터를 처리합니다.

b. `bit`는 컴퓨터의 처리 속도, 저장 용량, 통신 속도 등을 결정하는 중요한 요소입니다.

c. 컴퓨터의 비트 수는 해당 컴퓨터 시스템이 처리할 수 있는 데이터의 크기와 범위를 결정합니다.

d. 비트 수가 많을수록 컴퓨터의 처리 속도, 저장 용량, 통신 속도 등이 향상됩니다.

## 6. 컴퓨터 아키텍처(Architecture)

a. 컴퓨터 시스템의 아키텍처는 `bit`의 크기에 따라 결정됩니다.

b. 예를 들어, 32비트 아키텍처는 각 워드(`word`)가 32비트로 구성되어 있고, 64비트 아키텍처는 각 워드(`word`)가 64비트로 구성되어 있다.

i. 워드(`word`) : 여기서 워드는 컴퓨터 구조에서 하나의 연산을 통해 저장 장치로부터 프로세서의 레지스터에 옮겨놓을 수 있는 데이터 단위를 의미합니다.

ii. 즉, 컴퓨터에서 저장할 수 있는 데이터(연산)의 단위입니다.

c. 윈도우 `32 bit` : `4,294,967,296`의 데이터 크기 처리 가능합니다.

d. 윈도우 `64 bit` : `18,446,744,073,709,600,000`의 데이터 크기 처리 가능합니다.

## 1- 4. bit 연산 실습 예제

`code example`

```

public class BitOperationExample {
    public static void main(String[] args) {
        // 비트 연산 실습
        // java 2진수 표기법 : 0b
        // java 8진수 표기법 : 0
        // java 16진수 표기법 : 0x or 0X
        int a = 0b1010; // 10진수로 10
        int b = 0b0110; // 10진수로 6

        // 두 비트가 모두 1인 경우에만 결과가 1이 되고, 나머지는 모두 0이 됩니다.
        // AND 연산
        int andResult = a & b; // 결과: 0b0010 (2)
        System.out.println("AND 연산 결과: " + andResult);

        // 두 비트 중 하나라도 1이라면 결과가 1이 됩니다.
        // OR 연산
        int orResult = a | b; // 결과: 0b1110 (14)
        System.out.println("OR 연산 결과: " + orResult);

        // 두 비트가 다르다면 결과가 1이 되고, 같으면 0이 됩니다.
        // XOR 연산
        int xorResult = a ^ b; // 결과: 0b1100 (12)
        System.out.println("XOR 연산 결과: " + xorResult);

        // 0을 1로, 1을 0으로 반전시킵니다.
        // NOT 연산
        int notResult = ~a; // 결과: 0b0101 (-11)
        System.out.println("NOT 연산 결과: " + notResult);
    }
}

```

## 1-5. 바이트 단위 데이터 처리 실습 예제

code example

```

public class ByteOperationExample {
    public static void main(String[] args) {
        // 바이트 단위 데이터 처리 실습
        byte b1 = 0b01000001; // 65 (ASCII 'A')
        byte b2 = 0b00110100; // 52 (ASCII '4') // 바이트 값 확인
        System.out.println("b1: " + b1);
        System.out.println("b2: " + b2);

        // 바이트 배열 생성
        byte[] bytes = {b1, b2};

        // 바이트 배열 출력
        for (byte b : bytes) {
            System.out.print((char) b); // 출력: A4
        }
        System.out.println();
    }
}

```

## 2. Protocol

### 2-1. Protocol이 뭘까? (TCP/IP, HTTP)

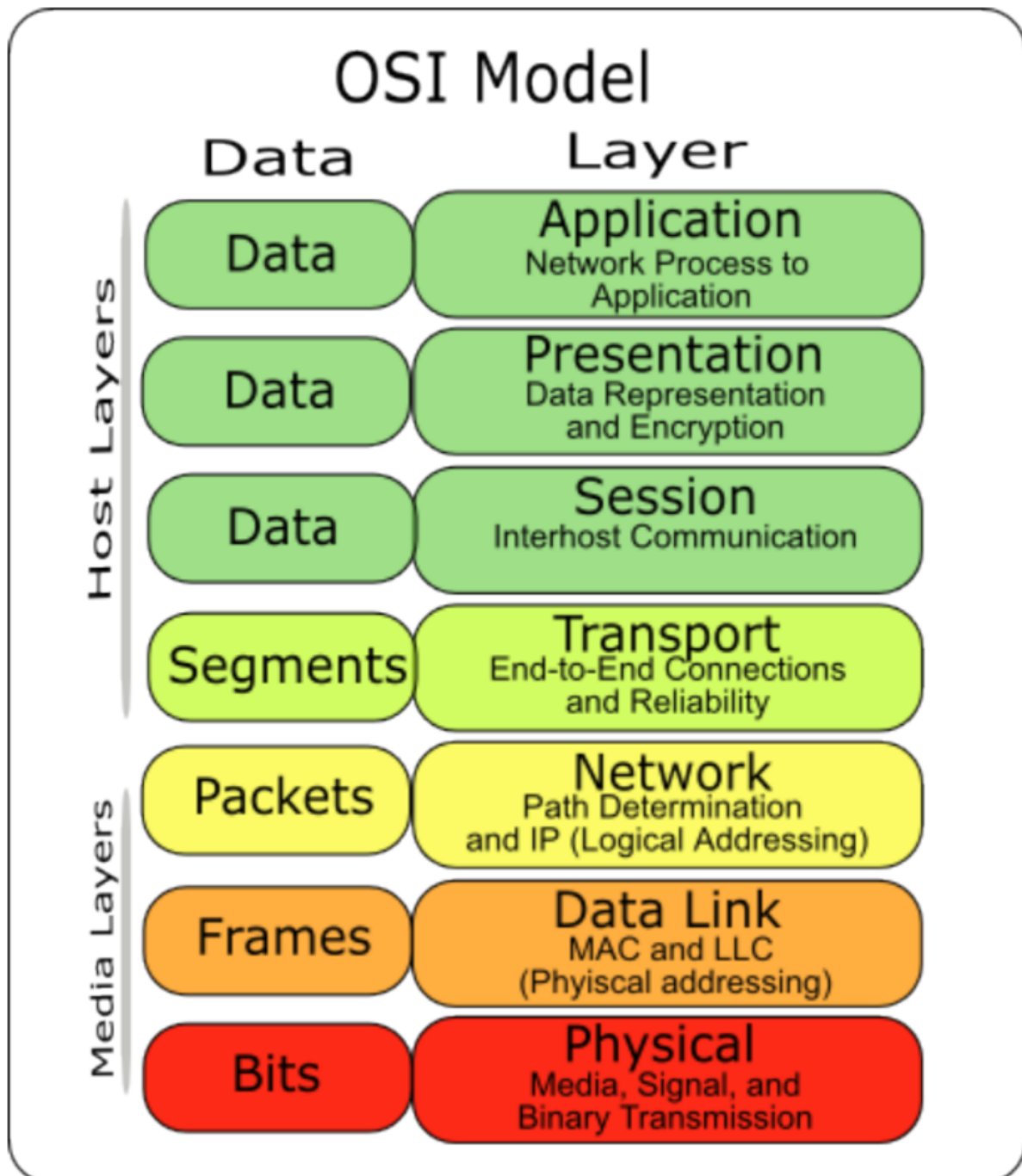


프로토콜 (**Protocol**)은 컴퓨터 네트워크에서 통신을 위한 규칙 및 규약을 의미합니다. 프로토콜은 네트워크 상에서 데이터를 주고받는 방법을 정의합니다. 대표적인 프로토콜로는 아래와 같이 있습니다.

### 2.2 OSI (Open Systems Interconnection Reference Model) 7계층이란?

1. 물리 계층( **Physical Layer** )개념
  - a. 전기적, 물리적 신호를 전송하는 계층
  - b. 예시: 물리적 케이블, 광섬유
2. 데이터 링크 계층( **Data Link Layer** )개념

- a. 신뢰성 있는 데이터 전송을 위한 계층
- b. 예시: HDLC, MAC
- 3. 네트워크 계층( **Network Layer** )개념
  - a. 데이터 경로 선택 및 논리적 주소 지정을 담당하는 계층
  - b. 예시: IP(IPv4, IPv6), ICMP, ARP, OSPF, BGP
- 4. 전송 계층( **Transport Layer** )개념
  - a. 애플리케이션 간 신뢰성 있는 데이터 전송을 담당하는 계층
  - b. 예시: TCP(Transmission Control Protocol), UDP(User Datagram Protocol)
- 5. 세션 계층( **Session Layer** )개념
  - a. 애플리케이션 간 대화 관리 및 동기화를 담당하는 계층
  - b. 예시: RPC, SSL/TLS
- 6. 표현 계층( **Presentation Layer** )개념
  - a. 데이터 형식 및 암호화/압축을 담당하는 계층
  - b. 예시: ASCII, JPEG, MPEG, MIDI, EBCDIC
- 7. 응용 계층( **Application Layer** )개념
  - a. 사용자 애플리케이션에서 직접 사용되는 계층
  - b. 예시: HTTP, FTP, SMTP, POP3, IMAP, DNS, DHCP



- [출처] : "[https://www.researchgate.net/figure/Open-Systems-Interconnection-OSI-reference-model\\_fig1\\_30850107](https://www.researchgate.net/figure/Open-Systems-Interconnection-OSI-reference-model_fig1_30850107)"

## 2-2. TCP/IP (Transmission Control Protocol/Internet Protocol)

- TCP/IP 프로토콜은 네트워크 계층(IP)과 전송 계층(TCP)으로 구성됩니다.
- 인터넷 통신의 기반이 되는 가장 대표적인 프로토콜입니다.
- IP 프로토콜은 데이터 패킷이 네트워크를 통해 이동하고 올바른 대상에 도착할 수 있도록 데이터 패킷을 라우팅하고 주소를 지정하기 위한 프로토콜 또는 규칙의 집합입니다.



- TCP는 연결 지향형 프로토콜로 클라이언트와 서버 간의 연결을 설정하고 데이터를 안정적으로 전송합니다.

## 2-3. HTTP(Hypertext Transfer Protocol)

- 저희가 흔히 아는 월드 와이드 웹(WWW)에서 클라이언트와 서버 간의 통신을 위해 사용되는 프로토콜입니다.
- 웹 브라우저와 웹 서버 간의 요청-응답 방식으로 동작합니다.
- 주로 HTML, CSS, JavaScript 등의 웹 리소스를 전송하는 데 사용됩니다.
- 이 외에도 FTP(File Transfer Protocol), SMTP(Simple Mail Transfer Protocol) 등 다양한 프로토콜이 존재합니다.

## 2-4. TCP/패킷과 세그먼트 차이가 뭘까?

- IP 패킷(IP Packet)의 개념:
  - IP 패킷은 네트워크 계층(IP 계층)에서 사용되는 데이터 단위입니다.
  - IP 패킷은 IP 헤더와 데이터 부분으로 구성됩니다.
  - IP 패킷은 네트워크를 통해 전송되는 데이터의 기본 단위입니다.
- TCP 세그먼트(TCP Segment)의 개념:
  - TCP 세그먼트는 전송 계층(TCP 계층)에서 사용되는 데이터 단위입니다.
  - TCP 세그먼트는 TCP 헤더와 데이터 부분으로 구성됩니다.
  - TCP 세그먼트는 TCP 계층에서 생성되어 IP 계층으로 전달됩니다.
- 차이점:
  - IP 패킷은 네트워크 계층에서 사용되는 데이터 단위이고, TCP 세그먼트는 전송 계층에서 사용되는 데이터 단위입니다.
  - IP 패킷에는 IP 헤더가 포함되지만, TCP 세그먼트에는 IP 헤더가 포함되지 않습니다.
    - IP 패킷: IP 프로토콜에서 사용되는 데이터 전송 단위입니다. IP 패킷에는 IP 헤더가 포함됩니다. IP 헤더에는 송신지 IP 주소, 수신지 IP 주소 등 IP 프로토콜에 필요한 정보가 담겨 있습니다.
    - TCP 세그먼트: TCP 프로토콜에서 사용되는 데이터 전송 단위입니다. TCP 세그먼트에는 TCP 헤더가 포함되지만, IP 헤더는 포함되지 않습니다.

즉, IP 패킷은 IP 프로토콜에 필요한 정보를 담고 있는 IP 헤더를 가지고 있지만, TCP 세그먼트는 TCP 프로토콜에 필요한 정보만 담고 있는 TCP 헤더만 가지고 있습니다.

- TCP 세그먼트는 TCP 계층에서 생성되어 IP 계층으로 전달되어 IP 패킷이 됩니다.

- TCP 세그먼트는 TCP 계층의 기능(흐름 제어, 오류 제어 등)을 수행하지만, IP 패킷은 이러한 기능을 수행하지 않습니다.

## 3. synchronous (동기) / asynchronous (비동기)

### 3-1. 동기 ? 비동기..?



동기 방식 : I/O 요청을 보내고 응답을 기다리는 동안 다른 작업을 할 수 없습니다.

비동기 방식 : I/O 요청을 보내고 다른 작업을 할 수 있습니다.

Netty의 비동기 모델을 이해하려면 이런 차이점을 알아야 합니다.

### 3-2. Sync(동기 처리)란?

- 동기 방식은 한 작업이 완료되어야만 다음 작업이 실행이 되는 방식입니다.
- 동기 방식의 장단점
  - 장점:
    - 코드의 실행 순서가 직관적이고 알기 쉽습니다.
    - 작업의 완료 여부를 명확하게 알 수 있습니다.
  - 단점:
    - 한 작업이 완료되어야만 다음 작업이 실행되므로, 오래 걸리는 작업이 있으면 전체 애플리케이션의 성능이 저하될 수 있습니다.
    - 작업이 완료될 때까지 다른 작업을 할 수 없어 비효율적일 수 있습니다.
- 동기 처리에 대한 예시
  - 식당에서 음식을 주문하면 그 사람은 주문한 음식이 나올 때까지 기다려야 합니다.
  - 그 사람은 다른 일을 할 수 없고 음식이 나올 때까지 가만히 있습니다.

### 3-3. Async(비동기 처리)란?

- 비동기 방식은 작업의 실행이 완료되지 않더라도, 다음 작업이 실행되는 방식입니다.
- 비동기 방식의 장단점
  - 장점
    - 시간이 오래 걸리는 작업이 있어도 다른 작업들이 계속해서 실행될 수 있어 애플리케이션의 전체 처리 속도와 성능을 향상시킬 수 있습니다.
    - 작업을 병렬적으로 처리할 수 있어 효율적입니다.

- 단점
  - 코드의 실행 순서가 직관적이지 않을 수 있습니다.
  - 여러 작업이 동시에 실행되는 경우 동기화와 같은 문제를 처리해야 할 수 있습니다.
- 비동기 처리에 대한 예시
  - 식당에서 음식을 주문하고 식사를 하면서 다른 일을 할 수 있습니다.
  - 음식이 나오면 알려주고, 그때 음식을 받으면 됩니다.
  - 다른 일을 하면서 음식이 나오기를 기다릴 수 있습니다.

### 3-4 코드 예시

- Sync / Async Example class

```

public class SyncAsyncExample {

    public static void main(String[] args) throws InterruptedException {

        // 음식을 주문합니다.
        Order order = new Order("pizza", 3000);

        // 레스토랑에서 음식을 주문하고 대기합니다.
        // 음식이 나올 때까지 일을 하지 못 합니다.
        // 동기 방식의 주문 처리
        Restaurant.processSyncOrder(order);

        // 레스토랑에서 음식을 주문하고 다른 일을 할 수 있습니다.
        Restaurant.processAsyncOrder(order);

        // is synchronous == true don't anything else until the food is served.
        // is asynchronous == true do anything else until the food is served.
        String threadName = Thread.currentThread().getName();
        System.out.println "[" + threadName + "] 음식이 나올 때까지 다른 일을 하고 싶어.");
        System.out.println "[" + threadName + "] 의자에 앉아서 책을 읽을래.");

        // 음식이 나올 때까지 기다립니다.
        Thread.sleep(15000);
    }
}

class Restaurant {
    private Order order;

    // 동기식으로 음식 준비
    public static void processSyncOrder(Order order) {
        String threadName = Thread.currentThread().getName();
        try {
            // 현재 쓰레드 이름 출력
            System.out.println "[" + threadName + "] processSyncOrder() start order [" + order + "]
default interval 10000 seconds.");
            Thread.sleep(10000);
            System.out.println "[" + threadName + "] processSyncOrder() finish!";
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // 비동기식으로 음식 준비
    public static void processAsyncOrder(Order order) {
        CompletableFuture.runAsync(() -> {
            String threadName = Thread.currentThread().getName();
            try {
                // 현재 쓰레드 이름 출력
                System.out.println "[" + threadName + "] processAsyncOrder() start order [" + order +
"] default interval 10000 seconds.");
                Thread.sleep(10000);
                System.out.println "[" + threadName + "] processAsyncOrder() finish!";
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
    }
}

/**
 * Order Object
 */
class Order {
    private final String foodName;
    private final int foodPrice;

    public Order(String foodName, int foodPrice) {
        this.foodName = foodName;
        this.foodPrice = foodPrice;
    }

    @Override
    public String toString() {
        return "Order{" +
            "foodName='" + foodName + '\'' +
            ", foodPrice=" + foodPrice +
            '\'';
    }
}

```

- `Restaurant class processSyncOrder` (동기 처리 메소드) 다른 작업을 할 수 없습니다.
- `Restaurant class processAsyncOrder` (비동기 처리 메소드) 다른 작업을 할 수 있습니다.

## 4. IO (블로킹) /NIO (논블로킹)

### 4-1. IO랑 NIO에 대해 알아야 하는 이유?



- Netty는 NIO를 기반으로 만들어진 프레임워크로서, NIO에 대한 이해가 있어야 합니다.
- IO와 NIO의 차이를 이해하면 Netty의 장점과 특징을 더 잘 파악할 수 있습니다.
- 또한 Netty를 사용할 때 발생하는 성능 문제 또는 버그를 해결하기 위해서는 NIO에 대한 이해가 필수적으로 필요합니다.

### 4-2. IO(Input/Output)이란?

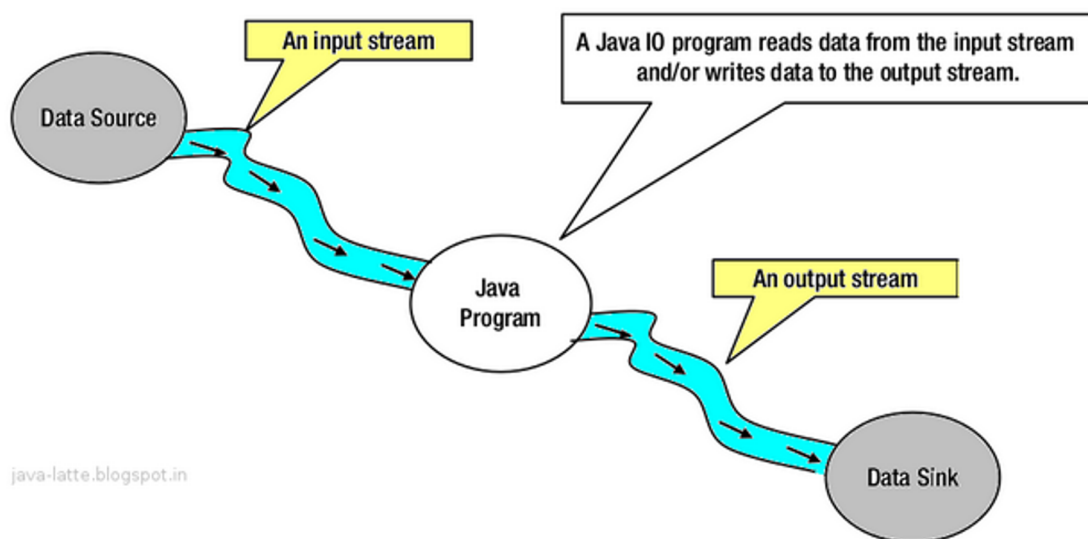
- IO는 기존의 자바 입출력 모델로, 입출력 작업을 수행할 때 스레드를 블로킹 하는 방식입니다.
- IO의 장단점
  - 입출력 작업이 완료될 때까지 스레드가 대기하므로, 입출력 작업이 오래 걸리면 해당 스레드는 그 시간 동안 다른 작업을 할 수 없게 됩니다.
  - 따라서 IO 모델은 애플리케이션에서 성능 문제가 발생할 수 있습니다.
- IO의 예시
  - 기존의 자바 입출력 모델에서 사용되는 개념은 `Stream`으로서 `InputStream`, `OutputStream` 등과 같은 클래스들을 사용합니다.

### 4-3. NIO(Non-blocking Input/Output) 이란?

- NIO는 비동기 입출력을 지원하며, 입출력 작업이 완료될 때까지 스레드를 블로킹하지 않습니다.
- NIO의 장단점
  - NIO 모델은 스트림 기반 IO모델에 비해 더 높은 성능과 확장성을 제공합니다.
  - 하지만 NIO 모델은 IO 모델에 비해 구현이 복잡하고 이해하기 어려울 수 있습니다.
- NIO의 예시
  - 자바의 NIO 모델에서 사용되는 개념은 `Channel`로서 `SocketChannel`, `ServerSocketChannel` 등과 같은 클래스들을 사용합니다.

### 4-4. Java IO에서 쓰는 Stream은 뭘까?

- Java IO에서 사용하는 **Stream** 은 데이터를 읽고 쓰는 데 사용되는 추상화된 클래스입니다.
- **Stream** 은 바이트 단위로 데이터를 처리하며, 입력 스트림( **InputStream** )과 출력 스트림( **OutputStream** )으로 구분됩니다.
- 이를 통해 애플리케이션은 파일, 네트워크 소켓, 메모리 버퍼 등 다양한 데이터 소스에서 데이터를 읽고 쓸 수 있습니다.
- **Stream** 의 특징
  - FIFO (First-In-First-Out) 구조:
    - 스트림은 FIFO 구조를 가지고 있습니다.
    - 먼저 들어간 데이터가 먼저 나오는 형태로, 데이터의 순서가 바뀌지 않습니다.
  - 단방향 (Unidirectional):
    - 하나의 스트림에서 입력과 출력이 동시에 이루어지지 않습니다.
    - 입력과 출력에 각각 별도의 스트림을 열어 사용해야 합니다.
  - 지연 가능 (Blocking)
    - 스트림 내의 데이터가 모두 전송되기 전까지 프로그램이 지연 상태에 빠질 수 있습니다.
    - 데이터가 완전히 전송될 때까지 프로그램이 대기하게 됩니다.



*Flow of data using an input/output stream in a Java program*

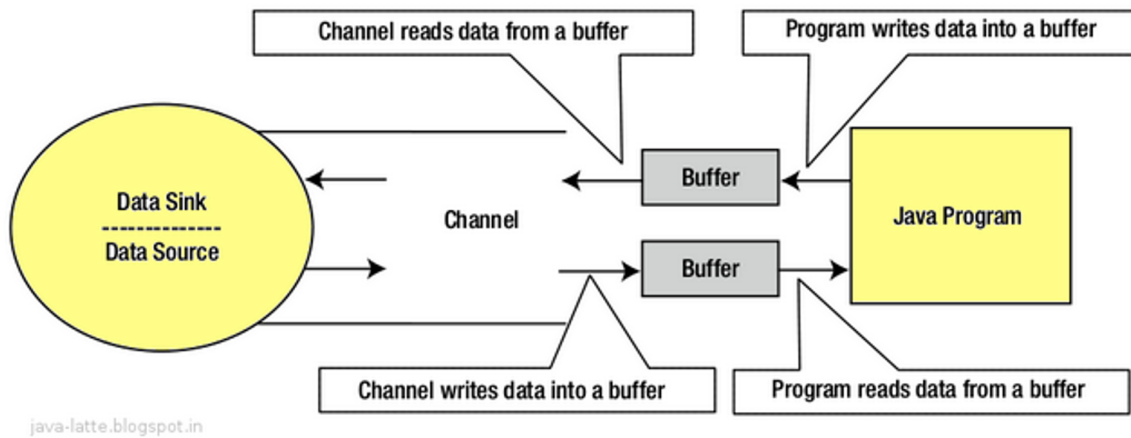
#### 4-5. Java NIO에서 쓰는 Channel은 뭘까?

- Java NIO에서 사용하는 **Channel** 은 입출력 작업을 수행하는 추상화된 클래스입니다.

- `Channel` 은 버퍼( `Buffer` )와 함께 사용되며, 파일, 네트워크 소켓, 데이터그램 소켓 등 다양한 I/O 소스에 대한 비동기적인 입출력을 지원합니다.
- `Channel` 은 스트림과 달리 데이터를 바이트 단위가 아닌 `버퍼 단위로 처리합니다` .

## 4-6. Stream / Channel 차이

- 데이터 처리 방식:
  - **Stream**: 바이트 단위로 데이터를 처리
  - **Channel**: 버퍼 단위로 데이터를 처리
  - 스트림 기반 I/O에서는 데이터를 `stream`에 직접 쓰지만 채널 기반 I/O에서는 데이터를 버퍼에 넣는다
- 동기/비동기 처리:
  - **Stream**: 기본적으로 동기 처리
  - **Channel**: 비동기 처리를 지원
- 성능:
  - **Stream**: 상대적으로 낮은 성능
  - **Channel**: 높은 성능, 대용량 데이터 처리에 유리
- 사용 목적:
  - **Stream**: 일반적인 입출력 작업
  - **Channel**: 대용량 데이터 처리, 비동기 I/O
- 예외 처리:
  - **Stream**: `IOException` 처리
  - **Channel**: `IOException` 외에 다양한 예외 처리 필요



Interaction between a channel, buffers, a Java program, a data source, and a data sink

- 채널은 데이터 소스로부터 데이터를 읽어서 버퍼로 전달합니다.
- 우리는 버퍼로부터 데이터를 읽을 수 있습니다.
- 위의 다이어그램은 채널, 버퍼, 데이터소스, 데이터싱크, 자바 프로그램 간의 상호 작용을 도식화한 것입니다.

## 4.7 버퍼가 자꾸 나오네.. 뭘까요?



**Buffer** 는 두 장치간 데이터를 주고 받을 때 전송 속도, 처리 속도 차이 해결을 위해 데이터를 임시 저장하는 공간입니다.

### 1. 속도 차이 조절

- 입력 장치(예: 키보드, 마우스)와 출력 장치(예: 모니터, 스피커) 간의 속도 차이를 해결하기 위해 사용됩니다.
- 입력이 빠르고 출력이 느린 경우, 버퍼에 데이터를 임시로 저장하여 출력 속도를 맞출 수 있습니다.

### 2. 데이터 전송 효율화

- 데이터를 한 번에 전송하는 것보다 버퍼에 모아두고 한꺼번에 전송하는 것이 효율적일 수 있습니다.
- 이를 통해 전송 오버헤드를 줄일 수 있습니다.

### 3. 버퍼 예시

- 아래 링크를 참조해 주시면 됩니다.



### [개념정리] 버퍼(Buffer)란? 버퍼 개념

버퍼(Buffer)란 버퍼란 임시 저장 공간을 의미 합니다. 임시 저장 공간이라고 해서 생동맞게 보일 수 있지만 정확히 말하면 A와 B가 서로 입출력을 수행하는데 있어서 속도차이를 극복하기 위해 사용하는 임시 저장 공간을

<https://dololak.tistory.com/84>

가능

초당 100개



### 버퍼를 사용하는 이유 (Feat. BufferedReader vs Scanner & cache)

최근 JAVA로 알고리즘 공부를 하면서 Scanner를 사용하면 Fail이 나오지만, BufferedReader를 사용하면 통과하는 기상천외한 상황을 겪으면서.. 이와 관련된 게시글들을 찾아보게 되었다. 그 과정에서 내가 사용하던 BufferedReader는 기존

<https://velog.io/@kkimbj18/버퍼를-사용하는-이유-Feat.-BufferedReader-vs-Scanner-cache>

< 버퍼를 사용하지 않는 입력 >

키보드의 입력이 키를 누르는 즉시 바로 전달됨

< 버퍼를 사용하는 입력 >



보드의 입력이 있을 때마다 문자씩 버퍼로 전송함

버퍼가 가득 차거나, 개행 문자 버퍼의 내용을 한 번에 전송함

- 이처럼 버퍼는 입출력 속도 차이를 조절하고 데이터 전송 효율을 높이는 데 매우 중요한 역할을 합니다.

## 4.8 IO Code Example

- Stream을 이용하여 File Read And Write

code example

```

public class IOExample {

    public static void main(String[] args) {
        // 입력 파일 처리 메서드 호출
        inputFile();
        // 출력 파일 처리 메서드 호출
        outputFile();
    }

    private static void inputFile() {
        // 입력 파일 객체 생성
        File file = new File("input.txt");
        // 파일이 존재하는 경우
        if (file.exists()) {
            try (FileInputStream fis = new FileInputStream(file)) {
                // 파일 읽기 변수
                int bytesRead;
                // 파일 끝까지 읽기 -1인 경우 더 이상 읽을 데이터가 없습니다.
                while ((bytesRead = fis.read()) != -1) {
                    // 콘솔에 문자 출력
                    System.out.print((char) bytesRead);
                }
            } catch (IOException e) {
                // 예외 처리
                e.printStackTrace();
            }
        }
    }

    private static void outputFile() {
        // 출력 파일 객체 생성
        File file = new File("output.txt");
        // 파일이 존재하는 경우
        if (file.exists()) {
            try (FileOutputStream fos = new FileOutputStream(file)) {
                // 출력할 문자열
                String outputText = "Hello Output Stream";
                // 문자열을 바이트로 변환
                byte[] outputBytes = outputText.getBytes();
                // 파일에 바이트 데이터 쓰기
                fos.write(outputBytes);
            } catch (IOException e) {
                // 예외 처리
                e.printStackTrace();
            }
        }
    }
}

```

## 4-9. NIO Code Example

- Channel을 이용하여 File Read And Write

code example

```
public class NioExample {

    public static void main(String[] args) {
        // 여기서 두 개의 메서드 nioInputFile()과 nioOutputFile()을 호출합니다.
        nioInputFile();
        nioOutputFile();
    }

    // nioInputFile() 메서드는 "input.txt" 파일을 읽어서 콘솔에 출력합니다.
    public static void nioInputFile() {
        String fileName = "input.txt";
        Path filePath = Paths.get(fileName);
        if (Files.exists(filePath)) {
            // FileChannel을 사용하여 파일을 읽습니다.
            try (FileChannel channel = FileChannel.open(filePath, StandardOpenOption.READ)) {
                // 파일 크기만큼의 ByteBuffer 생성
                ByteBuffer byteBuffer = ByteBuffer.allocate((int) channel.size());
                // 파일을 읽어서 ByteBuffer에 저장
                while (channel.read(byteBuffer) != -1) {
                    // ByteBuffer의 위치를 리셋
                    byteBuffer.flip();
                    // ByteBuffer의 내용을 콘솔에 출력
                    System.out.write(byteBuffer.array(), 0, byteBuffer.limit());
                    // ByteBuffer 초기화
                    byteBuffer.clear();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    // nioOutputFile() 메서드는 "output.txt" 파일에 아래 내용을 씁니다.
    public static void nioOutputFile() {
        String fileName = "output.txt";
        Path filePath = Paths.get(fileName);
        if (Files.exists(filePath)) {
            // FileChannel을 사용하여 파일에 쓰기
            try (FileChannel channel = FileChannel.open(filePath, StandardOpenOption.WRITE)) {
                String outputText = "Hello Nio ByteBuffer Output Channel";
                // 문자열을 바이트 배열로 변환
                byte[] outputBytes = outputText.getBytes(StandardCharsets.UTF_8);
                // 바이트 배열을 ByteBuffer로 래핑
                ByteBuffer byteBuffer = ByteBuffer.wrap(outputBytes);
                // ByteBuffer의 내용을 파일에 쓰기
                int writeSize = channel.write(byteBuffer);
            } catch (IOException e) {
                e.printStackTrace(); // 예외 처리
            }
        }
    }
}
```

## 5. 이벤트 주도 모델

## 5-1. 개념

- 이벤트 주도 모델은 시스템 내부에서 발생하는 이벤트(Event)를 중심으로 동작하는 프로그래밍 패턴입니다.
- 이벤트는 특정 상황이나 행동이 발생했음을 알리는 신호로, 이를 처리하는 코드를 이벤트 핸들러(Event Handler)라고 합니다.

## 5-2. 동작 방식

- 이벤트가 발생하면 이벤트 핸들러가 이를 감지하고 해당 이벤트를 처리합니다.
- 이벤트 핸들러는 이벤트에 대한 응답 코드를 실행하여 시스템의 상태를 변경하거나 다른 작업을 수행합니다.
- 이벤트 핸들러는 이벤트가 발생할 때까지 대기하는 비동기적인 방식으로 동작합니다.

## 5-3. 장점

- 이벤트 주도 모델은 시스템의 구성 요소 간 느슨한 결합을 가능하게 하여 유연성과 확장성을 높입니다.
- 이벤트 처리가 비동기적으로 이루어지므로 시스템의 응답성과 효율성이 높습니다.
- 이벤트 중심의 설계로 인해 시스템의 복잡성을 관리하기 쉽습니다.

## 5-4. 단점

- 이벤트 간의 의존성이 복잡해질 수 있어 시스템의 전체적인 흐름을 파악하기 어려울 수 있습니다.
- 이벤트 처리 과정에서 예기치 못한 상황이 발생할 수 있으며, 이를 디버깅하기 어려울 수 있습니다.
- 이벤트 핸들러 간의 순서 및 타이밍 문제로 인해 시스템의 동작이 예측하기 어려워질 수 있습니다.