면접 CS 공부

▼ JAVA

▼ equals(), hashcode()

- 공통
  - Object 메서드에서 포함된 메소드
  - 객체의 동등성 비교와 해시 기반 컬렉션 사용을 지원하기 위한 메서드
- equals
  - 두 객체의 레퍼런스를 비교하여 동일한 객체인지 비교합니다.
    (두 객체의 메모리 주소) - 동일성
  - 오버라이딩 규칙
    반사성 (x.equals(x) 는 반드시 true)
    대칭성 (x.equals(y) 가 true 면 y.equals(x)도 true)
    추이성 (x.equals(y) 가 true, y.equals(z) 도 true 면 x.equals(z) 도 true)
  - 구현 방법
    1. 자신이 참조인지 체크 한다. (==)
    2. instanceof 명령어로 자신의 타입이 맞는지 체크한다. (with null check)
    3. 핵심 필드 (동등성을 만족해주는 필드) 에 맞춰서 boolean 값을 반환한다.
- hashcode
  - native 메서드를 사용해서 메모리 주소를 참조한다.
    (즉, a.equals(b) 는 false, a.hashcode() == b.hashcode() 도 false 이다)
  - 구현 방법
    1. int 변수 하나를 만든다.
    2. 핵심 필드 타입의 따라 다름
       a. 기본 타입 이라면 기본타입.hashcode(field) (박싱된 기본타입)
       b. 참조 타입 이라면 해당 타입이 equals 과 hashcode 를 재구성했다면 재귀적으로 hashcode 를 만들어 호출한다.
       c. 배열 이라면 모든 원소가 핵심이라면 Arrays.hashcode 를 사용한다.
    3. 계산된 값을 지속적으로 갱신해준다.
       result = Integer.hashcode(a);
       result = 31 * result + Integer.hashcode(b);
  - 31 (쉬프트 연산, 소수(소수가 아닌 수를 곱한다면, 보다 같은 hash 값을 갖을 경우의 수가 많아짐으로 소수를 택한 것으로 생각됨.)
- 만약 new Point(1, 2), new Point(1, 2) 가 있다고 치면
  - 둘은 사실상 같은 객체(동일성)가 아니지만 동등하다고 볼 수 있다. (동등성)
    동등성을 이루기 위해서는 두 메서드를 반드시 오버라이드해야 한다.
- 만약 equals 를 override 하고 hashcode 를 override 하지 않았다면, Collection Framewor 를 사용할 때, 정상적인 로직을 기대할 수 없다.

▼ **String, StringBuilder, StringBuffer**

- 셋 다 문자열 표현
- String
  - 불변(한 번 생성되면 할당된 메모리 공간이 변하지 않음) 이기 때문에 + 연산시 새로운 객체를 생성한다. (성능 저하)
  - String a = “first name”;
    a = a + “ last name”;
    인 경우 “first name last name” 이라는 객체가 새로 생성되는 것이다.
  - 변외) String 객체는 String constant pool 을 사용하기 때문에
    String a = “str” String b = “str” 은 a == b true 이다.
    String constant pool 에서 “str” 라는 것을 저장후 다시 불러올 때 해당 “str” 이 있으면 해당 참조를 넘겨주기 때문에, 성능에 이점이 있다.
    즉, String 은 new 를 사용한 객체 생성보다 pool 을 사용하는 것이 성능 이점이 있다.
- StringBuilder
  - 가변으로 동일 객체내에서 문자열 크기를 변경이 가능해, String 보다 빠른 속도로 문자열 처리가 가능하다.
  - append 동시성 보장 X

```
@Override
public StringBuilder append(Object obj) {
    return append(String.valueOf(obj));
}
```

- StringBuffer
  - 가변으로 동일 객체내에서 문자열 크기를 변경이 가능해, String 보다 빠른 속도로 문자열 처리가 가능하다.
  - StringBuilder 와 다르게 동시성(쓰레드 안전성)이 보장된다.
  - append 동시성 보장 O

```
@Override
public synchronized StringBuffer append(Object obj) {
    toStringCache = null;
    super.append(String.valueOf(obj));
    return this;
}
```

▼ **call by reference vs call by value**

- Java 는 call by value 이다.
- c++ 의 pointer(*), reference(&) 를 지원하지 않는다.
- 즉, 기본 타입의 경우 메소드 호출시 사용되는 변수는 새로운 로컬 변수를 생성해 JVM 스택 영역에 할당함으로, 로컬 변수를 변경하더라도 메소드에 호출시 사용된 변수는 그대로 입니다.
  하지만, 참조 타입인 경우 객체를 힙에 저장하고 메소드 호출시 생성되는 Stack 변수는 heap의 해당 객체의 참조를 바라보게 됩니다. 그러무로, 해당 Stack 의 변수를 사용하면 해당 객체의 참조에 대한 내용을 변경하는 것임으로, 값을 변경하면 사용된 변수도 변경됩니다.

▼ **HashMap vs HashTable vs ConcurrentHashMap**

- HashMap
  - key와 value에 null을 허용한다.
  - 동기화를 보장하지 않는다. (싱글 쓰레드)
- HashTable
  - key와 value에 null을 허용하지 않는다.
  - 동기화를 보장한다.
  - HashTable 의 모든 메서드는 synchronized 키워드 가 붙어있다. 멀티 스레드 환경에서는 동시에 작업을 하려할 때 요청마다 Lock을 걸기 때문에 병목현상이 발생할 수 있다. (성능 저하) (Jdk 1.0 때 부터 있었기 때문에, JCF 가 이후 에는

ConcurrentHashMap 를 사용하는 것 이 좋다.)

- ConcurrentHashMap
  - key와 value에 null을 허용하지 않는다.
  - 동기화를 보장한다.
  - HashTable 보다 좋은 성능을 보여준다.

▼ **Java 8, Java 11**

- Java 8
  - LocalDate, LocalTime, LocalDateTime, Instant
    이전에 사용된 Date, Calendar 에 문제가 있어 새로운 Date-time API 를 도입해 날짜와 시간을 다루는 데 편리한 기능을 제공
  - Interface Default 메서드
    인터페이스에 기본적인 구현을 제공하는 디폴트 메서드를 정의.
    기존의 인터페이스를 확장할 때 하위 호환성을 유지 가능.
  - 람다식 ((a, b) -> a + b)
    람다식을 통해 함수형 프로그래밍 요소를 지원
    람다식은 익명 함수를 표현하는 방식으로, 코드의 간결성과 가독성을 높임
  - 메서드 참조 (::)
    메서드 참조를 사용하여 기존의 메서드를 람다식으로 대체
    코드의 가독성과 재사용성을 향상
  - 스트림 API
    스트림 API는 컬렉션을 다루는 데 사용되는 함수형 프로그래밍의 핵심 개념입니다. 스트림을 통해 데이터를 처리하고, 변환, 필터링, 집계 등의 작업을 효과적으로 수행할 수 있습니다.
- Java 11
  - Var
    람다식 및 익명 클래스의 파라미터로 사용되는 로컬 변수의 선언을 더 간결하게 할 수 있는 확장 문법을 도입
  - Epsilon GC
    GC 비용을 줄여 성능을 최적화할 수 있는 옵션을 제공
  - HTTP Client API
    HTTP/2 기술을 지원
  - 싱글 파일 실행
    별도의 컴파일 단계 없이 Java 파일을 실행 가능

▼ **Stream Parallel**

- Java 8부터 추가된 Parallel Streams는 스트림(Stream) API를 이용하여 병렬 처리 수행
- 기존의 순차적인 스트림 연산과는 달리 데이터를 병렬로 처리하여 작업을 더 빠르게 처리할 수 있습니다.
- Parallel Streams를 사용하려면 parallel() 메서드를 호출하여 병렬 스트림으로 변환하면 됩니다
- 주의점
  - 데이터 소스가 Stream.iterate (무한 스트림) 이라면 병렬 처리를 기대 할 수 없음
    (병렬 처리는 작업을 동시에 여러 개의 스레드에서 실행하여 속도를 향상시키는 것인데, 무한 스트림의 경우 처리할 작업이 끝나지 않기 때문에 작업의 분할과 병렬 실행이 어려움)
  - 중간 연산으로 limit 을 쓰면 파이프라인 병렬화로는 성능 개선을 기대할 수 없음.
    limit 이 parallel 이후에 있다면 문제인 것임. (무한 스크림에서 limit 으로 제한 후 parallel 한다면 병렬 처리 가능)
  - 종단 연산으로는 축소 (reduction) 가 되는 요소인 것이 좋다. (count, sum, anymatch)

▼ **ForkedJoinPool**

- Java 7부터 도입된 기능
- 병렬 처리를 위해 작업을 작은 단위로 분할하고 분할된 작업들을 병렬로 실행한 후 결과를 합치는 방식으로 동작
- RecursiveTask 또는 RecursiveAction 를 상속 받아 작업을 구현해야함. (compute 메서드)
  해당 작업을 fork로 분리한다. (분리된 작업은 compute 를 실행)

- ForkJoinPool 의 invoke 메소드로 작업을 시작

▼ 인터페이스와 추상클래스에 대해서 설명

- 공통
  - 둘다 IS-A 관계이다.
  - Default 메서드를 갖을 수 있다.
  - 인스턴스화 할 수 없다
  - 인터페이스 혹은 추상 클래스를 상속받아 구현한 구현체의 인스턴스를 사용해야 한다.
  - 인터페이스와 추상클래스를 구현, 상속한 클래스는 추상 메소드를 반드시 구현하여야 한다.
- 인터페이스
  - static 변수를 갖을 수 있지만... 열거 타입으로 쓰지는 말자. (Enum 을 쓰자)
  - 추상 메소드 하나만 갖고 있는 인터페이스를 함수형 인터페이스라고 한다. (@FunctionalInterface) - 람다로 사용가능
  - 다중 상속
- 추상 클래스
  - 하위 클래스들의 공통점들을 모아 추상화하여 만든 클래스
  - 단일 상속
  - 클래스간의 연관 관계를 구축을 초점을 둔다.

▼ volatile에 대해서 설명

- CPU 캐쉬에 등록되지 않고 메인 메모리에 등록되는 변수를 뜻한다.
- 수정 가능 쓰레드 1개, 읽기 가능 쓰레드 여러개 로 사용된다.
  - 왜?
    volatile int a = 0; 이라고 해보자
    a++; 를 하면 어떻게 될까? a 를 Main Thread 에서 불러서 a 에 + 1 을 처리 해줄 것이다.
    이때 만약 a를 부른 상태에서 +1 을 하기 직전에 다른 쓰레드에서 a++ 를 해서 a 를 가져왔다면 두 개 모두 1를 얻을 것이다.
    (안전 실패)
- 여러 쓰레드의 변경을 원한다면 synchronized 를 사용하거나 concurrent 라이브 러리를 사용하자

▼ 자바 접근 제어자

- 캡슐화(객체의 필드 및 메소드 은닉) 기능을 위함
- public
  - 모든 곳에서 호출 가능
- protected
  - 상속된 곳에서만 호출 가능
- default (package-private)
  - 같은 패키지 내에서만 호출 가능
- private
  - 해당 클래스 내부에서만 호출 가능
- Spring 에서 private 인 class는 @Component 어노테이션으로 bean 등록이 될까?
  된다. 왜냐면 Spring 에서는 리플렉션을 사용해 동적 로딩하고 어노테이션을 분석해서 bean 으로 등록하기 때문에 private 생성여도 bean 등록이 된다.

▼ 자바에서 불변 객체를 만드는 방식

- 불변 객체 장점
  - Thread-safe
  - 내부 상태 변경이 없어, cache, map, JCF 에 요소에 적합하다.
  - 외부에서 객체 변경이 불가능함으로 안정적임.
- String, Integer, Long, Double 등은 불변 객체

- 생성 방식
  - 모든 필드를 private, final 로 생성한다.
  - 가능하다면 필드에는 불변 객체, 기본 타입을 사용한다.
  - setter 메서드를 지원하지 않는다.
  - 클래스를 final 로 생성한다. (하위 클래스에서 override 금지)
  - 객체를 생성하기 위해서, 생성자나 정적 팩토리를 사용한다.
  - 가변 객체 - 기존 객체를 반환하지 않고 새로운 불변 객체를 생성해 반환
  - 가변 객체 JCF - 생성자나 다른 메서드에 인자로 넘겨온 변수를 그대로 사용하지 않고 생성하고, return 하는 경우에도 JCF 의 값을 그대로 주는게 아닌 새로 생성해서 return 한다.

▼ JVM 메모리 구조

- 메서드 영역(Method Area):
  - JVM이 시작될 때 생성되며, 모든 스레드가 공유하는 영역입니다.
  - 클래스 로더에 의해 로딩된 클래스의 바이트코드, 상수 풀(Constant Pool), static 변수 클래스 정보(필드 정보, 메서드 코드) 등이 저장됩니다.
- 힙(Heap):
  - 동적으로 생성된 객체와 배열이 할당되는 영역입니다.
  - 모든 스레드에서 공유되며, 가비지 컬렉션에 의해 관리됩니다.
  - 힙은 Young 영역(even, Survival1 , Survival2), Old 영역 로 나뉩니다.
  - Young 영역: 새로 생성된 객체들이 할당되는 공간입니다. 이 영역에서는 대부분의 객체가 빠르게 생성되고 소멸됩니다.
  - Old 영역: Young 영역에서 일정 시간 살아남은 객체들이 이동하는 공간입니다. 오랜 시간동안 살아남은 객체들이 저장됩니다.
- 스택(Stack):
  - 각 스레드마다 별도의 스택이 생성되는 영역입니다.
  - 스레드가 메서드를 호출할 때마다 스택 프레임(Frame)이 생성되어 매개변수, 지역 변수, 리턴 값 등이 저장됩니다.
  - 메서드 호출이 종료되면 해당 스택 프레임이 제거됩니다.
  - 스택의 크기는 미리 정해져 있으며, 스택 오버플로우(Stack Overflow)가 발생할 수 있습니다.
- PC 레지스터(Program Counter Register):
  - 각 스레드마다 현재 실행 중인 명령어의 주소를 가리키는 포인터입니다.
  - 스레드가 명령어를 실행하고 다음 명령어로 진행할 때, PC 레지스터의 값이 업데이트됩니다.
- 네이티브 메서드 스택(Native Method Stack):
  - Java 언어 외의 언어(C, C++ 등)로 작성된 네이티브 메서드의 호출 정보를 저장하는 스택입니다.
  - 네이티브 메서드는 JVM 바깥에서 실행되는 코드이므로, 네이티브 메서드 스택은 JVM의 메모리 영역에 포함되지 않습니다.

▼ GC (가비지 컬렉션)

- JVM 에서 자동으로 메모리 관리를 수행하는 기능
- C++ 처럼 개발자가 직접 메모리 할당, 해제하는 것 이 아닌,
  가비지 컬렉터가 더 이상 사용되지 않는 객체들을 자동으로 탐지하여 해제
- 메모리가 언제 해제되는지 정확하게 알 수 없어 제어하기 힘들며, 가비지 컬렉션이 동작하는 동안에는 다른 동작을 멈추기 때문에 오버헤드가 발생됨
- Java 8 부터 효율적인 메모리 관리를 위해 JVM 에서 heap 영역을 2가지 영역으로 분리 (Young, Old 영역)
  - Young
    - 새로운 객체에 할당 되는 영역
    - 대부분 Unreachable (참조 없음) 상태가 됨으로, 할당 후 바로 해제 됨으로 해당 영역을 (Minor GC) 라고 부른다.
    - even, survivor 1, 2 영역

            - even 메모리가 꽉차면 서바이벌 1로 이동한다면 even 과 서바이벌 2 영역 객체를 서바이벌 1로 이동하고, 그 이후 even 이 꽉차면 even 과 서바이벌 1 영역 객체를 서바이벌 2로 이동한다.
            - survivor 영역 1, 2 중 하나는 반드시 비어있어야 한다. (아래 질문과 일치)
            - survivor 영역은 왜 두 개 인가?

              한 개 인 경우를 상상해보자. 연속적 메모리를 사용 불가능할 것이다.
        - Old
            - Young영역에서 Reachable 상태를 유지하여 살아남은 객체가 복사되는 영역
            - Young 영역보다 크게 할당되며, 영역의 크기가 큰 만큼 가비지는 적게 발생한다.
            - 해당 영역을 Major GC 또는 Full GC라고 부른다.
    - heap dump

▼ 컴파일 과정 (+ 실행 과정)

1. 소스 코드 작성: 자바 소스 코드(java 파일)를 작성합니다. 이 소스 코드는 텍스트 파일로, 자바 프로그램의 구현을 포함하고 있습니다.
2. 컴파일: 자바 컴파일러(javac)를 사용하여 소스 코드를 바이트 코드(.class 파일)로 변환합니다. 컴파일러는 소스 코드를 구문 분석하고, 오류 검사, 타입 검사, 코드 최적화 등의 작업을 수행합니다. 변환된 바이트 코드는 JVM이 실행할 수 있는 형식입니다.
3. (컴파일 이후 실행) 클래스 로더는 동적로딩(Dynamic Loading)을 통해 필요한 클래스들을 로딩 및 링크하여 메서드 영역(JVM 메모리)에 올린다.
4. 실행엔진(Execution Engine)은 JVM 메모리에 올라온 바이트 코드들을 명령어 단위로 하나씩 가져와서 실행합니다. 이 때 실행 엔진은 두 가지 방식이 있다.
    a. 인터프리터 : 바이트 코드 명령어를 하나씩 읽어서 해석하고 실행합니다. 하나하나의 실행은 빠르나, 전체적인 실행 속도가 느리다는 단점을 가집니다.
    b. 프로파일러: 전체 프로그램에서 반복되는 코드 블록으로 식별합니다.
    메서드 단위로 발생하는 호출 수를 계산하는 카운터를 유지 관리한다.
    c. JIT컴파일러 : 인터프리터의 단점을 보완하기 위해 도입된 방식
    프로 파일러에서 관리하는 카운터가 JVM에 미리 정의된 임계값을 넘기면, JIT 컴파일러는 이를 기계어로 변환하여 해당 메서드를 더이상 인터프리팅 하지 않고, 직접 실행하는 방식입니다.
    해당 기계어는 캐쉬되어 재 사용된다.

▼ serializable, cloneable interface

- 둘 다 마커 인터페이스
    - 마커 인터페이스
        - 마커 인터페이스를 구현한 클래스의 인스턴스들을 구분하는 타입으로 쓸 수 있다.
            - 런타임에 발견될 오류를 컴파일 타임에 잡을 수 있다.
            - 자바의 직렬화는 마커 인터페이스를 보고 그 대상이 직렬화할 수 있는 타입인지 확인한다.
    - 마커 애너테이션
        - 거대한 어노테이션 시스템의 지원을 받을 수 있다. (Spring boot)
        - 클래스와 인터페이스 외의 프로그램 요소(모듈, 패키지, 필드 ,지역변수 등)에 마킹해야 할 때
- Serializable 인터페이스
    - Serializable 인터페이스는 직렬화를 지원하는 클래스와 인터페이스의 마커 인터페이스입니다.
    - 객체를 직렬화하기 위해서는 해당 객체가 Serializable 인터페이스를 구현해야 합니다.
    - 직렬화란, 객체의 상태를 바이트 스트림으로 변환하여 저장하거나 전송할 수 있는 형태로 만드는 과정을 말합니다.
    - 직렬화된 객체는 파일에 저장하거나 네트워크를 통해 전송할 수 있습니다.
    - ObjectOutputStream과 같은 클래스를 사용하여 객체를 직렬화하고, ObjectInputStream을 사용하여 직렬화된 객체를 역직렬화할 수 있습니다.
- Cloneable 인터페이스
    - Cloneable 인터페이스는 객체 복제를 지원하기 위한 인터페이스입니다.

    - 객체 복제란, 기존 객체의 동일한 복사본을 생성하는 과정을 말합니다.
    - Cloneable 인터페이스를 구현한 클래스에서 clone() 메서드를 사용하여 객체를 복제할 수 있습니다.
    - clone() 메서드는 Object 클래스에 정의되어 있지만, Cloneable 인터페이스를 구현하지 않은 클래스에서 clone() 메서드를 호출하면 CloneNotSupportedException이 발생합니다.
    - 객체 복제는 얕은 복사(Shallow Copy)를 수행하며, 필요에 따라 깊은 복사(Deep Copy)를 직접 구현해야 할 수도 있습니다.

▼ 직렬화 / 역직렬화

- 직렬화
    - 객체를 저장, 전송할 수 있는 특정 포맷 상태로 바꾸는 과정
- 역직렬화
    - 특정 포맷 상태의 데이터를 다시 객체로 변환하는 것
- 자바의 직렬화
    - 장점
        - 프로그래머가 어렵지 않게 분산 객체를 만들 수 있음
    - 단점
        - 보이지 않는 생성자, API와 구현 사이의 모호해진 경계, 잠재적인 정확성 문제, 성능, 보안, 유지보수성 등 대가가 큼.
- 자바의 직렬화 대신 크로스-플랫폼 구조화된 데이터 표현(JSON, protobuf) 를 사용하도록 하자.
- transient 키워드를 사용하여 해당 필드를 직렬화에서 제외할 수 있습니다. (기본값 0, null 과 같이 매핑됨)
- serialVersionUID를 직접 관리하는게 좋다. (버전 관리 측면)
  그렇지 않으면 런타임에 암호 해시 함수(SHA-1)를 적용해 serialVersionUID 를 생성한다.
  만약, 해당 클래스가 버전이 올라갔지만, 이전 직렬화된 상태와 호환된다면 수정하면 안되지만, 직접 구현하지 않았다면, 둘은 호환되지 않는다.
- 메시지큐 (kafka, RabbitMQ) 에서는 직렬화를 사용해야 할까?
    - 직렬화를 사용하면 객체를 이진 형태로 변환하고, 역직렬화를 사용하면 이진 데이터를 다시 객체로 변환할수 있음으로, 메시지큐를 이용해 데이터를 안전하게 전송 처리 가능하다. (안 써도 사용할 수 있음) (주관적인 생각으로는 보안과 성능은 반비례한다. 직렬화도 그런류 일 듯하다.)

▼ SOLID 원칙

- 해당 원칙들은 결합력은 낮추고, 응집도를 높여 객체 지향 설계에 도움을 주고, 코드 확장, 유지보수 관리가 쉬워지고, 불필요한 복잡성을 제거해 도움을 줄 수 있다.
- 디자인 패턴들도 SOLID 원칙을 지키면서 만들어 졌다.
- SRP (Single Responsibility Principle) - 단일 책임 원칙
    - 클래스(객체)는 단 하나의 책임만 가져야한다.
    - 만약 하나의 클래스에 책임이 여러개 있다면, 클래스 수정시, 수정할 코드가 많아진다.
    - 유지보수성
- OCP (Open Closed Principle) - 개방 폐쇄 원칙
    - 확장에는 열려있으며, 수정에는 닫혀있어야 한다.
    - 클래스를 확장해 쉽게 구현하고, 확장에 따른 클래스 수정은 최소화
    - 추상화 (다형성, 확장)
- LSP (Liskov Substitution Principle) - 리스코프 치환 원칙
    - 하위 타입은 항상 상위 타입으로 교체할 수 있어야 한다.
    - 다형성, 상속
- ISP (Interface Segregation Principle) - 인터페이스 분리 원칙
    - 인터페이스를 각각 사용에 맞게 끔 잘게 분리
    - 클라이언트의 목적과 용도에 적합한 인터페이스 만을 제공
- DIP (Dependency Inversion Principle) - 의존 역전 원칙

    - class 를 참조해야한다면, 해당 class 가 아닌 추상 클래스 or 인터페이스로 참조 하라는 뜻이다.
    - 결합도를 낮춤.

▼ 함수형 프로그래밍

- Java8 부터 람다 표현식과 함수형 인터페이스가 도입 이후 함수형 프로그래밍 가능해짐.
- Stream
    - Java Stream은 데이터 요소의 시퀀스를 처리하기 위한 연속된 연산을 제공하는 API
    - Stream은 데이터를 손쉽게 필터링, 변환, 정렬, 그룹화 등 다양한 작업을 수행할 수 있도록 합니다.
    - Stream은 데이터 소스(예: 컬렉션, 배열, 파일)에서 요소를 추출하고, 중간 연산과 최종 연산을 연결하여 원하는 결과를 얻을 수 있음
- Optional
    - Java Optional은 값이 있을 수도 있고 없을 수도 있는 상황에서 사용되는 컨테이너 클래스.
    - Optional은 NullPointerException을 방지하고 코드 안정성을 높이기 위해 도입됨
      그러니까 Optional 에 Null 을 넣는 미친짓은 하지 말자

▼ 캡슐화 vs 정보은닉

- 정보 은닉이란, 객체지향 언어적 요소를 활용하여 객체에 대한 구체적인 정보를 노출시키지 않도록 하는 기법이다.
- 정보 은닉이 캡슐화 보다 큰 범위이고 3 종류로 나눌 수 있다.
    - 객체의 구체적인 타입 은닉 (업캐스팅)
    - 객체의 필드 및 메소드 은닉 (캡슐화)
        - 접근 제어자 private 으로 필드나 메소드를 은닉 할 수 있다.
    - 구현 은닉 (인터페이스 & 추상 클래스)

▼ CheckedException vs UncheckedException

- 컴파일 에러과 런타임 에러를 비교 하는 것이다.
- Checked Exception은 프로그램이 예외를 처리하거나 전파할 수 있도록 하여 안정성과 신뢰성을 높이는 데에 사용. (에러를 수정 할 수 있어서 정상 로직에 다시 돌아갈 수 있도록 해야한다.)
- Unchecked Exception은 주로 프로그래머의 실수나 예외적인 상황을 나타내는데 사용되며, 명시적인 예외 처리를 강제하지 않아 코드의 가독성과 편의성을 높일 수 있습니다.
  (주로 프로그래밍 오류나 복구가 불가능한 상황)
- 궁금할 수도 있는 것
    - java 에서는 해당 프로세스가 down 되고, Spring 에서는 에러 던져진다. (예외가 발생하면 기본적으로 HTTP 응답 상태 코드를 반환하도록 설계되어 있음 - Spring의 예외 처리 메커니즘과 서블릿 컨테이너 으로 쓰레드 종료를 방지하고 예외를 처리하여 HTTP 응답을 반환)

▼ 쓰레드 풀

- 쓰레드를 미리 생성하고, 작업 요청이 발생할 때 마다 미리 생성된 쓰레드로 해당 작업을 처리하는 방식을 의미한다. 이때, 작업이 끝난 쓰레드는 종료되지 않으며 다음 작업 요청이 들어올때까지 대기한다.
- 들어오는 작업은 **작업 큐(task queue)**에 채워 넣은뒤 쓰레드 별로 할당하여 작업을 처리한다.
- 왜, 요청이 있을 때마다 쓰레드를 생성하는 것이 아닌, 미리 생성해둬야 하는건가?
    - 매번 발생하는 작업을 병렬처리하기 위해 쓰레드를 생성/수거하는 데 따른 부담은 프로그램 전체적인 성능을 저하시킨다.

▼ 세부 내용

- 대표적인 쓰레드 풀 사용시 생성법
    - ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
        - 특정시간 이후에 실행되거나 주기적으로 작업을 실행할 수 있다. (ScheduledExecutorService.schedule 메서드)
    - ExecutorService newFixedThreadPool(int nThreads)
        - 주어진 스레드개수만큼 생성하고, 그 수를 유지한다.
          이때 생성된 스레드중 일부가 작업시 종료되었다면 스레드를 다시생성하여 주어진 수를 맞춘다.

    - ExecutorService newCachedThreadPool()
        - 처리할 작업의 스레드가 많아지면 그 만큼 스레드를 증가하여 생성한다.
          만약 쉬는 스레드가 많다면 스레드를 종료시킨다. 반면 스레드를 제한두지 않기때문에 조심히 사용해야 한다.
    - ExecutorService ThreadPoolExecutor
        - 커스텀 할 수 있음

```
ExecutorService threadPool = new ThreadPoolExecutor(
    1, // 코어 스레드 개수
    10, // 최대 스레드 개수
    120L, // 최대 가용된 시간 (쓰레드가 놀고 있을 수 있는 최대 시간)
    TimeUnit.SECONDS, // 놀 수 있는 시간 단위
    new SynchronousQueue<Runnable>() // 작업 큐
);
```

- 쓰레드 풀 종료
    - void shutdown()
        - 현재 처리 중인 작업 뿐만 아니라 작업 큐에 대기하고 있는 모든 작업을 처리한 뒤에 스레드 풀을 종료한다.
    - List<Runnable>shutdownNow()
        - 현재 작업 처리 중인 스레드를 interrupt해서 작업 중지를 시도하고 스레드 풀을 종료한다.
        - 리턴 값은 작업 큐에 있는 미처리된 작업의 목록이다.
    - boolean awaitTermination(long timeout, TimeUnit unit)
        - shutdown() 메소드 호출 이후, 모든 작업 처리를 timeout 시간 내에 완료하면 true를 리턴하고, 그렇지 않으면 작업 처리 중인 스레드를 interrupt하고 false를 리턴한다.
- 작업 처리 요청 ExecutorService의 2가지 메소드
    - void execute
        - Runnable을 작업 큐에 저장하고, 작업 처리 결과를 받지 못함
        - 예외가 발생하면 해당 스레드를 스레드 풀에서 제거함
    - Future submit
        - Runnable 또는 Callable을 작업 큐에 저장
        - 리턴된 Future를 통해 작업 처리 결과를 알 수 있음
        - 예외가 발생하더라도 스레드는 종료되지 않고 다른 작업에 재사용될 수 있음
- Callable, Runnable
    - 둘다 구현된 함수를 수행한다는 공통점
    - Callable 특정 타입의 객체를 리턴합니다. Exception을 발생킬 수 있습니다.

```
public interface Callable<V> {
    V call() throws Exception;
}
```

    - Runnable 어떤 객체도 리턴하지 않습니다. Exception을 발생시키지 않습니다.

```
public interface Runnable {
    public abstract void run();
}
```

- 간단 예시

```
public static void main(String[] args) {
        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThre
adPool(2);
```

```
        for (int i = 0; i < 10; i++) {
            executor.execute(() -> {
                System.out.println(LocalTime.now() + " job num" + i);
            });
        }
        executor.shutdown();
    }
}
```

▼ **JPA vs MyBatis**

- JPA (Java Persistence API) (ORM)
  - 장점
    - 1차캐시, 쓰기지연, 변경감지, 지연로딩을 제공하여 성능상 이점을 얻을 수 있다.
    - 코드 레벨로 관리 되므로 사용하기 용이하고 생산성이 높다.
    - 컴파일 타임에 오류를 확인할 수 있다.
    - 데이터베이스에 종속적이지 않으므로 특정 쿼리를 사용하지 않아 추상적으로 기술 구현이 가능하다.
    - 엔티티로 관리되므로 스키마 변경시 엔티티만 수정하게 되면 엔티티를 사용하는 관련 쿼리는 자동으로 변경된 내역이 반영된다.
    - 개발 초기에는 쿼리에 대한 이해가 부족해도 코드 레벨로 어느 정도 커버가 가능하다.
    - 객체지향적으로 데이터를 관리할 수 있다.
    - 부족한 부분은 다양한 쿼리 빌더와 호환하여 보안할 수 있다.
  - 단점
    - JPA만 사용하여 복잡한 연산을 수행하기에는 다소 무리가 있다. (로직이 복잡하거나 불필요한 쿼리가 발생할 수 있다.)
    - 초기에는 생산성이 높을 수 있으나 점차 사용하다 보면 성능상 이슈가 발생할 수 있다.(N+1, FetchType, Proxy, 연관관계)
    - 고도화 될수록 학습 곡선이 높아질 수 있다. (성능 이슈의 연장선으로 해결 방안에 따라 복잡한 내부 로직을 이해해야 할 필요가 있다)
- MyBatis (SQL Mapper)
  - 장점
    - SQL 쿼리를 직접 작성하므로 최적화된 쿼리를 구현할 수 있다.
    - 엔티티에 종속받지 않고 다양한 테이블을 조합할 수 있다.
    - 복잡한 쿼리도 SQL 쿼리만 작성할 수 있다면 손쉽게 작성할 수 있다.
  - 단점
    - 스키마 변경시 SQL 쿼리를 직접 수정해주어야 한다.
    - 반복된 쿼리가 발생하여 반복 작업이 있다.
    - 런타임시에 오류를 확인할 수 있다.
    - 쿼리를 직접 작성하기 때문에 데이터베이스에 종속된 쿼리문이 발생할 수 있다. 데이터베이스 변경시 로직도 함께 수정해주어야 한다.

▼ Spring

▼ **Spring**

- Spring
  - 자바 엔터 프라이즈 개발을 편하게 해주는 오픈소스 애플리케이션 프레임워크
  - 개발자가 직접 설정 파일을 작성해 스프링 컨테이너를 구성하고 빈 객체를 등록, 빈 객체간의 의존성을 설정해야함.
- Spring Boot
  - Spring 프레임워크를 기반으로 한 도구
  - 개발자가 설정 파일을 작성할 필요 없이, 프로젝트의 설정과 라이브러리 의존성을 자동으로 처리 해주빈다.

- Spring 특징
  - 컨테이너 역할
    - SPring 컨테이너는 Java 객체의 LifeCycle 을 관리하며, Spring 컨테이너로 부터 필요한 객체를 가져와 사용
  - IoC/DI 지원
    - Spring 은 설정 파일이나 어노테이션을 통해 객체 간의 의존관계 설정 가능
  - AOP 지원
    - 공통적으로 필요한 모듈을 실제 핵심 모듈에서 분리해 적용가능
  - POJO 지원
    - Spring 컨테이너에 저장되는 Java 객체는 객체 지향의 집중하고 특정한 인터페이스를 구현하거나, 특정 클래스를 상속받지 않아도 된다.
  - PSA 지원
    - 환경의 변화와 관계없이 일관된 방식의 기술로의 접근 환경을 제공하는 추상화 구조

▼ **dispatcher-servlet (참고)**

- HTTP 프로토콜로 들어오는 모든 요청을 가장 먼저 받아 적합한 컨트롤러에 위임해주는 프론트 컨트롤러로 정의 한다.
- request 가 전송하면, Tomcat(톰캣)과 같은 서블릿 컨테이너가 요청을 받게 됩니다. 그리고 이 모든 요청을 프론트 컨트롤러인 디스패처 서블릿이 가장 먼저 받고 해당 요청을 핸들링하고 (해당 컨트롤러로 작업 위), 공통 작업을 처리 해줍니다.
- 정적 자원 요청까지 dispatcher-servlet 이 가져감으로 해당 요청이 실패하는 경우가 있음
  - 정적 자원 요청 URL, 기본 요청 URL 분리 (***deprecated)***
    모든 컨트롤러에 prefix URL 를 붙여줘야 함으로 직관적인 설계가 불가능
  - 요청에 해당하는 컨트롤러를 찾으면 (기본 요청), 못 찾으면 (정적 자원 요청)으로 변경됨.

▼ **Filter vs Interceptor**

- filter
  - Dispatcher Servlet에 요청이 전달되기 전/후에 url 패턴에 맞는 모든 요청에 대해 부가작업을 처리할 수 있는 기능을 제공한다.
  - 스프링 컨테이너가 아닌 톰캣과 같은 웹 컨테이너(**서블릿 컨테이너**)에 의해 관리됨.
  - 대표적으로 SpringSecurity 를 들 수 있다.
  - javax.servlet의 Filter 인터페이스를 구현(implements) 를 구현해야 한다.

```
public interface Filter {

    public default void init(FilterConfig filterConfig) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException;

    public default void destroy() {}
}
```

    - init 메소드는 필터 객체를 초기화하고 서비스에 추가하기 위한 메소드이다. 웹 컨테이너가 1회 init 메소드를 호출하여 필터 객체를 초기화하면 이후의 요청들은 doFilter를 통해 처리된다.
    - doFilter 메소드는 url-pattern에 맞는 모든 HTTP 요청이 디스패처 서블릿으로 전달되기 전에 웹 컨테이너에 의해 실행되는 메소드이다. doFilter의 파라미터로는 FilterChain이 있는데, FilterChain의 doFilter 통해 다음 대상으로 요청을 전달하게 된다. chain.doFilter() 전/후에 우리가 필요한 처리 과정을 넣어줌으로써 원하는 처리를 진행할 수 있다.
      - request, response 객체를 변경할 수 있음 (아예 다른 객체로)
    - destroy 메소드는 필터 객체를 서비스에서 제거하고 사용하는 자원을 반환하기 위한 메소드이다. 이는 웹 컨테이너에 의해 1번 호출되며 이후에는 이제 doFilter에 의해 처리되지 않는다.
- interceptor
  - Dispatcher Servlet이 컨트롤러를 호출하기 전과 후에 요청과 응답을 참조하거나 가공할 수 있는 기능을 제공

- **스프링 컨텍스트**에서 동작됨.
  - 스프링 예외처리 (ControllerAdvice와 ExceptionHandler) 사용 가능
- org.springframework.web.servlet의 HandlerInterceptor 인터페이스를 구현해야 함.

```
public interface HandlerInterceptor {

    default boolean preHandle(HttpServletRequest request, HttpServletResponse respo
        throws Exception {

        return true;
    }

    default void postHandle(HttpServletRequest request, HttpServletResponse respons
        @Nullable ModelAndView modelAndView) throws Exception {
    }

    default void afterCompletion(HttpServletRequest request, HttpServletResponse re
        @Nullable Exception ex) throws Exception {
    }
}
```

  - preHandle 메소드는 컨트롤러가 호출되기 전에 실행된다. 그렇기 때문에 컨트롤러 이전에 처리해야 하는 전처리 작업이나 요청 정보를 가공하거나 추가하는 경우에 사용할 수 있다. preHandle의 3번째 파라미터인 handler 파라미터는 핸들러 매핑이 찾아준 컨트롤러 빈에 매핑되는 HandlerMethod라는 새로운 타입의 객체로써, @RequestMapping이 붙은 메소드의 정보를 추상화한 객체이다. 또한 preHandle의 반환 타입은 boolean인데 반환값이 true이면 다음 단계로 진행이 되지만, false라면 작업을 중단하여 이후의 작업(다음 인터셉터 또는 컨트롤러)은 진행되지 않는다.
  - postHandle 메소드는 컨트롤러를 호출된 후에 실행된다. 그렇기 때문에 컨트롤러 이후에 처리해야 하는 후처리 작업이 있을 때 사용할 수 있다. 이 메소드에는 컨트롤러가 반환하는 ModelAndView 타입의 정보가 제공되는데, 최근에는 Json 형태로 데이터를 제공하는 RestAPI 기반의 컨트롤러(@RestController)를 만들면서 자주 사용되지는 않는다. 또한 컨트롤러 하위 계층에서 작업을 진행하다가 중간에 예외가 발생하면 postHandle은 호출되지 않는다.
  - afterCompletion 메소드는 이름 그대로 모든 뷰에서 최종 결과를 생성하는 일을 포함해 모든 작업이 완료된 후에 실행된다. 요청 처리 중에 사용한 리소스를 반환할 때 사용하기에 적합하다. postHandler과 달리 컨트롤러 하위 계층에서 작업을 진행하다가 중간에 예외가 발생하더라도 afterCompletion은 반드시 호출된다.

▼ **POJO**

- Plain Old Java Object
  간단히 POJO는 말 그대로 해석을 하면 오래된 방식의 간단한 자바 오브젝트라는 말로서 Java EE 등의 중량 프레임워크(EJB)들을 사용하게 되면서 해당 프레임워크에 종속된 "무거운" 객체를 만들게 된 것에
  반발해서 사용되게 된 용어
- 특정 기술에 종속되지 않는 순수한 자바 객체
- 객체지향적인 원리에 충실하면서, 환경과 기술에 종속되지 않고 필요에 따라 재활용될 수 있는 방식으로 설계된 오브젝트
- POJO 를 위해 PSA, IoC/DI, AOP 라는 개념을 적용한다.

▼ **PSA**

- Portable Service Abstraction
- 환경의 변화와 관계없이 일관된 방식의 기술로의 접근 환경을 제공하는 추상화 구조
- 스프링을 사용하면 서비스 추상화를 통해 특정 환경이나 서버, 기술에 종속되지 않으며 유연한 애플리케이션을 개발할 수 있다. 스프링에서는 추상화 계층을 통해 구체적인 기술과 환경에 종속되지 않도록 한다. 예를 들어 MyBatis나 JPA 등 세부 기술에 종속적인 에러들을 추상화하여 기술에 종속적이지 않은 에러들로 처리할 수 있도록 도와준다.

- PSA는 Spring에서 서로 다른 서비스 제공자들을 추상화하여 일관된 방식으로 사용할 수 있도록 도와줍니다. 여러 서비스 제공자(예: 데이터베이스, 메시징, 캐싱 등)와의 상호 작용을 추상화된 인터페이스를 통해 처리할 수 있게 해줍니다

▼ AOP (관점 지향 프로그래밍)

- 프로그래밍을 하다보면 공통적인 기능이 많이 발생한다. 이러한 공통 기능을 모든 모듈에 적용하기 위해 상속을 이용한다. 하지만 Java에서는 다중 상속이 불가능하며, 상속을 받아 공통 기능을 부여하기에는 한계가 있는데, AOP 가 이를 해결 해준다.
- 자동 프록시 생성
- 용어
  - target
    - 부가기능을 부여할 대상을 뜻함. (Service)
  - Join point
    - 추상적인 개념 으로 advice가 적용될 수 있는 모든 위치를 말합니다.
    - 스프링 AOP는 프록시 방식을 사용하므로 조인 포인트는 항상 메서드 실행 지점
  - Pointcut
    - 조인 포인트 중에서 advice가 적용될 위치를 선별하는 기능
  - Advice
    - 실질적으로 프록시에서 수행하게 되는 로직을 정의하게 되는 곳
    - Spring 에서는 주로 쓰이는 Around(메소드의 실행되기 전, 실행된 후 모두에서 동작) 과 Before, AfterReturning, AfterThrowing, After 이 있다.
  - Aspect
    - advice + pointcut을 모듈화 한 것
- 장점
  - 공통 관심 사항을 핵심 관심사항으로부터 분리시켜 핵심 로직을 깔끔하게 유지할 수 있다.
  - 그에 따라 코드의 가독성, 유지보수성 등을 높일 수 있다.
  - 각각의 모듈에 수정이 필요하면 다른 모듈의 수정 없이 해당 로직만 변경하면 된다.
  - 공통 로직을 적용할 대상을 선택할 수 있다
- 동작 원리
  1. 다이내믹 프록시 객체의 생성 요청
  2. 포인트컷을 통해 부가 기능 대상 여부 확인
  3. 어드바이스로 부가 기능 적용
  4. 실제 기능 처리
- 적용 순서 (boot 기준)
  - AOP 의존성 추가
  - main 메서드에 @EnableAspectJAutoProxy 어노테이션 추가
  - @Aspect어노테이션을 추가, Spring의 빈으로 등록 (@Component)
  - Advice 등록 (@Around("${pattern}") 어노테이션으로 부가 기능 로직을 추가하고
    pattern (pointcut 표현식)(* 모든 것, .. 0개 이상) 을 지정해 해당 부가 기능이 적용될 위치를 지정한다.
- 만약 여러 개의 Aspect 가 있다면, 순서를 보장하지 못한다. 이럴 경우 클래스 단위로 Order 어노테이션을 사용하면 순서를 보장해준다.
- 예시

```
@Component
@Aspect
public class AspectTest {
    @Around("execution(접근제어자? 반환타입 선언타입?메서드이름(파리미터) 예외?)")
    public Object doTransaction(ProceedingJoinPoint joinPoint) throws Throwable {
        try {
```

```
                        // @Before 수행
                        log.info("[트랜잭션 시작] {}", joinPoint.getSignature());
                        // @Before 종료
                        // Target 메서드 호출
                        Object result = joinPoint.proceed();
                        // Target 메서드 종료
                        // @AfterReturning 수행
                        log.info("[트랜잭션 커밋] {}", joinPoint.getSignature());
                        // @AfterReturning 종료
                        // 값 반환
                        return result;
        } catch (Exception e) {
                        // @AfterThrowing 수행
                        log.info("[트랜잭션 롤백] {}", joinPoint.getSignature());
            throw e;
                        // @AfterThrowing 종료
                } finally {
                        //@ After 수행
                        log.info("[리소스 릴리즈] {}", joinPoint.getSignature());
                        //@ After 종료
                }
    }
}
```

▼ **JDK Dynamic Proxy vs CGLib**

- 둘다 AOP (target 지정) 위한 기술
  순서 Proxy > JDK Dynamic Proxy > CGLib (Spring default)
- JDK Dynamic Proxy
  - JDK에서 지원하는 프록시 생성 방법
  - 프록시 팩토리에 의해 런타임 시 다이나믹하게 만들어지는 오브젝트
  - 프록시 팩토리에게 인터페이스 정보만 제공해주면 해당 인터페이스를 구현한 클래스 오브젝트를 자동으로 생성
  - Reflection API를 사용한다. (느리다)
  - 인터페이스가 반드시 존재해야한다. (클래스인경우 runtime 에러 발생)
  - InvocationHanlder 구현체(invoke Method Override)가 있어야 부가기능이 가능

```
Proxy.newProxyInstance(
    ClassLoader,
    new Class[] { targetInterface.class }, // target Interface
    new InvocationHandlerImpl(new target()) // InvocationHandler 구현체
);
```

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    if (method.getName().startsWith("test")) {
        return ((String) method.invoke(target, args)).toUpperCase();
    }
    return method.invoke(target, args);
}
```

- CGLIB Proxy
  - 프록시 팩토리에 의해 런타임 시 다이나믹하게 만들어지는 오브젝트
  - 클래스 상속을 이용하여 프록시 구현. 인터페이스가 존재하지 않아도 가능
    - 바이트 코드를 조작해서 프록시 생성함

        - 인터페이스에도 강제로 적용 가능. 이 경우 클래스에도 프록시를 적용해야 한다
    - Dynamic Proxy 보다 약 3배 가까이 빠르다
        - 메서드가 처음 호출되었을 때 동적으로 타깃 클래스의 바이트 코드를 조작
        - 이후 호출 시엔 조작된 바이트 코드를 재사용
    - MethodInterceptor 구현체 (intercept Method Override)가 있어야 부가기능이 가능
    - 메서드에 final을 붙이면 오버라이딩이 불가능

```
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(target.class); // target Interface, Class
enhancer.setCallback(new MethodInterceptorImpl()); // MethodInterceptor 구현체
Object proxy = enhancer.create();
```

```
@Override
public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) t
    if (method.getName().startsWith("test")) {
                log.info("prefix is test");
        return proxy.invoke(target, args);
    }
    return proxy.invoke(obj, args);
}
```

▼ DI - Dependency Injection (의존성 주입)

- 외부에서 두 객체 간의 관계를 결정해주는 디자인 패턴으로, 인터페이스를 사이에 둬서 클래스 레벨에서는 의존관계가 고정되지 않도록 하고 런타임 시에 관계를 동적으로 주입하여 유연성을 확보하고 결합도를 낮출 수 있다.
- 객체를 직접 생성하는 게 아니라 외부에서 생성한 후 주입 시켜주는 방식
- 방법 3가지
    - **생성자 주입**

```
@Service
public class UserService {

    private UserRoleService userRoleService;

    @Autowired
    public UserService(UserRoleService userRoleService) {
        this.userRoleService= userRoleService;
    }
}
```

        - 객체의 불변성 확보 (final 키워드 작성 및 Lombok과의 결합)
            - 실제 개발시, 의존 관계의 변경이 필요한 상황은 거의 없음.
              그러니 Setter, 필드는 불필요한 변경의 가능성이 있어서 유지보수성에 좋지 않음.
            - 생성자 주입을 하면 final 키워드 작성이 가능하고
              Lombok 에 @RequiredArgsConstructor 사용할 수 있다.
        - 순환 참조 에러 방지
            - 만약, UserRoleService 에 UserService가 있고, userService 에 A라는 메서드가 userRoleService 에 B라는 메서드를 호출 하고 그 B 라는 메서드가 userService 에 A 라는 메서드를 호출 한다고 하면, 두 메서드는 계속 서로를 호출 할 것임으로 Stack 에 CallStack이 계속 쌓여 StackOverflow 가 일어 날 것이다. 즉, 런타임 시점에 발생되어 서버가 다운될 것이다.
            - 하지만 생성자 주입을 하게 되면, 런타임시(객체 생성시) 바로 에러를 잡아내 서버를 실행 시킬 수 없게 되어, 서버가 다운될 가능성을 미리 잡아 낼 수 있다.
    - 수정자 주입

```
@Service
public class UserService {
    private UserRoleService userRoleService ;

    @Autowired
    public void setUserRoleService(UserRoleService userRoleService ) {
        this.userRoleService = userRoleService ;
    }
}
```

- Setter 주입은 생성자 주입과 다르게 주입받는 객체가 변경될 가능성이 있는 경우에 사용한다. (그럴 일 없다.. 아마)

- 필드 주입

```
@Service
public class UserService {
        @Autowired
    private UserRoleService userRoleService ;
}
```

- 필드 주입을 이용하면 코드가 간결해져서 과거에 상당히 많이 이용되었던 주입 방법이다. 하지만 필드 주입은 외부에서 접근이 불가능하다는 단점이 있어, Test 코드를 사용할 수 없어, 요즘은 잘 사용하지 않는다.

▼ IOC - Inversion of Control (제어 역전)

- 일반적으로는 클라이언트가 객체의 의존성을 주입하지만, Spring 에서는 DI (의존성 주입)을 통해서 관리됨으로, 클라이언트가 객체의 의존성을 신경 쓰지 않아도 됨으로, 코드의 결합도를 낮추고 유지보수성을 향상시킵니다.
- 간단히 말해 IOC는 객체의 제어를 프레임워크가 맡아서 처리하고, DI는 객체 간의 의존성을 주입하는 방식입니다. (Spring Framework에서는 Spring IOC 컨테이너를 통해 DI를 구현하여 객체 간의 의존성을 주입함)

▼ Spring - 트랜잭션

- 트랜잭션 - 더 이상 쪼갤 수 없는 최소 작업 단위
- Spring 에서 제공하는 기술 3가지
  - 트랜잭션 동기화
    - 트랜잭션을 시작하기 위한 Connection 객체를 특별한 저장소에 보관해두고 필요할 때 꺼내쓸 수 있도록 하는 기술
    - 작업 쓰레드마다 Connection 객체를 독립적으로 관리하기 때문에, 멀티쓰레드 환경에서도 충돌이 발생할 여지가 없다
    - Hibernate에서는 Connection이 아닌 Session이라는 객체를 사용하여 동기화 코드에 문제를 일으킨다. (그래서 트랜잭션 추상화가 추가됨)
  - 트랜잭션 추상화
    - 트랜잭션 기술의 공통점을 담은 트랜잭션 추상화 기술을 제공함으로써, 애플리케이션에 각 기술마다(JDBC, JPA, Hibernate 등) 종속적인 코드를 이용하지 않고도 일관되게 트랜잭션을 처리가 가능해짐.
    - Spring이 제공하는 트랜잭션 경계 설정을 위한 추상 인터페이스 PlatformTransactionManager
    - 추상된 코드를 구체화하면 트랜잭션 코드와 비지니스 코드가 결합되어 2가지 책임을 갖게되어 Spring 개발 목표 (객체 지향적 코드)를 이룰 수 없음 (그래서 AOP 를 이용한 트랜잭션 분리를 추가하게 됨)
  - @Transactional - AOP 를 이용한 트랜잭션 분리 (선언적 트랜잭션)
    - 해당 로직을 클래스 밖으로 빼내서 별도의 모듈로 만드는 AOP
    - 여러 트랜잭션 적용 범위를 묶어서 커다란 하나의 트랜잭션 경계를 만들 수 있다는 점
    - 세부 기능
      - 트랜잭션 전파 (propagation)
        - 트랜잭션의 경계에서 이미 진행중인 트랜잭션이 있거나 없을 때 어떻게 동작할 것인가를 결정하는 방식을 의미함.

        ▼ 속성

        - REQUIRED

        - default 속성, 모든 트랜잭션 매니저가 지원함
        - 대부분 이 속성으로 충분한 사용 가능
        - 미리 시작된 트랜잭션이 있으면 참여하고 없으면 새로 시작
    - SUPPORTS
        - 이미 시작된 트랜잭션이 있으면 참여하고, 그렇지 않으면 트랜잭션 없이 진행함
    - MANDATORY
        - 이미 시작된 트랜잭션이 있으면 참여하고, 없으면 새로 시작하는 대신 없으면 예외를 발생시킴
        - MANDATORY는 혼자서 독립적으로 트랜잭션을 진행하면 안되는 경우에 사용
    - REQUIRES_NEW
        - 항상 새로운 트랜잭션을 시작해야 하는 경우에 사용
        - 이미 진행중인 트랜잭션이 있으면 이를 보류시키고 새로운 트랜잭션을 만들어 시작
    - NOT_SUPPORTED
        - 이미 진행중인 트랜잭션이 있으면 이를 보류시키고, 트랜잭션을 사용하지 않도록 함
    - NEVER
        - 이미 진행중인 트랜잭션이 있으면 예외를 발생시키며, 트랜잭션을 사용하지 않도록 강제함
    - NESTED
        - 이미 실행 중인 트랜잭션이 존재한다면, 중첩 트랜잭션(부모 트랜잭션에서 새로운 트랜잭션을 내부에 만드는 것)을 만든다.
        - 중첩 트랜잭션은 부모 트랜잭션의 커밋과 롤백에는 영향을 받지만, 중첩 트랜잭션 자기 자신은 부모 트랜잭션에 영향을 주지 않는다.
        - REQUIRED와 마찬가지로 부모 트랜잭션이 존재하지 않으면 독립적으로 트랜잭션을 생성해서 사용한다.
- 격리수준 (isolation)
    - 동시에 여러 트랜잭션이 진행될 때 트랜잭션의 작업 결과를 여타 트랜잭션에게 어떻게 노출할 것인지를 결정 (commit 되지 않는 rows 를 보여줄 것인가 와 같은 내용)

    ▼ 속성

    - DEFAULT (아래 4중 하나 사용됨)
        - 사용하는 데이터 액세스 기술, DB 드라이버의 디폴트 설정을 따름 (DB 격리 수준을 따름)
          psql 15 - READ COMMITTED
          mysql InnoDB - REPEATABLE READ
    - READ_UNCOMMITTED
        - 가장 낮은 격리수준으로써 하나의 트랜잭션이 커밋되기 전에 그 변화가 다른 트랜잭션에 그대로 노출되는 문제가 있음
        - 하지만 가장 빠르기 때문에 데이터의 일관성이 조금 떨어지더라도 성능을 극대화할 때 의도적으로 사용함
    - READ_COMMITTED
        - DB는 일반적으로 READ_COMMITED가 기본 속성이므로 가장 많이 사용됨
        - 다른 트랜잭션이 커밋하지 않은 정보는 읽을 수 없다. 대신 하나의 트랜잭션이 읽은 로우를 다른 트랜잭션이 수정가능함.
    - REPEATABLE_READ
        - 하나의 트랜잭션이 읽은 로우를 다른 트랜잭션이 수정할 수 없도록 막아주지만 새로운 로우를 추가하는 것은 막지 않음
        - 따라서 SELECT로 조건에 맞는 로우를 전부 가져오는 경우 트랜잭션이 끝나기 전에 추가된 로우가 발견될 수 있음
    - SERIALIZABLE

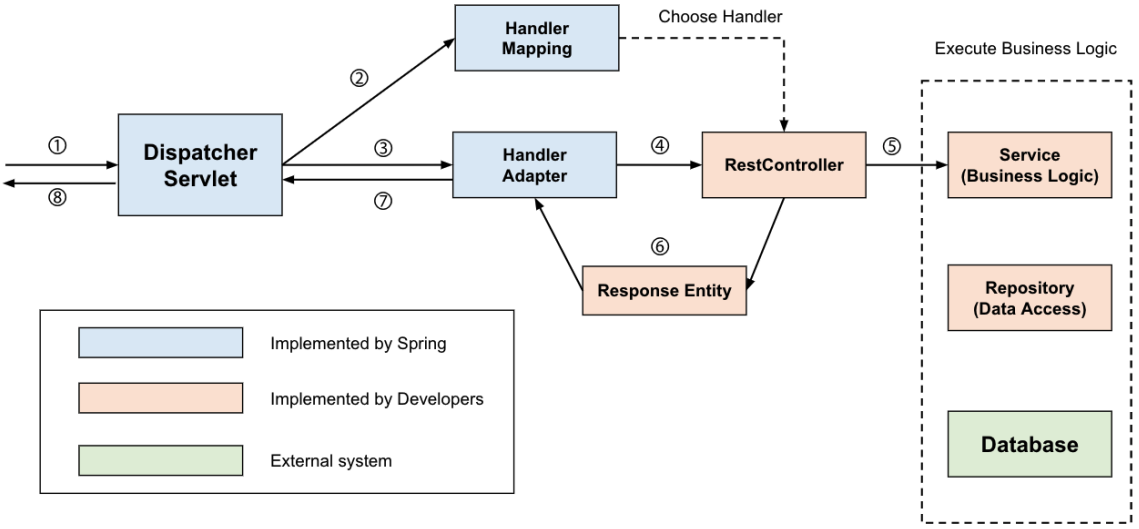
        - 가장 강력한 트랜잭션 격리 수준으로, 이름 그대로 트랜잭션을 순차적으로 진행시켜줌
        - 그러므로 SERIALIZABLE은 여러 트랜잭션이 동시에 같은 테이블의 정보를 액세스할 수 없음
        - SERIALIZABLE은 가장 안전하지만 가장 성능이 떨어지므로 극단적으로 안전한 작업이 필요한 경우에만 사용
    - 읽기전용 (readOnly)
        - 읽기 전용으로 설정함으로써 성능을 최적화함
        - 쓰기 작업 (DML)이 일어나는 것을 의도적으로 방지함
        - default: false
        - `@Transactional(readOnly = true)`
    - 롤백 처리
        - default 로 check 예외는 롤백되지 않고 runtime 예외는 롤백된다.
        - 커밋 대상이지만 롤백을 발생시킬 예외나 클래스 이름은 각각 rollbackFor 또는 rollbackForClassName으로 지정할 수 있으며, 반대로 롤백 대상인 런타임 예외를 트랜잭션 커밋 대상으로 지정하기 위해서는 noRollbackFor 또는 noRollbackForClassName을 이용할 수 있다.
        - default: RuntimeException, Error
        - `@Transactional(rollbackFor = {IOException.class})`
    - 제한시간
        - 지정한 시간 내에 해당 메소드 수행이 완료되이 않은 경우 rollback 수행
        - default: -1
        - `@Transactional(timeout = 2)`
    - 우선 순위
        - 클래스 메소드 > 클래스 > 인터페이스 메소드 > 인터페이스
        - 로 되어 있지만, @Transaction 을 사용하는 경우는 비즈니스 로직단 service 단 메서드에서 처리함으로, 클래스 메서드에서만 사용될 것이다.
          클래스에 선언하고 클래스 메소드에도 선언하면 Override 하는 것으로 판단된다.
- 트랜잭션 관리
    - TransactionSynchronizationManager는 트랜잭션에 관한 정보들을 ThreadLocal로 관리한다.

      즉 트랜잭션은 하나의 스레드에서만 관리(생성, 종료 등)될 수 있다.

      때문에 트랜잭션이 걸린 메서드에서 Async를 사용해 비동기(다른 스레드로) 하던, 쓰레드를 새로 생성해 메서드 호출을 하면,

      호출한 메서드와 호출된 메서드는 서로 다른 스레드에서 동작하기 때문에 트랜잭션을 공유할 수 없다.
- 성능을 개선하기 위해 메서드 내에서 스레드를 생성하여 비동기로 쿼리를 날리면 어떻게 될까요?(힌트: 병렬 처리보다 트랜잭션에 대해 묻는거다)
    - 주관적인 생각으로는...
      에러를 뽑거나, 정상적인 트랜잭션을 기대할 수 없을 듯 합니다.
      왜냐면, 스프링에서 트랜잭션은 쓰레드 단위로 동작하는데, 새로운 쓰레드를 생성한다면, 독자적인 실행 흐름을 갖을 것으로 판단됩니다.
- 스프링 내부에서 트랜잭션이 어디에 저장 될까요?
    - PlatformTransactionManager - 스프링 프레임워크와 스프링 부트에서 제공되는 인터페이스 (트랜잭션을 시작하고 커밋 또는 롤백하는 등의 트랜잭션 관리 기능 정의)
    - 스프링 프레임워크와 스프링 부트에서는 여러 구현체를 제공합니다. 예를 들어, 데이터베이스의 경우 JDBC 기반의 DataSourceTransactionManager와 JPA 기반의 JpaTransactionManager가 제공됩니다.
    - 즉, PlatformTransactionManager 인터페이스를 구체화한 클래스 (...TransactionManager) 에서 트랜잭션이 관리 됩니다.

▼ 스프링 통신 과정



1. 클라이언트의 요청을 디스패처 서블릿이 받음
2. HandlerMapping 가 요청을 위임할 컨트롤러를 찾음
   HandlerMapping 은 컨트롤러를 찾는 역할
3. 요청을 컨트롤러로 위임할 HandlerAdapter를 찾아서 전달함
   HandlerAdapter 는 컨트롤러를 실행하는 역할
4. HandlerAdapter가 컨트롤러로 요청을 위임함
5. 비지니스 로직을 처리함 (Service)
6. 컨트롤러가 반환값을 반환함
7. HandlerAdapter가 반환값을 처리함
8. 서버의 응답을 클라이언트로 반환함

▼ **Spring 예외 (subtitle - ControllerAdvice ExceptionHandler) (참고)**

- Spring은 예외가 발생하면 가장 구체적인 예외 핸들러를 먼저 찾고, 없으면 부모 예외의 핸들러를 찾는다.
- Spring boot 에러 처리 과정

```
WAS -> filter -> servlet(Dispatcher-servlet) -> interceptor -> controller
-> controller (예외 발생) -> intercepter -> servlet(Dispatcher-servlet) -> filter -> WAS
-> WAS -> filter -> servlet(Dispathcer-servlet) -> intercepter -> controller(BasicErro
```

  - 기본적인 에러 처리 방식은 결국 에러 컨트롤러를 한번 더 호출하는 것이다. 그러므로 필터나 인터셉터가 다시 호출될 수 있는데, 이를 제어하기 위해서는 별도의 설정이 필요하다.
- BasicErrorController
  - 기본적인 에러 처리를 담당하는 컨트롤러인 BasicErrorController는 accept 헤더에 따라 에러 페이지(errorHtml 메서드)를 반환하거나 에러 메세지(error 메서드)을 반환한다.
  - errorHtml()과 error()는 모두 getErrorAttributeOptions를 호출해 반환할 에러 속성을 얻는데, 기본적으로 DefaultErrorAttributes로부터 반환할 정보를 가져온다. DefaultErrorAttributes는 전체 항목들에서 설정에 맞게 불필요한 속성들을 제거한다.
  - DefaultErrorAttributes는 잘 되어 있지만, 클라이언트가 원하는 에러를 보여주는 것에는 유용하지 않다.
- Spring 예외 처리
  - Spring 은 예외 처리라는 공통 관심사를 메인 로직에 분리하는 방식을 고안하였고, 예외 처리 전략을 추상화한 HandlerExceptionResolver 인터페이스를 생성했다. (전략 패턴)
  - 대부분의 HandlerExceptionResolver는 발생한 Exception을 catch하고 HTTP 상태나 응답 메세지 등을 설정한다. 그래서 WAS 입장에서는 해당 요청이 정상적인 응답인 것으로 인식한다.

```
public interface HandlerExceptionResolver {
    ModelAndView resolveException(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex);
}
```

    - 위의 Object 타입인 handler는 예외가 발생한 컨트롤러 객체이다. 예외가 던져지면 디스패처 서블릿까지 전달되는데, 적합한 예외 처리를 해 HandlerExceptionResolver 구현체들을 빈으로 등록해서 관리한다. 그리고 적용 가능한 구현체를 찾아 예외 처리를 하는데, 우선순위대로 아래의 4가지 구현체들이 빈으로 등록되어 있다.
        - DefaultErrorAttributes: 에러 속성을 저장하며 직접 예외를 처리하지는 않는다.
        - ExceptionHandlerExceptionResolver: 에러 응답을 위한 Controller나 ControllerAdvice에 있는 ExceptionHandler를 처리함
        - ResponseStatusExceptionResolver: Http 상태 코드를 지정하는 @ResponseStatus 또는 ResponseStatusException를 처리함
        - DefaultHandlerExceptionResolver: 스프링 내부의 기본 예외들을 처리한다.
    - DefaultErrorAttributes는 직접 예외를 처리하지 않고 속성만 관리하므로 성격이 다르다. 그래서 내부적으로 DefaultErrorAttributes를 제외하고 직접 예외를 처리하는 3가지 ExceptionResolver들을 HandlerExceptionResolverComposite로 모아서 관리한다.
- Spring은 아래와 같은 도구들로 ExceptionResolver를 동작시켜 에러를 처리를 한다.
    1. ResponseStatus
    2. ResponseStatusException
    3. **ExceptionHandler**
    4. **ControllerAdvice**, RestControllerAdvice
- @ResponseStatus
    - @ResponseStatus는 에러 HTTP 상태를 변경하도록 도와주는 어노테이션
    - 예외가 발생하면 ResponseStatusExceptionResolver가 에러 처리함.
    - 사용 가능 경우
        - Exception 클래스 자체
        - 메소드에 @ExceptionHandler와 함께
        - 클래스에 @RestControllerAdvice와 함께
    - 한계
        - 에러 응답의 내용(Payload)를 수정할 수 없음(DefaultErrorAttributes를 수정하면 가능하긴 함)
        - 예와 클래스와 강하게 결합되어 같은 예외는 같은 상태와 에러 메세지를 반환함
        - 별도의 응답 상태가 필요하다면 예외 클래스를 추가해야 됨
        - WAS까지 예외가 전달되고, WAS의 에러 요청 전달이 진행됨
        - 외부에서 정의한 Exception 클래스에는 @ResponseStatus를 붙여줄 수 없음
- ResponseStatusException
    - 외부 라이브러리에서 정의한 코드는 우리가 수정할 수 없으므로 @ResponseStatus를 붙여줄 수 없다. Spring5에는 @ResponseStatus의 프로그래밍적 대안으로써 손쉽게 에러를 반환할 수 있는 ResponseStatusException가 추가되었다.
    - ResponseStatusException는 HttpStatus와 함께 선택적으로 reason과 cause를 추가할 수 있고, 언체크 예외을 상속받고 있어 명시적으로 에러를 처리해주지 않아도 된다.
    - 예외가 발생하면 ResponseStatusExceptionResolver가 에러 처리함.
    - 장점
        - 기본적인 예외 처리를 빠르게 적용할 수 있으므로 손쉽게 프로토타이핑할 수 있음
        - HttpStatus를 직접 설정하여 예외 클래스와의 결합도를 낮출 수 있음
        - 불필요하게 많은 별도의 예외 클래스를 만들지 않아도 됨

        - 프로그래밍 방식으로 예외를 직접 생성하므로 예외를 더욱 잘 제어할 수 있음
    - 한계
        - 직접 예외 처리를 프로그래밍하므로 일관된 예외 처리가 어려움
        - 예외 처리 코드가 중복될 수 있음
        - Spring 내부의 예외를 처리하는 것이 어려움
        - 예외가 WAS까지 전달되고, WAS의 에러 요청 전달이 진행됨
- @ExceptionHandler
    - @ExceptionHandler는 매우 유연하게 에러를 처리할 수 있는 방법을 제공하는 기능이다. @ExceptionHandler는 다음에 어노테이션을 추가함으로써 에러를 손쉽게 처리할 수 있다.
    - @ExceptionHandler는 Exception 클래스들을 속성으로 받아 처리할 예외를 지정할 수 있다. 만약 ExceptionHandler 어노테이션에 예외 클래스를 지정하지 않는다면, 파라미터에 설정된 에러 클래스를 처리하게 된다. 또한 @ResponseStatus와도 결합가능한데, 만약 ResponseEntity에서도 status를 지정하고 @ResponseStatus도 있다면 ResponseEntity가 우선순위를 갖는다.
    - ExceptionHandler는 @ResponseStatus와 달리 에러 응답(payload)을 자유롭게 다룰 수 있다는 점에서 유연하다. 예를 들어 응답을 다음과 같이 정의해서 내려준다면 좋을 것이다.
        - code: 어떠한 종류의 에러가 발생하는지에 대한 에러 코드
            - code 는 내부적인 커스텀 code 보다 Http 상태 코드를 사용하는게 좋다.
        - message: 왜 에러가 발생했는지에 대한 설명
        - erros: 어느 값이 잘못되어 @Valid에 의한 검증이 실패한 것인지를 위한 에러 목록
    - 사용 가능 경우
        - 컨트롤러의 메소드
        - @ControllerAdvice나 @RestControllerAdvice가 있는 클래스의 메소드
    - 주의점
        - @ExceptionHandler에 등록된 예외 클래스와 파라미터로 받는 예외 클래스가 동일해야 한다는 것이다. 만약 값이 다르다면 스프링은 컴파일 시점에 에러를 내지 않다가 런타임 시점에 에러를 발생시킨다.
        - Controller 클래스에서 예외를 처리하게 되면, 컨트롤러 클래스마다 공통되는 부분이 있을 수 있음으로, 전역적 @ExceptionHandler를 적용해야한다.
- @ControllerAdvice와 @RestControllerAdvice
    - 두 어노테이션는 여러 컨트롤러에 대해 전역적으로 ExceptionHandler를 적용해준다.
    - 장점
        - 하나의 클래스로 모든 컨트롤러에 대해 전역적으로 예외 처리가 가능함
        - 직접 정의한 에러 응답을 일관성있게 클라이언트에게 내려줄 수 있음
        - 별도의 try-catch문이 없어 코드의 가독성이 높아짐
    - 주의점
        - 여러 ControllerAdvice가 있을 때 @Order 어노테이션으로 순서를 지정해야한다.
        - 한 프로젝트당 하나의 ControllerAdvice만 관리하는 것이 좋다.
        - 만약 여러 ControllerAdvice가 필요하다면 basePackages나 annotations 등을 지정해야 한다.
        - 직접 구현한 Exception 클래스들은 한 공간에서 관리한다.

▼ 실제 사용 예

```
@ControllerAdvice
@Order(Ordered.HIGHEST_PRECEDENCE)
public class ApiExceptionHandler
{
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(ApiException.class)
    @ResponseBody
```

```
    public ApiExceptionResponse handleApiException(ApiException e)
    {
        logger.error("ApiExceptionHandler.handleException: ", e);
        return new ApiExceptionResponse(e);
    }

    public static class ApiExceptionResponse
    {
        public int code;
        public String message;

        public ApiExceptionResponse(ApiException e)
        {
            if (e == null)
            {
                code = ErrorCodes.INTERNAL_ERROR;
                message = "null exception was passed to ApiExceptionHandler";
            }
            else
            {
                code = e.code;
                message = CommonUtil.getExceptionReasons(e);
            }
        }
    }

    private static final Logger logger = LoggerFactory.getLogger(ApiExceptionHandler.c
}

public class ApiException extends RuntimeException
{
    public int code;

    public ApiException(int code)
    {
        this.code = code;
    }

    public ApiException(int code, String message)
    {
        super(message);
        this.code = code;
    }
}
```

  - Ordered.HIGHEST_PRECEDENCE 로 가장 우선순위가 빠르다.
  - @ExceptionHandler(ApiException.class) 로 ApiException 에러 발생시 예외 처리한다.
- 예외 처리 순서
  1. ExceptionHandlerExceptionResolver가 동작함
     a. 예외가 발생한 컨트롤러 안에 적합한 @ExceptionHandler가 있는지 검사함
     b. 컨트롤러의 @ExceptionHandler에서 처리가능하다면 처리하고, 그렇지 않으면 ControllerAdvice로 넘어감
     c. ControllerAdvice안에 적합한 @ExceptionHandler가 있는지 검사하고 없으면 다음 처리기로 넘어감
  2. ResponseStatusExceptionResolver가 동작함
     a. @ResponseStatus가 있는지 또는 ResponseStatusException인지 검사함

b. 맞으면 ServletResponse의 sendError()로 예외를 서블릿까지 전달되고, 서블릿이 BasicErrorController로 요청을 전달함

3. DefaultHandlerExceptionResolver가 동작함
   a. Spring의 내부 예외인지 검사하여 맞으면 에러를 처리하고 아니면 넘어감
4. 적합한 ExceptionResolver가 없으므로 예외가 서블릿까지 전달되고, 서블릿은 SpringBoot가 진행한 자동 설정에 맞게 BasicErrorController로 요청을 다시 전달함

▼ DB (MySQL, PostgreSQL)

▼ 인덱스

- 인덱스(Index)는 데이터베이스의 테이블에서 데이터를 빠르게 검색하기 위해 사용되는 데이터 구조입니다. 인덱스는 특정 열 또는 열의 조합에 대해 생성됩니다. 인덱스를 사용하면 데이터베이스 엔진은 인덱스를 통해 테이블을 스캔하는 대신 인덱스를 스캔하여 원하는 결과를 빠르게 찾을 수 있습니다. 이로 인해 조회 쿼리 응답 시간이 향상될 수 있습니다. 그러나 인덱스는 테이블에 대한 쓰기 작업(DML)의 성능에 영향을 줄 수 있으므로 적절하게 사용해야 합니다. (인덱스 테이블에도 추가적인 작업이 발생되기 때문)
- 인덱스 선택

카디널리티 (Cardinality)	높을 수록 적합	값의 다양성의 척도 (값의 중복 과 반비례)
선택도 (Selectivity)	낮을 수록 적합 (10% 이하 적정)	컬럼의 특정 값의 row 수 / 테이블의 총 row 수 * 100
활용도	높을 수록 적합	where 절, join 절에 자주 사용되는지
인덱스 중복도	없을 수록 적합	중복되는 인덱스 여부

  - 복합 컬럼 인덱스 (**인덱스 선정기준에 따라서 순서를 정한다**).
    ex) 사람의 테이블에 성별, 이름인 경우 이름이 값의 다양성이 더 많기 때문에, 인덱스를 생성시 이름, 성별 순서로 생성해야 한다.
- 인덱스 동작이 불가능 한 경우
  - (전제 user 테이블에 salary(int4), id(char) 각각 단일 인덱스로 있다)
  - 인덱스 컬럼 절의 변형
    - `select * from user where salary * 2 > 400` 인덱스 불가
    - `select * from user where salary > 200` 인덱스 가능
  - 내부적인 데이터 변환
    - `select * from user where id = 591234` 인덱스 불가 (내부 변형)
    - `select * from user where id = to_char(591234`) 인덱스 가능
  - NULL 조건의 사용
    - `select * from user where id is not null` 인덱스 불가
      인덱스에는 NULL 값을 포함하지 않음.
  - 부정형 조건의 사용
    - `select * from user where id != '591234'` 인덱스 불가
  - LIKE 연산자 사용
    - `select * from user where id like '%5'` 인덱스 불가
    - `select * from user where id like '5%'` 인덱스 가능
- 종류 (PostgreSQL 기준)
  - BTree
    B-트리는 데이터를 정렬된 형태로 저장하여 효율적인 검색과 정렬된 결과를 제공합니다. 이진 트리의 확장 버전인 B-트리는 각 노드에 여러 개의 키를 가지고 있고, 각 키에 해당하는 값 또는 자식 노드로 연결됩니다. B-트리는 빠른 검색과 삽입/삭제 연산을 지원함.
    equal(=)뿐만 아니라 >, >=, <, <= 또는 between
  - GIN
    GIN은 텍스트 검색, 배열, JSONB와 같은 복잡한 데이터 유형에 대한 검색을 지원하는 데 유용함.

인덱스는 Full text 검색 속도를 높이는데 적합한 index이다.

- Gist
  GiST는 데이터의 공간 관계를 표현하는 데 사용되며, 공간 검색, 전문 검색, 유사도 검색 등 다양한 검색 유형을 지원합니다.
  인덱스의 핵심 시스템은 Geometry 데이터 유형에 대한 R-Tree 기능을 제공

▼ 파티션

- 파티션(Partition)은 대량의 데이터를 더 작고 관리 가능한 조각으로 분할하는 데이터 관리 기법입니다. 파티셔닝을 사용하면 테이블을 논리적 또는 물리적으로 여러 개의 파티션으로 분할할 수 있습니다. 각 파티션은 개별적으로 관리되며, 데이터를 효율적으로 분산시키거나 쿼리의 성능을 향상시키기 위해 사용될 수 있습니다. 예를 들어, 파티션을 사용하면 특정 시간 범위에 속한 데이터만 검색하거나 삭제하는 등의 작업을 더 빠르게 수행할 수 있습니다. 또한 파티션은 대량의 데이터를 다수의 저장 장치에 분산하여 저장할 수 있으므로 데이터베이스의 용량 한계를 극복하는 데에도 도움이 될 수 있습니다. (delete 시 dead tuple(PostgreSQL 기준) 같은 개념이 발생되어, optimize(MySQL), vacuum(PostgreSQL) 작업이 필요하고 delete 는 테이블 크기가 크면 해당 delete 문을 처리하는 기간이 길다. 하지만, 파티션 관리 후 drop partition 하면 금방되며 dead tuple 같은 개념을 걱정하지 않아도 된다.)
- 파티션 선택
  - 쿼리 패턴, 데이터 분포 등을 가능한 컬럼 기준으로 파티션을 건다.
  - ex) 날짜, 위치 컬럼이 파티션 설정하는게 좋다.
- 종류
  - Range Partitioning - 범위 (ex. 매달의 1일 부터 말일까지)
  - List Partitioning - 특정 값 (ex. 서울, 부산, 인천 등)
  - Hash Partitioning - 현재 어떤 컬럼을 기준으로 파티션하기 어려운 경우 사용됨

▼ ACID

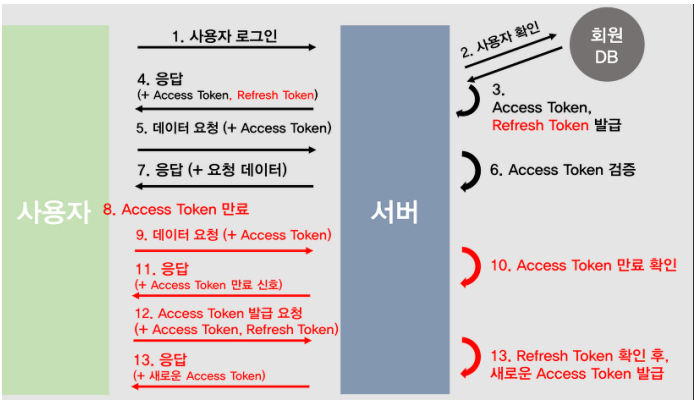
- 트랜잭션(Transaction)은 데이터베이스의 상태를 변환시키는 하나의 논리적 기능을 수행하기 위한 작업의 단위 또는 한꺼번에 모두 수행되어야 할 일련의 연산들을 의미
- 원자성 (Atomicity)
  트랜잭션의 연산은 DB에 모두 반영되던지 아니면 전혀 반영되지 않아야 한다
- 일관성 (Consistency)
  트랜잭션 실행이 성공적으로 완료하면 항상 일관성 있는 DB 상태로 유지하는 것을 의미
  데이터의 일관된 상태를 유지하여 데이터의 무결성을 보장
- 독립성 (Isolation)
  트랜잭션 수행시 다른 트랜잭션이 끼어들지 못하도록 보장하는 것을 의미
- 지속성 (Durability)
  성공적으로 수행된 트랜잭션은 영원히 반영되어야 함을 의미
  트랜잭션이 커밋되면 그 결과가 영구적으로 저장되어 데이터의 지속성을 보장

▼ Web

▼ 쿠키 vs 세션 vs JWT

- 인증 (Authentication)
  - 인증은 쉽게 말하자면, 로그인 이다. 클라이언트가 자기자신이라고 주장하고 있는 사용자가 맞는지를 검증하는 과정
- 인가 (Authorization)
  - 인가는 인증 작업 이후에 행해지는 작업으로, 인증된 사용자에 대한 자원에 대한 접근 확인 절차를 의미.
- HTTP 통신
  - HTTP는 상태를 저장하지 않습니다.(Stateless)
  - HTTP는 요청과 응답을 한번 주고받으면 바로 연결을 끊어버리는 특성을 가지고 있다. (비연결성)
  - HTTP의 핵심은 상태가 없는 것이지만 HTTP 쿠키는 상태가 있는 세션을 만들도록 해줍니다.
  - 헤더 확장성을 사용하여, 동일한 컨텍스트 또는 동일한 상태를 공유하기 위해 각각의 요청들에 세션을 만들도록 HTTP 쿠키가 추가됩니다.
- 쿠키
  - 쿠키는 HTTP 요청과 응답에 함께 실려 전송된다.

    - 쿠키는 클라이언트 즉, 브라우저에 저장된다.
    - 웹 서버가 클라이언트로 보내는 응답의 헤더 중 `Set-Cookie` 라는 헤더에 키와 값을 함께 실어 보내면
      그 응답을 받은 브라우저는 해당 쿠키를 저장하고, 그 다음 요청부터 자동으로 쿠키를 헤더에 넣어 송신한다.
- 세션
    - 세션은 쿠키와 다르게 정보를 서버측에 저장하는 방식이다.
    - 세션이란 브라우저로 웹서버에 접속한 시점부터 브라우저를 종료하여 연결을 끝내는 시점까지의 일련의 요청을 하나의 상태로 간주하고,
      그 상태를 일정하게 유지하는 기술이다.
    - 사용자가 HTTP 요청의 Body에 인증 정보 (유저이름이나 패스워드 같은 것들) 을 실어 서버로 보낸다.
      서버에서는 해당 인증정보가 유효하면 사용자와 데이터를 식별하는 Session ID를 생성한다. (tomcat - JSESSIONID)
      생성된 Session ID는 응답의
      `Set-Cookie` 헤더에 생성된 세션 아이디를 실어 보내진다.
    - 클라이언트는 해당 세션 아이디를 쿠키에 저장하고, 매 요청마다 세션 아이디를 `Cookie` 헤더에 실어 전송한다.
      서버는 전달받은 세션 아이디를 통해 해당 요청의 송신자가 누구인지 식별할 수 있다.
- JWT (Json Web Token)
    - JWT 는 정보가 토큰 자체에 포함된 (Self-Contained) 클레임 (Claim) 기반 토큰이다.
    - 인증 (Authentication) 과 권한부여(Authorization) 에 사용되는 것이 가장 일반적이다. 인증 절차를 거쳐 서버에서 JWT를 발급해주면, 클라이언트는 이를 잘 보관하고 있다가 API 등을 사용할 때에 서버에 JWT를 함께 제출하며 서버로부터 행위에 대해 인가 받을 수 있다.
    - JWT 는 해시 혹은 비대칭키 방식을 사용하여 서명 (Signature) 하기 때문에 무결성을 검증할 수 있다.
    - 서버는 '발급' 과 '검증' 두가지 역할만 함으로 따로 저장하고 있지 않아 Stateless 하다.
      (refresh 토큰을 사용하는 경우 별도의 저장소가 있어야 함으로 그렇게 되면 Stateful 하다)
    - Refresh 토큰 생성 히스토리
        - Access Token 만을 통한 인증 방식의 문제는 만일 제 3자에게 탈취당할 경우 보안에 취약하다는 점이다.
          Access Token은 발급된 이후, 서버에 저장되지 않고 토큰 자체로 검증을 하며 사용자 권한을 인증하기 때문에,
          Access Token이 탈취되면 토큰이 만료되기 전 까지, 토큰을 획득한 사람은 누구나 권한 접근이 가능하다.
          JWT는 발급한 후 삭제가 불가능하기 때문에, 접근에 관여하는 토큰에 유효시간을 부여하는 식으로 탈취 문제에 대해 대응을 하여야 한다.
        - Access Token 유효 시간은 짧으면, 새로 로그인 해야 하는 과정이 너무 많아짐으로 Refresh Token (Access Token 과 형식은 똑같고 유효 기간이 보다 길음) 을 새로 발급해 다시 로그인 하는 과정을 스킵할 수 있다.
        - Access Token 은 인가를 받아 서버의 응답 데이터를 받은 용도
          Refresh Token 은 Access Token 을 새로 발급하는 용도
    - Refresh 토큰 생성 과정



    - 구조
        - **헤더 (Header), 페이로드 (Payload), 서명 (Signature) - (** `헤더.페이로드.서명` **)**
        - 헤더 - 토큰의 유형과 암호화 알고리즘 두가지 정보
        - 페이로드 - 페이로드는 사용자의 정보 혹은 데이터 속성 등을 나타내는 클레임(Claim) 이라는 정보

        - 서명 - 특정 암호화 알고리즘을 사용하여, Base64 인코딩된 헤더와 Base64 인코딩된 페이로드 그리고 비밀키를 이용하여 암호화한다. 서명을 통해 서버는 헤더 혹은 페이로드가 누군가에 의해 변조되었는지 그 무결성을 검증하고 보장할 수 있다.
        - header json byte64 encoding → payload json byte64 encoding → Signature Encoding 암호화(header 에서 정의된 알고리즘)(header json byte64 encoding + . + payload json byte64 encoding + server's key)
- 역사
    1. 쿠키
        - 로그인시 인증을 성공하면, 쿠키 (key, value)로 만 userId, pw 를 전송해 관리함
        - 쿠키 노출시 중요 정보를 노출하게 된다.
        - 쿠키 사이즈가 낮은 사이즈로 고정되어 있다.
    2. 쿠키 + 세션
        - Cookie에 ID, PW와 같은 중요 정보들을 담는것이 아닌, 중요 정보가 아닌 인증을 위한 별개의 정보를 세션 저장소에 저장하고,
          클라이언트는 이 정보를 쿠키에 대신 담아서 요청하고 서버는 세션 저장소에 있는 정보랑 일치하는지 확인하는 방식이다.
        - http의 가장 큰 특성중 하나인 `stateless`를 위배한다는 것이다.

          `stateless`라면 서버는 클라이언트의 상태를 저장하지 않아야 하지만 세션 저장소라는 곳에서 클라이언트의 상태를 저장하게 되므로 `stateful`하게 된다.
    3. JWT
        - 두 개체(client, server)에서 JSON 객체를 사용하여 가볍고 자가수용적인 (self-contained) 방식으로 정보를 안전성 있게 전달한다.
        - 인증에 필요한 정보가 토큰에 들어있어서 별도의 저장소가 필요 없다.
            - 하지만 보안성을 높이기위해 `Refresh Token`을 사용하는경우 별도의 저장소에 저장하면서 사용하는 경우에는 해당하지 않는다.
        - Cookie와 Session 사용시 문제점이였던 stateful 특성을 JWT 사용시 stateless하게 가져갈 수 있다. 즉 서버는 클라이언트의 상태를 가질 필요가 없다.
        - 다양한 언어에서 지원한다.
        - HTTP 헤더에 넣어서 쉽게 전달 가능하다.
        - MSA 환경에서 유용하다
        - 오버헤드가 발생된다. (거의 모든 요청에 토큰이 포함되므로 트래픽 크기에 영향 - 네트워크 부하 발생 가능)
        - 페이로드는 암호화된게 아니라 `BASE64`로 인코딩 된 것이므로 중간에 토큰을 탈취하면 페이로드의 데이터를 모두 볼 수 있다.

▼ 클러스터 (세션, Server)

- 클러스터(Cluster)는 여러 대의 컴퓨터로 이루어진 하나의 묶음입니다.
  클러스터를 구성하는 각 컴퓨터들을 노드(node)라고 하며, 이 노드들은 특별한 기술로 연결되어 하나의 컴퓨터처럼 동작할 수 있습니다.
  클러스터는 높은 수준의 가용성, 안정성, 확장성 을 제공 하기 위해 하나의 시스템을 이용하는것보다, 두개 또는 그이상의 시스템을 이용한다.
- 서버 클러스터(Server Cluster - Scale Out)는 여러 대의 서버 컴퓨터들로 이루어진 클러스터(Cluster)입니다. 서버 클러스터를 구성하는 각 서버들은 공유된 저장소(Shared Storage)에 접근하여 데이터를 공유하고, 클라이언트(Client)의 요청에 따라 부하 분배를 수행하며 (로드밸런싱), 장애 발생 시 서비스의 지속성을 보장하기 위해 상호 백업 등의 기능을 수행합니다.
  서버 클러스터는 고가용성(High Availability)을 보장하여, 장애 발생 시 다른 서버로 자동 이전되므로 시스템 다운 타임을 최소화하고, 이를 통해 서비스의 안정성을 높일 수 있습니다. 또한, 서버의 자원 활용률을 높일 수 있어, 처리량 향상과 비용 절감 효과를 얻을 수 있습니다.
- 서버 한 대에 하나의 세션 저장소가 생성되는데, 이때 서버 클러스터링으로 동작하고 세션에 대한 처리가 없다면 정합성 이슈(여러 서버에서 사용자 세션을 나눠 갖기 때문에 발생하는 이슈)가 발생됩니다. 이를 위한 별도의 처리는 3가지 방식이 있습니다.
    - Sticky Session 방식은 세션을 최초에 생성한 서버로 요청을 고정하는 방식입니다. 정합성 이슈를 해결할 수 있으나, 로드밸런싱과 가용성면에서 문제가 있을 수 있습니다.

    - Session Clustering 은 세션을 생성될 때마다 복제하여 각 서버의 세션 정보를 일치시켜 정합성 이슈를 해결합니다. 하지만, 매번 세션 객체를 복제하는데 오버헤드가 발생하므로 사용 시, 이를 고려해야 합니다.
    - 세션 스토리지 분리 방식 별도의 세션 저장소를 사용하는 것으로, 서버가 아무리 늘어난다고 할 지라도 세션 스토리지에 대한 정보만 각각의 서버에 입력해주면 세션을 공유할 수 있게 됩니다.
        - 빈번한 Read/Write가 이루어지는 세션 저장소로써 Disk 기반의 데이터베이스는 상대적으로 I/O 속도가 느리기 때문에 적합하지 않습니다.
        - In-Memory 데이터베이스를 사용하면 데이터를 메모리에서 Read/Write 할 수 있다는 점에서 빠른 속도로 데이터를 처리할 수 있기 때문에 세션 저장소로써 적합합니다.
        - In-Memory 데이터베이스는 전원 공급이 중단되면 데이터를 잃어버리지만, 세션 저장소에 저장되는 데이터는 상대적으로 피해가 적기 때문에 In-Memory 데이터베이스 사용이 적합합니다. (일부 데이터베이스는 Replication을 지원하기 때문에 가용성을 확보할 수 있습니다.)
            - In-Memory 데이터베이스로 key value 모델인 Redis와 Memcached 를 주로 사용한다.
            - 성능적인 측면에서는 Memcached는 쓰기 성능, Redis는 읽기 성능 및 메모리 사용 효율을 갖는다.

▼ CORS

- 정의
    - 교차 출처 리소스 공유(Cross-Origin Resource Sharing, CORS)는 웹 페이지의 리소스 요청을 다른 도메인으로부터 수신하기 위해 브라우저와 서버 간의 교차 출처 통신을 가능하게 하는 메커니즘입니다. 기본적으로 웹 브라우저는 동일 출처 정책(Same-Origin Policy)을 따르므로, 서로 다른 도메인 간의 리소스 공유가 제한되어 있습니다. CORS는 이러한 제약을 완화할 수 있는 기술입니다.
    - origin(출처) 이란 protocol, host, port 로 구성된다. 이때 3가지 요소가 같으면 SOP(Same-Origin Policy) 동일 출처 라고 한다. SOP 는 같은 출처에서만 리소스를 공유할 수 있다는 규칙으로, 브라우저에서 다른 서버에서 요청할 경우에 해당되고, 브라우저를 거치지 않고 서버 간 통신할 때는 이 정책이 적용되지 않는다.
- 동작 방식
    - 클라이언트가 서버에 교차 출처 리소스 요청을 보냅니다. 이 때 요청 헤더에는 Origin이라는 필드가 포함되며, 요청이 어느 출처로부터 왔는지 알려줍니다.
    - 서버는 클라이언트 요청을 받고 CORS 정책을 확인합니다. 서버는 이를 허용하는 출처를 나타내는 Access-Control-Allow-Origin이라는 헤더를 응답에 포함시킵니다.
    - 브라우저는 서버의 응답을 받아 분석합니다. Access-Control-Allow-Origin 헤더에 허용되는 출처가 포함되어 있다면, 요청 성공 처리를 하고 리소스를 사용할 수 있게 됩니다. 만약 출처가 허용되지 않거나 CORS 관련 헤더가 누락되면 브라우저는 오류를 발생시킵니다.
    - 추가적으로 서버는 Access-Control-Allow-Methods, Access-Control-Allow-Headers, Access-Control-Allow-Credentials 등의 헤더를 사용하여 요청을 처리하는 세부 사항을 설정할 수 있습니다.
        - `Access-Control-Allow-Origin` : 요청을 허용할 출처를 명시할 때 사용하며, *를 사용하면 모든 출처의 리소스 요청을 허용합니다.
        - `Access-Control-Allow-Methods` : 어떤 메서드를 허용할 것인지 명시합니다.
        - `Access-Control-Allow-Headers` : 어떤 헤더들을 허용할 것인지 명시합니다.
        - `Access-Control-Max-Age` : preflight 요청에 대한 응답을 브라우저에서 얼마만큼 캐싱하고 있을지 설정할 때 사용합니다.
        - `Access-Control-Allow-Credentials` : 클라이언트에서 보낸 자격 인증 정보(세션 ID가 저장되어있는 쿠키(Cookie) 혹은 Authorization 헤더에 설정하는 토큰)를 허용할 것인지 명시합니다.
        - `Access-Control-Expose-Headers` : 브라우저가 스크립트에 노출시킬 헤더의 목록을 명시할 때 사용합니다. 기본적으로 다음 7가지 헤더(일명 CORS safe-listed header)는 따로 설정하지 않아도 노출시킵니다.
            - `Cache-Control`
            - `Content-Language`
            - `Content-Length`
            - `Content-Type`
            - `Expires`
            - `Last-Modified`
            - `Pragma`

▼ **RESTAPI**

- Representational State Transfer
  HTTP 프로토콜을 그대로 활용하기 때문에 웹의 장점을 최대한 활용할 수 있는 아키텍처 스타일
- 구조
  - Resource - HTTP URI
  - Verb - HTTP METHOD
  - Representations - HTTP Message Pay Load (Content-type, Content-Encoding)
- 특징
  - Uniform (유니폼 인터페이스)
    - URI 로 지정한 리소스에 대한 조작을 통일되고 한정적인 인터페이스로 수행하는 아키텍쳐 스타일
  - Stateless (무상태성)
    - 작업을 위한 상태 정보를 저장하지 않음, API 서버는 요청만 단순히 처리함으로
      서비스의 자유도가 높아지고 서버에 정보를 저장하지 않음으로 구현이 단순화 됨
  - Cache 가능
    - HTTP 프로토콜 표준에서 사용하는 Last-Modified태그나 E-Tag를 이용하면 캐싱 구현
  - Client - Server 구조
    - REST 서버는 API 제공
      클라이언트는 사용자 인증, 컨텍스트 (로그인 정보) 등을 직접 관리하는 구조
    - 즉, 서로 독립적인 구조로 서로간 의존성이 줄어듬
  - 계층형 구조
    - 서버는 다중 계층(보안, 로드 밸런싱, 암호화 등)으로 구성될 수 있어, 구조 유연성을 제공
- 장점
  - 간결하고 직관적인 인터페이스
    - HTTP 메서드와 URI를 통해 리소스에 접근 가능
  - 확장성, 유연성
    - 클라이언트와 서버간 역할이 명확히 분리하고, 각 리소스에 고유한 URI 를 부여함
    - HTTP 프로토콜 기반으로 HTTP 를 사용하는 플랫폼이면 RESTAPI Call 이 가능하다.
  - 캐싱 기능
    - HTTP 프로토콜의 캐시를 사용할 수 있음
- 단점
  - 표준이 없다.
    - 관리의 어려움과 (공식화 된) 좋은 API 디자인 가이드가 존재하지 않음
- 대중적인? 설계 (표준이 없지만...)
  - 슬래시(/) 계층 관계 표현
  - URI 마지막 슬래시(/) 포함 X
  - 언더바(_) 대신 하이픈(-) 사용
  - URI 소문자 적합
  - 파일인 경우 파일 확장자 포함 X
  - 명사로 표현
  - HTTP 상태 코드 사용

▼ **HTTP 통신**

- HTTP (HyperText Transfer Protocol)
  - TCP/IP 를 이용하는 응용 프로토콜

    - HTTP는 연결 상태를 유지하지 않는 비연결성 프로토콜
- 클라이언트 ↔ 서버로 나눠진 구조
    - Request(c → s)
        - Http Request 구조
            - Request Line
                - Method, URL, Http Version
            - Header
                - HTTP 전송에 필요한 모든 부가 정보 (크기, 인증, 서버 정보, 캐시 등)
            - Body
                - 실제 전송할 데이터 (Html 문서, 이미지, JSON 등)
    - Response(s → c)
        - Http Response 구조
            - Response Line
                - Status Code, Status Message, Http Version
            - Header
                - HTTP 전송에 필요한 모든 부가 정보 (크기, 인증, 서버 정보, 캐시 등)
            - Body
                - 전송 받은 데이터
    - 클라이언트와 서버가 나눠져있어 각자의 역할에 집중할 수 있음
- HTTP 메서드

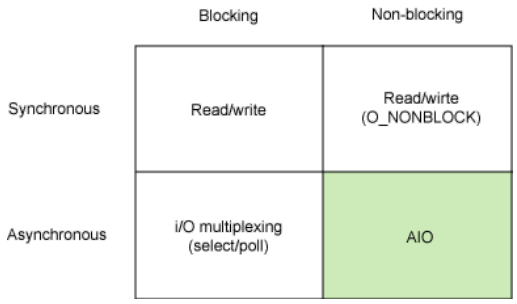
메서드	설명	Request Body	Response Body	안정 (호출해도 리소스 변경이 없음)	멱등 (여러번 call 해도 한 번 call 한 것과 같음)	캐시 (응답 결과를 캐시 가능)
GET	리소스 요청	X	O	O	O	O
HEAD	GET 과 동일한 응답을 요청하지만 Response Line + Header 만 응답	X	X	O	O	O
POST	내용 전송	O	O	X	X	X
PUT	내용 갱신	O	O	X	O	X
DELETE	리소스 삭제	권장하지 않으나 가능	권장하지 않으나 가능	X	O	X
CONNECT	목적 리소스로 식별되는 서버 터널 설정	X	O	X	X	X
OPTIONS	웹 서버측 제공 메소드 질의	X	O	O	O	X
TRACE	목적 리소스의 경로를 따라 메시지 loop-back 테스트	X	O	X	O	X
PATCH	내용 부분 갱신	O	O	X	X	X

▼ MSA 2PC commit

▼ MSA SAGA pattern

▼ 기타

▼ Non-Blocking vs Bloking, Sync vs Async



- 블로킹 Blocking
  - A함수가 B 함수를 호출 할 때, B 함수가 자신의 작업이 종료되기 전까지 A 함수에게 제어권을 돌려주지 않는 것
- 논블로킹 Non-blocking
  - 함수가 B 함수를 호출 할 때, B 함수가 제어권을 바로 A 함수에게 넘겨주면서, A 함수가 다른 일을 할 수 있도록 하는 것.
- 동기 Synchronous
  - A 함수가 B 함수를 호출 할 때, B 함수의 결과를 A 함수가 처리하는 것.
- 비동기 Asynchronous
  - A 함수가 B 함수를 호출 할 때, B 함수의 결과를 B 함수가 처리하는 것. (callback)
- 관점의 차이
  - 블로킹/논블로킹
    호출되는 함수가 바로 제어권을 넘겨주냐 안 넘겨 주냐
  - 동기/비동기
    호출되는 함수의 작업 완료 여부를 누가 신경쓰냐
- 주로 사용되는 모델
  - Sync-Blocking
    - 일반적인 동기 프로그래밍 모델
      A 함수 내 B 함수 호출 시, B 함수가 끝나야 A 함수에서 B 함수 호출 이후 코드를 진행 할 수 있음.
  - Async-NonBlocking
    - 일반적인 비동기 프로그래밍 모델
      A 함수가 B 함수를 호출하고 바로 제어권을 반환받는다. 제어권을 반환받은 A 함수는 호출된 B 함수의 작업 결과에 상관없이 바로 다른 작업을 할 수 있기 때문에 다음 라인으로 넘어간다. 호출된 B 함수는 작업을 수행하고 그 결과를 콜백 함수로 반환한다.
- 잘 사용되지 않는 모델
  - Sync-NonBlocking
    - A 함수가 B 함수를 호출하고 바로 제어권을 반환 받는다. A 함수는 다른 작업을 할 수는 있지만, B 함수 작업이 완료되었는지 A 함수는 계속 확인이 필요하다.
  - Async-Blocking
    - A 함수가 B 함수를 호출하고 바로 제어권을 반환하지 않고, A 함수는 호출된 B 함수의 작업 완료 여부를 신경쓰지 않는 것이다.

▼ **디자인 패턴**

- **디자인 패턴**은 소프트웨어 디자인 과정에서 자주 발생하는 문제들에 대한 전형적인 해결책입니다. 이는 코드에서 반복되는 디자인 문제들을 해결하기 위해 맞춤화할 수 있는 미리 만들어진 청사진과 비슷합니다.
- 표준화된 라이브러리들이나 함수들을 코드에 복사해 사용하는 것처럼 패턴들을 붙여넣기식으로 사용할 수 없습니다. 패턴은 재사용할 수 있는 코드 조각이 아니라 특정 문제를 해결하는 방식을 알려주는 일반적인 개념입니다. 당신은 패턴의 세부 개념들을 적용하여 당신의 프로그램에 맞는 해결책을 구현할 수 있습니다.
- 종류
  - 디자인 패턴 [생성]은 기존 코드의 유연성과 재사용을 증가시키는 객체를 생성하는 다양한 방법을 제공합니다.
    ex) 팩토리 메서드 패턴, 싱글톤 패턴, 프로토 타입 패턴[clone 메서드], 빌더 패턴 [
    이펙티브 자바] 등

- 디자인 패턴 [구조]은 구조를 유연하고 효율적으로 유지하면서 객체들과 클래스들을 더 큰 구조로 조립하는 방법을 설명합니다.
  ex) 어댑터 패턴, 데코레이터 패턴, 프록시 패턴 등
- 디자인 패턴 [행동]은 알고리즘들 및 객체 간의 책임 할당과 관련이 있습니다.
  ex) 템플릿 메서드 패턴, 옵저버 패턴, 전략 패턴 등

▼ 디자인 패턴 [생성]

▼ 싱글톤 패턴

- 애플리케이션에서 **하나의 인스턴스만 존재하며**, 주로 무겁고 공통된 객체를 사용해야 하는 경우에 사용된다.
  DB connection pool, ThreadPool, Cache 등
- 간단한 싱글톤 패턴

```
public class Singleton {

    private static Singleton instance;

    private Singleton() {
        throw new IllegalStateException("Private Constructor");
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- 멀티 쓰레드 안전 싱글턴 패턴
  더블 채킹 라킹 기법이 있지만, 주로 사용되는 정적 멤버 클래스(Holder 클래스)를 사용한다.

```
public class Singleton {
        private Singleton() {}

        private static class SingletonHolder {
            private static final Singleton INSTANCE = new Singleton();
        }

        public static Singleton getInstance() {
            return SingletonHolder.INSTANCE;
        }
}
```

- 싱글톤은 안티패턴이다?
  - Java 에서 일반적으로 싱글톤을 생성하면 안티 패턴이 될 수가 있다. 하지만 Spring 에서의 싱글톤은 IoC 컨테이너에서 직접 싱글톤으로 객체를 관리함으로 안티 패턴이 아니다.
  - 안티 패턴이 될 수 있는 경우 (Java)
    - private 생성자를 갖고 있어서 상속이 불가능하다.
    - 전역 상태를 만들 수 있기 때문에 바람직하지 못하다.

      싱글톤 인스턴스가 너무 많은 일을 하거나 많은 데이터를 공유시킬 경우 다른 클래스의 인스턴스들 간에 결합도가 높아져 OCP(개방-폐쇄 원칙)를 위배하게 된다.
    - 테스트하기가 어려워진다.
  - 하지만, 객체의 생성을 프레임워크 (Spring)에 위임할으로 써, 아래 처럼만 코드를 작성하면 됨으로, 위에 작성된 단점들을 모두 극복이 가능하다.

```
@Component
public class Bean { ... }
```

▼ **팩토리 메소드 패턴**

- **팩토리 메서드는 부모 클래스에서 객체들을 생성할 수 있는 인터페이스를 제공하지만, 자식 클래스들이 생성될 객체들의 유형을 변경할 수 있도록 하는 생성 패턴**
- 장점
  - 생성자(Creator)와 구현 객체(concrete product)의 강한 결합을 피할 수 있다.
  - 팩토리 메서드를 통해 객체의 생성 후 공통으로 할 일을 수행하도록 지정해줄 수 있다.
  - 캡슐화, 추상화를 통해 생성되는 객체의 구체적인 타입을 감출 수 있다
  - 단일 책임 원칙. 제품 생성 코드를 프로그램의 한 위치로 이동하여 코드를 더 쉽게 유지관리할 수 있습니다.
  - 개방/폐쇄 원칙. 기존 클라이언트 코드를 훼손하지 않고 새로운 유형의 제품들을 프로그램에 도입할 수 있습니다.
- 단점
  - 각 제품 구현체마다 팩토리 객체들을 모두 구현해주어야 하기 때문에, 구현체가 늘어날때 마다 팩토리 클래스가 증가하여 서브 클래스 수가 폭발한다.
  - 코드의 복잡성이 증가한다.
- 예제

```
public abstract class RobotFactory {
    abstract Robot createRobot(String name);
}

public class SuperRobotFactory extends RobotFactory {
    @Override
    Robot createRobot(String name) {
        switch(name) {
        case "super" :
            return new SuperRobot();
        case "power" :
            return new PowerRobot();
        }
        return null;
    }
}
```

▼ **디자인 패턴 [구조]**

▼ **어뎁터 패턴**

- 클래스를 바로 사용할 수 없는 경우가 있음 (다른 곳에서 개발했다거나, 수정할 수 없을 때) **중간에서 변환 역할**을 해주는 클래스가 필요하다면 어뎁터 패턴을 적용하면 됨.
- 호환되지 않은 인터페이스를 사용하는 클라이언트 그대로 활용 가능
  향후 인터페이스가 바뀌더라도, 변경 내역은 어댑터에 캡슐화 되므로 클라이언트 바뀔 필요X
  ex) 110V 제품 → 220V adapter → 220V 콘센트 사용, Arrays.asList
- 장점으로는 코드 유지 관리가 용이하고, 코드 재사용성이 향상됩니다.
  하지만, 둘 사이에 어뎁터 코드를 새로 작성해야 함으로 추가적인 코드 작성이 필요합니다.
- 예시

```
interface Duck {
      public void quack();
      public void fly();
}

class MallardDuck implements Duck {
```

```
      @Override
      public void quack() {
        System.out.println("Quack");
      }
      @Override
      public void fly() {
        System.out.println("I'm flying");
      }
}
interface Turkey {
      public void gobble();
      public void fly();
}

class WildTurkey implements Turkey {
      @Override
      public void gobble() {
        System.out.println("Gobble gobble");
      }
      @Override
      public void fly() {
        System.out.println("I'm flying a short distance");
      }
}

public class TurkeyAdapter implements Duck {
      Turkey turkey;

      public TurkeyAdapter(Turkey turkey) {
           this.turkey = turkey;
      }
      @Override
      public void quack(){
           turkey.gobble();
      }
      @Override
      public void fly() {
           turkey.fly();
      }
 }

public class Main {
        public static void main(String[] args) {
            Duck duck = new MallardDuck();

            Turkey turkey = new WildTurkey();
            Duck turkeyAdapter = new TurkeyAdapter(turkey);

            testDuck(duck);
            testDuck(turkeyAdapter);
        }

        public static void testDuck(Duck duck){
             duck.quack();
             duck.fly();
        }
}
```

▼ 데코레이터 패턴

- 주어진 상황 및 용도에 따라 어떤 객체에 책임을 덧붙이는 패턴으로, 기능 확장이 필요할 때 서브클래싱 대신 쓸 수 있는 유연한 대안이 될 수 있다.
- 데코레이터의 합성은 항상 클라이언트에 의해 제어된다는 점입니다. **(기능 추가)**
- 장점
  - 기존 코드를 수정하지 않고도 데코레이터 패턴을 통해 행동을 확장시킬 수 있습니다.
  - 구성과 위임을 통해서 실행중에 새로운 행동을 추가할 수 있습니다.
- 단점
  - 의미없는 객체들이 너무 많이 추가될 수 있습니다.
  - 데코레이터를 너무 많이 사용하면 코드가 필요 이상으로 복잡해질 수 있습니다.
- 예시

```
public interface Component {
    String add(); //재료 추가
}

public class BaseComponent implements Component {

    @Override
    public String add() {
        // TODO Auto-generated method stub
        return "에스프레소";
    }
}

abstract public class Decorator implements Component {
    private Component coffeeComponent;

    public Decorator(Component coffeeComponent) {
        this.coffeeComponent = coffeeComponent;
    }

    public String add() {
        return coffeeComponent.add();
    }
}

public class WaterDecorator extends Decorator {
    public WaterDecorator(Component coffeeComponent) {
        super(coffeeComponent);
    }

    @Override
    public String add() {
        // TODO Auto-generated method stub
        return super.add() + " + 물";
    }
}

public class MilkDecorator extends Decorator {
    public MilkDecorator(Component coffeeComponent) {
        super(coffeeComponent);
    }

    @Override
    public String add() {
        // TODO Auto-generated method stub
```

```
        return super.add() + " + 우유";
    }
}

public class Main {

    public static void main(String[] args) {
        Component espresso = new BaseComponent();
        System.out.println("에스프레소 : " + espresso.add());

        Component americano = new WaterDecorator(new BaseComponent());
        System.out.println("아메리카노 : " + americano.add());

        Component latte = new MilkDecorator(new WaterDecorator(new BaseComponent())
        System.out.println("라떼 : " + latte.add());
    }
}
```

▼ 프록시 패턴

- 다른 객체에 대한 대체 또는 자리표시자를 제공할 수 있는 구조 디자인 패턴입니다.
  프록시는 원래 객체에 대한 접근을 제어하므로, 당신의 요청이 원래 객체에 전달되기 전 또는 후에 무언가를 수행할 수 있도록 합니다.
  프록시는 일반적으로 자체적으로 자신의 서비스 객체의 수명 주기를 관리.
  (접근 제어가 목적)
- 장점
  - 사이즈가 큰 객체(ex : 이미지)가 로딩되기 전에도 프록시를 통해 참조를 할 수 있다.
  - 실제 객체의 public, protected 메소드들을 숨기고 인터페이스를 통해 노출시킬 수 있다.
  - 로컬에 있지 않고 떨어져 있는 객체를 사용할 수 있다.
  - 원래 객체의 접근에 대해서 사전처리를 할 수 있다.
- 단점
  - 객체를 생성할때 한단계를 거치게 되므로, 빈번한 객체 생성이 필요한 경우 성능이 저하될 수 있다.
  - 프록시 내부에서 객체 생성을 위해 스레드가 생성, 동기화가 구현되야 하는 경우 성능이 저하될 수 있다.
  - 로직이 난해해져 가독성이 떨어질 수 있다.
- 예제

```
public interface Image {
   void displayImage();
}

public class Real_Image implements Image {

    private String fileName;

    public Real_Image(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    private void loadFromDisk(String fileName) {
                try {
                        Thread.sleep(1000);
                }
                catch(Exception e) {
```

```
                }
        System.out.println("Loading " + fileName);
    }

    @Override
    public void displayImage() {
        System.out.println("Displaying " + fileName);
    }
}

public class Proxy_Image implements Image {
    private Real_Image realImage;
    private String fileName;

    public Proxy_Image(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void displayImage() {
        if (realImage == null) {
            realImage = new Real_Image(fileName);
        }
        realImage.displayImage();
    }
}

public class Proxy_Main {
    public static void main(String[] args) {
        Image image1 = new Proxy_Image("test1.png");
        Image image2 = new Proxy_Image("test2.png");

        image1.displayImage();
        System.out.println();
        image2.displayImage();
    }
}
```

▼ 디자인 패턴 [행동]

▼ 템플릿 메소드 패턴

- 템플릿 메서드는 **부모 클래스에서 알고리즘의 골격을 정의**하지만, 해당 알고리즘의 구조를 변경하지 않고 **자식 클래스들이 알고리즘의 특정 단계들을 오버라이드(재정의)**할 수 있도록 하는 행동 디자인 패턴입니다.
- 장점

  1. 중복코드를 줄일 수 있다.

  2. 자식 클래스의 역할을 줄여 핵심 로직의 관리가 용이하다.

  3. 좀더 코드를 객체지향적으로 구성할 수 있다.
- 단점

  1. 추상 메소드가 많아지면서 클래스 관리가 복잡해진다.

  2. 클래스간의 관계와 코드가 꼬여버릴 염려가 있다.
- 예시

```
//추상 클래스
abstract class Teacher{
    public void start_class() {
        inside();
```

```
        attendance();
        teach();
        outside();
    }
    public void inside() {
        System.out.println("선생님이 강의실로 들어옵니다.");
    }

    public void attendance() {
        System.out.println("선생님이 출석을 부릅니다.");
    }

    public void outside() {
        System.out.println("선생님이 강의실을 나갑니다.");
    }

    // 추상 메서드
    abstract void teach();
}

class Korean_Teacher extends Teacher{
    @Override
    public void teach() {
        System.out.println("선생님이 국어를 수업합니다.");
    }
}

class Math_Teacher extends Teacher{

    @Override
    public void teach() {
        System.out.println("선생님이 수학을 수업합니다.");
    }
}

class English_Teacher extends Teacher{

    @Override
    public void teach() {
        System.out.println("선생님이 영어를 수업합니다.");
    }
}

public class Main {
    public static void main(String[] args) {
        Korean_Teacher kr = new Korean_Teacher(); //국어
        Math_Teacher mt = new Math_Teacher(); //수학
        English_Teacher en = new English_Teacher(); //영어

        kr.start_class();
        System.out.println("----------------------------");
        mt.start_class();
        System.out.println("----------------------------");
        en.start_class();
    }
}
```

▼ 옵저버 패턴

- 옵서버 패턴(observer pattern)은 객체의 상태 변화를 관찰하는 관찰자들, 즉 옵저버들의 목록을 객체에 등록하여 상태 변화가 있을 때마다 메서드 등을 통해 객체가 직접 목록의 각 옵저버에게 통지하도록 하는 디자인 패턴이다. 주로 분산 이벤트 핸들링 시스템을 구현하는 데 사용된다. **발행/구독** 모델로 알려져 있기도 하다.
- 장점
  - Subject의 상태 변경을 주기적으로 조회하지 않고 자동으로 감지할 수 있다.
  - 발행자의 코드를 변경하지 않고도 새 구독자 클래스를 도입할 수 있어 개방 폐쇄 원칙(OCP) 준수한다
  - 런타임 시점에서에 발행자와 구독 알림 관계를 맺을 수 있다.
  - 상태를 변경하는 객체(Subject)와 변경을 감지하는 객체(Observer)의 관계를 느슨하게 유지할 수 있다. (느슨한 결합)
- 단점
  - 구독자는 알림 순서를 제어할수 없고, 무작위 순서로 알림을 받음
  - 옵저버 패턴을 자주 구성하면 구조와 동작을 알아보기 힘들어져 코드 복잡도가 증가한다.
  - 다수의 옵저버 객체를 등록 이후 해지하지 않는다면 메모리 누수가 발생할 수도 있다.
- 예시

```
// 관찰 대상자 / 발행자
interface ISubject {
    void registerObserver(IObserver o);
    void removeObserver(IObserver o);
    void notifyObserver();
}

class ConcreteSubject implements ISubject {
    // 관찰자들을 등록하여 담는 리스트
    List<IObserver> observers = new ArrayList<>();

    // 관찰자를 리스트에 등록
    @Override
    public void registerObserver(IObserver o) {
        observers.add(o);
        System.out.println(o + " 구독 완료");
    }

    // 관찰자를 리스트에 제거
    @Override
    public void removeObserver(IObserver o) {
        observers.remove(o);
        System.out.println(o + " 구독 취소");
    }

    // 관찰자에게 이벤트 송신
    @Override
    public void notifyObserver() {
        for(IObserver o : observers) { // 관찰자 리스트를 순회하며
            o.update(); // 위임
        }
    }
}
```

```
// 관찰자 / 구독자
interface IObserver {
    void update();
}

class ObserverA implements IObserver {
```

```
    public void update() {
        System.out.println("ObserverA 한테 이벤트 알림이 왔습니다.");
    }

    public String toString() { return "ObserverA"; }
}

class ObserverB implements IObserver {
    public void update() {
        System.out.println("ObserverB 한테 이벤트 알림이 왔습니다.");
    }

    public String toString() { return "ObserverB"; }
}
```

```
public class Client {
    public static void main(String[] args) {

        // 발행자 등록
        ISubject publisher = new ConcreteSubject();

        // 발행자를 구독할 관찰자들 리스트로 등록
        IObserver o1 = new ObserverA();
        IObserver o2 = new ObserverB();
        publisher.registerObserver(o1);
        publisher.registerObserver(o2);

        // 관찰자에게 이벤트 전파
        publisher.notifyObserver();

        // ObserverB가 구독 취소
        publisher.removeObserver(o2);

        // ObserverA 한테만 이벤트 전파
        publisher.notifyObserver();
    }
}
```

▼ 전략 패턴

- 전략 패턴은 **실행(런타임) 중에 알고리즘 전략을 선택하여 객체 동작을 실시간 변경** 할 수 있게 하는 행위 디자인 패턴 이다.
- 어떤 일을 수행하는 알고리즘이 여러가지 일때, 동작들을 미리 전략으로 정의함으로써 손쉽게 전략을 교체할 수 있는, 알고리즘 변형이 빈번하게 필요한 경우에 적합한 패턴
- 장점
  - 런타임에 한 객체 내부에서 사용되는 알고리즘들을 교환할 수 있습니다.
  - 알고리즘을 사용하는 코드에서 알고리즘의 구현 세부 정보들을 고립할 수 있습니다.
  - 상속을 합성으로 대체할 수 있습니다.
  - 개방/폐쇄 원칙. 콘텍스트를 변경하지 않고도 새로운 전략들을 도입할 수 있습니다.
- 단점
  - 알고리즘이 몇 개밖에 되지 않고 거의 변하지 않는다면, 패턴과 함께 사용되는 새로운 클래스들과 인터페이스들로 프로그램을 지나치게 복잡하게 만들 이유가 없습니다.
  - 클라이언트들은 적절한 전략을 선택할 수 있도록 전략 간의 차이점들을 알고 있어야 합니다.
- 예시

```
public interface MovableStrategy {
    public void move();
}
public class RailLoadStrategy implements MovableStrategy{
    public void move(){
        System.out.println("선로를 통해 이동");
    }
}
public class LoadStrategy implements MovableStrategy{
    public void move() {
        System.out.println("도로를 통해 이동");
    }
}

public abstract class Moving {
    private MovableStrategy movableStrategy;

    public void move(){
        movableStrategy.move();
    }

    public void setMovableStrategy(MovableStrategy movableStrategy){
        this.movableStrategy = movableStrategy;
    }
}
public class Bus extends Moving{

}
public class Train extends Moving{

}

public class Client {
    public static void main(String args[]){
        Moving train = new Train();
        Moving bus = new Bus();

        train.setMovableStrategy(new RailLoadStrategy());
        bus.setMovableStrategy(new LoadStrategy());

        train.move();
        bus.move();

        /*
            선로를 따라 움직이는 버스가 개발
         */
        bus.setMovableStrategy(new RailLoadStrategy());
        bus.move();
    }
}
```

▼ 버블 정렬/ 퀵 정렬

- 버블 정렬
  - 정의
    - 두 개의 인접한 원소를 비교하여 정렬하는 방식
  - 동작 순서

1. 앞에서부터 현재 원소와 바로 다음의 원소를 비교한다.
2. 현재 원소가 다음 원소보다 크면 원소를 교환한다.
3. 다음 원소로 이동하여 해당 원소와 그 다음원소를 비교한다.

- 시간 복잡도
  - 사이즈가 n 일 때, n - 1, n - 2 ... 3, 2, 1
  - 즉 n (n - 1) / 2 로 $O(n^2)$ 이다.

▼ 코드 (st-lab.tistory.com)

```
public class Bubble_Sort {

    public static void bubble_sort(int[] a) {
        bubble_sort(a, a.length);
    }

    private static void bubble_sort(int[] a, int size) {

        // round는 배열 크기 - 1 만큼 진행됨
        for(int i = 1; i < size; i++) {

            boolean swapped = false;

            // 각 라운드별 비교횟수는 배열 크기의 현재 라운드를 뺀 만큼 비교함
            for(int j = 0; j < size - i; j++) {

                /*
                 *  현재 원소가 다음 원소보다 클 경우
                 *  서로 원소의 위치를 교환하고
                 *  비교수행을 했다는 표시로 swapped 변수를 true로 변경한다.
                 */
                if(a[j] > a [j + 1]) {
                    swap(a, j, j + 1);
                    swapped = true;
                }
            }

            /*
             * 만약 swap된적이 없다면 이미 정렬되었다는 의미이므로
             * 반복문을 종료한다.
             */
            if(swapped == false) {
                break;
            }
        }
    }

    private static void swap(int[] a, int i, int j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

- 퀵 정렬
  - 정의
    - 하나의 리스트를 **피벗(pivot)**을 기준으로 **두 개의 부분리스트**로 나누어 하나는 **피벗보다 작은 값들의 부분리스트,** 다른 하나는 **피벗보다 큰 값들의 부분리스트**로 정렬한 다음**, 각 부분리스트에 대해 다시 위 처럼 재귀적으로 수행**하여 정렬하

는 방법

- 동작 순서

  1. 피벗을 하나 선택한다.

  2. 피벗을 기준으로 양쪽에서 피벗보다 큰 값, 혹은 작은 값을 찾는다. 왼쪽에서부터는 피벗보다 큰 값을 찾고, 오른쪽에서부터는 피벗보다 작은 값을 찾는다.

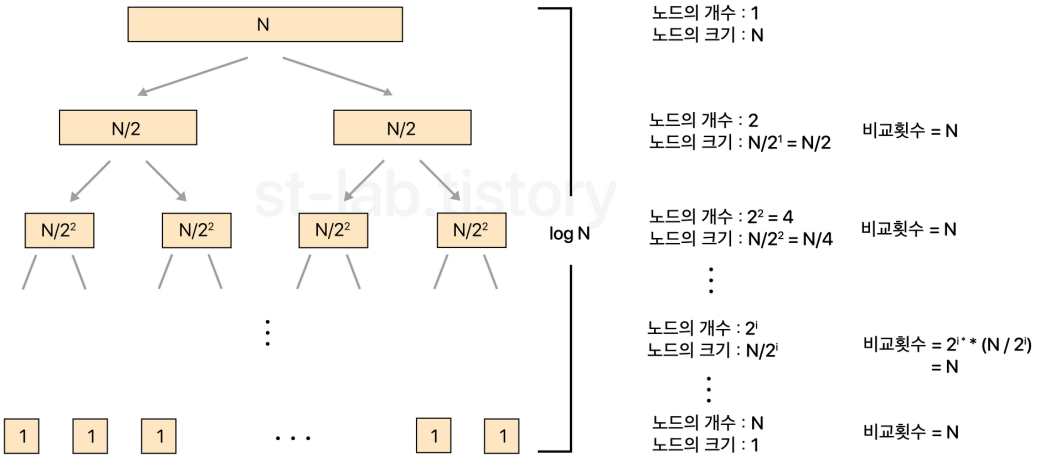
  3. 양 방향에서 찾은 두 원소를 교환한다.

  4. 왼쪽에서 탐색하는 위치와 오른쪽에서 탐색하는 위치가 엇갈리지 않을 때 까지 2번으로 돌아가 위 과정을 반복한다.

  5. 엇갈린 기점을 기준으로 두 개의 부분리스트로 나누어 1번으로 돌아가 해당 부분리스트의 길이가 1이 아닐 때 까지 1번 과정을 반복한다. (Divide : 분할)

  6. 인접한 부분리스트끼리 합친다. (Conqure : 정복)

- 시간 복잡도



  - O(nlogn)

▼ 코드 (st-lab.tistory.com)

```
public class QuickSort {

    public static void sort(int[] a) {
        l_pivot_sort(a, 0, a.length - 1);
    }

    /**
     *  왼쪽 피벗 선택 방식
     * @param a     정렬할 배열
     * @param lo    현재 부분배열의 왼쪽
     * @param hi    현재 부분배열의 오른쪽
     */
    private static void l_pivot_sort(int[] a, int lo, int hi) {

        /*
         *  lo가 hi보다 크거나 같다면 정렬 할 원소가
         *  1개 이하이므로 정렬하지 않고 return한다.
         */
        if(lo >= hi) {
            return;
        }

        /*
         * 피벗을 기준으로 요소들이 왼쪽과 오른쪽으로 약하게 정렬 된 상태로
```

```
     * 만들어 준 뒤, 최종적으로 pivot의 위치를 얻는다.
     *
     * 그리고나서 해당 피벗을 기준으로 왼쪽 부분리스트와 오른쪽 부분리스트로 나누어
     * 분할 정복을 해준다.
     *
     * [과정]
     *
     * Partitioning:
     *
     *   a[left]          left part              right part
     * +-------------------------------------------------------+
     * |  pivot  |    element <= pivot    |    element > pivot   |
     * +-------------------------------------------------------+
     *
     *
     *  result After Partitioning:
     *
     *         left part          a[lo]          right part
     * +-------------------------------------------------------+
     * |   element <= pivot    |  pivot  |    element > pivot    |
     * +-------------------------------------------------------+
     *
     *
     *  result : pivot = lo
     *
     *
     *  Recursion:
     *
     * l_pivot_sort(a, lo, pivot - 1)     l_pivot_sort(a, pivot + 1, hi)
     *
     *         left part                          right part
     * +-----------------------+             +-----------------------+
     * |   element <= pivot    |    pivot    |    element > pivot    |
     * +-----------------------+             +-----------------------+
     * lo                pivot - 1        pivot + 1                hi
     *
     */
    int pivot = partition(a, lo, hi);

    l_pivot_sort(a, lo, pivot - 1);
    l_pivot_sort(a, pivot + 1, hi);
}



/**
 * pivot을 기준으로 파티션을 나누기 위한 약한 정렬 메소드
 *
 * @param a     정렬 할 배열
 * @param left  현재 배열의 가장 왼쪽 부분
 * @param right 현재 배열의 가장 오른쪽 부분
 * @return      최종적으로 위치한 피벗의 위치(lo)를 반환
 */
private static int partition(int[] a, int left, int right) {

    int lo = left;
    int hi = right;
    int pivot = a[left];        // 부분리스트의 왼쪽 요소를 피벗으로 설정
```

```
        // lo가 hi보다 작을 때 까지만 반복한다.
        while(lo < hi) {

            /*
             * hi가 lo보다 크면서, hi의 요소가 pivot보다 작거나 같은 원소를
             * 찾을 때 까지 hi를 감소시킨다.
             */
            while(a[hi] > pivot && lo < hi) {
                hi--;
            }

            /*
             * hi가 lo보다 크면서, lo의 요소가 pivot보다 큰 원소를
             * 찾을 때 까지 lo를 증가시킨다.
             */
            while(a[lo] <= pivot && lo < hi) {
                lo++;
            }

            // 교환 될 두 요소를 찾았으면 두 요소를 바꾼다.
            swap(a, lo, hi);
        }


        /*
         *  마지막으로 맨 처음 pivot으로 설정했던 위치(a[left])의 원소와
         *  lo가 가리키는 원소를 바꾼다.
         */
        swap(a, left, lo);

        // 두 요소가 교환되었다면 피벗이었던 요소는 lo에 위치하므로 lo를 반환한다.
        return lo;
    }

    private static void swap(int[] a, int i, int j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

▼ compiler vs interpreter

- 컴파일러
  - 전체 소스 코드를 한 번에 컴파일하여 기계어로 변환하는 프로그램
  - 컴파일러는 소스 코드를 분석하고, 문법적인 오류를 체크한 뒤, 기계어로 변환하는 과정을 거치고 변환된 기계어 코드는 컴퓨터에서 직접 실행됨
  - 컴파일러는 소스 코드를 한 번만 컴파일하고, 컴파일된 결과물을 실행하기 때문에 실행 속도가 빠르고, 실행 중에 추가적인 변환 과정이 필요하지 않습니다.
  - 대표적인 예시로는 C++, Java와 같은 언어가 있습니다.
- 인터프리터
  - 소스 코드를 한 줄씩 읽어 해석하고, 실행하는 프로그램
  - 소스 코드를 한 줄씩 해석하기 때문에 디버깅이 쉽고, 언어에 대한 동적인 기능이 잘 작동합니다.
  - 실행할 때마다 소스 코드를 해석하므로 컴파일러에 비해 실행 속도가 느릴 수 있습니다.
  - 대표적인 예시로는 Python, JavaScript와 같은 언어가 있습니다.

▼ 면접 예상 내용

- ETL
  - 왜 만듬?
    - 기존 DB 가 MySQL 로 되어 있었고 몇몇 일부 테이블은 일정 주기마다 업데이트만 되는 방식이였음. 그런데 사용자 요구 사항에 시간대 변경하여 조회가 가능해야 되었고, 몇몇 테이블은 Join 을 너무 많이 해야 하는 상황이 나와 ETL 에서 미리 Join 된 마트 테이블을 생성하거나, 경량화된 테이블로 변경할 수 있었습니다. 또한, 이노 개발팀에서 앞으로의 RDB 방향성은 PostgreSQL 이였기 때문에, ETL 시스템을 만들었습니다.
  - Node.js 로 만들었던데 왜? Spring batch, java 가 아니고?
    - 해당 ETL 시스템 작업은 하루 빨리 만들어야 다른 개발자들에게 화면이나, WAS 단에서 코드를 작성할 수 있었기 때문에, 러닝 타임이 있는 Spring bacth 로는 시간이 부족했습니다.
      Java는 컴파일 언어이고, Node.js는 인터프리터 언어임으로, 컴파일 시간을 줄이고 싶었습니다. 또한 Java 의 장점인 멀티 쓰레드 방식이 별로 필요 없다고 느껴졌습니다. 왜냐하면 일단 공유될 데이터가 없고, 테이블 update 주기가 5분, 1시간, 1일 이렇게 진행되고 1분, 30분 이런식으로도 추가될 가능성이 있다고 판단되어, 확장성에 용이하게 Node.js 를 사용하고 주기 단위로 node 를 여러 프로세스로 관리하면 된다고 판단했기 때문입니다.
- 왜 NHN PAYCO 을 오려고 하는가?
  - 저는 애플리케이션을 잘 구축된 환경을 경험해 성장하고 싶기 때문입니다. 성공된 애플리케이션 특징으로는 신뢰성, 확장성, 유지 보수성으로 볼 수 있습니다. 이에 신뢰성에 중요하다고 생각된 PAYCO 결제 시스템을 경험해보고 싶었으며, 결제 시스템뿐만 아니라 다양한 도메인을 경험해 백엔드 개발자로 더욱 성장하고 싶었기 때문입니다.
- 이전 회사에서 프론트 작업을 많이 한 것 같은데..? 백엔드가 되고 싶은 이유는?
  - 다양한 트래픽을 경험하면서 시스템을 안정적이고 효율적으로 만들어 보고 싶다는 생각이 들었기 때문입니다. 예를 들어서 **데이터 중심 애플리케이션 설계** 책에서 확인 했던 내용으로 트위터 사용자에 대한 응답 시간의 대한 목표를 갖고 해당 목표를 이루기 위해 팔로워 수에 따라 데이터를 불러오는 방식을 다르게 설계 한다는 것에 큰 영감을 받았습니다.
- JPA 를 사용한 적이 없는지?
  - 개인 사이드 프로젝트로 H2, JPA 를 사용한 적이 있긴 합니다만.. 크게 사용하는 법을 알지는 못합니다. 하지만, 이전 회사에서는 통신 도메인으로 SQL에서 Join 쿼리가 많아 JPA 보다는 MyBatis 가 더 효율적이 였었고, 직접 DB를 관리 할 수 있어서 RDB 에 많은 역량을 키울 수 있었습니다. (limit/offset, index, partition, Explain analysis Query) JPA 도 DB 지식이 많아야 이해가 빨라져 습득력이 높을 것으로 예상되어 업무를 진행하는데 큰 차질은 없을 것입니다.

▼ 지원 회사

- 젤 가고 싶은 곳
  - 네이버 파이낸셜 인재 pool 등록
- 가고 싶은 곳 (결제 + B2C + 백엔드)
  - ~~NHN PAYCO Java 개발자 - 서탈~~
  - ~~NHN PAYCO 플랫폼 개발자 - 서탈~~
  - 카카오 페이 플랫폼 개발자
  - 카카오 페이 증권 서버 개발자
  - 야놀자 백엔드 플랫폼 개발자
- 2차 (B2C + 백엔드)
  - 패스오더 백엔드 개발자