**1. Java**

**- OOP 개념**  
프로그램을, 메소드와 변수(데이터)로 구성된 객체라는 기본 단위로 나누고, 이 객체들의 상호작용으로 서술하는 방식

**- JVM 동작 방식**  
Java Source(.java)를 Java Compiler(javac)가 Java Byte Code(.class)로 컴파일하면,  
바이트코드가 JVM의 Class Loader에 전달  
클래스로더는 동적로딩(Dynamic Loading)을 통해 필요한 클래스들을 로딩 및 링크하여,  
런타임 데이터 영역(Runtime Data Area) 즉, JVM의 메모리에 올린다.  
실행엔진(Execution Engine)이 JVM 메모리에 올라온 바이트코드를  
명령어 단위로 하나씩 가져와서 실행

**- JVM Memory  
1) Method Area**: 클래스 멤버 변수의 이름, 데이터 타입, 접근 제어자 정보 같은 필드 정보와 메소드의 이름, 리턴 타입, 파라미터, 접근 제어자 정보 같은 메소드 정보, type 정보(class, interface), constant pool(상수 풀: 문자 상수, 타입, 필드, 객체 참조), static 변수, final class 변수 등이 생성  
**2) Heap Area**: new 키워드로 생성된 객체와 배열이 생성되는 영역. 메소드 영역에 로드된 클래스만 생성이 가능하고, GC에 의해 참조되지 않는 메모리는 확인, 제거된다.  
**3) Stack Area**: 지역변수, 파라미터, 리턴값, 연산에 사용되는 임싯값 등이 생성되는 영역.  
int a = 10;의 경우, stack에 값이 10인 메모리 공간을 만들고, 이름을 a라고 붙여준다.  
Person p = new Person();의 경우, Person p는 stack에 생성되고, new로 생성된 Person 클래스의 인스턴스는 heap에 생성되는데, stack에 생성된 p의 값이 heap에 위치한 인스턴스의 주솟값이다. 즉, stack에 생성된 p가 heap에 생성된 객체를 참조하는 것  
**4) PC Register**: 스레드가 생성될 때마다 생성되는 영역으로, program counter 즉, 현재 스레드가 실행되는 부분의 주소와 명령을 저장. 이것을 이용해서 스레드가 돌아가면서 수행을 할 수 있다.  
**5) Native Method Stack**: Java 외 언어로 작성된 네이티브 코드를 위한 메모리 영역. 보통 C/C++ 등의 코드를 수행하기 위한 stack이다.(JNI)

**- Garbage Collection 매커니즘**JVM 메모리는 크게 Heap Area / Method Area / JVM Stack / Native Method Stack 네 영역으로 나눠진다. (+ PC Register)  
**가비지 콜렉터는 힙 영역을 다루게 된다.**  
JVM Heap은 Young, Old, Perm 세 영역으로 나뉜다.  
Perm(Permanent) 영역은 거의 사용되지 않는 영역으로, 클래스, 메소드 등의 수준, 자바 언어 레벨에서 사용되지 않는다.  
Young 영역은 Eden 영역과 2개의 Survivor 영역으로 나뉜다.  
따라서, 우리가 고려해야 할 Java의 메모리 영역은 총 4개(Young(Eden, Survivor, Survivor), Old)라고 할 수 있다.  
**Young Generation 영역**: 새롭게 생성된 객체가 위치한다. 많은 객체가 이 영역에 생성되었다 사라지며, 그 과정을 Minor GC라고 한다. New 키워드를 통해 생성된 객체는 우선 Eden 영역에 위치하게 된다. Eden 영역에서 한 번의 GC 후 살아남은 객체는 2개의 Survivor 영역 중 하나로 이동한다. 이 때, Eden에서 Survivor로 이동하는 과정을 Minor GC라고 한다. 이 Minor GC가 반복되며 Survivor 영역이 가득 차게 되면, Old 영역으로의 이동이 시작된다.  
(2개의 Survivor 영역에 모두 데이터가 존재하거나, 모두 사용량이 0이면 비정상 상태이다.)  
**Old Generation 영역**: Unreachable 상태가 되지 않아 young 영역에서 살아남은 객체가 이 영역으로 복사된다. Young 영역보다 크게 할당되며 GC는 적게 발생한다. 그 과정을 Major GC 또는 Full GC라고 한다.  
**Mark**: GC가 스택의 모든 변수 또는 Reachable 객체를 스캔하면서, 각각 어떤 객체를 참조하고 있는지 찾는 과정  
**Sweep**: Mark 되어 있지 않은 객체들을 힙에서 제거하는 과정  
**Stop-the-world**: GC가 실행되기 전, JVM이 애플리케이션의 실행을 멈추는 작업. GC를 처리하는 스레드 외의 스레드는 모두 작업을 중단하기 때문에, stop-the-world를 줄이는 작업을 GC 튜닝이라고 한다.  
**1) Serial GC**: Mark-sweep-compact 알고리즘이 사용된다. Old 영역에서 살아 있는 객체를 mark 하고 heap의 앞부분부터 살아있는 객체를 sweep 한 후, heap의 앞부분부터 객체를 쌓는다. (Compact) CPU 코어수가 적고, 메모리가 작을 때 좋다  
**2) Parallel GC**: Serial GC와 기본적인 알고리즘(Mark-sweep-compact)은 같지만, GC를 처리하는 스레드의 개수가 많다. CPU 코어수가 많을수록, 메모리가 클수록 좋다.  
**3) Parallel Old GC**: Parallel GC와 같지만, Old 영역의 GC 알고리즘만 다르다. Mark-summary-compact 알고리즘이며 Mark-sweep-compact에 비해 조금 더 복잡하다.  
**4) CMS GC**: Stop-the-world 이후, initial marking 시 살아있는 객체만 찾는다. 이후 concurrent mark 단계에서 참조를 따라가며 새로 추가되거나 참조가 끊긴 객체들을 remark 한다. 모든 작업이 멀티 스레드 환경에서 동시에 진행되기 때문에 stop-the-world 시간이 매우 짧은 대신, 메모리와 CPU 자원을 많이 사용하고, compaction 단계가 생략된다.  
**5) GI GC**: young 영역과 old 영역 구분 없이 매우 빠르게 객체를 할당하고 GC한다.

*GC 튜닝이 모든 자바 애플리케이션에 필수적인 것은 아니지만, 크리티컬한 요청을 담당하는 서버나 코어 엔진의 경우 GC 튜닝이 필요하다. GC 튜닝을 위해 지켜야 할 기본적인 원칙은 GC 튜닝이 로직에 영향을 미치지 않도록 가능한 늦게 수행하도록 하고, 객체 생성을 최소화하는 것이다.  
즉, GC 튜닝은 Java 코드의 최적화와 맞물려 있는 영역이라고도 할 수 있다. 가령 String의 append 시, + 연산으로 2개 이상의 String을 더하는 대신, StringBuilder 등을 사용하는 것도 일종의 튜닝이라고 할 수 있다. 그 외에는 설정적인 부분으로 JVM 옵션으로 메모리 크기를 조절하고 GC 방식을 지정하는 방법 등이 있다. 그러나, GC 튜닝이 비약적인 시스템 성능 향상을 가져오리라 보장할 수는 없다. 결국 GC 튜닝은 지속적인 모니터링을 통해 이루어지는 것으로, 복잡한 시스템의 메모리 구조를 공식처럼 맞춰서 설정하기보다는 지속적인 모니터링과 성능향상을 위한 노력이 필요하다.*

**- Vector vs ArrayList vs LinkedList  
1) Vector**: 동기화를 보장. 공유 자원이나 복수의 사용자가 존재할 때 좀더 안전하게 프로그램 작성 가능. 하나의 스레드가 하나의 자원을 이용하는 경우에는 오히려 성능 저하. 공간이 부족할 때, 부족분의 2배만큼 확보하기 때문에 메모리를 많이 잡아먹는다.  
**2) ArrayList**: 인덱스가 있어서 검색에 용이하나, 삽입/삭제 시 shifting이 발생하기 때문에 삽입/삭제가 빈번한 데이터에는 부적합. 동기화를 보장하지 않는다. 공간이 부족할 때, 부족한 만큼 확보.  
**3) LinkedList**: 노드가 서로 연결된 구조. 검색에 적합하지 않음. 삽입/삭제가 빈번한 데이터에 적합

**- String vs StringBuilder vs StringBuffer**String은 짧은 문자열을 더할 경우  
StringBuilder는 스레드에 안전한지 여부가 중요하지 않은 프로그램을 개발할 때 사용하면 좋음  
StringBuffer는 스레드에 안전한 프로그램이 필요할 때나, 개발 중인 시스템의 부분이 스레드에 안전한지 모를 경우 **1) String**: 한 번 생성되면, 할당된 메모리 공간이 변하지 않는다. + 연산자 또는 concat 메소드를 통해 기존에 생성된 String 클래스 객체 문자열에 다른 문자열을 붙여도 기존 문자열에 새로운 문자열을 붙이는 것이 아니라, 새로운 String 객체를 만든 후, 새 String 객체에 연결된 문자열을 저장하고, 그 객체를 참조하도록 함  
**2) StringBuilder**: 동기화를 지원하지 않음. 기존의 버퍼 크기를 늘리며 유연하게 동작  
**3) StringBuffer**: 동기화를 지원. 기존의 버퍼 크기를 늘리며 유연하게 동작  
JDK 1.5 이후 버전에서는 String 클래스를 사용해도, StringBuilder와 성능상 차이가 없어졌으나, 반복 루프를 사용해서 문자열을 더할 때 객체가 계속 추가된다는 사실은 변함이 없기 때문에, String 클래스를 쓰는 대신, 스레드 사용 여부에 따라서 StringBuilder와 StringBuffer를 구분하여 사용하면 된다.  
연산이 많을 경우, 단순 성능은 StringBuilder > StringBuffer > String

**- final 키워드**  
1) 상수라고 불린다. 변수를 선언함과 동시에 초기화하며, 이후에 값을 수정할 수 없다. get만 가능  
프로그램 전체에 걸쳐 사용되는 경우가 많아서, 특정 메소드 내부에서 선언하기보다는  
클래스에 static 키워드와 함께 사용한다.  
2) 메소드에 사용할 경우, Overriding이 불가능하다.  
3) 클래스에 사용할 경우, 상속이 불가능하며, subclass를 만들 수 없다.

**- static 키워드**  
1) 클래스를 설계할 때, 멤버변수 중 모든 인스턴스에 공통적으로 사용해야 하는 것에는 static을 붙인다.  
2) static이 붙은 멤버변수는 인스턴스를 생성하지 않아도 사용할 수 있다. Static이 붙은 멤버변수는 클래스가 메모리에 올라갈 때 이미 자동적으로 생성되기 때문  
3) static이 붙은 메소드에서는 인스턴스 변수를 사용할 수 없다. Static이 붙은 메소드는 인스턴스 생성 없이 호출이 가능한데, 인스턴스 변수는 인스턴스가 생성되었을 때만 존재하기 때문에, static 메소드가 호출될 때 인스턴스가 생성되어 있을지 알 수 없기 때문에, static 메소드에서의 인스턴스 변수 사용은 허용하지 않는다. (반대로, 인스턴스 변수나 인스턴스 메소드에서는 static 멤버를 언제든지 사용할 수 있다. 인스턴스 변수가 존재한다는 것은, static 멤버가 이미 메모리에 존재한다는 것을 의미하기 때문)  
4) 메소드 내에 인스턴스 변수를 사용하지 않는다면, static을 붙이는 것을 고려한다. 메소드의 호출시간이 짧아지기 때문이다. Non-static 메소드는 실행 시 호출되어야 할 메소드를 찾는 과정이 추가적으로 필요하기 때문에 시간이 더 걸린다.

**- 접근제어자(default, public, private, protected)  
1) default**: 접근제어자를 별도로 설정하지 않는다면, 해당 패키지 내에서만 접근이 가능하다.  
**2) public**: 어떤 클래스에서든지 접근이 가능하다.  
**3) private**: 해당 클래스에서만 접근이 가능하다.  
**4) protected**: 동일 패키지 내의 클래스, 해당 클래스를 상속 받은 클래스(외부 패키지도 가능)

**- Overloading vs Overriding**Java가 다형성을 확보하는 방법  
1) Overloading: 같은 이름으로 파라미터를 다르게 해서 받음  
2) Overriding: 하위 클래스에서 재정의해서 사용. 파라미터, 리턴 타입 같음

**- Interface vs Abstract Class**  
자바가 상속성과 다형성을 보장하는 방법  
추상클래스: 상속 받아 기능 확장  
인터페이스: 상속 받은 모든 클래스에 특정한 메소드가 반드시 존재하도록 강제

**- Call-by-reference vs Call-by-value  
1) Call-by-reference**  
장점: 복사하지 않고 직접 참조하기 때문에 빠르다.  
단점: 직접 참조하기 때문에 원래 값이 영향을 받는다는 위험성이 있다.  
**2) Call-by-value**  
장점: 복사하여 처리하기 때문에 안전하다. 원래 값이 보존된다.  
단점: 복사를 하기 때문에 메모리 사용량이 늘어난다.

**- Java 8버전의 Functional Interface, Stream, Lambda Expression  
Functional Interface**: 1개의 추상 메소드를 갖고 있는 인터페이스  
SAM(Single Abstract Method)이라고 불리기도 함  
Lambda Expression이 함수형 인터페이스로만 접근이 가능하기 때문에 사용함  
함수형 인터페이스는, Runnable, Consumer, Supplier, Function, Predicate, UnaryOperator, BinaryOperator, BiPredicate, BiConsumer, BiFunction, Comparator 등이 있다.  
**Stream**: Java 8에서 추가된, 람다를 활용할 수 있는 기술 중 하나. 컬렉션, 배열 등의 저장 요소를 하나씩 참조하고, 람다식을 적용하며 반복적으로 처리할 수 있도록 해주는 기능

**2. DB**

**- Transaction 이란? + ACID 개념**  
**트랜잭션**: 데이터베이스의 상태를 변화시키기 위해서 수행하는 작업의 단위  
**작업의 단위**: 질의어 한 문장이 아니라, 많은 질의어 명령문들을 기준에 따라 정하는 것을 의미한다.  
**Atomicity(원자성)**: 트랜잭션이 DB에 모두 반영이 되든지, 전혀 반영이 되지 않든지  
**Consistency(일관성)**: 트랜잭션이 진행되는 동안에 DB가 변경되더라도, 업데이트된 트랜잭션이 진행되는 것이 아니라, 처음에 트랜잭션을 진행하기 위해 참조한 DB로 진행되어, 사용자가 일관성 있는 DB를 볼 수 있어야 함  
**Isolation(독립성)**: 둘 이상의 트랜잭션이 동시에 실행되고 있을 경우, 어떤 하나의 트랜잭션이라도, 다른 트랜잭션의 연산에 끼어들 수 있어서는 안 됨  
**Durability(지속성)**: 트랜잭션이 성공적으로 완료된 경우, 결과는 영구적으로 반영되어야 함  
**Commit**: 하나의 트랜잭션이 성공적으로 끝났다는 것을 알려주는 연산  
**Rollback**: 하나의 트랜잭션 처리가 비정상적으로 종료되어 트랜잭션의 원자성이 훼손된 경우, 트랜잭션을 처음부터 다시 시작하거나, 해당 트랜잭션의 연산된 결과를 부분적으로 취소시킴

**- Query 튜닝(성능 향상시키는 법)  
1. 필요한 column만 명시한다.**  
 SELECT \*를 사용하는 것은 피한다.  
 사용하지 않는 데이터를 호출하는 것만으로도 이미 부하가 크다.  
 특히 text 타입의 데이터를 호출하는 경우는 더욱 심각하다.  
 data type의 byte가 적은 열을 주로 사용하자  
**2. COUNT(\*)을 사용한다**  
 COUNT(특정 column)으로 호출하는 경우가 있는데,  
 이 경우 해당 컬럼의 null값을 제외한 COUNT를 가져오게 된다.  
 null값을 일일이 체크하면 호출 속도가 저하되므로  
 꼭 null을 체크해야 하는 경우가 아닐 때는, COUNT(\*)을 하도록 하자.  
 COUNT(\*)은 null도 모두 count하지만, null 여부를 확인하지 않기 때문에  
 성능의 저하가 줄어든다.  
**3. List 호출이 아닌 1 row 호출을 하는 경우 TOP 1을 사용한다.**  
 성능상의 이유로 권장하는 것은 아니다.  
 WHERE 조건절이 동적으로 변하는 경우를 고려하여 작성하라는 것이다.  
 물론, query를 호출하기 전(前) 비즈니스 로직 단계에서  
 validation 체크를 충분히 해야겠지만,  
 호출하는 쪽과 DB 양쪽에서 빈틈이 없어야 한다.  
**4. WHERE 조건문의 왼쪽은 되도록 변형되지 않은 순수한 column만을 선언한다.**  
 WHERE NAME + ‘’ = ’조건’과 같이 하지 말고,  
 WHERE NAME = ‘’ + ‘조건’과 같이 오른쪽에 조건 선언을 하라.  
 조건 일치를 매 행마다 확인할 때 왼쪽 조건을 변형하게 되므로,  
 부하가 눈에 띄게 증가한다.  
**5. 커서 및 임시테이블의 사용을 최대한 자제한다.**  
 커서 < 임시 테이블 < 테이블 변수 순으로 성능이 좋다.  
 커서의 경우 내부적으로 임시 테이블을 사용하지만,  
 임시 테이블을 쓴다고 부하가 더 발생하는 것은 아니고,  
 오히려 커서의 부가기능 때문에 서버 자원이 더 낭비되게 된다.  
 커서로 처리할 수 있는 것은 모두 임시 테이블이나 테이블 변수로도 처리가 가능하므로  
 되도록 커서를 쓰지 않는 것이 좋다.  
**6. JOIN을 사용하는 경우에는 되도록 INNER JOIN을 사용한다.**  
 LEFT OUTER JOIN 보다 INNER JOIN이 빠르다  
7.

- DB Indexing

- RDB vs NoSQL

- Join이란? (Left/Right Join, Left/Right Outer Join)

**- DB 이중화(Database Replication)**  
시스템 오류로 인한 DB 서비스의 중단이나 물리적 손상 발생 시 이를 복구하기 위해 동일한 DB를 복제하여 관리  
**Eager 기법**: 트랜잭션 수행 중 데이터 변경이 발생하면 이중화 된 모든 데이터베이스에 즉시 전달하여 변경이 즉시 적용되도록 하는 기법  
**Lazy 기법**: 트랜잭션의 수행이 종료되면 변경 사실을 새로운 트랜잭션에 작성하여 각 DB에 전달하는 기법으로, DB마다 새로운 트랜잭션이 수행되는 것으로 간주됨  
**Active-active**: 2개의 DB가 서로 다른 서비스를 제공하다가 하나의 DB에 문제가 발생하면 나머지 DB가 이어서 서비스를 제공. 2개의 DB가 모두 처리하기 때문에 처리율이 높지만 구성 방법 및 설정이 복잡  
**Active-standby**: 하나의 DB가 활성 상태로 서비스 되고 있으면 다른 DB는 대기하고 있다가 활성 DB에 장애 발생 시 대기 중이던 DB가 자동으로 모든 서비스를 대신 수행. 구성 방법과 관리가 쉬워 많은 기업에서 이용

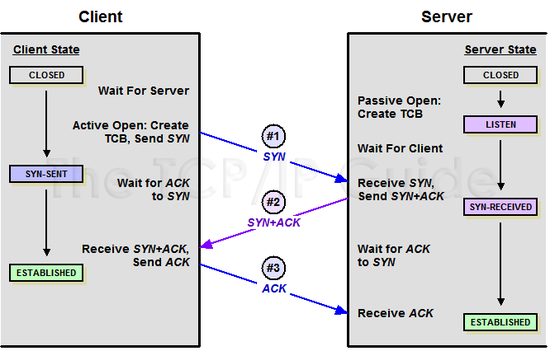
**- Function vs Procedure vs Trigger**  
**1) Function**  
 CREATE [OR REPLACE] FUNCTION [함수명]  
 리턴값이 있고, DB 객체로 저장되어 컴파일 된 상태로 수행된다.  
 단독적인 사용보다 SQL에서 많이 사용, 예외 처리 가능  
**2) Procedure**  
 CREATE [OR REPLACE] PROCEDURE [프로시저명]  
 빠르다, 파라미터 사용 가능, 여러 애플리케이션과 공유 가능  
 모듈식 프로그래밍 가능, 네트워크 전송량 감소(긴 쿼리의 단순화)  
**3) Trigger**  
 테이블, 뷰, 스키마 또는 PL/SQL를 블록으로 설정하여  
 특정 이벤트 발생 시 자동으로 해당 블록이 실행 되게 만드는 것

**3. Network**

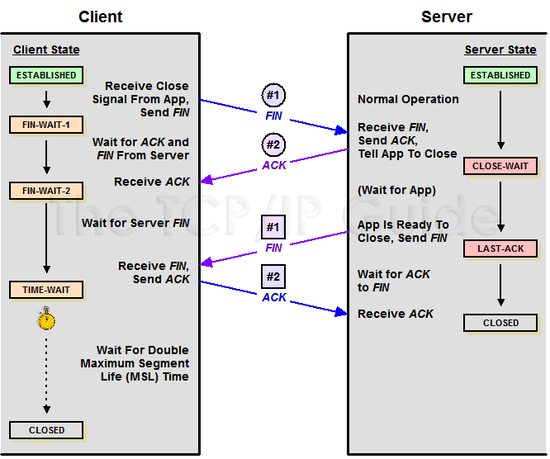
**- OSI 모델의 철학, 구조, Protocol**프로토콜을 기능별로 나눈 것이다.  
각 계층은 하위 계층의 기능만을 이용하고, 상위 계층에게 기능을 제공한다.  
‘프로토콜 스택’ 혹은 ‘스택’은 이러한 계층들로 구성되는 프로토콜 시스템이 구현된 시스템을 가리키는데, 프로토콜 스택은 하드웨어나 소프트웨어 혹은 둘의 혼합으로 구현될 수 있다. 일반적으로 하위 계층들은 하드웨어로, 상위 계층들은 소프트웨어로 구현됨.  
**1계층: 물리 계층**  
**2계층: 데이터 링크 계층  
3계층: 네트워크 계층** IP, ARP, RARP, ICMP  
**4계층: 전송 계층(Transport Layer)**  
 TCP, UDP  
**5계층: 세션 계층(Session Layer)  
6계층: 표현 계층(Presentation Layer)  
7계층: 응용 계층(Application Layer)**  
 telnet, FTP, DHCP, IFTP, HTTP, SMTP, DNS, SNMP

**- TCP vs UDP  
TCP(Transmission Control Protocol): 가상회선 패킷 교환 방식**  
 연결형 서비스로 가상 회선 방식을 제공한다  
 연결은 3-way handshaking   
 해제는 4-way handshaking  
 흐름 제어 및 혼잡 제어 가능  
 높은 신뢰성  
 UDP보다 속도가 느림  
 전이중(Full-duplex), 점대점(point-to-point) 방식  
 손실된 경우 재전송을 요청하므로 스트리밍 서비스에 불리  
 스트림 전송을 하므로 전송 데이터의 크기가 무제한  
**UDP(User Datagram Protocol): 데이터그램 패킷 교환 방식**  
 비연결형 서비스  
 UDP 헤더의 CheckSum 필드를 통해 최소한의 오류만 검출  
 신뢰성이 낮음  
 TCP보다 속도가 빠름  
 연결이라는 개념이 없어서 서버 소켓과 클라이언트 소켓의 구분이 없다.  
 소켓 대신 IP를 기반으로 데이터를 전송한다.  
 서버와 클라이언트는 1대1, 1대n, n대n 등으로 연결될 수 있다.  
 데이터그램(메시지) 단위로 전송되며, 크기는 65535 바이트로,  
 크기가 초과하면 잘라서 보낸다.  
 흐름제어(Flow Control)가 없어서, 패킷이 제대로 전송되었는지,  
 오류가 없는지 확인할 수 없다.  
 신뢰성보다 성능이 중요시 되는 경우 사용된다.

**- 3-way, 4-way Handshaking**  
**3-way Hand-shaking)**  
 Server(Listen)  
 Client(Closed -> SYN Sent) -> Server(Listen -> SYN Received): TCP SYN  
 Server -> Client(SYN Sent -> Established): TCP SYN+ACK  
 Client -> Server(SYN Received -> Established): TCP ACK



**4-way Handshaking)**



**- 브라우저에 URI 입력 시 발생하는 일**기본적으로 URL의 구조는 다음과 같다.



**1) URL 파싱**  
 프로토콜, URL, 포트를 해석  
**2) HSTS 목록 조회**  
 HSTS(HTTP Strict Transport Security) HTTP의 사용을 허용하지 않고, HTTPS의 사용만 허용  
 브라우저는 HSTS 목록 조회를 통해 해당 요청을 HTTPS로 보낼지 판단  
**3) URL -> IP주소 변환**  
 (1) 로컬 hosts 파일과 브라우저 캐시에 해당 URL이 존재하는지 확인  
 (2) (존재하지 않는 경우) DNS에 요청  
**4) 라우터를 통해 해당 서버의 게이트웨이까지 이동**  
 라우팅 테이블을 통해 경로 지정  
**5) ARP(Address Resolution Protocol)를 통해 IP주소를 MAC주소로 변환**  
**6) 대상 서버와 TCP 소켓 연결**  
 HTTP: 3-way Handshake  
 HTTPS: 3-way Handshake + TLS Handshake  
**7) HTTP(S)로 요청 및 응답**  
**8) 브라우저에서 응답을 해석**  
 HTML, CSS, Javascript 등

- Spring Framework 동작 원리

- JSP vs Servlet

- Apache vs Tomcat

- Bean 이란?

- Servlet Lifecycle

- IOC(Inversion of Control) / DI(Dependency Injection)

- Spring MVC vs SpringBoot

**4. OS**

- Process vs Thread

- Mutex vs Semaphore vs Monitor

- Critical Section, Race Condition

- Dead Lock

**5. Design Pattern**

- MVC vs MVP vs MVVM

- Factory Method Pattern