3주차 - 예상 질문 10가지

1. 순수 추상 클래스와 인터페이스의 차이

추상 클래스 (Abstract Class)

- **정의**: 하나 이상의 추상 메서드를 포함하는 클래스로, 직접 인스턴스화할 수 없음
- **구성**: 일반 메서드, 추상 메서드, 인스턴스 변수, 생성자를 모두 포함할 수 있음
- **상속**: 단일 상속만 가능 (extends 키워드 사용)
- **용도**: 관련된 클래스들 간의 공통 기능을 구현하고, 일부 기능은 하위 클래스에서 구현하도록 강제

인터페이스 (Interface)

- **정의**: 구현체 없이 메서드 시그니처만 정의한 계약서 역할
- **구성**: public static final 상수와 abstract 메서드만 포함 (Java 8 이후 default, static 메서드 지원)
- **구현**: 다중 구현 가능 (implements 키워드 사용)
- **용도**: 클래스가 어떤 메서드를 구현해야 하는지 명세를 정의

핵심 차이점

구분	추상 클래스	인터페이스
목적	공통 기능 구현 + 강제 구현	계약 정의
상속/구현	단일 상속	다중 구현
멤버	모든 종류 가능	상수, 추상 메서드 위주
접근 제한자	다양하게 사용	기본적으로 public
진화?	새 메서드 추가시 하위 클래스 영향	default 메서드로 하위 호환성 제공

언제 사용할까?

- **추상 클래스**: "is-a" 관계, 공통된 기능과 상태를 공유하는 경우
- **인터페이스**: "can-do" 관계, 서로 다른 클래스들이 같은 동작을 해야 하는 경우

---

2. IOC와 DI

IOC (Inversion of Control) - 제어의 역전

- **기본 개념**: 프로그램의 제어권이 개발자에게서 프레임워크로 넘어가는 것
- **전통적 방식**: 개발자가 직접 객체 생성, 생명주기 관리, 메서드 호출 시점 결정
- **IOC 방식**: 프레임워크가 객체 생성부터 소멸까지 전체 생명주기를 관리
- **Hollywood 원칙**: "Don't call us, we'll call you" - 개발자가 프레임워크를 호출하는 것이 아니라 프레임워크가 개발자 코드를 호출

DI (Dependency Injection) - 의존성 주입

- **핵심 개념**: 객체가 사용할 의존 객체를 직접 생성하지 않고 외부에서 주입받는 방식
- **IOC와의 관계**: DI는 IOC를 구현하는 구체적인 방법 중 하나
- **의존성이란**: 한 객체가 다른 객체를 사용(의존)하는 관계

DI의 3가지 방법

1. **생성자 주입** (Constructor Injection)
   - **장점**: 불변성 보장, 순환 참조 방지, 테스트 용이성
   - **권장 이유**: 의존성이 확실히 주입됨을 보장
2. **세터 주입** (Setter Injection)
   - **용도**: 선택적 의존성, 변경 가능한 의존성
   - **단점**: 객체 생성 후 의존성이 설정되기 전까지 불완전한 상태
3. **필드 주입** (Field Injection)
   - **편의성**: 코드가 간결함
   - **문제점**: 테스트하기 어렵고, 불변성을 보장할 수 없음

DI의 핵심 이점

- **결합도 감소**: 구체 클래스가 아닌 인터페이스에 의존
- **테스트 용이성**: Mock 객체를 쉽게 주입 가능
- **유연성**: 런타임에 다른 구현체로 교체 가능
- **단일 책임 원칙**: 객체 생성 책임과 비즈니스 로직 분리

---

3. VO, DTO, DAO

VO (Value Object) - 값 객체

- **본질**: 값 자체를 나타내는 객체, 식별자가 없고 값으로만 구분
- **불변성**: 생성 후 값이 변경되지 않음 (Immutable)
- **동등성**: 값이 같으면 같은 객체로 취급 (equals/hashCode 기반)
- **도메인 의미**: 비즈니스 도메인의 개념을 나타냄 (돈, 주소, 색상 등)
- **생명주기**: 값 자체가 중요하므로 생명주기가 단순함

DTO (Data Transfer Object) - 데이터 전송 객체

- **목적**: 계층 간 또는 시스템 간 데이터 전송을 위한 컨테이너
- **특징**: 로직 없이 오로지 데이터만 담는 단순한 객체
- **구성**: getter/setter 메서드와 필드로만 구성
- **직렬화**: 네트워크를 통한 전송을 위해 직렬화 가능해야 함
- **변환**: 도메인 객체와 DTO 간 변환 로직 필요
- **사용 예**: API 응답, 화면 전송 데이터, 시스템 간 통신

DAO (Data Access Object) - 데이터 접근 객체

- **역할**: 데이터베이스나 파일 시스템 등 영속성 계층에 대한 추상화
- **책임**: CRUD 연산과 쿼리 로직을 캡슐화
- **분리 효과**: 비즈니스 로직과 데이터 접근 로직의 명확한 분리
- **데이터베이스 독립성**: 구체적인 데이터베이스 기술을 숨김
- **패턴 적용**: Repository 패턴과 유사한 개념

세 객체의 관계와 데이터 흐름

- **Controller → DTO → Service → Domain(VO) → DAO → Database**
- 각 계층에서 적절한 객체로 변환하여 사용
- 계층 간 데이터 전달 시 불필요한 정보 노출 방지

---

4. 스프링 프레임워크의 생명 주기

스프링 컨테이너 생명주기

- **생성**: ApplicationContext 구현체 생성 및 설정 정보 로드
- **빈 등록**: @Component, @Bean 등으로 정의된 빈들을 컨테이너에 등록
- **의존 관계 설정**: 등록된 빈들 간의 의존 관계 주입
- **초기화**: 빈들의 초기화 작업 수행
- **사용**: 애플리케이션에서 빈 사용
- **소멸**: 컨테이너 종료 시 빈들의 정리 작업 및 컨테이너 해제

스프링 빈의 상세 생명주기

1. **스프링 빈 생성**: 스프링 컨테이너가 빈의 인스턴스를 생성
2. **의존관계 주입**: 필요한 의존 객체들을 주입
3. **초기화 콜백 실행**: 빈이 사용 준비가 완료된 후 실행
    - **@PostConstruct**: 어노테이션 기반 초기화
    - **InitializingBean**: afterPropertiesSet() 메서드 구현
    - **@Bean(initMethod)**: 초기화 메서드 직접 지정
4. **빈 사용**: 애플리케이션에서 실제 빈 사용
5. **소멸 전 콜백 실행**: 컨테이너 종료 전 정리 작업
    - **@PreDestroy**: 어노테이션 기반 소멸
    - **DisposableBean**: destroy() 메서드 구현
    - **@Bean(destroyMethod)**: 소멸 메서드 직접 지정

빈 스코프별 생명주기 차이

- **singleton** (기본): 컨테이너당 하나의 인스턴스, 컨테이너와 생명주기 동일
- **prototype**: 매번 새로운 인스턴스 생성, 생성 후 관리하지 않음
- **request/session**: HTTP 요청/세션 단위로 생명주기 관리
- **application**: 서블릿 컨텍스트 단위로 생명주기 관리

생명주기 관리의 중요성

- **자원 관리**: 데이터베이스 연결, 파일 핸들 등 적절한 해제
- **초기화**: 복잡한 설정이나 외부 연결이 필요한 경우

- **정리 작업**: 캐시 정리, 연결 종료 등 안전한 종료 보장

---

5. REST API

REST의 개념과 철학

- **REST**: REpresentational State Transfer - 웹의 기존 기술을 최대한 활용하는 아키텍처 스타일
- **철학**: 웹의 우수한 설계를 그대로 활용하여 분산 시스템을 구축하자
- **상태 전이**: 클라이언트가 서버의 자원 상태를 전이(변경)시키는 방식

REST 6원칙

1. **Client-Server 분리**: 사용자 인터페이스와 데이터 저장소의 분리로 독립적 진화 가능
2. **Stateless (무상태성)**: 각 요청은 독립적이며 이전 요청의 상태 정보를 저장하지 않음
3. **Cacheable (캐시 가능)**: 응답은 캐시 가능하거나 불가능함을 명시해야 함
4. **Uniform Interface (일관된 인터페이스)**: 자원 식별, 메시지를 통한 자원 조작 등
5. **Layered System (계층화)**: 클라이언트는 서버와 직접 연결되었는지 중간 서버와 연결되었는지 구분할 수 없어야 함
6. **Code on Demand (선택적)**: 서버가 클라이언트에 실행 가능한 코드를 전송할 수 있음

HTTP 메서드의 의미와 활용

- **GET**: 자원 조회 (Safe, Idempotent)
- **POST**: 자원 생성, 데이터 처리 (Non-Safe, Non-Idempotent)
- **PUT**: 자원 전체 수정, 없으면 생성 (Non-Safe, Idempotent)
- **PATCH**: 자원 부분 수정 (Non-Safe, Non-Idempotent)
- **DELETE**: 자원 삭제 (Non-Safe, Idempotent)

RESTful API 설계 핵심 원칙

- **자원(Resource) 중심**: URI는 자원을 나타내고 행위는 HTTP 메서드로 표현
- **명사 사용**: URI에는 동사가 아닌 명사 사용 (/users, /orders)
- **계층 구조**: 자원 간의 관계를 URI 경로로 표현 (/users/1/orders)
- **일관성**: 전체 API에서 일관된 네이밍과 구조 유지

HTTP 상태 코드의 의미

- **2xx Success**: 요청이 성공적으로 처리됨
  - 200 OK, 201 Created, 204 No Content
- **4xx Client Error**: 클라이언트 요청에 문제가 있음
  - 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
- **5xx Server Error**: 서버에서 오류가 발생함
  - 500 Internal Server Error, 503 Service Unavailable

REST의 장단점

장점: 웹 표준 활용, 플랫폼 독립적, 간단하고 직관적
단점: 표준이 없어 모호함, HTTP 메서드 제한, 복잡한 쿼리 표현의 한계

---

6. AJAX (Asynchronous JavaScript And XML)

AJAX의 핵심 개념

- **비동기 통신**: 페이지 전체를 새로고치지 않고 필요한 데이터만 서버와 주고받음
- **사용자 경험**: 페이지 깜빡임 없는 매끄러운 인터랙션 제공
- **부분 업데이트**: 페이지의 특정 영역만 동적으로 변경
- **XMLHttpRequest**: 브라우저에서 제공하는 비동기 통신 API (현재는 fetch API도 사용)

AJAX의 동작 원리

1. **JavaScript 이벤트 발생**: 사용자 액션 또는 프로그램적 트리거
2. **XMLHttpRequest 객체 생성**: 서버와 통신할 객체 생성
3. **비동기 요청 전송**: 서버에 데이터 요청 (GET, POST 등)
4. **서버 처리**: 요청을 받아 적절한 응답 데이터 생성
5. **응답 수신**: JavaScript가 서버 응답을 받음
6. **DOM 조작**: 받은 데이터로 페이지 일부분만 업데이트

AJAX의 주요 특징

- **비동기성**: 요청 후 응답을 기다리는 동안 다른 작업 수행 가능

- **선택적 업데이트**: 필요한 부분만 변경하여 효율적
- **다양한 데이터 형식**: XML, JSON, HTML, 텍스트 등 지원
- **브라우저 지원**: 모든 현대 브라우저에서 지원

AJAX 사용 기술

- **XMLHttpRequest**: 전통적인 AJAX 구현 방식
- **Fetch API**: 모던 브라우저의 새로운 HTTP 클라이언트 API
- **jQuery AJAX**: jQuery 라이브러리의 간편한 AJAX 메서드
- **Axios**: 인기 있는 HTTP 클라이언트 라이브러리

AJAX의 장점

- **향상된 사용자 경험**: 페이지 새로고침 없는 부드러운 인터랙션
- **네트워크 효율성**: 필요한 데이터만 전송하여 대역폭 절약
- **응답성**: 빠른 페이지 반응속도
- **풍부한 인터랙션**: 데스크톱 애플리케이션과 유사한 경험

AJAX의 단점과 고려사항

- **SEO 문제**: 검색 엔진이 동적 콘텐츠를 크롤링하기 어려움
- **브라우저 히스토리**: 뒤로 가기 버튼 동작 관리 복잡
- **접근성**: 스크린 리더 등 보조 기술 지원 고려 필요
- **JavaScript 의존**: JavaScript 비활성화 시 동작하지 않음
- **보안**: CSRF, XSS 등 보안 취약점 주의 필요

---

7. 웹스퀘어의 서브미션에 대해 설명해주세요

서브미션의 정의

"웹스퀘어의 서브미션은 클라이언트에서 서버로 데이터를 전송하는 통신 객체입니다. 사용자 인터페이스에서 입력된 데이터를 서버 측 비즈니스 로직으로 전달하고, 처리 결과를 다시 클라이언트로 받아오는 역할을 담당합니다."

주요 특징

- **비동기 통신**: Ajax 기반으로 페이지 새로고침 없이 서버와 통신
- **데이터 바인딩**: DataCollection과 연동하여 자동으로 데이터 매핑
- **다양한 통신 방식 지원**: HTTP, HTTPS, JSON, XML 등
- **에러 핸들링**: 통신 실패 시 예외 처리 기능 제공

구성 요소

- **요청 정보**: URL, HTTP 메서드, 파라미터 설정
- **응답 처리**: 성공/실패 콜백 함수 정의
- **데이터 매핑**: 입력/출력 데이터의 구조 정의

실무 활용 예시

"예를 들어, 사용자 정보 수정 화면에서 수정 버튼을 클릭했을 때, 서브미션을 통해 입력된 데이터를 서버의 사용자 정보 업데이트 API로 전송하고, 처리 결과에 따라 성공 메시지를 표시하거나 오류 처리를 수행할 수 있습니다."

8. 호이스팅 (Hoisting)

호이스팅의 본질

- **끌어올림**: 변수와 함수 선언이 해당 스코프의 최상단으로 "끌어올려지는" 현상
- **JavaScript 엔진의 동작**: 실제로 코드가 이동하는 것이 아니라 메모리 할당 과정에서 발생
- **컴파일 단계**: JavaScript 엔진이 코드를 실행하기 전에 변수와 함수 선언을 먼저 처리

JavaScript 실행 과정과 호이스팅

1. **컴파일 단계**: 변수와 함수 선언을 찾아 메모리에 공간 확보
2. **실행 단계**: 실제 코드를 한 줄씩 실행하며 값 할당
3. **스코프 생성**: 각 함수나 블록마다 새로운 스코프 생성

변수 호이스팅의 차이점

var 변수

- **선언과 초기화**: 선언 시 undefined로 자동 초기화
- **함수 스코프**: 함수 단위로 스코프가 결정됨
- **중복 선언**: 같은 스코프에서 중복 선언 허용
- **문제점**: 의도치 않은 undefined 값으로 인한 버그 발생 가능

let, const 변수

- **선언만 호이스팅**: 선언은 되지만 초기화되지 않음
- **TDZ (Temporal Dead Zone)**: 선언 전까지 접근할 수 없는 영역
- **블록 스코프**: 블록({}) 단위로 스코프가 결정됨
- **엄격한 규칙**: 중복 선언 금지, const는 재할당 금지

함수 호이스팅의 종류

함수 선언식 (Function Declaration)

- **완전한 호이스팅**: 함수 전체가 호이스팅되어 선언 전에도 호출 가능
- **함수 스코프**: var와 동일한 스코프 규칙 적용
- **안정성**: 함수 전체가 메모리에 올라가므로 안정적

함수 표현식 (Function Expression)

- **변수 호이스팅 적용**: 변수명만 호이스팅되고 함수는 실행 시점에 할당
- **초기값**: 변수 선언만 호이스팅되므로 undefined 상태
- **에러 발생**: 할당 전 호출 시 "is not a function" 에러

호이스팅의 실용적 영향

- **코드 구조**: 변수를 사용하기 전에 선언하는 것이 좋은 습관
- **디버깅**: 호이스팅을 이해하면 예상치 못한 undefined 값의 원인 파악 가능
- **최적화**: 함수 선언식은 호이스팅되므로 코드 구조에 유연성 제공
- **모던 JavaScript**: let, const 사용으로 호이스팅 관련 문제 최소화

호이스팅 관련 모범 사례

- **var 대신 let, const 사용**: 예측 가능한 스코프와 TDZ로 안정성 향상
- **함수 표현식 선호**: 호이스팅으로 인한 혼란 방지

- **선언 후 사용**: 가독성과 유지보수성을 위해 선언 후 사용 원칙

---

9. 깊은 복사와 얕은 복사

복사의 필요성과 배경

- **객체의 특성**: JavaScript에서 객체는 참조 타입으로 메모리 주소를 공유
- **원본 보호**: 원본 데이터의 의도치 않은 변경 방지
- **독립적 조작**: 복사본을 독립적으로 수정할 필요성
- **함수형 프로그래밍**: 불변성(Immutability) 원칙과 관련

얕은 복사 (Shallow Copy)

- **1단계 복사**: 객체의 최상위 프로퍼티만 새로 복사
- **참조 공유**: 중첩된 객체나 배열은 원본과 참조 공유
- **부분적 독립성**: 원시값 프로퍼티는 독립적, 참조값 프로퍼티는 공유
- **성능**: 빠르고 메모리 효율적

얕은 복사가 발생하는 상황

- **Object.assign()**: 첫 번째 인자에 나머지 객체들의 프로퍼티 복사
- **스프레드 연산자 (...)**: ES6에서 도입된 간편한 복사 방법
- **Array.slice()**: 배열의 특정 부분을 새 배열로 복사
- **Array.from()**: 이터러블한 객체를 배열로 변환

깊은 복사 (Deep Copy)

- **완전 복사**: 중첩된 모든 객체와 배열까지 새로 생성
- **완전한 독립성**: 원본과 복사본이 완전히 분리됨
- **재귀적 처리**: 모든 깊이의 중첩 구조를 순회하며 복사
- **메모리 사용**: 더 많은 메모리 사용, 성능상 비용 발생

깊은 복사 구현 방법들

- **JSON 방식**: JSON.stringify + JSON.parse 조합
    - **장점**: 간단하고 이해하기 쉬움

    - **한계**: 함수, undefined, Symbol, Date 객체 등 제한사항
- **재귀 함수**: 직접 구현한 깊은 복사 함수
    - **장점**: 완전한 제어 가능, 모든 타입 지원 가능
    - **단점**: 순환 참조 처리, 복잡한 구현
- **외부 라이브러리**: Lodash의 cloneDeep, Ramda의 clone 등
    - **장점**: 검증된 구현, 다양한 에지 케이스 처리
    - **단점**: 추가 의존성

언제 어떤 복사를 사용할까?

얕은 복사 사용 상황

- **단순한 객체**: 중첩이 없거나 1단계만 있는 객체
- **성능이 중요**: 빠른 복사가 필요한 상황
- **메모리 제약**: 메모리 사용량을 최소화해야 할 때
- **참조 공유 허용**: 중첩 객체의 참조 공유가 문제되지 않는 경우

깊은 복사 사용 상황

- **복잡한 중첩 구조**: 여러 단계의 중첩된 객체/배열
- **완전한 독립성 필요**: 원본 데이터 보호가 중요한 경우
- **상태 관리**: Redux, 불변성이 중요한 상태 관리
- **데이터 변환**: 원본을 보존하면서 데이터를 변환해야 할 때

복사 관련 모범 사례

- **목적에 맞는 선택**: 요구사항에 따라 적절한 복사 방법 선택
- **불변성 라이브러리**: Immutable.js, Immer 등 전문 도구 활용
- **구조 공유**: 성능과 메모리를 위한 구조 공유 기법 고려
- **테스트**: 복사 후 원본과의 독립성 확인하는 테스트 작성

---

10. React Hooks

React Hooks의 탄생 배경

- **클래스 컴포넌트의 문제점**: 복잡한 생명주기, this 바인딩, 로직 재사용의 어려움
- **함수형 컴포넌트의 한계**: 상태 관리와 생명주기 메서드 사용 불가
- **Hook의 해결책**: 함수형 컴포넌트에서 클래스 컴포넌트의 기능을 사용할 수 있게 함
- **React 16.8**: 2019년 2월에 정식 도입된 새로운 패러다임

Hooks의 핵심 개념

- **함수형 프로그래밍**: 클래스 없이 상태와 생명주기 기능 사용
- **재사용성**: 커스텀 Hook을 통한 로직 재사용
- **간결성**: 클래스 컴포넌트 대비 간결한 코드
- **조합성**: 여러 Hook을 조합하여 복잡한 로직 구성

Hook 규칙 (Rules of Hooks)

1. **최상위에서만 호출**: 반복문, 조건문, 중첩 함수 내부에서 Hook 호출 금지
    - **이유**: Hook의 호출 순서가 바뀌면 상태 추적 불가능
    - **보장**: 매 렌더링에서 동일한 순서로 Hook 호출
2. **React 함수에서만 호출**: React 함수형 컴포넌트나 커스텀 Hook에서만 사용
    - **금지**: 일반 JavaScript 함수에서 Hook 사용 불가
    - **목적**: React의 상태 관리 시스템과 연동

주요 내장 Hooks

useState - 상태 관리

- **목적**: 함수형 컴포넌트에서 지역 상태 관리
- **반환값**: [현재 상태값, 상태 변경 함수] 배열
- **초기값**: 원시값, 객체, 함수 모두 가능
- **배치 처리**: 여러 setState 호출을 배치로 처리하여 성능 최적화
- **함수형 업데이트**: 이전 상태를 기반으로 새 상태 계산

useEffect - 사이드 이펙트 처리

- **목적**: 컴포넌트의 사이드 이펙트(API 호출, DOM 조작, 구독 등) 처리

- **생명주기 대체**: componentDidMount, componentDidUpdate, componentWillUnmount 기능 통합
- **의존성 배열**: 두 번째 인자로 의존성을 지정하여 실행 조건 제어
  - **빈 배열 []**: 컴포넌트 마운트 시에만 실행 (componentDidMount)
  - **의존성 있음**: 의존성 변경 시에만 실행
  - **의존성 없음**: 매 렌더링마다 실행
- **클린업**: 반환 함수를 통해 정리 작업 수행

useContext - Context 사용

- **목적**: React Context API를 Hook으로 간편하게 사용
- **Props Drilling 해결**: 깊은 컴포넌트 트리에서 데이터 전달 문제 해결
- **전역 상태**: 테마, 언어, 사용자 정보 등 전역적으로 사용되는 데이터 관리
- **성능 고려**: Context 값 변경 시 모든 Consumer 리렌더링

useReducer - 복잡한 상태 로직

- **목적**: useState보다 복잡한 상태 로직을 관리할 때 사용
- **Redux 유사**: action과 reducer 패턴을 사용한 상태 관리
- **예측 가능**: 상태 변경이 reducer 함수를 통해서만 이루어짐
- **디버깅**: 상태 변경 과정을 추적하기 쉬움
- **사용 시기**: 여러 하위 값이 있는 복잡한 상태, 다음 상태가 이전 상태에 의존적인 경우

useMemo - 메모이제이션

- **목적**: 비용이 큰 계산 결과를 메모이제이션하여 성능 최적화
- **의존성 기반**: 의존성이 변경되지 않으면 이전 계산 결과 재사용
- **사용 시기**: 무거운 연산, 복잡한 객체 생성 시
- **주의점**: 과도한 사용은 오히려 성능 저하 가능

useCallback - 함수 메모이제이션

- **목적**: 함수를 메모이제이션하여 불필요한 리렌더링 방지
- **자식 컴포넌트 최적화**: props로 전달되는 함수가 변경되지 않으면 자식 컴포넌트 리렌더링 방지

- **의존성 관리**: 함수 내부에서 사용하는 값들을 의존성 배열에 포함
- **useMemo와의 관계**: useCallback(fn, deps) === useMemo(() ⇒ fn, deps)

커스텀 Hooks

- **개념**: 재사용 가능한 상태 로직을 추상화한 사용자 정의 Hook
- **네이밍 규칙**: 반드시 'use'로 시작하는 함수명 사용
- **조합**: 여러 내장 Hook을 조합하여 복잡한 로직 캡슐화
- **테스트**: 로직을 분리하여 독립적으로 테스트 가능

커스텀 Hook의 장점

- **로직 재사용**: 여러 컴포넌트에서 동일한 상태 로직 공유
- **관심사 분리**: UI와 비즈니스 로직 분리
- **테스트 용이성**: 로직만 따로 테스트 가능
- **가독성**: 복잡한 로직을 의미 있는 이름으로 추상화

React Hooks의 장점

- **간결한 코드**: 클래스 컴포넌트 대비 boilerplate 코드 감소
- **로직 재사용**: 커스텀 Hook을 통한 강력한 재사용 메커니즘
- **관심사 분리**: 관련 로직을 한 곳에 모을 수 있음
- **함수형 패러다임**: 함수형 프로그래밍의 장점 활용
- **타입스크립트 친화적**: 타입 추론과 검증이 용이

Hook 사용 시 주의사항

- **의존성 배열**: useEffect, useMemo, useCallback의 의존성을 정확히 명시
- **무한 루프**: 잘못된 의존성 설정으로 인한 무한 루프 방지
- **메모리 누수**: useEffect의 클린업 함수로 구독 해제, 타이머 정리
- **과도한 최적화**: useMemo, useCallback의 남용은 오히려 성능 저하
- **ESLint 플러그인**: eslint-plugin-react-hooks로 Hook 규칙 검사

Hooks의 미래와 발전 방향

- **Concurrent Features**: React 18의 동시성 기능과 함께 발전

- **Server Components**: 서버 컴포넌트와의 통합
- **커뮤니티 생태계**: 다양한 커스텀 Hook 라이브러리 등장
- **성능 최적화**: 더욱 정교한 최적화 기법 개발

1. 순수 추상 클래스와 인터페이스의 차이

"추상 클래스는 일반 메서드와 추상 메서드를 모두 가질 수 있고, 인터페이스는 메서드 시그니처만 정의합니다. 가장 큰 차이는 추상 클래스는 단일 상속만 가능하지만 인터페이스는 다중 구현이 가능하다는 점입니다. 또한 추상 클래스는 생성자와 인스턴스 변수를 가질 수 있어서 공통 기능을 구현할 때 유용합니다."

2. IOC와 DI

"IOC는 제어의 역전으로, 객체 생성과 관리를 개발자가 아닌 스프링 컨테이너가 담당하는 개념입니다. DI는 의존성 주입으로 IOC를 구현하는 방법 중 하나인데, 객체가 필요한 의존 객체를 직접 생성하지 않고 외부에서 주입받습니다. 이렇게 하면 결합도가 낮아지고 테스트하기 쉬워집니다."

3. VO, DTO, DAO

"VO는 값 자체를 나타내는 불변 객체로 주로 도메인에서 사용하고, DTO는 계층 간 데이터 전송을 위한 객체입니다. DAO는 데이터베이스 접근 로직을 캡슐화한 객체로 CRUD 연산을 담당합니다. 간단히 말하면 VO는 값 표현, DTO는 데이터 운반, DAO는 데이터 접근 역할을 합니다."

4. 스프링 프레임워크의 생명 주기

"스프링 빈의 생명주기는 객체 생성, 의존관계 주입, 초기화 콜백 실행, 사용, 소멸 전 콜백 실행 순서로 진행됩니다. @PostConstruct로 초기화 작업을, @PreDestroy로 정리 작업을 할 수 있습니다. 기본적으로 싱글톤 스코프라서 컨테이너와 생명주기를 함께 합니다."

5. REST API

"REST는 웹의 장점을 활용한 아키텍처 스타일입니다. 자원을 URI로 표현하고 HTTP 메서드로 조작하는데, GET은 조회, POST는 생성, PUT은 수정, DELETE는 삭제에 사용합니다. 무상태성이라서 각 요청이 독립적이고, JSON 형태로 데이터를 주고받는 것이 일반적입니다."

6. AJAX

"AJAX는 페이지 새로고림 없이 서버와 비동기 통신하는 기술입니다. XMLHttpRequest나 fetch API를 사용해서 백그라운드에서 서버와 데이터를 주고받고, 받은 데이터로 페이지 일부만 업데이트합니다. 사용자 경험이 훨씬 부드러워지는 장점이 있습니다."

7. 웹스퀘어 submission

"웹스퀘어에서 클라이언트와 서버 간 데이터 통신을 담당하는 컴포넌트입니다. 화면에서 입력한 데이터를 서버로 전송하고 응답을 받아서 처리하는 역할을 합니다. 동기와 비동기 방식을 모두 지원해서 상황에 맞게 사용할 수 있습니다."

8. 호이스팅

"호이스팅은 변수나 함수 선언이 해당 스코프 맨 위로 끌어올려지는 자바스크립트 특성입니다. 자바스크립트 엔진이 코드를 실행하기 전에 선언부를 먼저 메모리에 저장하기 때문에 발생합니다. var는 undefined로 초기화되고, let과 const는 선언만 호이스팅되어서 접근하면 에러가 발생합니다."

9. 깊은 복사와 얕은 복사

"얕은 복사는 객체의 최상위 속성만 복사해서 중첩된 객체는 원본과 참조를 공유합니다. 깊은 복사는 중첩된 모든 객체까지 완전히 새로 만드는 것입니다. 스프레드 연산자는 얕은 복사이고, 깊은 복사는 JSON.stringify를 사용하거나 라이브러리를 활용해야 합니다."

10. React Hooks

"Hooks는 함수형 컴포넌트에서 상태와 생명주기 기능을 사용할 수 있게 해주는 리액트 기능입니다. useState로 상태를 관리하고, useEffect로 생명주기나 사이드 이펙트를 처리합니다. 클래스형 컴포넌트보다 코드가 간결해지고 로직 재사용이 쉬워집니다."