**JAVA**

**1. Multithreading and concurrency**

- Introduction to Multithreading and Concurrency- Concurrency vs. Parallelism

- Common Use Cases for Multithreading

- Creating and Managing Threads in Java - Thread Class- Thread Lifecycle and States

- Synchronization and Thread Safety

- Inter-Thread Communication- Executor Framework

- Advanced Concurrency Utilities- Best Practices for Multithreaded Programming- Practical Examples

**2. Networking in java**

- Introduction to Java Networking APIs

- Java Networking Package (java.net)

- Importance of Networking in Modern Applications

- Sockets and Server Sockets in Java

- Working with URLs in Java

- Building a Simple Client-Server Application

- Handling Network Exceptions

- Security Considerations in Networked Applications

- Optimizing Network Performance

- Exercises and Hands-On Lab

**3. Database connectivity with JDBC**

- Introduction to JDBC

- JDBC Architecture and Components- Setting Up Database Connection

- Establishing a Database Connection

- CRUD Operations Overview

- Executing CRUD Operations with JDBC

- Very Lightweight DB for Java: H2 Database

- SQL Practices

- PreparedStatement and ResultSet- Transaction Management

**4. Design patterns and best practices**

- Introduction to Design Patterns

- Key Principles Achieved by Design Patterns

- Activity – Identifying Design Issues- Creational Patterns Overview

- Singleton Pattern

- Factory Method Pattern

- Builder Pattern

- Design Patterns

- Structural Design Patterns- Adapter Pattern- Decorator Pattern

- Composite Pattern- Structural Design Patterns

- Behavior Design Patterns

- Strategy Pattern- Observer Pattern- Template Method Pattern- Behavior Design Patterns

**< Multithreading and concurrency >**

**Introduction to Multithreading and Concurrency**

**1. Definition of Multithreading**

- Thread: 프로그램 내에서 실행되는 단일 실행 흐름, 운영체제에 의해 스케줄링될 수 있는 가장 작은 처리 단위

- Multithreading: CPU 또는 단일 프로그램이 여러 실행 스레드를 동시에 관리할 수 있는 기능

**2. Importance of Multithreading**

- Performance Enhancement: 특히 다중 코어 프로세서에서 유리

- Responsiveness: 백그라운드 스레드가 시간이 오래 걸리는 작업을 처리하는 동안, 사용자 인터페이스는 계속해서 사용자와의 상호작용에 응답

- Resource Utilization

**Concurrency vs. Parallelism**

**1. Concurrency, 동시성**

- 여러 작업을 일정 기간 동안 관리하고 실행할 수 있는 능력을 의미하며, 이 작업들은 시간적으로 **겹칠 수 있음**

- 단일 코어 환경에서 실행이 교차(interleaving)되기 때문에 스레드들이 동시에 실행되는 것처럼 보임

**2. Parallelism, 병렬성**

- 여러 작업을 **동시에 실행**

- 주로 다중 코어 프로세서

- 각 작업이 개별 코어에서 실행되기 때문에 **진정한 동시 실행**

- **여러 개의 코어 또는 프로세서가 있는 하드웨어**가 필요

-> **동시성**은 여러 작업을 다루고 시간에 따라 실행을 관리하는 것이고, **병렬성**은 여러 작업을 **동시에 실행**하는 것

**Creating and Managing Threads in Java - Thread Class**

**1. Extending the Thread Class**

- **Thread 클래스**를 확장하고 run() 메서드를 오버라이드

- run() 메서드에는 스레드가 시작될 때 실행될 코드가 포함

- 클래스가 다른 클래스를 확장해야 할 경우 제한적

- Java는 **다중 상속을 지원하지 않음**

**2. Implementing Runnable Interface**

- **Thread 클래스를 확장하지 않고도** run() 메서드를 정의 가능

- 클래스가 다른 클래스를 **확장해야 할 경우**에도 사용 가능

- 작업(task)인 run() 메서드와 **스레드 자체**를 분리

- 코드의 **재사용성**이 높음

**3. Using Lambda Expressions for Threads**

- 코드를 단순화하고 **함수형 스타일**로 작성

- **일회성 작업**에 유용

**Thread Lifecycle and States**

**1. Lifecycle of a Thread**

- New: 스레드가 생성되었지만 아직 시작되지 않은 상태

- Runnable: 스레드가 실행 준비가 되어 CPU 할당을 기다리는 상태. start() 메서드를 호출하면 이 상태로 진입, 스레드 스케줄러가 언제 스레드에 CPU 시간을 할당할지를 결정

- Blocked: 스레드가 동기화된 블록 또는 메서드에 들어가려고 하지만, **락**을 얻기 위해 기다리는 상태. 다른 스레드가 락을 해제할 때까지 이 상태

- Waiting: 스레드가 **무기한 대기**하며, 다른 스레드가 특정 동작(예: notify() 호출)을 수행할 때까지 기다리는 상태

- Timed Waiting: 스레드가 **지정된 시간 동안 대기**하는 상태. 타임아웃이 만료되면 스레드는 다시 **Runnable** 상태

- Terminated: 스레드의 실행이 완료. run() 메서드가 끝나면 스레드는 자동으로 이 상태

**2. Methods to Control Thread State**

- start(): **New** 상태에서 **Runnable** 상태로

- sleep(milliseconds): **Timed Waiting** 상태

- yield(): 스레드 스케줄러에게 현재 스레드가 프로세서를 **양보할 의향이 있다**는 힌트를 제공, **Runnable** 상태를 유지

- join(): 현재 스레드를 **Waiting** 상태로 만들고, 지정된 스레드가 실행을 **완료할 때까지 대기**

- interrupt(): **Waiting** 상태나 **Timed Waiting** 상태에 있을 때 인터럽트를 걸어, 해당 스레드를 다시 **Runnable** 상태로 전환

**3. Examples of Thread State Transitions**

- New -> Runnable: start() 메서드를 호출

- Runnable -> Running: 스레드 스케줄러가 CPU 시간을 할당

- Running -> Blocked: 스레드가 동기화된 블록이나 메서드에 진입하려 하지만 락을 얻기 위해 기다려야 할 때

- Running -> Waiting: wait() 메서드를 **타임아웃 없이** 호출, 다른 스레드가 notify()나 notifyAll()을 호출할 때까지 대기

- Running -> Timed Waiting: sleep(milliseconds) 또는 wait(milliseconds)와 같은 메서드 호출, 지정된 시간이 지나면 스레드는 다시 **Runnable** 상태

- Any State -> Terminated: run() 메서드 실행이 완료

**Synchronization and Thread Safety**

**1. Need for Synchronization**- Race Conditions: 두 개 이상의 스레드가 **공유 데이터를 동시에 접근하고 수정**하려고 할 때. **예측 불가능하고 일관되지 않은 결과**를 초래

- Data Inconsistency: 동시 실행 환경에서 공유 자원에 **동기화되지 않은 접근**이 이루어지면 데이터가 불일치하거나 손상될 수 있음

- Ensuring Thread Safety: 여러 스레드가 중요한 코드 영역 (critical section)을 동시에 실행하지 못하게 방지하여 공유 데이터가 **통제된 방식으로 접근되고 수정되도록 보장**

**2. Synchronized Methods and Blocks**

- Synchronized Method: **동일한 객체**에 대해 한 번에 하나의 스레드만 이 메서드를 실행할 수 있도록 보장

- Synchronized Block: 메서드 전체가 아닌 **특정 코드 영역**만 동기화하는 데 사용 -> **세밀한 제어**가 가능

**3. Using Lock Objects – ReentrantLock**

- ReentrantLock: java.util.concurrent.locks 패키지에 포함, synchronized 키워드보다 **유연한 잠금 메커니즘**

- 장점

> Explicit Locking: lock()과 unlock() 메서드를 통해 **잠금 획득과 해제**를 명시적으로 제어

> Try Locking: tryLock() 메서드를 사용하면 잠금을 시도하고, **잠금이 불가능하면 다른 작업을 진행**

> Fairness: **공정 모드**로 설정 가능; **스레드가 요청한 순서대로 잠금을 획득**하도록 보장

\* lock: 공유 데이터에 대한 접근 권한

\* deadlock: 두개 이상 스레드가 서로 보유한 자원을 기다리며 무한 대기

**4. Volatile and Atomic Variables**

- Volatile 키워드: volatile로 선언된 변수에 대한 **읽기 및 쓰기 작업**은 항상 주 메모리(main memory)를 직접 접근하므로, 모든 스레드가 항상 최신 값 접근 가능

- Atomic Variables: java.util.concurrent.atomic 패키지에 **AtomicInteger**, **AtomicBoolean** 등과 같은 클래스를 제공하여 **단일 변수**에 대한 원자적 연산을 수행, synchronized 블록이나 **락(lock)** 없이도 원자적 연산을 수행

\* Atomicity: 작업이 **부분적으로 실행되거나 중간에 중단되지 않고** **모두 실행되거나 전혀 실행되지 않아야 한다**는 것

**Inter-Thread Communication1. Inter-Thread Communication with wait(), notify(), and notifyAll()**- 스레드들이 **신호를 교환하거나 서로를 기다림**

- 공유 데이터 접근 시 **충돌이나 데드락**을 방지

- **의존하는 스레드 간의 동기화**를 가능

- 조건이 충족될 때까지 스레드가 **대기**하도록 함으로써 불필요한 CPU 사용을 줄임

**2. Methods for Inter-Thread Communication**- wait(): 현재 스레드가 **락을 해제하고 대기** 상태, 다른 스레드가 동일한 객체에 대해 notify() 또는 notifyAll()을 호출하면 깨어남

- notify(): 대기 중인 **스레드 하나**를 깨움, 깨어난 스레드는 **락을 다시 획득**한 후 실행

- notifyAll(): 대기 중인 **모든 스레드를 깨움**, 모든 스레드는 **락을 경쟁**

**4. Simplified Producer-Consumer with BlockingQueue**

- BlockingQueue는 **스레드 안전 컬렉션**으로 동기화를 자동으로 처리

- put()과 take()와 같은 **블로킹 연산**을 제공

- 수동으로 락(lock)과 조건을 관리할 필요가 없어 **오류와 복잡성** 감소

**Executor Framework**

**1. Introduction to the Executor Framework**

- java.util.concurrent 패키지의 일부로, **스레드 관리 및 작업 실행**을 위한 고수준 API를 제공

- **Executor**: 작업 실행을 위한 execute(Runnable command) 메서드를 정의하는 기본 인터페이스

- **ExecutorService**: **Executor**를 확장한 인터페이스로, 작업 제출 및 스레드 종료(lifecycle)를 관리하는 메서드를 제공

**2. Types of Thread Pools in the Executor Framework**- Fixed Thread Pool: 고정된 개수의 스레드를 유지하며 여러 작업을 재사용하여 처리

- Cached Thread Pool: 필요에 따라 새로운 스레드를 생성하고, 사용 가능한 기존 스레드를 재사용

- Scheduled Thread Pool: 작업을 **지연 후 실행**하거나 **주기적으로 실행**하도록 스케줄링

**4. ForkJoin Framework**

- 작업을 **재귀적으로 작은 작업**으로 나눈 후 결과를 결합하여 최종 출력을 얻는 데 최적화된 프레임워크

- **분할 정복(divide-and-conquer)** 알고리즘에 적합

- ForkJoinPool: java.util.concurrent 패키지에 포함된 **특수 스레드 풀**로, ForkJoinTask와 함께 동작, 작업자 스레드 풀을 관리하며 작은 작업을 병렬로 실행

- 재귀적 작업 분할: 작업을 작은 단위(그레인 수준)로 **분할** -> 작은 작업은 **병렬 실행**되며, 결과를 결합하여 최종 출력을 생성

**Advanced Concurrency Utilities1. Concurrent Collections**

- thead-safe 버전의 전통적인 Java 컬렉션으로, 외부 동기화 없이도 **동시 수정**을 처리하도록 설계

- ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue

**2. Synchronization Utilities**- CountDownLatch: 다른 스레드에서 **일련의 작업이 완료될 때까지** 하나 이상의 스레드가 기다리도록 돕는 동기화 도구, 초기화 시 설정된 카운트는 countDown() 메서드로 감소되며, 카운트가 **0이 되면** 대기 중인 스레드가 계속 실행

- CyclicBarrier: 여러 스레드가 **공통의 장벽 지점**에 도달할 때까지 서로를 기다리도록 돕는 동기화 도구, 모든 스레드가 장벽에 도달하면 동시에 실행을 계속하며, **장벽은 재사용 가능**

**3. Semaphore**- **공유 자원에 접근할 수 있는 스레드 수를 제한**하는 동기화 도구

- 사용 가능한 허가(permit)의 개수를 유지하며, 스레드는 허가를 획득한 후 실행하고, 완료되면 허가를 **반환**

**Best Practices for Multithreaded Programming**- 공유 데이터 사용 최소화

- 고수준 동시성 API 활용

- 확장성과 유지 보수성을 고려한 설계

- 동시성 애플리케이션 테스트 및 디버깅

**< Networking in java >**

**Introduction to Java Networking APIs**

**1. Overview of Java Networking Capabilities**

- Java에서의 네트워킹: 서로 다른 **장치**나 **애플리케이션 간 연결**을 만들어 **데이터 교환**을 가능하게 하는 것

- 로컬 네트워크(LAN)와 **인터넷** 모두에서 애플리케이션 간의 통신을 지원

- Java의 네트워킹 API: **java.net 패키지**에 내장된 네트워킹 API를 제공, **클라이언트-서버 통신**과 **P2P(peer-to-peer) 데이터 교환**을 모두 지원

**2. Role of Networking in Java Applications**

- 분산 애플리케이션 구축: 여러 애플리케이션이 네트워크를 통해 **통신**하고 **자원을 공유**

**Java Networking Package (java.net)**

**1. Overview of Key Classes and Interfaces**

- Socket: 두 머신 간 통신의 끝점(Endpoint)을 나타내며, **TCP 연결**을 통해 직접적인 데이터 전송을 가능하게 함

- ServerSocket: 서버 측 클래스이며, **들어오는 클라이언트 연결**을 수신

- URL: **Uniform Resource Locator**를 나타내며, 인터넷상의 리소스에 접근하기 위한 주소를 제공

- HttpURLConnection: URLConnection의 하위 클래스이며, HTTP 전용 기능을 지원

**2. High-Level vs. Low-Level Networking**

- Socket-Based Networking: Socket과 ServerSocket을 사용하여 **저수준 통신**을 제공, 데이터 흐름에 대한 **세밀한 제어**가 가능

- URL-Based Networking: 인터넷상의 리소스에 접근할 수 있는 **고수준** 네트워킹 기능을 제공

**Importance of Networking in Modern Applications**

**2. Advantages of Using Java for Networking**

- Platform Independence: Java의 **크로스 플랫폼** 특성 덕분에 네트워크 애플리케이션은 다양한 **운영체제**에서 실행 가능

- Robust Libraries: 잘 **문서화**되어 있음, **안정적이고 일관된 성능**을 제공

**Sockets and Server Sockets in Java**

**1. What is a Socket?**

- 네트워크에서 데이터를 **송수신**하는 endpoint

- 두 장치가 네트워크를 통해 **통신**할 수 있도록 **전용 채널**을 제공

- 클라이언트는 **소켓**을 생성하여 서버에 연결을 요청

- 서버는 ServerSocket을 사용해 들어오는 연결을 수신하고, 연결이 수립되면 클라이언트와 통신을 시작

**2. Types of Sockets**

- TCP 소켓: 신뢰할 수 있는 **연결 지향형** 통신

> 데이터 **전송 보장, 데이터 순서 유지, 오류 검사 제공**

- UDP 소켓: 연결을 수립하지 않고 **빠른 통신**을 제공

> 오류 검사나 데이터 순서 보장이 없음

**3. Creating a Client Socket with the Socket Class**

- connect(): 소켓을 서버의 **주소와 포트**에 연결

- getInputStream(): 서버로부터 데이터를 읽을 수 있는 **입력 스트림**을 가져옴

- getOutputStream(): 서버로 데이터를 전송하는 **출력 스트림**을 가져옴

- close(): 소켓 연결을 닫아 자원을 해제

**4. ServerSocket Class for Listening to Client Connections**

- 여러 클라이언트의 요청을 처리하기 위해 **각 연결마다 새로운 소켓**을 생성

- accept(): 들어오는 클라이언트 연결을 **대기**하고, 연결이 수립되면 **통신**을 시작

- bind(): ServerSocket을 특정 **포트**와 **로컬 IP 주소**에 바인딩

- close(): 서버 소켓을 닫아 더 이상 연결 요청을 받지 않음

**Working with URLs in Java**

**1. The URL Class in Java**

- URL: Uniform Resource Locator, 웹 주소를 나타내며, Java 애플리케이션이 **인터넷 리소스에 접근**할 수 있도록 함

- 구조

> **Protocol** : 통신에 사용되는 프로토콜

> **Host** : 리소스를 호스팅하는 서버의 이름 또는 IP 주소

> **Port** : 서버가 수신 대기 중인 네트워크 포트

> **File** : 서버의 특정 리소스에 대한 경로 및 파라미터

- 주요 메서드

> **getProtocol()**: URL에서 사용된 프로토콜을 반환

> **getHost()**: URL의 호스트 구성 요소 반환

> **getPort()**: 포트 번호를 반환

> **getFile()**: 파일 경로 및 파라미터를 반환

**2. Connecting to Web Resources with HttpURLConnection**

- 웹 리소스에 연결하고 데이터를 전송하거나 응답을 읽는 데 사용

- 주요 메서드

> **connect()**: URL에 대한 **연결을 열어줌**

> **getResponseCode()**: HTTP 응답 코드를 반환

> **getInputStream()**: 서버에서 반환된 응답 데이터를 읽을 수 있는 **입력 스트림**을 제공

> **disconnect()**: 연결을 닫아 리소스를 해제

**Handling Network Exceptions**

**1. Common Network Exceptions**

- UnknownHostException: 지정된 **호스트**를 확인할 수 없음

- IOException: 입력/출력 작업과 관련된 **일반적인 예외**

- SocketException: 소켓과 관련된 **문제**, ex. **연결 재설정** 또는 **타임아웃**

**2. Best Practices for Exception Handling**

- Try-Catch 블록

- graceful error handling: **사용자 친화적인 에러 메시지**나 대체 작업(fallback actions) 제공

- 자원 관리: **소켓**과 **스트림**은 항상 finally 블록에서 닫아야 함, 또는 **try-with-resources** 구문을 사용해 **자동으로 리소스를 해제**

**Security Considerations in Networked Applications**

- Data Encryption: **SSL/TLS**와 같은 암호화 프로토콜을 사용하여 안전한 통신 채널을 구축, Java는 SSLSocket을 통해 **SSL** 지원을 제공

- Authentication: **인증**을 통해 오직 **권한이 있는 클라이언트와 서버**만 연결 ex. API 키, 토큰, SSL 인증서

- Access Control: 특정 **IP 주소**에만 접근을 허용하거나 **방화벽**을 사용해 **무단 사용자**의 접근을 차단

- Validation: 들어오는 데이터를 **항상 유효성 검사**하여 SQL 인젝션과 같은 **공격**이나 악성 입력을 방지

**Optimizing Network Performance**

**1. Definition of Multithreading**

- 효율적인 데이터 전송 기술

> Using Buffers: **네트워크에 대한 직접적인 읽기/쓰기 연산**의 횟수를 줄여 **효율성**을 높임

> Managing Data Packet Sizes: 데이터가 **적절한 크기**의 청크로 전송되도록 함

> Minimizing Latency: **서버를 사용자와 가까운 위치**에 배치하여 지연 시간을 줄임

**< Database connectivity with JDBC >**

**Introduction to JDBC**

**1. Definition of a Database**

- Database: 데이터를 **체계적으로 조직**하여 **저장, 검색, 관리**할 수 있도록 하는 시스템

- 관계형 데이터베이스 (SQL)는 테이블 형식,

NoSQL 데이터베이스는 문서 형식으로 데이터를 저장

- 데이터베이스 사용의 장점

> Centralized Data Management: 여러 사용자와 시스템이 데이터를 **일관되게 저장, 업데이트, 검색**

> Data Integrity: **규칙**과 **제약 조건**을 적용해 데이터의 정확성과 일관성을 보장

> Scalability

> Efficient Data Retrieval: **복잡한 쿼리**를 지원하여 특정 데이터를 **빠르고 효과적으로 검색**

**2. Defining JDBC**

- JDBC: Java Database Connectivity, **Java 애플리케이션**이 **관계형 데이터베이스**에 연결하고 상호작용할 수 있도록 제공되는 API

- JDBC의 주요 기능

> 데이터베이스 연결, SQL 문 실행, 데이터 검색 및 업데이트

\* SQL: Structured Query Language, 구조화된 질의 언어

**JDBC Architecture and Components**

**1. JDBC Core Components**

- JDBC 드라이버 관리자: 데이터베이스 드라이버를 관리하고 **연결 요청**을 처리

- JDBC 드라이버: JDBC 호출을 **데이터베이스가 이해할 수 있는 형식**으로 변환

- JDBC API: 데이터베이스에 **연결**, **SQL 실행**, **쿼리 결과 관리**를 위한 표준 인터페이스와 클래스 집합을 제공

**2. How JDBC Components Work Together**

- 애플리케이션이 JDBC API를 통해 **데이터베이스 연결 요청**

- DriverManager가 **연결 URL**을 기반으로 적절한 JDBC 드라이버를 선택

- JDBC 드라이버가 Java 애플리케이션과 데이터베이스 간 **통신**을 수행

**3. Types of JDBC Drivers**- JDBC-ODBC Bridge Driver

> JDBC 호출을 ODBC 호출로 변환하여 **ODBC 호환 데이터베이스**와 연결

- Native-API/Partially Java Driver

> 데이터베이스의 **네이티브 코드 라이브러리**를 호출

- Network Protocol/All-Java Driver

> 네트워크 기반 드라이버로, **미들웨어 서버**와 통신하고 미들웨어가 데이터베이스와 상호작용

- Thin Driver/Pure Java Driver

> **JDBC 호출**을 데이터베이스 전용 프로토콜 호출로 **직접 변환**하는 **순수 Java 드라이버**

> **플랫폼 독립적**이며 별도의 네이티브 라이브러리 없이 동작하므로 **가장 선호**

**Setting Up Database Connection**

**1. Prerequisites for Establishing a JDBC Connection**

- Database Server: 데이터베이스 서버가 **설치**되어 있고 **실행 중**

- Database Driver

> Driver Requirement: 각 데이터베이스에는 **고유한 JDBC 드라이버**가 필요

> Driver Download

> Adding Driver to Project

**Establishing a Database Connection**

**1. Connection String and Database Credentials**

- Connection String

: **jdbc:<database-type>://<host>:<port>/<database-name>**

> jdbc: JDBC 프로토콜을 나타냄

> <database-type>: 사용할 database의 유형을 지정

> <host>: database server의 **호스트명** 또는 **IP 주소**

> <port>: database에 접근할 때 사용하는 포트 번호

> <database-name>: 연결할 database의 이름

- Database Credentials: database에 **연결**하고 인증받기 위해 **username**과 **password**가 필요

**2. Using Manager to Obtain a Connection**

- DriverManager.getConnection(): **connection**을 설정**, connection URL**, **username**, **password**를 파라미터로 받음

**CRUD Operations Overview**

- CRUD: **Create, Read, Update, Delete**는 **데이터베이스와 상호작용하기 위한 네 가지 기본 작업**

> **Create** – **INSERT**: 테이블에 새로운 데이터를 추가

> **Read** – **SELECT**: 데이터베이스에서 데이터를 **조회**

> **Update** – **UPDATE**: 기존의 레코드를 **수정**

> **Delete** – **DELETE**: 테이블에서 레코드를 **삭제**

**\*\*쿼리 작성하기\*\***

**Executing CRUD Operations with JDBC**

**1. CREATE Operation – Inserting Data**

> INSERT INTO <테이블명> (컬럼1, 컬럼2, ...) VALUES (값1, 값2, ...);

ex. "INSERT INTO users (id, name, email) VALUES (1, 'John Doe', 'john@example.com')"

**2. READ Operation – Retrieving Data**

> SELECT 컬럼명 FROM 테이블명 WHERE 조건;

ex. "SELECT id, name, email FROM users WHERE id = 1"

- **ResultSet** 객체: 조회된 데이터를 행(row)과 **열(column)** 단위로 접근 가능

**3. UPDATE Operation – Modifying Data**

> UPDATE 테이블명 SET 컬럼1=값1, 컬럼2=값2 WHERE 조건;

ex. "UPDATE users SET email = 'john.new@example.com' WHERE id = 1"

**- executeUpdate():** 업데이트를 수행하며, ***영향을*** 받은 행의 수를 반환

**4. DELETE Operation – Removing Data**

> DELETE FROM 테이블명 WHERE 조건;

ex. "DELETE FROM users WHERE id = 1"

- **executeUpdate()** : 삭제를 수행하며, 영향을 받은 **행의 수**를 반환

**Very Lightweight DB for Java: H2 Database**

**SQL Practices**

**1. Creating Tables in SQL**

- CREATE TABLE: 데이터베이스에 **새로운 테이블**을 정의

- 각 column은 **이름, 데이터 타입**, 제약 조건(constraints)을 가질 수 있음

- **제약 조건 (Constraints)**: 데이터의 무결성(Integrity)을 보장

**2. CREATE TABLE Syntax**

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

- 자주 사용되는 제약 조건

> **PRIMARY KEY**: 각 레코드를 고유하게 식별

> **NOT NULL**: 해당 컬럼에 **NULL 값을 허용하지 않음**

> **UNIQUE**: 컬럼에 **중복된 값**이 없도록 보장

> **OREIGN KEY**: 다른 테이블의 **레코드와 연결**

- Basic SQL Query: Where

> 특정 조건을 적용

**3. SQL JOINs Overview**

- JOIN: JOIN은 두 개 이상의 테이블에서 **관련된 컬럼**을 기준으로 행을 결합하여 **통합된 데이터**

- JOIN의 종류

> INNER JOIN: 두 테이블에서 **일치하는 값**이 있는 행만 반환

> LEFT JOIN: **왼쪽 테이블**의 모든 행과 **오른쪽 테이블**의 일치하는 행을 반환

> RIGHT JOIN: **오른쪽 테이블**의 모든 행과 **왼쪽 테이블**의 일치하는 행을 반환

> FULL OUTER JOIN: 두 테이블에서 일치하는 행을 반환하고, 일치하지 않는 행은 **NULL**

**4. Join Operation Comparison**

텍스트, 원, 스크린샷, 도표이(가) 표시된 사진

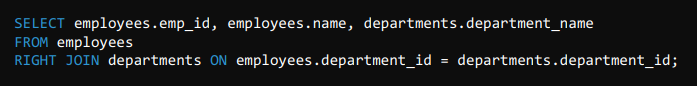
자동 생성된 설명

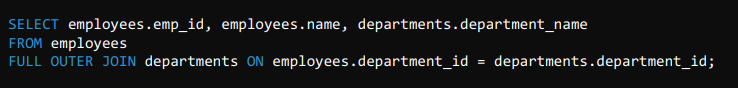
스크린샷, 폰트, 텍스트, 라인이(가) 표시된 사진

자동 생성된 설명

스크린샷, 텍스트, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명





**5. GROUP BY Clause**

- **데이터를 그룹화**하여 특정 컬럼을 기준으로 묶음

- **집계 함수**와 함께 사용

- 함수 예시

> **COUNT, SUM, AVG, MAX, MIN**

**6. ORDER BY Clause**

- 쿼리 결과를 정렬

- 정렬 방식

> ASC, DESC

**7. LIMIT Clause**

- 반환되는 **결과 행의 수를 제한**

**PreparedStatement and ResultSet**

**1. PreparedStatement Overview**

- PreparedStatement: **미리 컴파일된 SQL 문**으로, **파라미터화된 쿼리**를 지원

**2. Advantages of PreparedStatement**

- SQL Injection 방지: ?를 사용해 SQL 구조와 데이터를 분리하므로, 입력값이 SQL 구조를 변경하지 못함

- 성능 향상: Precompilation, Reusability

- 유지보수 용이: 파라미터화된 쿼리

**3. Understanding SQL Injection**

- SQL Injection: 입력 필드를 통해 악의적인 SQL 코드를 삽입해 데이터베이스를 조작하는 **보안 취약점**

**4. Risks and Consequences of SQL Injection**

- Data Theft: 공격자가 사용자 데이터, 신용카드 정보, 회사 기록과 같은 **민감한 정보 가져감**

- Data Manipulation and Corruption: 데이터를 삭제하거나 무단 변경

- Authentication Bypass: 공격자가 로그인 화면을 우회해 **시스템에 접근**

- System Takeover: 공격자가 **데이터베이스 서버**에 추가 명령을 실행해 시스템을 장악**5. How PreparedStatement Works**

- Parameterized Queries: SQL 명령어와 데이터 입력을 분리, **?**(플레이스홀더)를 사용하고 값은 나중에 메서드를 통해 제공

- SQL 구조는 **미리 정의되고 컴파일**되므로, 입력값을 통해 SQL 구조를 바꿀 수 없음**6. Best Practices for SQL Injection Prevention**

- PreparedStatement 사용: 특히 사용자 입력을 다룰 때

- 입력값 검증 및 정제

- 데이터베이스 권한 제한

- ORM 도구 사용: **파라미터화된 쿼리**를 자동으로 적용

**7. Introduction to ResultSet**

- ResultSet: **JDBC 객체**로, 데이터베이스 쿼리의 결과를 저장하며, **행 단위**로 데이터를 탐색하고 추출

- **next()**: 커서를 다음 행으로 이동

- **getString(columnLabel)**, **getInt(columnLabel)**: 컬럼 값을 이름 또는 인덱스로 가져옴

- **구조적으로 쿼리 결과**에 접근할 수 있어 애플리케이션에서 데이터를 쉽게 처리

**8. Closing ResultSet and Best Practices**

- ResultSet은 닫지 않으면 **메모리 누수**와 **성능 저하**를 유발

- Always Close Resources

- Try-With-Resources 사용: 예외가 발생하더라도 리소스가 **자동으로 닫힘**

**Transaction Management**

**1. Introduction to Transaction Management**

- Transaction: **하나의 단위 작업**으로 실행되는 일련의 데이터베이스 작업

- 모든 작업이 **성공**해야만 완료되며, 하나라도 실패하면 **모두 실패**하는 **불가분의 작업 단위**

**2. ACID Properties of Transactions**

- Atomicity (원자성)

> All-or-Nothing Principle: 일부 작업이 실패하면 트랜잭션은 **중단**되고, 변경사항은 **저장되지 않음**

- Consistency (일관성)

> Data Integrity: 실행 전후의 데이터베이스는 **일관된 상태**

- Isolation (격리성)

> Concurrent Transactions: 서로 독립적으로 실행

- Durability (지속성)

> Persistence: 커밋되면 그 결과는 **영구적**으로 유지되며, 시스템 오류가 발생해도 **유실되지 않**

**3. JDBC Transaction Control - Auto-Commit Mode**

- JDBC는 기본적으로 **auto-commit 모드**에서 동작

- 각 SQL 문장이 독립된 트랜잭션으로 처리

- INSERT, UPDATE, DELETE, SELECT와 같은 SQL 작업이 **자동으로 커밋**

- **한 번 커밋되면** 작업은 **롤백**할 수 없음

**4. Manual Transaction Management in JDBC**

- auto-commit을 false로 설정하면 여러 SQL 문장을 **하나의 트랜잭션**으로 묶을 수 있음

- commit(): 트랜잭션을 **완료**하고 모든 변경 사항을 **영구적**으로 저장

- rollback(): 트랜잭션 시작 이후의 모든 변경 사항을 **취소**하고 데이터베이스를 **이전 상태로 복원**

**5. Best Practices in Transaction Management**

- try-catch-finally 사용

- Auto-Commit 복원: 트랜잭션이 끝난 후, 추가로 독립적인 SQL 문장이 필요하면 **auto-commit**을 다시 true로 설정

- 트랜잭션 범위 최소화: 트랜잭션 코드 블록을 가능한 한 작게 유지

- 격리 수준 고려

**< Design patterns and best practices >**

**Introduction to Design Patterns**

- 소프트웨어 설계에서 반복적으로 발생하는 문제들에 대한 검증된 해결책

- **재사용 가능한 템플릿**을 제공하여 **견고하고, 확장 가능하며, 유지보수 가능한 시스템**을 구축

> Efficiency, Scalability, Communication

**Key Principles Achieved by Design Patterns**

- Open/Closed Principle: 확장에는 열려있고, 수정에는 닫혀 있어야 함

- Single Responsibility Principle: 각 클래스는 **하나의 책임**만 가져야 함

- DRY (Don’t Repeat Yourself): 공통 기능을 하나로 통합

**Activity – Identifying Design Issues**

**2. Poorly Designed Class Example**

- Tight Coupling

- Hard-coded Dependencies

- Single Responsibility Violation

- No Abstraction

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**Creational Patterns Overview**

- 객체 생성 방식을 효율적으로 관리하는 데 중점을 둔 디자인 패턴

> **Singleton**: 단 하나의 인스턴스만을 보장

> **Factory Method**: 객체 생성을 **하위 클래스**에 위임

> **Builder**: 복잡한 객체를 **단계별로** 구성

**Singleton Pattern**

**1. Introduction to Singleton Patterns**

- 클래스의 인스턴스가 애플리케이션 전체에서 **단 하나**만 존재하도록 보장하고, 그 인스턴스에 접근할 수 있는 **전역 접근점**을 제공

> Single Instance, Global Access, Controlled Instantiation**2. When to Use the Singleton Pattern**

- Shared Resource: 시스템 전체에서 **단 하나의 객체**가 필요

- Global State Management: **공유된 상태나 리소스**를 하나의 접근점에서 일관되게 관리**3. Advantages and Disadvantages**

- 장점

> Controlled Access: **공유 리소스**에 대한 접근을 제어

> Reduced Memory Overhead

> Consistency

- 단점

> Global State Issues: 디버깅과 테스트가 어려움

> Thread Safety: 멀티스레드 환경에서 Singleton 구현은 주의 깊게 설계 필요

> Difficulty in Unit Testing: 모의 객체(Mock)로 대체하기 어려움

**4. Implementation in Java**

- Eager Initialization (즉시 초기화): 클래스 로딩 시점에 인스턴스를 생성

- Lazy Initialization (지연 초기화): 인스턴스가 **필요할 때만** 생성

- Thread-Safe Singleton (스레드 안전 싱글톤): 멀티스레드 환경에서 동기화(synchronization)를 사용해 단일 인스턴스를 보장

**예시 ->** logger manager, configuration manager

**Factory Method Pattern**

**1. Introduction to Factory Method Pattern**

- 객체 생성을 위한 **인터페이스**를 제공하지만,  
**하위 클래스**가 생성될 객체의 **타입을 결정하도록** 하는 **생성 패턴**

> Encapsulation of Object Creation: 객체 생성 로직은 **팩토리 메서드**에 숨겨져 있음

> Global Access: 하위 클래스가 팩토리 메서드에서 생성할 객체의 타입을 정의할 수 있으므로, 새로운 객체 타입이 추가되어도 **클라이언트 코드**는 변경되지 않음

> Controlled Instantiation: **확장에는 열려 있고(Open)**, 기존 코드 수정에는 닫혀 있는(Closed) 상태

**2. When to Use the Factory Method Pattern**

- 생성될 객체의 정확한 타입을 모를 때

- 새로운 클래스를 추가하되 기존 코드를 수정하지 않아야 할 때

- 객체 생성 책임을 클라이언트 코드에서 위임하고 싶을 때

**3. Structure of Factory Method Pattern**

- Product (제품): 팩토리 메서드가 생성할 객체의 **타입**을 정의하는 인터페이스

- ConcreteProduct (구체적인 제품): Product를 구현하거나 확장한 클래스

- Creator (생성자): 객체를 생성하는 **팩토리 메서드**를 선언하는 **추상 클래스**

- ConcreteCreator (구체적인 생성자): Creator의 하위 클래스이며, **팩토리 메서드**를 구현해 **특정 ConcreteProduct** 객체를 생성

**4. Advantages of Factory Method Pattern**

- Loose Coupling: 클라이언트 코드는 **추상 Product**와 **Creator**에만 의존

- Scalability: 새로운 제품 타입을 추가하더라도 **클라이언트 코드를 수정할 필요가 없음**

- Encapsulation: 객체 생성 로직이 **팩토리 메서드**에 집중

**5. Disadvantages of Factory Method Pattern**

- Complexity: 여러 클래스를 도입

- Overhead for Small Applications: 작은 애플리케이션에서는 단순히 **new 키워드**를 사용하는 것이 더 실용적

**예시 ->** document editor, notification system

**Builder Pattern**

**1. Introduction to Builder Pattern**

- 복잡한 객체를 **단계별로** 구성할 수 있도록 하는 **생성 디자인 패턴**

- 다른 생성 패턴과 달리, **세부적인 제어**를 제공하여 **많은 선택적 매개변수**나 **구성이 필요한 객체**를 쉽게 생성

**2. Key Characteristics of the Builder Pattern**

- Step-by-Step Construction

- Separation of Representation and Construction: 객체 **생성 과정**은 객체의 **최종 표현**과 분리

- Immutability: 최종적으로 생성된 객체는 **불변 객체**로 만들어짐

- Readable Code

**3. Structure of the Builder Pattern**

> Product, Builder, Concrete Builder, Director

**4. When to Use the Builder Pattern**

- 복잡한 객체 생성

- 코드 가독성 향상

- 불변 객체 생성

**5. Advantages of the Builder Pattern**

- Readability, Flexibility, Immutability, Scalability

**6. Disadvantages of the Builder Pattern**

- Complexity, Overhead

**예시 ->** creating house

**Structural Design Patterns**

**1. Introduction to Structural Design Patterns**

- 클래스와 객체의 **구성(composition)** 방식을 다루는 디자인 패턴

**2. Key Characteristics of Structural Patterns**

- Composition Over Inheritance: **클래스 상속** 대신 객체 조합(구성)을 강조

- Enhance Relationships: 컴포넌트간의 관계를 **명확하고 효율적으로** 정의

- Improve Modularity: 클래스와 객체를 잘 **조직화**하고 결합함

- Adaptability: 기존 시스템에 **큰 변경 없이** 새로운 기능을 추가하거나 확장 가능

**3. When to Use Structural Patterns**

- 한 클래스의 **인터페이스**를 다른 클래스에 맞게 **변경**해야 할 때

- 시스템의 **구조를 단순화**하거나 **통일**

- **관계가 없는 클래스**들이 서로 함께 작동

- 한 객체에 **책임**을 동적으로 추가하면서도 **다른 객체에 영향을 주지 않아야** 할 때

**Adapter Pattern**

**1. Definition**

- 호환되지 않는 인터페이스들이 함께 작동할 수 있도록 해줌

- **두 클래스**의 **인터페이스**를 변환하여, **클라이언트**가 기대하는 인터페이스와 호환되도록 **다리(Bridge)** 역할

**2. Key Concepts of the Adapter Pattern**

- Bridge Between Incompatible Interfaces

- Single Responsibility Principle

- Flexibility

**3. When to Use the Adapter Pattern**

- 기존 클래스가 다른 인터페이스와 함께 작동하도록 해야 할 때

- 기존 클래스를 **재사용**하려고 하지만 해당 인터페이스가 애플리케이션과 **호환되지 않을 때**

- 클라이언트가 기대하는 인터페이스로 **변환이 필요**할 때

**4. Components of the Adapter Pattern**

> Target Interface: 클라이언트가 기대하는 인터페이스

> Adaptee: 기존에 존재하는 클래스이며, **타겟 인터페이스와 호환되지 않는 인터페이스**를 가짐

> Adapter: 타겟 인터페이스를 구현하며, 클라이언트의 요청을 **Adaptee**에게 전달하고 변환

> Client: 타겟 인터페이스와 상호작용

**5. Advantages of the Adapter Pattern**

> Reusability, Flexibility, Decoupling

**6. Disadvantages of the Adapter Pattern**

- Increased Complexity, Limited Use Cases

**예시 ->** media player, payment system

**Decorator Pattern**

**1. Definition**

- 같은 클래스의 다른 객체들에 영향을 주지 않으면서 개별 객체의 동작을 런타임(runtime)에 동적으로 추가 가능**2. Key Concepts of the Decorator Pattern**

- Dynamic Behavior Addition: 상속은 컴파일 타임(compile-time)에 기능을 확장하지만, 데코레이터 패턴은 **런타임(runtime)에 동작을 추가**

- Composition Over Inheritance: 객체 조합에 의존하며, 서브클래스를 생성하는 대신 데코레이터가 원래 객체를 래핑(wrap)하여 기능을 추가

- Transparent Interface: 데코레이터와 원래 객체는 동일한 인터페이스를 공유**3. When to Use the Decorator Pattern**

- 객체의 클래스를 수정하지 않고 **동적으로 객체의 동작을 추가하거나 수정**하고 싶을 때

- 동작 조합을 모두 포함하기 위해 서브클래스가 **폭발적으로 증가**할 우려가 있을 때

- 핵심 기능과 선택적 기능을 분리하여 단일 책임 원칙(Single Responsibility Principle)을 준수하고 싶을 때; 확장에는 열려 있고, 수정에는 닫혀 있음**4. Components of the Decorator Pattern**

- Component: 핵심 객체와 데코레이터 모두를 위한 공통 인터페이스

- Concrete Component: 동작이 확장되는 **원래 객체**

- Decorator: **컴포넌트 인터페이스**를 구현

- Concrete Decorators

**5. Benefits of the Decorator Pattern**

- Flexibility

- Open/Closed Principle

- Combining Behaviors

**6. Disadvantages of the Decorator Pattern**

- Increased Complexity

- Overhead

- Dependency on Composition

**예시 ->** coffee shop

**Composite Pattern**

**1. Definition**

- 객체를 **부분-전체 계층 구조**로 구성

- 클라이언트가 개별 객체와 객체의 집합(조합)을 **동일하게 취급**할 수 있게 함

**2.** **Key Concepts of the Composite Pattern**

- Tree Structure: 각 노드는 **리프(파일)** 또는 컴포지트(폴더)가 될 수 있음

- Uniformity: 리프 객체와 컴포지트 객체는 **공통 인터페이스**를 공유

- Recursive Composition

- Transparency: 공통 인터페이스를 통해 모든 구성 요소를 동일하게 다룸

**3. When to Use the Composite Pattern**

- **부분-전체 계층 구조**를 표현하고 싶을 때

- 개별 객체와 객체 집합을 **동일하게 취급**해야 할 때

- **파일 시스템, 조직도, 메뉴**와 같은 트리 구조를 작업할 때

**4. Components of the Composite Pattern**

- Component: 리프와 컴포지트 모두에 적용되는 공통 **인터페이스**

- Leaf: 조합의 개별 객체

- Composite: 컴포넌트 객체 그룹

- Client: 컴포넌트 인터페이스와 상호작용하며, 리프인지 컴포지트인지 신경 쓰지 않음

**5. Benefits of the Composite Pattern**

- Uniformity

- Scalability

- Flexibility

- Transparency

**6. Disadvantages of the Composite Pattern**

- Complexity

- Overhead

- Not Always Necessary**예시 ->** file system, organization structure

**Structural Design Patterns**

**1. Advantages of Structural Patterns**

- Improved Modularity

- Flexibility

- Reusability

- Separation of Concerns

**Behavior Design Patterns**

**1. Introduction to Behavior Design Patterns**

- **객체 간의 커뮤니케이션 및 상호작용**에 초점을 맞춘 설계 패턴

- **런타임 상호작용**을 다루어, 시스템이 변화하는 요구 사항에 유연하도록 함

**2. Key Characteristics of Behavioral Design Patterns**

- Object Interaction: 객체의 **책임**과 **커뮤니케이션 방법**을 정의

- Flexibility

- Reusability

- Separation of Concerns: 각 객체가 특정 작업에만 집중하도록 하고, 다른 책임은 협력 객체에게 **위임**

**3. When to Use Behavioral Patterns**

- 여러 객체가 **복잡하게 상호작용**하는 경우

- **런타임**에 객체의 동작을 동적으로 변경해야 하는 경우

- **책임**이나 **알고리즘**을 여러 객체 간에 **공유**하거나 **위임**해야 하는 경우

**4. Advantages of Behavioral Patterns**

> Encapsulation, Scalability, Loose Coupling, Code Reusability

**5. Disadvantages of Behavioral Patterns**

> Complexity, Learning Curve, Overhead

**Strategy Pattern**

**1. Definition**

- **알고리즘의 집합**을 정의하고 이를 각각의 클래스에 캡슐화하여 서로 **교환 가능**하게 함

**2. Key Concepts of the Strategy Pattern**

- Encapsulation of Algorithms

- Interchangeability

- Open/Closed Principle

- Delegation of Behavior**3. When to Use the Strategy Pattern**

- 특정 작업을 수행하기 위한 **여러 알고리즘이나 동작**이 존재하며, 실행 시점에 정확한 동작을 선택해야 할 때

- 동작을 결정하기 위해 큰 조건문(if-else, switch-case 등)을 사용하는 것을 피하고 싶을 때

- 알고리즘이나 동작을 쉽게 **교체 가능하고 확장 가능**하게 만들고 싶을 때**4. Components of the Strategy Pattern**

- Strategy Interface: 지원하는 모든 알고리즘이나 동작에 대한 **공통 인터페이스**

- Concrete Strategies: 전략 인터페이스에 정의된 **특정 알고리즘**

- Context: 전략 객체에 대한 참조를 유지하며, 선택된 전략에 동작을 위임

**5. Benefits of the Strategy Pattern**

- Eliminates Conditional Logic

- Flexibility

- Extensibility

- Reusability

**6. Disadvantages of the Strategy Pattern**

- Increased Complexity

- Client Responsibility

- Overhead

**예시 ->** GPS navigation system, payment processing system

**Observer Pattern**

**1. Definition**

- 객체 간의 **일대다(one-to-many)** 의존성을 정의

- 한 객체(주제, Subject)의 상태가 변경되면, 이를 의존하는 다른 객체들(옵저버, Observers)이 **자동으로 알림**을 받고 업데이트

**2. Key Concepts of the Observer Pattern**

- Subject: 관찰되는 객체

- Observers: 주제의 상태 변화에 관심이 있는 객체

- One-to-Many Relationship: 하나의 주제가 다수의 옵저버에게 변경 사항을 알릴 수 있도록 하며, 시스템의 **일관성**을 유지

**3. When to Use the Observer Pattern**

- 한 객체의 변경 사항이 다른 **의존 객체들**로 자동으로 전파되어야 할 때

- 객체가 상태 변화를 여러 **관계없는 객체들**에게 **브로드캐스트**해야 할 때

- 제와 옵저버 간의 느슨한 결합(loose coupling)을 통해 유연한 설계를 원할 때**4. Structure of the Observer Pattern**

- Subject: 옵저버 목록을 유지하며, 옵저버를 추가/삭제하는 메서드와 상태 변경 시 알리는 메서드를 제공

- Observer: 업데이트 메서드를 정의하는 **인터페이스**

- Concrete Subject

- Concrete Observer

**5. Benefits of the Observer Pattern**

- Loose Coupling

- Scalability

- Flexibility

- Event-Driven Programming

**6. Disadvantages of the Observer Pattern**

- Complexity

- Performance Overhead

- Memory Leaks

**예시 ->** news letter subscribe system, weather monitoring system

**Template Method Pattern**

**1. Definition**

- 알고리즘의 뼈대를 슈퍼클래스에서 정의하고, 세부 단계는 서브클래스에서 구현

**2. Key Concepts of the Template Method Pattern**

- Skeleton of Algorithm

- Abstract and Concrete Steps

> **추상 메서드**는 서브클래스가 반드시 구현해야 하는 알고리즘의 단계

> **구체 메서드**는 모든 서브클래스에서 동일하게 유지되는 단계로, 기본 클래스에 구현

- Inversion of Control: 알고리즘의 흐름은 기본 클래스가 제어하며, 특정 구현은 서브클래스에서 호출

- Code Reuse

- Hook Methods: 서브클래스가 전체 템플릿 메서드를 재정의하지 않고도 **행동을 수정하거나 확장3. When to Use the Template Method**

- **고정된 구조**를 가진 알고리즘에서 일부 단계만 **커스터마이징**해야 할 때

- 다양한 구현에서 **일관된 워크플로우**를 강제하고 싶을 때

- 공통 로직을 기본 클래스에 두어 **코드 재사용**을 촉진하고 싶을 때**4. Components of the Template Method**

- Abstract Class(Template Class): 알고리즘의 뼈대를 정의

- Template Method: 알고리즘의 흐름을 정의하는 **고수준 메서드**

- Abstract Methods: 알고리즘의 단계 중 서브클래스에서 반드시 구현해야 하는 부분

- Concrete Methods: 기본 클래스에서 구현되며, 모든 서브클래스에서 공통적으로 사용되는 단계

- Concrete Subclasses: 추상 메서드를 구현하여 특정 알고리즘 단계를 정의

**5. Benefits of the Template Method**

- Code Reuse

- Flexibility

- Consistency

- Ease of Maintenance

**6. Disadvantages of the Template Method**

- Inheritance Dependency: **상속**에 의존하므로, 조합(composition) 기반 솔루션에 비해 유연성이 떨어짐

- Overridden Behavior Complexity

- Limited Scalability: 템플릿 메서드나 단계들을 수정하거나 추가하려면, 모든 서브클래스를 변경

**예시 ->** recipe templete, document generation system