4. SQL - 데이터 관리

#0.강의/2.데이터베이스로드맵/1.입문

DDL - 테이블 생성1

이제 본격적으로 쇼핑몰 테이블을 설계해보자.

쇼핑몰 테이블 실전 설계

우리 쇼핑몰의 요구사항은 다음과 같다.

- 고객
  - 고객 id, 이름, 이메일, 비밀번호, 주소, 가입 시각이 관리되어야 한다.
- 상품
  - 상품 id, 이름, 설명, 가격, 재고 수량이 관리되어야 한다.
- 주문
  - 주문 id, 주문 고객, 주문 상품, 주문 수량, 주문 시각, 주문 상태가 관리되어야 한다.
  - 주문이 등록되면 최초의 주문 상태는 `주문접수` 상태가 된다.
  - 예제를 단순화 하기 위해 한 번의 주문 시에 한 종류의 상품만 주문할 수 있다. 한 종류의 상품을 여러 개 주문하는 것은 가능하다.

이제 배운 모든 것을 총동원하여 우리 쇼핑몰의 핵심 테이블 3개를 설계하고 생성해 보자.

> **SQL 소스 파일 참고**
> 강의 자료가 PDF 파일이라 복잡한 SQL 코드를 복사할 때 오류가 발생할 수 있다.
> 이 경우, 섹션 1. 강의 소개와 수업 자료 ➜ SQL 소스 파일을 다운로드해서 사용하자.

1. 고객 테이블 (`customers`)

```
CREATE TABLE customers (
    customer_id   INT AUTO_INCREMENT PRIMARY KEY,
    name          VARCHAR(50) NOT NULL,
    email         VARCHAR(100) NOT NULL UNIQUE,
    password      VARCHAR(255) NOT NULL,
    address       VARCHAR(255) NOT NULL,
    join_date     DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

- `customer_id`: 고객을 식별하는 기본 키(`PRIMARY KEY`). `AUTO_INCREMENT`로 자동 번호가 부여된다.
  - `AUTO_INCREMENT` 덕분에 데이터를 저장할 때마다 값이 1씩 자동으로 증가한다. 따라서 모든 행을 구분할 수 있다.
- `name`, `email`, `password`, `address`: 모두 필수 값이므로 `NOT NULL`을 설정했다.
- `email`: 고객마다 유일해야 하므로 `UNIQUE` 제약 조건을 추가했다.
- `join_date`: 가입 시각. 값을 따로 넣지 않으면 `DEFAULT` 설정에 따라 현재 시각(`CURRENT_TIMESTAMP`)이 자동으로 기록된다.

날짜와 기본값 설정 옵션

```
CREATE TABLE test (
    ...
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

이 기능을 사용하면 행이 추가되거나 변경된 날짜를 편리하게 관리할 수 있다.

- `DEFAULT CURRENT_TIMESTAMP`: 새로운 데이터 행(row)이 추가될 때, 해당 컬럼에 별도의 값을 지정하지 않으면 현재의 날짜와 시간이 자동으로 입력된다.
- `DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP`: 새로운 데이터 행(row)이 추가될 때는 물론이고, 같은 행의 컬럼 값이 변경되어 업데이트될 때, 이 컬럼의 값은 현재 날짜와 시간으로 자동 갱신된다.

2. 상품 테이블 (`products`)

```
CREATE TABLE products (
    product_id     INT AUTO_INCREMENT PRIMARY KEY,
    name           VARCHAR(100) NOT NULL,
    description    TEXT,
    price          INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0
);
```

- `product_id`: 상품을 식별하는 기본 키. `AUTO_INCREMENT`를 사용했다.
- `price`: 금액이므로 `INT` 타입을 사용했고, 필수 값이므로 `NOT NULL`이다.
- `stock_quantity`: 재고 수량. 필수 값이지만, 값을 넣지 않으면 기본적으로 `0`이 되도록 `DEFAULT`를 설정했다.
- `description`: 상품 설명은 길 수 있으므로 `TEXT` 타입을 사용했고, 필수는 아니므로 `NOT NULL` 제약 조건을 걸지 않았다.

3. 주문 테이블 (`orders`)

```
CREATE TABLE orders (
    order_id     INT AUTO_INCREMENT PRIMARY KEY,
    customer_id  INT NOT NULL,
    product_id   INT NOT NULL,
    quantity     INT NOT NULL,
    order_date   DATETIME DEFAULT CURRENT_TIMESTAMP,
    status       VARCHAR(20) NOT NULL DEFAULT '주문접수',

    CONSTRAINT fk_orders_customers FOREIGN KEY (customer_id)
        REFERENCES customers(customer_id),

    CONSTRAINT fk_orders_products FOREIGN KEY (product_id)
        REFERENCES products(product_id)

);
```

- `order_id`: 주문을 식별하는 기본 키. `AUTO_INCREMENT`를 사용했다.
- `customer_id`: **가장 중요한 부분.** 이 주문을 한 고객이 누구인지를 나타낸다.
  - 이 값은 `customers` 테이블의 `customer_id`를 참조한다. 이곳을 확인하면 고객의 이름을 찾을 수 있다.
- `product_id`: **가장 중요한 부분.** 어떤 상품을 주문했는지 나타낸다.
  - 이 값은 `products` 테이블의 `product_id`를 참조한다. 이곳을 확인하면 상품명을 찾을 수 있다.

외래 키 제약조건

외래 키 제약조건은 다음과 같은 명세를 가진다.

```
CONSTRAINT [제약조건_이름]
FOREIGN KEY ([자식_테이블의_컬럼명])
REFERENCES [부모_테이블명]([부모_테이블의_컬럼명])
[ON DELETE 옵션] [ON UPDATE 옵션]
```

`fk_orders_customers`

- `CONSTRAINT fk_orders_customers FOREIGN KEY (customer_id) REFERENCES customers(customer_id)`
- `orders` 테이블의 `customer_id`를 `customers` 테이블의 `customer_id`와 연결하는 외래 키 설정이다. 이로써 두 테이블 사이에 '관계'가 형성된다. 이제 존재하지 않는 **고객 ID**로 주문을 넣으려는 시도는 데이터베이스 단에서 차단된다.
- 외래 키 제약조건의 이름에는 관계 있는 테이블의 참조 순서와 이름을 사용했다. `orders -> customers` 이렇게 하면 이름으로 관계를 쉽게 파악할 수 있다.

fk_orders_products

- `CONSTRAINT fk_orders_products FOREIGN KEY (product_id) REFERENCES products(product_id)`
- `orders` 테이블의 `product_id`를 `products` 테이블의 `product_id`와 연결하는 외래 키 설정이다. 이로써 두 테이블 사이에 '관계'가 형성된다. 이제 존재하지 않는 **상품 ID**로 주문을 넣으려는 시도는 데이터베이스 단에서 차단된다.
- 외래 키 제약조건의 이름에는 관계 있는 테이블의 이름을 사용해서 `fk_orders_products`라고 했다. 이렇게 하면 이름으로 관계를 쉽게 파악할 수 있다. `orders -> products`

> **실무에서 다루는 주문 테이블들은 더 복잡하다.**
> 지금은 데이터베이스 설계를 배우는 것이 목적이 아니기 때문에, 예제를 쉽게 이해할 수 있도록 테이블의 수를 최소화 했다. 실무에 가까운 복잡한 설계 예시는 **실전 데이터베이스 설계 강의**에서 다루겠다.

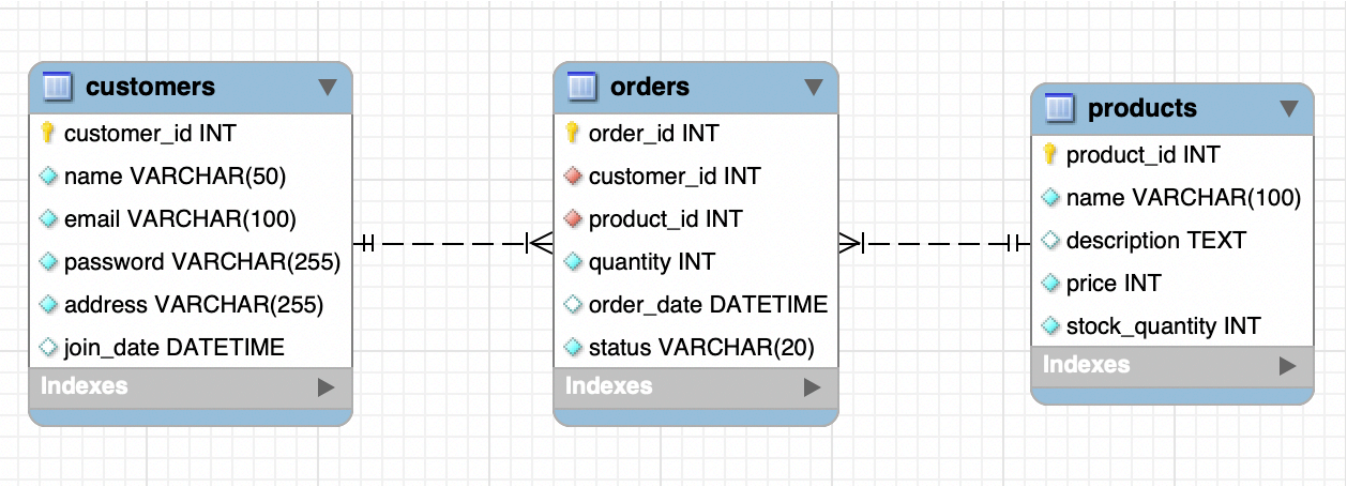
DDL - 테이블 생성2

ERD: 데이터베이스의 지도, 관계를 한눈에 파악하기

이렇게 만들어둔 테이블을 한눈에 파악할 수 있는 지도를 ERD(Entity Relationship Diagram)라 한다.

MySQL 워크벤치로 ERD 만들기

- MySQL 워크벤치 실행
- 상단 메뉴 바에서 Database > Reverse Engineer.. 실행
- Continue 버튼으로 계속 넘어가다가 우리가 만든 `my_shop` 데이터베이스 선택
- 이후 계속 넘어가면 다음과 같은 ERD 확인 가능



왜 ERD가 필요할까? 지금까지 우리는 `customers`, `products`, `orders`라는 3개의 테이블을 만들었다. 테이블이 몇 개 안 될 때는 머릿속으로 그 구조와 관계를 그릴 수 있지만, 실무에서는 수십, 수백 개의 테이블이 복잡하게 얽히게 된다. 이때, 테이블과 그들 사이의 관계(Relationship)를 그림으로 표현한 설계도가 없다면 우리는 데이터의 구조라는 거대한 숲에서 길을 잃고 말 것이다. ERD가 바로 그 설계도, 즉 '데이터베이스의 지도' 역할을 한다.

우리가 만든 쇼핑몰의 관계를 간단히 표현하면 다음과 같다.

1. **고객(Customers)과 주문(Orders)의 관계**
   - 한 명의 고객(1)은 여러 번 주문(N)할 수 있다.
   - 이를 **1:N (일대다, One-to-Many)** 관계라고 한다.
   - `customers (1)` -----< `orders (N)`

2. **상품(Products)과 주문(Orders)의 관계**
   - 하나의 상품(1)은 여러 번 주문(N)될 수 있다.
   - 이 또한 **1:N (일대다, One-to-Many)** 관계다.
   - `products (1)` -----< `orders (N)`

> 까마귀발 표기법

> `---<` 기호는 까마귀발 표기법(Crow's Foot Notation)을 간단히 표현한 것으로, `<`가 '다수(Many)' 쪽을 의미한다.

이 관계를 종합해 보면, `orders` 테이블은 `customers`와 `products`의 중간에서 두 테이블을 연결하는 중요한 역할을 하고 있음을 알 수 있다.

- `orders` 테이블의 `customer_id` (FK)가 `customers` 테이블의 `customer_id` (PK)를 바라보고,
- `orders` 테이블의 `product_id` (FK)가 `products` 테이블의 `product_id` (PK)를 바라보는 구조다.

이처럼 ERD를 통해 데이터의 전체적인 청사진을 이해하고, 개발자나 기획자 등 다른 팀원과 원활하게 소통할 수 있다. 좋은 데이터베이스 설계는 좋은 ERD를 그리는 것에서 시작된다.

> **연관 관계와 데이터베이스를 설계**
> 테이블 간의 연관 관계는 1:N, N:1, N:N, 1:1 같은 다양한 관계가 있다.
> 연관 관계와 ERD를 설계에 관한 자세한 내용은 데이터베이스 설계 강의에서 다룬다.

테이블과 컬럼 이름 규칙

기본 규칙 (백틱 미사용)

백틱으로 감싸지 않는 이름은 가장 일반적이며 권장되는 방식이다. 예) `product_name`

- **허용 문자**
  - 영문 대소문자 (`a-z`, `A-Z`)
  - 숫자 (`0-9`)
  - **밑줄 (`_`)**
  - **달러 기호 (`$`)**
  - UTF-8과 같은 다국어 문자 (예: **한글**)
- **제한 사항**
  - 이름이 **숫자로 시작할 수 없다.**
  - MySQL **예약어** (예: `SELECT`, `TABLE`, `ORDER`)는 사용할 수 없다.
  - **총 길이는 64자**를 넘을 수 없다.

확장 규칙 (백틱 사용)

이름을 백틱(`` ` ``)으로 감싸면 훨씬 더 자유로운 이름 생성이 가능하다.

- **허용 문자**: **거의 모든 문자**가 허용된다.
  - **공백** (예: `user name`)
  - **하이픈** (`-`) 및 기타 특수문자 (예: `item-code`, `@email`)

- **숫자로 시작**하는 이름 (예: `2025_report`)
- **MySQL 예약어** (예: `order`, `select`)

> 백틱(`)은 키보드 숫자1 왼쪽에 있다.

권장되는 명명 규칙 (Best Practice)

기술적인 규칙을 넘어, 실무에서는 다음과 같은 규칙을 따르는 것이 좋다.

- **영문 소문자와 밑줄 사용**: `user_orders`, `product_name`처럼 영문 소문자와 단어를 구분하는 밑줄(`_`)을 사용하는 것이 가장 일반적이다. (스네이크 케이스, snake_case)
- **일관성 유지**: 프로젝트 내에서 명명 규칙을 하나로 정하고 일관되게 적용하는 것이 중요하다.
- **예약어 피하기**: 백틱을 사용하면 예약어도 이름으로 쓸 수 있지만, 혼란을 피하기 위해 사용하지 않는 것이 좋다.
- **간결하고 명확하게**: 이름만 보고도 데이터의 의미를 파악할 수 있도록 만든다. (예: `prdct_nm` 보다는 `product_name`)
- **다국어 이름 피하기**: 한글 이름도 기술적으로는 가능하지만, 호환성 및 인코딩 문제를 예방하기 위해 영문으로 작성하는 것을 권장한다.

DDL - 테이블 변경, 제거

ALTER TABLE: 이미 만든 테이블의 구조 변경

쇼핑몰을 한창 운영하다 보면, 이미 만들어 둔 테이블의 구조를 변경해야 할 때가 생긴다. 예를 들어, 고객에게 포인트를 지급하는 제도를 새로 도입해서 `customers` 테이블에 `point` 열을 추가해야 할 수 있다. 이럴 때 사용하는 것이 `ALTER TABLE` 명령어다.

열(Column) 추가하기 - ADD COLUMN

`customers` 테이블에 `point` 열을 추가해보자.

```
ALTER TABLE customers
ADD COLUMN point INT NOT NULL DEFAULT 0;
```

- `DESC customers`로 확인해보면 `point` 열이 추가된 것을 볼 수 있다.

열(Column) 데이터 타입 변경하기 - MODIFY COLUMN

`customers` 테이블의 `address` 열을 더 길게 만들어야 한다면?

```
ALTER TABLE customers
MODIFY COLUMN address VARCHAR(500) NOT NULL;
```

- `DESC customers`로 확인해보면 `address` 컬럼의 길이가 500으로 변경된 것을 확인할 수 있다.

열(Column) 삭제하기 - DROP COLUMN

`point` 제도를 없애기로 했다면?

```
ALTER TABLE customers
DROP COLUMN point;
```

- `DESC customers`로 확인해보면 `point` 컬럼이 제거된 것을 확인할 수 있다.

> **실무 경고**
> `ALTER TABLE`은 유용한 기능이지만, 조심해서 사용해야 한다. 수백만, 수천만 건의 데이터가 들어있는 거대한 테이블의 구조를 변경하는 작업은 엄청나게 많은 시간과 시스템 자원을 소모한다. 작업 중에는 테이블이 잠겨서 서비스가 일시적으로 멈출 수도 있다. 따라서 실무에서는 사용자가 적은 새벽 시간을 이용해 점검 시간에 맞춰 작업하는 것이 일반적이다.
> 참고로 최신 버전의 MySQL에서는 열을 추가하는 것 정도는 실시간으로 적용해도 크게 문제가 되지는 않는다.
> 그래도 대부분의 라이브 변경 작업은 데이터베이스 버전에 맞는 매뉴얼을 확인한 다음에 어느정도 잠금이 발생하는지, 라이브 상황에 수행할 수 있는지 확인한 다음에 실행하는 것을 권장한다.

DROP TABLE vs TRUNCATE TABLE

테이블을 다루다 보면 테이블의 내용을 비워야 할 때가 있다. 이때 `DROP`과 `TRUNCATE`는 비슷해 보이지만 아주 큰 차이가 있다.

`DROP TABLE` : **테이블의 존재 자체를 삭제한다.**

- `DROP TABLE orders;` 를 실행하면, `orders` 테이블의 모든 데이터는 물론, `orders` 라는 테이블의 구조(설계도)까지 완전히 사라진다. 테이블을 다시 사용하려면 `CREATE TABLE`부터 다시 해야 한다. 마치 건물을 통째로 철거하는 것과 같다.

`TRUNCATE TABLE` : **테이블의 구조는 남기고, 내부 데이터만 모두 삭제한다.**

- `TRUNCATE TABLE orders;` 를 실행하면, `orders` 테이블 안의 모든 주문 데이터가 순식간에 사라진다. 하지만 `orders` 라는 테이블의 구조(열, 제약조건 등)는 그대로 남아있어서, 바로 새로운 데이터를 `INSERT` 할 수 있다. 건물의 내부만 싹 비우고 뼈대는 그대로 두는 것과 같다.
- **특징**
  - `DELETE FROM orders;` (WHERE 절 없는 DELETE)와 결과적으로는 같아 보이지만, `TRUNCATE`가 훨씬 빠르다. `DELETE`는 한 줄씩 지우면서 삭제 기록을 남기는 반면, `TRUNCATE`는 테이블을 초기화하는 개념이라 내부 처리 방식이 더 간단하고 빠르다.
  - `TRUNCATE`는 `AUTO_INCREMENT` 값도 초기화한다. 만약 `orders` 테이블에 1000개의 주문이 있어서 다음 `order_id`가 1001일 차례였다면, `TRUNCATE` 이후에는 다시 1부터 시작한다. (DELETE는 `AUTO_INCREMENT` 값을 초기화하지 않는다.)

이 둘의 차이를 명확히 이해하고 상황에 맞게 사용하는 것은 매우 중요하다. 테스트 데이터를 모두 지우고 처음부터 다시 시작하고 싶을 때는 `TRUNCATE`가 유용하고, 테이블 자체가 더 이상 필요 없을 때는 `DROP`을 사용한다.

제약 조건 무시하기

`products` 테이블은 `orders`에서 참조하고 있다. 이렇게 참조 당하는 테이블은 `DROP`, `TRUNCATE`를 할 수 없다.

```
DROP TABLE products;
```

[실행 결과]

```
Error Code: 3730. Cannot drop table 'products' referenced by a foreign key
constraint 'fk_orders_products' on table 'orders'.
```

- 실행 결과를 보면 `orders` 테이블에 있는 `fk_orders_products` 외래 키 제약 조건 때문에 테이블을 제거할 수 없다는 뜻이다.
- 만약 이 테이블을 제거할 수 있게 둔다면 `orders` 테이블 입장에서는 **주문된 상품의 정보를 찾을 수 없는(상품 없는 주문)** 심각한 문제가 나타난다!

```
TRUNCATE TABLE products;
```

[실행 결과]

```
Error Code: 1701. Cannot truncate a table referenced in a foreign key
constraint (`my_shop`.`orders`, CONSTRAINT `fk_orders_products`)
```

- `TRUNCATE`의 경우도 마찬가지다. `TRUNCATE`는 내부의 데이터를 확인하지 않고, 모든 데이터를 빠르게 제거한다. 따라서 이 테이블의 모든 데이터를 제거한다면 `orders` 테이블 입장에서는 **주문된 상품의 정보를 찾을 수 없는(상품 없는 주문)** 심각한 문제가 발생할 수 있다.

만약 이런 외래 키 제약조건을 임시로 무시하고 싶다면 외래 키 체크를 비활성화 하면 된다.

```
SET FOREIGN_KEY_CHECKS = 0; -- 비활성화
SET FOREIGN_KEY_CHECKS = 1; -- 활성화
```

```
SET FOREIGN_KEY_CHECKS = 0; -- 비활성화

TRUNCATE TABLE products;

SET FOREIGN_KEY_CHECKS = 1; -- 활성화
```

주의사항

- `FOREIGN_KEY_CHECKS`를 비활성화하면 데이터 무결성이 깨질 수 있으므로, 필요한 작업을 마친 후에는 반드시 다시 활성화해야 한다.
- `SET XXX` 방식의 설정은 데이터베이스 접속이 연결되는 동안만 유효하다. 연결이 끊기면 설정이 사라진다. 따라서 다시 접속하는 경우 설정을 다시 해야 한다.

이제 건물의 뼈대를 세웠으니(DDL), 그 안에 실제 내용물을 채워 넣을 차례다(DML). 쇼핑몰에 신상품을 등록하고, 새로운 회원을 가입시키며, 변경된 정보를 수정하는 모든 과정이 DML을 통해 이루어진다.

DML - 등록

실제 운영 환경에서 가장 빈번하게 일어나는 작업들이 바로 DML 명령어들이다. 앞서 맛보기로 경험했던 `INSERT`, `UPDATE`, `DELETE`를 이제는 깊이 있게 파고들어 보자.

`INSERT`는 테이블에 새로운 데이터 행(row)을 추가하는 명령어다. 신규 상품을 등록하고 새로운 회원을 가입시키는 상황을 통해 `INSERT`의 사용법을 익혀보자.

INSERT 문법

INSERT 문의 기본 문법은 다음과 같다.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- `table_name`: 데이터를 추가할 테이블의 이름이다.
- `(column1, column2, ...)`: 값을 지정할 열의 목록이다. 이 목록을 생략하면 테이블의 모든 열에 순서대로 값을 넣어야 한다.
- `VALUES (value1, value2, ...)`: 지정된 각 열에 삽입될 값이다. 열 목록과 값 목록은 순서와 개수가 정확히 일치해야 한다.

방법 1: 모든 열(컬럼) 추가하기

가장 기본적인 방법은 데이터를 추가할 열의 목록을 생략하는 것이다. 그리고 열의 순서대로 모든 값을 (VALUES)에 제공하는 것이다. 우리가 앞서 설계한 `customers` 테이블에 새로운 회원을 가입시켜 보자.

```
INSERT INTO customers VALUES (NULL, '강감찬', 'kang@example.com',
'hashed_password_123', '서울시 관악구', '2025-06-11 10:30:00');
INSERT INTO customers VALUES (NULL, '이순신', 'lee@example.com',
'hashed_password_123', '서울시 관악구', '2025-06-12 10:30:00');
```

- `customer_id`에 `NULL`을 넣은 이유: 이 열은 `AUTO_INCREMENT`로 설정했기 때문이다. `NULL`을 명시하거나 아예 열 목록에서 생략하면 MySQL이 알아서 다음 순번의 숫자를 자동으로 채워준다.
- `join_date`에 직접 날짜와 시간을 입력했다.

[실행 결과]

```
SELECT * FROM customers;
```

customer_id	name	email	password	address	join_date
1	강감찬	kang@example.com	hashed_password_123	서울시 관악구	2025-06-11 10:30:00

2	이순신	lee@example.com	hashed_password_123	서울시 관악구	2025-06-12 10:30:00

> ⚠ 실무에서 `password`는 암호화해서 저장해야 한다.

방법 2: 원하는 특정 열만 골라서 데이터 추가하기

모든 열에 데이터를 넣을 필요가 없을 때, 예를 들어 선택 입력 항목이 있을 때 사용하는 방식이다.

예를 들어 `AUTO_INCREMENT`나 `DEFAULT` 값이 설정된 열은 굳이 명시할 필요가 없다. 이들을 생략하면 코드가 훨씬 간결해진다.

```
INSERT INTO customers (name, email, password, address)
VALUES ('세종대왕', 'sejong@example.com', 'hashed_password_456', '서울시 종로구');
```

[실행 결과]

```
SELECT * FROM customers;
```

customer_id	name	email	password	address	join_date
1	강감찬	kang@example.com	hashed_password_123	서울시 관악구	2025-06-11 10:30:00
2	이순신	lee@example.com	hashed_password_123	서울시 관악구	2025-06-12 10:30:00
3	세종대왕	sejong@example.com	hashed_password_456	서울시 종로구	(현재 시각)

위 코드를 실행하면 `customer_id`는 자동으로 다음 번호가 부여되고, `join_date`는 `DEFAULT` 값인 현재 시각(`CURRENT_TIMESTAMP`)이 자동으로 기록된다. 더 간단하다.

`products` 테이블에 신상품을 등록하는 예시를 보자. `description`(상품 설명) 열은 필수(`NOT NULL`)가 아니었다.

```sql
INSERT INTO products (name, price, stock_quantity)
VALUES ('베이직 반팔 티셔츠', 19900, 200);
INSERT INTO products (name, price, stock_quantity)
VALUES ('초록색 긴팔 티셔츠', 30000, 50);
```

[실행 결과]

```sql
SELECT * FROM products;
```

product_id	name	description	price	stock_quantity
1	베이직 반팔 티셔츠	NULL	19900	200
2	초록색 긴팔 티셔츠	NULL	30000	50

이렇게 하면 `product_id`는 자동으로 생성되고, `name`, `price`, `stock_quantity`에는 지정된 값이 들어간다. 값을 지정하지 않은 `description` 열에는 `NULL` 값이 자동으로 들어간다. 이처럼 필요한 데이터만 골라서 넣는 것이 실무에서 가장 흔하게 사용하는 방식이다.

> **실무 이야기: 열 목록을 사용하자**
>
> 열 목록을 생략하고 `INSERT INTO products VALUES (NULL, '베이직 반팔 티셔츠', ...)`와 같이 값만 순서대로 나열하는 방법은 추천하지 않는다. 나중에 테이블 구조가 변경(예: `ALTER TABLE`로 열 추가)되면 기존의 `INSERT` 코드는 모두 오류를 일으키기 때문이다. 항상 데이터를 추가할 열의 목록을 명시적으로 작성하는 것이 안전하고 좋은 습관이다. 물론 학습이나 테스트를 위한 것이라면 편의상 이런 방식을 사용해도 괜찮다.

방법 3: 한 번에 등록하기

```sql
INSERT INTO products (name, price, stock_quantity) VALUES
('검정 양말', 5000, 100),
('갈색 양말', 5000, 150),
('흰색 양말', 5000, 200);
```

다음과 같이 VALUES에 여러 항목을 한 번에 넣어서 하나의 쿼리로 많은 데이터를 등록할 수 있다. 중간에 쉼표(`,`)를

사용해서 데이터를 구분하면 된다.

[실행 결과]

```
SELECT * FROM products;
```

product_id	name	description	price	stock_quantity
1	베이직 반팔 티셔츠	NULL	19900	200
2	초록색 긴팔 티셔츠	NULL	30000	50
3	검정 양말	NULL	5000	100
4	갈색 양말	NULL	5000	150
5	흰색 양말	NULL	5000	200

DML - 수정

`UPDATE`는 이미 존재하는 데이터의 내용을 수정하는 명령어다.

UPDATE 문법

UPDATE 문의 기본 문법은 다음과 같다.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- `table_name`: 데이터를 수정할 테이블의 이름이다.
- `SET column1 = value1, ...`: 수정할 열과 새로운 값이다. 쉼표(,)를 사용해 여러 열을 한 번에 수정할 수 있다.
- `WHERE condition`: 수정할 행을 식별하는 조건이다. 이 부분을 생략하면 테이블의 모든 행이 수정되므로 극도로 주의해야 한다.

UPDATE와 SET 기본 구문

`product_id=1`인 베이직 반팔 티셔츠를 조회해보자.

```
SELECT * FROM products
WHERE product_id = 1
```

- WHERE를 사용해서 원하는 행을 지정할 수 있다.

product_id	name	description	price	stock_quantity
1	베이직 반팔 티셔츠	NULL	19900	200

이 상품은 '베이직 반팔 티셔츠'이다. 이 상품의 가격을 9800원으로 할인하고, 재고를 580개로 변경해보자.

```
UPDATE products
SET price = 9800, stock_quantity = 580
WHERE product_id = 1;
```

[실행 결과]

```
SELECT * FROM products
WHERE product_id = 1
```

product_id	name	description	price	stock_quantity
1	베이직 반팔 티셔츠	NULL	9800	580

가격은 19900 ➜ 9800원으로, 재고는 200 ➜ 580으로 변경된 것을 확인할 수 있다.

SQL 안전 업데이트 모드

데이터를 변경하는 것은 생각보다 위험할 수 있다.

만약 "베이직 반팔 티셔츠" 상품만 특가로 판매하기 위해 가격을 990원으로 변경하려고 했는데, 실수로 WHERE를 생략한다면 어떻게 될까?

```
```

```
UPDATE products
SET price = 990; -- WHERE product_id = 1; -- 실수로 생략
```

이러면 모든 우리 쇼핑몰 모든 상품의 가격이 990원이 된다! 단 한 번의 실수로 서비스가 망할 수도 있는 것이다.
한번 과감하게 실행해보자.

[실행 결과]

```
Error Code: 1175. You are using safe update mode and you tried to update a
table without a WHERE that uses a KEY column.  To disable safe mode, toggle
the option in Preferences -> SQL Editor and reconnect.
```

- 다행히도 실행되지 않고, 오류가 발생한다. 오류 메시지를 보면 안전 업데이트 모드를 사용중이라고 나온다.

이런 실수를 방지하기 위해 MySQL은 "안전 업데이트 모드"를 제공한다.

```
SELECT @@SQL_SAFE_UPDATES;
```

- **실행 결과**: 1
- 실행 결과 1은 안전 업데이트 모드가 활성화 상태이고, 0은 비활성화 상태이다.

안전 업데이트 모드 활성화 상태에서는 데이터를 **변경하거나 삭제**할 때 WHERE 절에 기본 키 컬럼을 반드시 지정해야 한다.
(또는 `LIMIT`를 사용해서 변경하거나 삭제할 데이터의 양을 조절해야 한다. `LIMIT`는 뒤에서 배운다.)

그래서 기본 키 컬럼을 사용하지 않은 다음과 같은 SQL도 실행이 되지 않는다.

```
UPDATE products
SET price = 990
WHERE name = '베이직 반팔 티셔츠'; -- name 컬럼을 사용했다.
```

[실행 결과]

```
Error Code: 1175. You are using safe update mode and you tried to update a
table without a WHERE that uses a KEY column.  To disable safe mode, toggle
the option in Preferences -> SQL Editor and reconnect.
```

참고로 MySQL이 제공하는 안전 업데이트 모드의 기본 설정은 비활성화(0) 상태이다.
그런데 MySQL 워크벤치는 이런 실수를 방지하기 위해 MySQL에 접근할 때 안전 업데이트 설정을 활성화 한다.

정리하면 다음과 같다.

- **MySQL 서버 (전역):** 기본적으로 OFF (0)
- **애플리케이션 (커넥션):** 기본적으로 OFF (0)
- **MySQL 워크벤치:** 기본적으로 ON (1)

다음 문구를 실행하면 안전 업데이트 모드의 설정을 임시로 변경할 수 있다.

```
SET SQL_SAFE_UPDATES = 0; -- 안전 업데이트 모드 끄기
SET SQL_SAFE_UPDATES = 1; -- 안전 업데이트 모드 활성화
```

MySQL 워크벤치의 기본 설정을 변경하고 싶다면

- 윈도우: 메뉴바 ➜ Edit ➜ Preference ➜ SQL Editor ➜ 맨 아래의 Safe Updates 체크 해제
- MAC: 메뉴바 ➜ MySQLWorkbench ➜ Settings... ➜ SQL Editor ➜ 맨 아래의 Safe Updates 체크 해제

안전 모드를 제거하고 다시 UPDATE 문을 실행해보자.

```
SET SQL_SAFE_UPDATES = 0; -- 안전 모드 제거

UPDATE products
SET price = 990;

SET SQL_SAFE_UPDATES = 1; -- 안전 모드 활성화
```

- 안전 모드를 끄고, 필요한 기능을 실행하고 나면 안전 모드를 다시 활성화해주는 것이 안전하다.

[실행 결과]

```
SELECT * FROM products;
```

product_id	name	description	price	stock_quantity
1	베이직 반팔 티셔츠	NULL	990	580
2	초록색 긴팔 티셔츠	NULL	990	50

3	검정 양말	NULL	990	100
4	갈색 양말	NULL	990	150
5	흰색 양말	NULL	990	200

- 모든 상품의 가격이 990원이 된 것을 확인할 수 있다.

위 명령어를 실행하는 순간, `WHERE` 조건이 없으므로 `products` 테이블의 **모든 상품 가격이 990원으로 바뀌는 재앙**이 발생한다. 특정 상품 하나만 특가 할인 행사를 하려다 쇼핑몰 전체를 거의 공짜로 내주는 셈이다. 데이터는 복구하기 매우 어렵거나 불가능할 수 있으며, 이는 곧 비즈니스의 막대한 손실로 이어진다.

특히 MySQL 워크벤치와 같이 SQL을 직접 실행할 수 있는 도구에서 SQL을 직접 다룰 때 이런 문제가 자주 발생한다. 그리고 이런 문제를 예방하기 위해 MySQL 워크벤치는 안전 모드를 기본으로 적용한다.

변경 대상을 먼저 확인하는 습관을 들이자
예방이 최선이다. `UPDATE` 나 `DELETE` 문을 실행하기 전에는, **반드시 동일한** `WHERE` **절을 사용한** `SELECT` **문을 먼저 실행해서 내가 변경하려는 데이터가 정확히 맞는지를 눈으로 확인하는 습관**을 들여야 한다!!!

```
-- 1단계: 변경할 대상을 눈으로 먼저 확인한다.
SELECT * FROM products
WHERE name = '베이직 반팔 티셔츠';
-- 아, 이 상품이 맞구나. 확인 완료.

-- 2단계: 확인된 대상에 대해서만 UPDATE를 실행한다.
SET SQL_SAFE_UPDATES = 0; -- 안전 모드 제거

UPDATE products
SET price = 19800
WHERE name = '베이직 반팔 티셔츠';

SET SQL_SAFE_UPDATES = 1; -- 안전 모드 활성화
```

> ⚠ **실무에서는 SQL을 사용해서 데이터를 변경하거나 삭제할 때, 반드시 SELECT 쿼리를 먼저 수행해서 내가 수정할 데이터를 꼭! 먼저 확인하자.**

DML - 삭제

`DELETE FROM`은 테이블에서 행을 삭제하는 명령어다.

DELETE 문법

DELETE 문의 기본 문법은 다음과 같다.

```
DELETE FROM table_name
WHERE condition;
```

- `table_name`: 데이터를 삭제할 테이블의 이름이다.
- `WHERE condition`: 삭제할 행을 식별하는 조건이다. UPDATE와 마찬가지로 이 부분을 생략하면 테이블의 모든 데이터가 삭제되는 재앙이 발생하므로 반드시 작성해야 한다.

DELETE 기본 구문

'베이직 반팔 티셔츠' 상품이 단종되어 우리 쇼핑몰에서 더 이상 판매하지 않기로 했다고 가정해 보자.

```
SELECT * FROM products
WHERE product_id = 1; -- 상품 ID가 1번이라고 가정
```

product_id	name	description	price	stock_quantity
1	베이직 반팔 티셔츠	NULL	19800	580

DELETE FROM으로 삭제하자.

```
DELETE FROM products
WHERE product_id = 1; -- 상품 ID가 1번이라고 가정
```

[실행 결과]

```
```

```sql
SELECT * FROM products;
```

product_id	name	description	price	stock_quantity
2	초록색 긴팔 티셔츠	NULL	990	50
3	검정 양말	NULL	990	100
4	갈색 양말	NULL	990	150
5	흰색 양말	NULL	990	200

- `product_id=1` 베이직 반팔 티셔츠 상품이 제거된 것을 확인할 수 있다.

`WHERE` 절의 절대적인 중요성

`DELETE`는 `UPDATE`보다 더 파괴적이다. `UPDATE`는 데이터를 잘못 바꿔도 원래 값을 추적할 여지라도 있지만, `DELETE`는 행 자체를 없애버린다. `DELETE` 문에서 `WHERE` 절을 빠뜨리는 것은 **핵폭탄 발사 버튼을 누르는 것과 같다.**

```sql
-- 절대로, 절대로 따라하지 말 것!
DELETE FROM customers;
```

[실행 결과]

```sql
SELECT * FROM customers;
```

customer_id	name	email	password	address	join_date

만약 안전 업데이트 모드가 활성화되어 있지 않다면 쇼핑몰의 모든 고객 정보가 순식간에 사라진다. 변명의 여지가 없는 대형 사고다. `UPDATE`와 마찬가지로, `DELETE` 역시 **실행 전** `SELECT` **문으로 삭제 대상을 확인하는 것**이 철칙이다.

전체 `UPDATE`**,** `DELETE`**를 실행했을 때 벌어지는 끔찍한 상황과 대처법**

1. **사고 발생**: `WHERE` 절 없는 `UPDATE`를 실행했다는 사실을 깨닫는다. 심장이 내려앉는다.
2. **즉시 보고 및 서비스 점검**: 즉시 DBA를 호출하고, 상급자에게 상황을 보고한다. 가능하다면 추가적인 피해를 막

기 위해 서비스를 점검 상태로 전환한다. 전문적인 지식을 갖춘 DBA가 있다면 데이터베이스 복원 기능으로 빠르게 복구할 수도 있다.

3. **백업 확인**: 가장 먼저 확인해야 할 것은 가장 최근의 데이터베이스 백업이다. 정기적인 백업이 있었다면, 백업 시점 이후의 데이터 유실을 감수하고 복구(Restore)를 진행할 수 있다. 이것이 백업이 중요한 이유다.
4. **로그 분석**: 백업이 없다면, 데이터베이스 로그를 분석하여 변경 이전의 값을 추적하는 복잡한 작업을 시도해 볼 수 있으나, 이는 매우 어렵고 시간이 오래 걸린다.

다시 한번 비교하기: DELETE vs TRUNCATE

앞서 `TRUNCATE`를 배웠다. 둘 다 데이터를 삭제하는데, 무슨 차이가 있을까?

구분	`DELETE FROM table;`	`TRUNCATE TABLE table;`
종류	**DML** (데이터 조작어)	**DDL** (데이터 정의어)
처리 방식	한 줄씩, 조건에 따라 삭제 가능 (`WHERE` 사용 가능)	테이블 전체를 한 번에 삭제 (`WHERE` 사용 불가)
속도	느림 (각 행의 삭제를 기록)	매우 빠름 (테이블을 잘라내고 새로 만드는 개념)
`AUTO_INCREMENT`	초기화되지 않음	**1부터 다시 시작하도록 초기화됨**
되돌리기(Rollback)	트랜잭션 내에서 `ROLLBACK` 가능	`ROLLBACK` 불가능 (즉시 적용)

언제 무엇을 써야 할까?

- "탈퇴한 회원 한 명의 정보만 지우고 싶다" 또는 "특정 조건에 맞는 주문 기록만 삭제하고 싶다" 와 같이 **선별적인 삭제**가 필요할 때는 `DELETE`를 사용해야 한다. 일반적인 비즈니스 로직은 항상 `DELETE`를 사용한다.
- "테스트용으로 넣었던 수백만 건의 데이터를 모두 지우고 처음부터 다시 시작하고 싶다" 와 같이 **테이블의 모든 데이터를 깨끗하게 비울 목적**이라면 `TRUNCATE`가 훨씬 빠르고 효율적인 선택이다.

제약 조건 활용

테이블을 만들 때 설정한 제약 조건(Constraints)은 실수로 혹은 의도적으로 잘못된 데이터가 들어오는 것을 막는 강력한 수문장 역할을 한다. 데이터베이스에 데이터를 저장하거나 변경할 때 이 규칙을 어기면 어떻게 될까? 데이터베이스는 단호하게 입력을 거부하며 오류를 발생시킨다.

예제를 실행하기 전에 먼저 기존 테이블을 초기화하자.

```
SET FOREIGN_KEY_CHECKS = 0; -- 비활성화
truncate table products;
truncate table customers;
truncate table orders;
SET FOREIGN_KEY_CHECKS = 1; -- 활성화
```

1. NOT NULL 제약 조건 위반: "필수 항목을 입력해주세요."

가장 기본적이고 단순한 제약 조건은 `NOT NULL`이다.
`NULL`은 값이 없다는 뜻이다. `NOT NULL`은 '`NULL`을 허용하지 않는다'는 뜻으로, 필수 입력 항목을 지정할 때 사용한다.

`customers` 테이블의 `name` 열은 `NOT NULL`로 설정되어 있으므로, 회원 가입 시 이름은 반드시 입력해야 한다. 만약 이름을 빼고 `INSERT`를 시도하면 어떻게 될까?

```
-- `name` 열을 빼고 INSERT를 시도한다.
INSERT INTO customers (email, password, address)
VALUES ('noname@example.com', 'password123', '서울시 마포구');
```

[실행 결과]

```
Error Code: 1364. Field 'name' doesn't have a default value
```

데이터베이스는 "`name` 필드에 기본값이 없다"라는 오류를 반환하며 INSERT를 거부한다. `name` 열은 `NOT NULL`인데 값을 주지 않았고, `DEFAULT`로 지정된 값도 없으므로 어떤 데이터를 넣어야 할지 알 수 없기 때문이다. 이처럼 `NOT NULL` 제약 조건은 데이터의 완전성을 보장하는 첫 번째 방어선이다.

2. UNIQUE 제약 조건 위반: "이미 사용 중인 이메일입니다."

가장 흔하게 마주치는 제약 조건 오류 중 하나는 `UNIQUE` 키 위반이다. `customers` 테이블의 `email` 열에는 `UNIQUE` 제약 조건이 걸려 있으므로, 모든 고객은 서로 다른 이메일 주소를 가져야 한다. 만약 기존에 있는 이메일로 새로운 회원을 가입시키려고 하면 어떻게 될까?

```
INSERT INTO customers (name, email, password, address)
VALUES ('강감찬', 'kang@example.com', 'new_password_789', '서울시 강남구');
```

- 강감찬 회원을 저장하는 쿼리는 문제 없이 실행되고, 데이터베이스에 저장된다.

[실행 결과]

```
SELECT * FROM customers;
```

customer_id	name	email	password	address	join_date
1	강감찬	kang@example.com	new_password_789	서울시 강남구	(저장 시각)

```
-- 'kang@example.com'은 이미 '강감찬' 고객이 사용 중인 이메일이다.
INSERT INTO customers (name, email, password, address)
VALUES ('홍길동', 'kang@example.com', 'new_password_123', '서울시 송파구');
```

- 추가로 저장할 홍길동 회원의 이메일 `kang@example.com`은 이미 강감찬 고객이 사용 중인 이메일이다.

[실행 결과]

```
Error Code: 1062. Duplicate entry 'kang@example.com' for key 'customers.email'
```

홍길동 회원을 저장할 때 데이터베이스는 "`customers.email`키에 중복된 값 `kang@example.com`이 있습니다"라는 명확한 오류 메시지를 반환하며 `INSERT` 요청을 거부한다. 이처럼 `UNIQUE` 제약 조건은 데이터의 고유성을 보장하는 중요한 역할을 한다.

3. 외래 키(FK) 제약 조건: 관계의 무결성 지키기

이제 관계형 데이터베이스의 핵심, 외래 키(Foreign Key) 제약 조건을 살펴보자. `orders` 테이블은 `customers` 테이블과 `products` 테이블에 의존하는 '자식 테이블'이다. 즉, `orders` 테이블에 데이터를 추가하려면, 그 주문을 한 고객(`customer_id`)과 주문된 상품(`product_id`)이 반드시 `customers`와 `products` 테이블에 실제로 존재해야 한다.

먼저, 정상적인 주문 데이터를 입력해 보자.

`customers` 테이블에 `customer_id`가 1인 고객이 있고, `products` 테이블에 `product_id`가 1인 '베이직 반팔 티셔츠'가 있다고 가정하자.

고객은 앞서 저장한 강감찬 고객(`customer_id=1`)이 있으니 상품만 하나 추가하자.

```
INSERT INTO products (name, price, stock_quantity)
VALUES ('베이직 반팔 티셔츠', 19900, 200);
```

[실행 결과]

```
SELECT * FROM products;
```

product_id	name	description	price	stock_quantity
1	베이직 반팔 티셔츠	NULL	19900	200

```
-- 1번 고객이 1번 상품을 1개 주문한다.
INSERT INTO orders (customer_id, product_id, quantity)
VALUES (1, 1, 1);
```

[실행 결과]

```
SELECT * FROM orders;
```

order_id	customer_id	product_id	quantity	order_date	status
1	1	1	1	(주문 시각)	주문접수

이 `INSERT` 문은 성공적으로 실행된다. `customer_id` 1과 `product_id` 1이 부모 테이블에 모두 존재하기 때문이다. `order_id`와 `order_date`, `status`는 각각 `AUTO_INCREMENT`와 `DEFAULT` 설정에 따라 자동으로 채워졌다.

만약 실패한다면 `customers` 테이블에 `customer_id`가 없거나, `products` 테이블에 `product_id`가 존재하지 않을 것이다.

4. 외래 키(FK) 제약 조건 위반: "존재하지 않는 고객의 주문입니다."

이번에는 **존재하지 않는 고객**의 주문을 넣어보자. `customers` 테이블에는 `customer_id`가 999인 고객이 없다.

```
-- 존재하지 않는 999번 고객이 1번 상품을 1개 주문하려고 시도한다.
INSERT INTO orders (customer_id, product_id, quantity)
VALUES (999, 1, 1);
```

[실행 결과]

```
Error Code: 1452. Cannot add or update a child row: a foreign key constraint
fails (`my_shop`.`orders`, CONSTRAINT `fk_orders_customers` FOREIGN KEY
(`customer_id`) REFERENCES `customers` (`customer_id`))
```

데이터베이스는 "자식 행을 추가하거나 업데이트할 수 없습니다: `fk_orders_customers` 외래 키 제약 조건이 실패했습니다"라는 오류를 뱉어낸다. 이 메시지는 `orders` 테이블에 `INSERT`하려던 값 `999`가 부모 테이블인 `customers`의 `customer_id`에 존재하지 않아서 외래 키 제약조건을 위반했다는 뜻이다.

이처럼 외래 키 제약 조건은 **데이터의 정합성**과 **무결성**을 지키는 핵심 장치다. 이 기능 덕분에 우리는 유령 회원이나 단종된 상품이 주문되는 것과 같은 데이터 불일치 상황을 원천적으로 방지할 수 있다.

> 외래 키 제약조건을 포함한 제약조건들은 **INSERT 뿐만 아니라 UPDATE, DELETE 등 모든 상황의 데이터 불일치를 원천 방지한다.**

문제와 풀이

> 문제와 풀이 진행 방법
>
> 1. **반드시 스스로 문제를 풀어보자**
> 2. 문제를 10분 이상 고민한다면 기존 내용을 복습하고 문제를 풀어보자.
> 3. 그래도 문제가 풀리지 않으면 정답을 보고 문제를 푼 이후에, 스스로 문제를 다시 풀어보자.

문제1: 데이터베이스와 테이블 생성하기

[문제]

쇼핑몰 프로젝트를 위해 `my_test` 데이터베이스를 생성하고, 해당 데이터베이스를 사용하도록 선택해라. 그 후, 고객 정보를 저장할 `members` 테이블을 생성해라. `members` 테이블은 다음과 같은 열(column)을 가진다.

- `id`: 정수(INT), 기본키 설정
- `name`: 최대 50글자의 문자열(VARCHAR), NOT NULL 제약 조건
- `join_date`: 날짜(DATE)

`DESC members;` 실행 시 다음과 같은 결과가 출력된다.

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	
name	varchar(50)	NO		NULL	
join_date	date	YES		NULL	

[정답]

```sql
-- 1. my_test 데이터베이스 생성
CREATE DATABASE my_test;

-- 2. my_test 데이터베이스 선택
USE my_test;

-- 3. members 테이블 생성
CREATE TABLE members (
    id INT PRIMARY KEY, -- 기본키 설정
    name VARCHAR(50) NOT NULL,
    join_date DATE
);
```

```sql
-- 4. 테이블 구조 확인
DESC members;
```

문제2: 데이터 추가 및 조회하기 (INSERT, SELECT)

[문제]

1번 문제에서 생성한 `members` 테이블에 아래 두 명의 회원 데이터를 추가하고, 테이블의 전체 내용을 조회해라.

1. ID: `1`, 이름: `션`, 가입일: `2025-01-10`
2. ID: `2`, 이름: `네이트`, 가입일: `2025-02-15`

`SELECT * FROM members;` 실행 시 다음과 같은 결과가 출력된다.

id	name	join_date
1	션	2025-01-10
2	네이트	2025-02-15

[정답]

```sql
-- 1. 첫 번째 회원 데이터 추가
INSERT INTO members (id, name, join_date)
VALUES (1, '션', '2025-01-10');

-- 2. 두 번째 회원 데이터 추가
INSERT INTO members (id, name, join_date)
VALUES (2, '네이트', '2025-02-15');

-- 3. 전체 데이터 조회
SELECT * FROM members;
```

문제3: 데이터 수정 및 삭제하기 (UPDATE, DELETE)

[문제]

2번 문제에서 추가한 데이터에 다음 두 가지 작업을 수행하고, 최종 결과를 조회해라.

1. `id`가 2인 회원 '네이트'의 이름을 '네이트2'로 변경해라.
2. `id`가 1인 회원 '션'의 정보를 삭제해라.

`SELECT * FROM members;` 실행 시 다음과 같은 결과가 출력된다.

id	name	join_date
2	네이트2	2025-02-15

[정답]

```sql
-- 1. 이름 변경 (UPDATE)
UPDATE members
SET name = '네이트2'
WHERE id = 2;

-- 2. 회원 정보 삭제 (DELETE)
DELETE FROM members
WHERE id = 1;

-- 3. 최종 데이터 조회
SELECT * FROM members;
```

문제4: 제약 조건을 포함한 테이블 생성하기

[문제]

쇼핑몰의 `products`(상품) 테이블을 다음 요구사항과 제약 조건에 맞게 생성하고, 테이블 구조를 확인해라.
(데이터베이스가 `my_test`로 다르기 때문에 기존 테이블에 영향을 주지 않는다)

- `product_id`: 정수, 자동으로 1씩 증가하는 기본 키(PRIMARY KEY)
- `product_name`: 최대 100글자의 문자열, 비어 있을 수 없음(NOT NULL)
- `product_code`: 최대 20글자의 문자열, 값이 중복될 수 없음(UNIQUE)
- `price`: 정수, 비어 있을 수 없음(NOT NULL)
- `stock_count`: 정수, 비어 있을 수 없으며(NOT NULL), 값을 지정하지 않으면 기본으로 0이 입력됨(DEFAULT)

`DESC products;` 실행 시 다음과 같은 결과가 출력된다.

Field	Type	Null	Key	Default	Extra
product_id	int	NO	PRI	NULL	auto_increment
product_name	varchar(100)	NO		NULL	
product_code	varchar(20)	YES	UNI	NULL	
price	int	NO		NULL	
stock_count	int	NO		0	

[정답]

```
CREATE TABLE products (
    product_id   INT AUTO_INCREMENT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    product_code VARCHAR(20) UNIQUE,
    price        INT NOT NULL,
    stock_count  INT NOT NULL DEFAULT 0
);

DESC products;
```

문제5: 외래 키(Foreign Key)로 테이블 관계 맺기

[문제]

고객(`customers`)과 주문(`orders`) 테이블을 생성해라. `orders` 테이블의 `customer_id`는 `customers` 테이블의 `customer_id`를 참조하는 외래 키(Foreign Key) 관계를 맺어야 한다.

- `customers` **테이블**
  - `customer_id`: 정수, 기본 키, 자동 증가
  - `name`: 문자열(50자), 필수
- `orders` **테이블**
  - `order_id`: 정수, 자동 증가 기본 키
  - `customer_id`: 정수, 필수
  - `order_date`: 날짜와 시간(DATETIME), 기본값은 현재 시각
  - `FOREIGN KEY` 설정: `orders.customer_id`가 `customers.customer_id`를 참조하도록 한다.

두 테이블을 생성한 후, '홍길동' 고객과 그 고객이 주문한 데이터를 각각 1개씩 추가하고, 두 테이블의 전체 내용을 조회해라.

`SELECT * FROM customers;`

customer_id	name
1	홍길동

`SELECT * FROM orders;`

order_id	customer_id	order_date
1	1	(주문 시각)

[정답]

```sql
-- customers 테이블 생성
CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    name        VARCHAR(50) NOT NULL
);
```

```sql
-- orders 테이블 생성
CREATE TABLE orders (
    order_id    INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date  DATETIME DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT fk_orders_customers FOREIGN KEY (customer_id)
        REFERENCES customers(customer_id)
);

-- 고객 데이터 추가
INSERT INTO customers (name) VALUES ('홍길동');

-- 주문 데이터 추가 (방금 추가된 홍길동 고객의 customer_id는 1)
INSERT INTO orders (customer_id) VALUES (1);

-- 데이터 조회
SELECT * FROM customers;
SELECT * FROM orders;
```

문제6: 제약 조건 위반 상황 만들기

[문제]

5번 문제에서 생성한 테이블들을 이용해 아래 두 가지 잘못된 데이터 추가를 시도하고, 왜 실패하는지 그 이유를 설명해라.

1. `customers` 테이블에 존재하지 않는 `customer_id` (예: 999)를 사용하여 `orders` 테이블에 새로운 주문을 추가해라.
2. `customers` 테이블에 고객을 추가할 때, 필수 항목인 `name`을 빼고 추가해라.

[정답]

```sql
-- 시도 1: 존재하지 않는 고객의 주문 추가
INSERT INTO orders (customer_id) VALUES (999);

-- 시도 2: 필수 항목(name) 누락
```

```sql
INSERT INTO customers (customer_id) VALUES (2);
```

[실행 결과]

시도 1 결과:

```
Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`my_test`.`orders`, CONSTRAINT `fk_orders_customers` FOREIGN KEY (`customer_id`) REFERENCES `customers` (`customer_id`))
```

- **실패 이유**: 외래 키(Foreign Key) 제약 조건 위반. `orders` 테이블에 주문을 추가하려면, `customer_id`에 해당하는 값이 부모 테이블인 `customers`에 반드시 존재해야 한다. `customer_id` 999는 `customers` 테이블에 없으므로 데이터 추가가 거부된다.

시도 2 결과:

```
Error Code: 1364. Field 'name' doesn't have a default value
```

- **실패 이유**: `NOT NULL` 제약 조건 위반. `customers` 테이블의 `name` 열은 `NOT NULL`로 설정되어 있어 값을 반드시 입력해야 한다. `name` 값을 지정하지 않고 추가를 시도했기 때문에 데이터 추가가 거부된다.

문제와 풀이가 끝나면 다음 명령을 실행해서 사용 데이터베이스를 원래대로 변경하자.

```sql
USE my_shop;
```

정리

DDL - 테이블 생성

- `customers, products, orders` 3개 테이블을 만들어 쇼핑몰 핵심 데이터를 저장
- 기본 키는 `AUTO_INCREMENT`로 자동 번호를 부여
- 날짜 시간 컬럼은 `DEFAULT CURRENT_TIMESTAMP`(+ `ON UPDATE`)로 자동 관리
- 외래 키(FK)로 `orders.customer_id → customers.customer_id`, `orders.product_id → products.product_id` 관계 설정

- ERD를 그려 테이블 관계(1:N)를 한눈에 파악

DDL - 테이블 변경, 제거

- 구조 변경은 `ALTER TABLE` `ADD / MODIFY / DROP COLUMN`으로 수행
- 대용량 테이블은 변경 시 잠금, 속도를 고려해 새벽에 작업
- `DROP TABLE`은 테이블 자체를, `TRUNCATE TABLE`은 데이터만 삭제한다. FK가 걸려 있으면 두 명령 모두 차단된다.
- FK를 잠시 무시하려면 `SET FOREIGN_KEY_CHECKS=0` 후 작업하고 즉시 1로 돌려라.

DML - 등록

- `INSERT`는 (1) 모든 컬럼, (2) 필요 컬럼만, (3) 다중 VALUES 세 가지 패턴이 있다.
- 열 목록을 명시해서 스키마 변경 시 오류를 막아라.
- `AUTO_INCREMENT`, `DEFAULT`가 있는 컬럼은 생략가능하다.
  - 물론 NULL을 입력할 수 있는 컬럼도 생략할 수 있다.

DML - 수정

- `UPDATE … SET … WHERE` 형태로 수정해라. WHERE를 반드시 써라.
- MySQL Workbench는 기본으로 "안전 업데이트 모드(`SQL_SAFE_UPDATES=1`)"를 켜둔다.
- 변경 전 동일 WHERE로 `SELECT`해 확인해라.
- 대량 수정이 필요하면 `SET SQL_SAFE_UPDATES=0`로 잠시 끄고 끝나면 다시 켜라.

DML - 삭제

- `DELETE FROM … WHERE`로 행을 삭제해라. WHERE를 반드시 써라. 전부 날아간다.
- `DELETE`(DML) vs `TRUNCATE`(DDL) 차이를 명확히 기억해라.
  - `DELETE`는 느리지만 조건 삭제,트랜잭션 롤백 가능, AUTO_INCREMENT 유지.
  - `TRUNCATE`는 빠르지만 조건 불가,롤백 불가, AUTO_INCREMENT 초기화.
- 실행 전 반드시 `SELECT`로 대상 행을 확인해라.

제약 조건 활용

- `NOT NULL`: 필수값을 비워 두면 오류 발생
- `UNIQUE` 키: 중복 입력 시 오류 발생
- FK: 부모에 없는 값을 넣으면 오류 발생
- INSERT, UPDATE, DELETE 모두 제약조건을 우선 검증하므로 데이터 무결성이 보호된다.