

6. 도메인 계층 (Domain Layer)

도메인 모델과 엔티티의 정의

🔗 1. 도메인 모델(Domain Model)이란?

◆ 정의

현실 세계의 비즈니스 개념, 규칙, 행동을 소프트웨어 안에서 표현한 객체 모델
즉, 시스템이 해결하고자 하는 문제 영역(=도메인)의 핵심 개념과 로직을
클래스로 명확하게 표현한 것이 도메인 모델이다.

◆ 역할

기능	설명
💡 개념 표현	고객, 주문, 결제, 배송, 재고 등 비즈니스 개체와 행위를 표현
📦 상태 관리	도메인 객체의 필드, 상태값 등
💬 비즈니스 규칙	"VIP 고객은 배송비 무료", "5개 이상 구매 시 할인" 같은 규칙
⚠️ 불변성, 일관성	상태 전이를 직접 관리하며 데이터 무결성을 보장

📦 2. 엔티티(Entity)란?

◆ 정의

도메인 모델 중에서 식별자(ID)를 가지고 있으며, 상태 변경이 가능한 객체
즉, 동일성을 유지해야 하는 비즈니스 객체가 시간이 지나도 같은 것으로 인식될 수 있도록
ID 기반으로 관리되는 객체를 엔티티(Entity)라고 한다.

◆ 엔티티의 특징

특징	설명
ID로 식별	동일한 속성이라도 ID가 다르면 다른 객체
🔄 상태 변경 가능	시간이 흐르면서 내부 값이 변함
⚙️ 식별자 중심 설계	equals/hashCode는 ID 기준으로 작성

3. 도메인 모델 ≠ 무조건 엔티티

- 도메인 모델 = 엔티티 + 밸류 객체(Value Object) + 도메인 서비스 + 이벤트
- 엔티티는 도메인 모델의 한 구성 요소

4. 도메인 모델 vs 엔티티 비교

구분	도메인 모델	엔티티
정의	문제 도메인을 코드로 표현한 전체 구조	ID로 식별되는 상태 변경 가능한 객체
예시	주문(Order), 재고(Inventory), 배송(Shipping) 규칙 전체	주문(Order), 사용자(User)
포함 요소	Entity, Value Object, 도메인 서비스, 이벤트 등	필드, 메서드, ID
설계 목적	의미, 로직, 규칙 중심	식별성과 영속성 중심
단위	추상 개념, 도메인 전체	구체적 인스턴스 중심

5. 실전 예시: User 엔티티

```
1 @Entity
2 public class User {
3     @Id @GeneratedValue
4     private Long id;
5
6     private String name;
7     private String email;
8     private String password;
9
10    protected User() {}
11
12    public User(String name, String email, String password) {
13        this.name = name;
14        this.email = email;
15        this.password = password;
16    }
17
18    public void changePassword(String oldPw, String newPw) {
19        if (!this.password.equals(oldPw)) {
20            throw new IllegalArgumentException("기존 비밀번호가 다릅니다");
21        }
22        this.password = newPw;
23    }
24 }
```

분석	설명
@Entity	엔티티임을 나타냄 (JPA)

분석	설명
@Id	식별자 필드
비즈니스 규칙 메서드	<code>changePassword()</code> 는 도메인 로직
생성자 캡슐화	<code>new User(...)</code> 로 도메인 규칙에 맞게 객체 생성

6. 도메인 모델 설계 시 핵심 원칙

원칙	설명
✅ 불변성과 일관성 유지	외부에서 setter 노출 금지, 생성자나 메서드로 상태 전이 제한
✅ 의미 있는 메서드 작성	<code>setStatus()</code> ❌ → <code>markAsPaid()</code> ✅
✅ 도메인이 책임진다	서비스가 상태를 바꾸는 것이 아니라, 도메인이 스스로 바뀌어야 함

7. 안티패턴 주의

잘못된 예	문제
setter만 있는 빈 껍데기 객체	도메인 객체의 의미 상실 (Anemic Domain Model)
모든 로직을 Service에 위임	도메인 로직이 분산되어 재사용성과 일관성 저하
상태를 외부에서 직접 변경	일관성, 유효성 무너짐 → <code>order.setStatus("완료")</code> ❌

✅ 마무리 요약

항목	도메인 모델	엔티티
위치	도메인 계층 전체	그 중 식별 가능한 개체
역할	규칙, 개념, 흐름 모델링	식별 가능한 상태 관리 주체
구성	Entity, VO, Domain Service	필드, ID, 도메인 메서드
특징	비즈니스 의미 중심	ID 기반 동등성, 영속 대상
주의점	로직 캡슐화 필수	상태 전이 책임 유지해야 함

@Entity, @Id, @GeneratedValue 개념

1 @Entity — 도메인 객체 + 영속성 객체

✓ 정의

JPA에서 DB 테이블에 매핑되는 클래스이자,
DDD에서 식별자(ID)를 가진 도메인 객체

✓ 특징

- @Entity 어노테이션으로 선언
- 필수로 식별자 필드(@Id)가 있어야 함
- 영속성 컨텍스트(1차 캐시)에서 관리됨
- 상태 변경 가능 (Mutable)
- 동일성 비교: ID 기준

✓ 예시

```
1 @Entity
2 public class User {
3     @Id @GeneratedValue
4     private Long id;
5
6     private String name;
7     private String email;
8
9     // 도메인 로직 포함 가능
10    public void changeEmail(String newEmail) {
11        this.email = newEmail;
12    }
13 }
```

2 @Id — 식별자 지정

✓ 정의

JPA에서 Entity를 식별하기 위한 기본 키 필드에 지정하는 어노테이션

✓ 특징

- JPA는 @Id 가 있어야 Entity로 인식함
- @GeneratedValue 를 통해 자동 생성 가능
- Entity의 동등성/해시 비교 기준이 됨

```
1 @Id
2 @GeneratedValue
3 private Long id;
```

3 Value Object (VO, 값 객체)

✓ 정의

DDD에서 불변이며, 의미를 가지는 값 자체를 표현하는 객체
ID가 없고, 상태보다 값의 내용 자체에 의미가 있음

✓ 특징

항목	설명
불변성	생성 이후 절대 변경 불가 (setter 없음)
ID 없음	식별자가 아니라 값 자체로 비교
equals/hashCode	내용 기반 동등성 비교
JPA에서 @Embeddable, @Embedded 사용 가능	
한 엔티티 내부에 내장됨 (별도 테이블 없음)	

✓ 예시 1: 일반 VO

```
1 public record Email(String value) {
2     public Email {
3         if (!value.matches(".*@.*")) {
4             throw new IllegalArgumentException("잘못된 이메일");
5         }
6     }
7 }
```

이건 Java 16+ `record` 로 표현한 순수 VO.
값 자체에 의미가 있고, 불변성 + 유효성을 보장함.

✓ 예시 2: JPA와 함께 쓰는 VO

```
1 @Embeddable
2 public class Address {
3
4     private String city;
5     private String street;
6
7     protected Address() {} // JPA용
8
9     public Address(String city, String street) {
10         this.city = city;
11         this.street = street;
12     }
13 }
```

```

13
14 // equals, hashCode 반드시 override (내용 기반 비교)
15 }

```

```

1 @Entity
2 public class User {
3     @Id @GeneratedValue
4     private Long id;
5
6     @Embedded
7     private Address address;
8 }

```

Entity vs Value Object 비교표

항목	@Entity	Value Object
식별자	있음 (@Id)	없음
상태 변경	가능 (Mutable)	불가능 (Immutable)
영속성	개별 테이블 존재	내장형 (@Embeddable)
동등성	ID 기준	내용 기준 (equals)
주 사용 위치	도메인 핵심 모델, Root	엔티티 내부 속성 (ex: Address, Money)
책임	비즈니스 상태 관리	상태 설명, 표현, 검증 역할

설계 시 주의할 점

잘못된 설계	문제점
VO에 setter 제공	불변성 무너짐 → 값이 바뀌면 VO가 아님
Entity가 VO 없이 모든 값을 갖음	도메인 모델이 지나치게 커지고, 응집력 약화
VO가 ID를 가짐	VO의 철학과 모순됨. ID가 있다면 Entity임

마무리 정리

요소	역할
@Entity	상태 관리, 식별 가능한 핵심 도메인 객체
@Id	식별자 지정 및 영속성 관리 기준
Value Object	값 자체로 의미를 가지며, 불변성을 갖는 도메인 속성 표현 수단





도메인 이벤트, 도메인 서비스

1. 도메인 이벤트 (Domain Event)

정의

도메인 모델 내부에서 중요한 일이 발생했음을 알리는 메시지 객체
즉, 도메인 상태 변화에 따라 “무언가 일어났다”는 사실을 시스템 내부에 전달하는 객체

목적

목적	설명
 관심사 분리	도메인 로직과 후처리 로직을 분리 (ex. 알림 전송, 로깅)
 비즈니스 개념 반영	"회원가입이 완료되었다", "결제가 승인되었다" 등 실제 업무 용어 모델링
 비동기 처리 가능	이벤트 기반으로 외부 작업을 지연 실행하거나 메시지 브로커 연계 가능
 확장성	새로운 후처리 로직 추가 시 도메인 수정 없이 리스너만 추가하면 됨

실전 예시

이벤트 클래스

```
1 public class UserRegisteredEvent {
2     private final Long userId;
3     private final String email;
4
5     public UserRegisteredEvent(Long userId, String email) {
6         this.userId = userId;
7         this.email = email;
8     }
9
10    // getter
11 }
```

도메인 로직 내부에서 이벤트 발행

```
1 public class User {
2     public User(String email, String password) {
3         this.email = email;
4         this.password = password;
5         // 도메인 이벤트 등록
6         DomainEvents.raise(new UserRegisteredEvent(this.id, this.email));
7     }
8 }
```

`DomainEvents` 는 이벤트 발행을 추상화한 유틸. (Spring에서는 `ApplicationEventPublisher` 로 연동 가능)

📧 리스너에서 후처리

```
1 @Component
2 public class UserEventHandler {
3
4     @EventListener
5     public void handle(UserRegisteredEvent event) {
6         emailSender.sendWelcomeEmail(event.getEmail());
7     }
8 }
```

✓ Spring과 통합

- `@EventListener` 기반의 동기 이벤트 처리
- `@TransactionalEventListener` 로 트랜잭션 완료 후에 이벤트 처리 가능
- Kafka, RabbitMQ 연동 시 외부 브로커 발행기로 연계

🏠 2. 도메인 서비스 (Domain Service)

✓ 정의

하나의 엔티티나 밸류 객체에 넣기 어려운 비즈니스 규칙을
도메인 계층 내에서 캡슐화한 순수 자바 서비스 객체

✓ 목적

목적	설명
💬 비즈니스 규칙 분리	특정 엔티티 하나에 속하지 않는 복합 도메인 로직
🔄 엔티티 간 협력 조정	복수 객체 협업, 외부 상태 판단 등
📦 순수 도메인 계층 유지	기술 의존 없이 오직 규칙만 포함

✓ 언제 사용하는가?

상황	도메인 서비스 대상
한 엔티티 안에 넣기 애매할 때	할인 정책, 배송비 계산
외부 정보 기반 판단이 필요할 때	회원 등급 정책, 거리 계산 등
도메인 계층에 계산만 필요할 때	금액 비교, 비밀번호 강도 검사 등

✓ 실전 예시

```
1 public class DiscountPolicy {
2
3     public int calculateDiscount(Order order, Member member) {
4         if (member.isVip()) return order.totalPrice() * 10 / 100;
5         return 0;
6     }
7 }
```

```
1 public class Order {
2     public int applyDiscount(DiscountPolicy discountPolicy, Member member) {
3         int discount = discountPolicy.calculateDiscount(this, member);
4         return this.totalPrice - discount;
5     }
6 }
```

도메인 서비스는 상태를 가지지 않으며, **순수 계산/판단 로직만 포함**한다.

3. 구조적 비교

항목	도메인 이벤트	도메인 서비스
정의	무언가 발생했음을 알리는 도메인 메시지	비즈니스 규칙 중 복수 객체 또는 외부 판단이 필요한 로직
위치	도메인 로직의 결과/후처리	도메인 로직의 핵심 계산/판단
주체	“알려주는 역할”	“계산하고 판단하는 역할”
트리거	엔티티 내부, 유스케이스 종료 시	유스케이스 내부, 도메인에서 직접 호출
상태	단순 데이터 보유	계산 로직만 포함, 상태 없음
Spring 통합	@EventListener, @TransactionalEventListener	아무 의존 없음, 단위 테스트 쉬움

✓ 마무리 요약

개념	정의	사용 시점
도메인 이벤트	도메인 내에서 발생한 사실을 알리는 객체	엔티티 상태 변경 이후, 후처리 분리
도메인 서비스	비즈니스 규칙을 순수하게 계산하는 서비스 객체	복수 객체 협력, 외부 판단이 필요할 때

도메인 주도 설계의 원칙

1. DDD란 무엇인가?

Domain-Driven Design이란,

소프트웨어를 비즈니스의 핵심 도메인 모델에 집중하여
의미 있는 언어(Ubiquitous Language)와 구조로 설계하는 방법론이다.

단순히 "엔티티-서비스-레포지토리 구조"가 아니라,
"도메인 자체를 중심에 두고 시스템을 이해하고 조직한다"는 철학적 접근이 핵심이다.

2. DDD의 핵심 원칙 10가지

1 도메인 모델은 비즈니스의 중심 개념이다

- 도메인 모델은 DB 구조가 아니라,
비즈니스 규칙과 개념을 표현한 코드 구조
- 현실 세계의 핵심 개념(ex: 주문, 결제, 할인 정책)을 객체로 추상화

2 Ubiquitous Language (공통 언어)를 사용하라

- 개발자, 기획자, 비즈니스 전문가가 동일한 용어 체계로 말하고 모델링
- 코드, 문서, 회의, 커밋 메시지까지 용어를 일치시켜야 함

3 도메인 로직은 도메인 안에 캡슐화하라

- 도메인 객체는 단순 데이터 컨테이너가 아님
- 비즈니스 로직은 반드시 도메인 엔티티/VO/도메인 서비스 내부에 있어야 한다

```
1 // ❌ 나쁜 예
2 order.setStatus("PAID");
3
4 // ✅ 좋은 예
5 order.markAsPaid();
```

4 도메인은 상태 변경의 책임 주체다

- 외부에서 직접 값을 수정하는 게 아니라
도메인 객체가 자신의 상태를 스스로 변경해야 함 (자기 책임 원칙)

5 Entity와 Value Object를 구분하라

개념	Entity	Value Object
식별자	있음 (@Id)	없음
변경 가능성	상태 변경 가능	불변
비교 기준	ID	값 자체
예시	User, Order	Money, Address

6 도메인 서비스를 도메인 내부에만 정의하라

- 복수 객체에 걸친 규칙 or 외부 정보 기반 판단은 도메인 서비스 객체로 구현
- 상태를 가지지 않는 순수한 계산/판단 로직만 포함해야 함

7 도메인 이벤트로 관심사를 분리하라

- 도메인 상태 변경 후 발생하는 일은 이벤트로 분리
- 예: 주문 완료 → 이메일 발송, 포인트 적립 등

8 애그리게이트(Aggregate)는 도메인의 일관성을 관리하는 단위다

- 하나의 루트 엔티티 + 연관된 내부 객체 묶음
- 외부에서는 루트 엔티티를 통해서만 접근 가능해야 함

9 응용 서비스(Application Layer)는 흐름 조율자이다

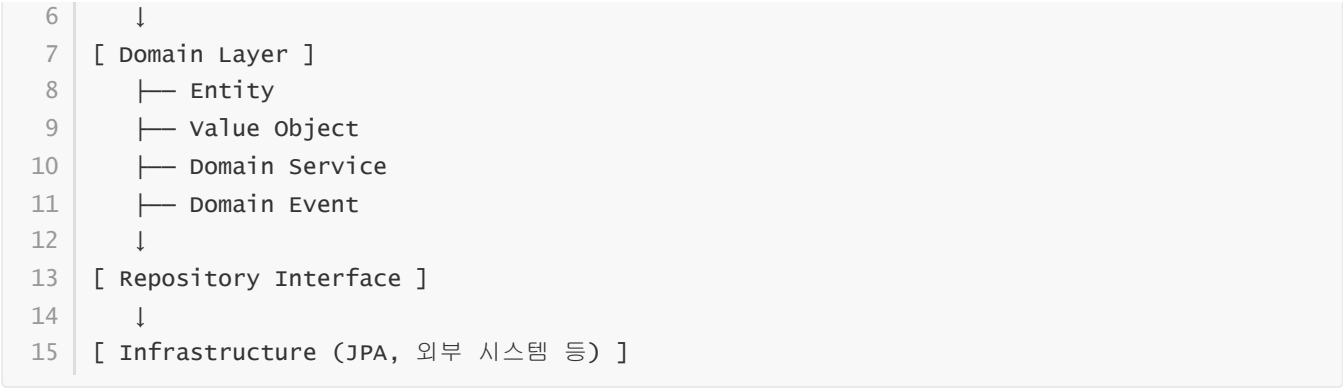
- 도메인 객체를 조합하고 호출할 뿐, 로직을 직접 담으면 안 됨
- 트랜잭션 경계를 설정하고 유스케이스를 조립하는 계층

10 Repository는 도메인 객체의 컬렉션처럼 동작해야 한다

- `save()`, `findById()` 는 도메인 객체의 삽입/조회에 집중
- 기술 세부 사항(DB, JPA)은 숨기고 도메인 친화적 인터페이스 설계

3. 도메인 중심 구조 설계도

```
1 [ 사용자 ]
2   ↓
3 [ Controller / API Layer ]
4   ↓
5 [ Application Service ]
```



💡 DDD를 적용할 때 고려해야 할 현실적인 기준

항목	설명
프로젝트 규모	단순 CRUD만 있는 소규모 앱에는 과할 수 있음
비즈니스 복잡성	규칙과 상태 전이가 많을수록 DDD 효과가 큼
팀 구성	개발자와 도메인 전문가 간 커뮤니케이션이 잦을수록 효과적
기술 프레임워크	Spring, JPA는 DDD와 잘 통합됨 (Entity, Repository 매핑)

✅ 마무리 요약표

원칙	설명
도메인 중심 설계	코드의 중심은 데이터가 아닌 비즈니스 의미
캡슐화	도메인 로직은 도메인 객체 안에 존재해야 함
책임 분리	Application ↔ Domain ↔ Infra 계층 분리
용어 일치	비즈니스와 개발이 같은 언어로 말해야 함
확장 가능성 확보	이벤트 기반 설계, 도메인 서비스, 밸류 객체 등으로 유연하게 변화 대응

애그리거트 루트, 엔티티, VO(Value Object)

✂ 1. 개념 요약

개념	정의
엔티티(Entity)	식별자(ID)가 있고 상태가 변할 수 있는 도메인 객체
값 객체(Value Object)	식별자가 없고 값 자체에 의미가 있는 불변 객체
애그리거트 루트(Aggregate Root)	연관된 엔티티와 VO들을 하나의 단위로 묶고, 일관성을 책임지는 대표 엔티티

2. 각 요소 상세 설명

✓ Entity

- ID를 기반으로 구분됨 (ex: `User`, `Order`)
- 상태 변경 가능 (mutable)
- DB 테이블과 1:1로 매핑되는 경우 많음
- `@Entity`, `@Id`로 표시 (Spring Data JPA)

```
1 @Entity
2 public class User {
3     @Id @GeneratedValue
4     private Long id;
5     private String name;
6     private String email;
7 }
```

✓ Value Object (VO)

- 식별자가 없음
- 값 자체가 의미
- 불변이어야 하며, 상태 변경 시 새 객체를 생성해야 함
- equals/hashCode는 내용 기준
- `@Embeddable` / `@Embedded` 사용 가능 (JPA)

```
1 @Embeddable
2 public class Address {
3     private String city;
4     private String street;
5
6     protected Address() {}
7     public Address(String city, String street) {
8         this.city = city;
9         this.street = street;
10    }
11 }
```

```
1 @Entity
2 public class Customer {
3     @Id @GeneratedValue
4     private Long id;
5
6     @Embedded
7     private Address address;
8 }
```

✅ Aggregate Root

- 도메인의 **일관성 경계(boundary)** 단위
- 외부에서 이 경계 내부의 객체에 직접 접근할 수 없고, **루트를 통해서만 조작 가능**
- 하나의 애그리거트에 속하는 객체들은 **트랜잭션이 공유됨**
- 반드시 **엔티티이며 식별자**를 가짐
- Repository는 **루트 단위**로만 존재함

예시: Order Aggregate

```
1 @Entity
2 public class Order { // ← Aggregate Root
3
4     @Id @GeneratedValue
5     private Long id;
6
7     @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
8     private List<OrderItem> items = new ArrayList<>();
9
10    public void addItem(OrderItem item) {
11        this.items.add(item);
12    }
13 }
```

```
1 @Entity
2 public class OrderItem { // ← Aggregate 내부 Entity
3     @Id @GeneratedValue
4     private Long id;
5     private String productName;
6     private int quantity;
7 }
```

외부에서 `OrderItem`을 직접 수정하지 않고, 반드시 `Order`를 통해서만 수정함

🎯 3. 설계 원칙

📌 애그리거트 경계 원칙

원칙	설명
경계 내부는 일관성을 보장	트랜잭션 내에서 루트가 모든 변경을 조정
루트를 통해서만 변경 가능	내부 객체 직접 변경은 허용하지 않음
루트 외부에는 ID만 공개	연관된 Entity나 VO는 외부에 노출되지 않음

📌 VO 설계 원칙

원칙	설명
불변이어야 함	생성 시에만 값 설정, 이후 변경 불가
equals/hashCode 재정의 필수	값 기반 비교가 핵심
논리적 의미를 가져야 함	<code>new Email(String)</code> , <code>new Money(BigDecimal)</code> 처럼 타입의 의미를 표현

✅ 4. 역할 구분 비교

항목	Entity	Value Object	Aggregate Root
식별자(ID)	있음	없음	있음 (루트)
상태 변경	가능	불변	가능
비교 기준	ID	값 자체	ID
영속 대상	O (@Entity)	종속적으로 O (@Embeddable)	O
책임	개별 객체 상태	의미 표현, 검증	일관성 유지, 트랜잭션 경계
접근성	내부 외부 모두 가능	내부에서만 조립	외부 진입점, 내부 캡슐화

🏠 실제 구조 예시

```
1 Order (Aggregate Root)
2   └─ OrderItem (Entity)
3   └─ ShippingAddress (Value Object)
```

- 외부에서는 오직 `Order` 를 통해서만 접근 가능해야 하며,
- `OrderItem` 이나 `ShippingAddress` 는 직접적으로 수정되면 안 됨

✅ 마무리 요약

개념	요약 정의	역할
Entity	ID로 구분되는 객체	상태 변화와 영속 대상
VO	값 자체에 의미가 있는 불변 객체	의미 표현, 값 검증
Aggregate Root	엔티티 중 트랜잭션/일관성 책임지는 루트	경계 관리, 도메인 외부와의 인터페이스

순수 자바 객체 설계

✂ 1. 순수 자바 객체란?

어떠한 프레임워크나 라이브러리의 의존 없이,
순수하게 **Java 문법과 문법적 개념만으로** 구성된 객체

즉, `@Component`, `@Entity`, `@Service` 등의 **Spring, JPA, Lombok**과 같은
특정 기술 스택에 **의존하지 않는 순수 Java 클래스**를 말한다.

🎯 목적

목적	설명
💡 비즈니스 의미 중심 설계	기술이 아니라 업무 개념 중심으로 객체를 표현
🧪 테스트 용이성	외부 의존성 없이 단위 테스트 가능
🔄 유지보수 유리	기술 교체나 확장에 유연
🧩 깔끔한 구조	SRP(단일 책임 원칙), 캡슐화, 불변성 등 객체지향 원칙을 살릴 수 있음

🔧 2. 어떤 객체가 POJO인가?

항목	설명
❌ <code>@Component</code> , <code>@Service</code> 등 없음	Spring에 의존하지 않음
❌ <code>@Entity</code> , <code>@Id</code> 등 없음	JPA에 의존하지 않음
❌ Lombok 없이도 명확한 설계	getter/setter 남발 금지, 명확한 생성자/메서드 중심
✅ <code>new</code> 로 직접 생성 가능	DI나 프록시 없이도 독립 사용 가능
✅ 순수 Java 코드	오직 Java 문법 + 도메인 개념만 존재

📦 3. 실전 설계 예시

예: `Money` 값 객체 (Value Object)

```
1 public class Money {
2
3     private final int amount;
4
5     public Money(int amount) {
6         if (amount < 0) throw new IllegalArgumentException("금액은 음수일 수 없습니다.");
7         this.amount = amount;
8     }
9 }
```



```

10     public Money add(Money other) {
11         return new Money(this.amount + other.amount);
12     }
13
14     public boolean isGreaterThan(Money other) {
15         return this.amount > other.amount;
16     }
17
18     public int getAmount() {
19         return amount;
20     }
21
22     @Override
23     public boolean equals(Object o) {
24         if (this == o) return true;
25         if (!(o instanceof Money money)) return false;
26         return amount == money.amount;
27     }
28
29     @Override
30     public int hashCode() {
31         return Objects.hash(amount);
32     }
33 }

```

- 기술 의존 없음
- 값 자체의 의미, 검증, 계산 책임을 **내부에 캡슐화**
- 어디서나 재사용 가능 (서비스, 엔티티, 테스트 등)

예: **Order** 도메인 엔티티 (순수 객체)

```

1  public class Order {
2
3      private final Long userId;
4      private final List<OrderItem> items;
5      private boolean paid;
6
7      public Order(Long userId, List<OrderItem> items) {
8          if (items == null || items.isEmpty()) throw new IllegalArgumentException("주문
항목 필수");
9          this.userId = userId;
10         this.items = new ArrayList<>(items);
11         this.paid = false;
12     }
13
14     public int totalAmount() {
15         return items.stream().mapToInt(OrderItem::subTotal).sum();
16     }
17
18     public void markAsPaid() {
19         if (this.paid) throw new IllegalStateException("이미 결제됨");

```

```
20         this.paid = true;
21     }
22
23     public boolean isPaid() {
24         return paid;
25     }
26 }
```

`@Entity`, `@Transactional` 없이도 도메인 자체를 표현하고
비즈니스 규칙을 메서드 안에 명확히 표현

🧠 4. 순수 객체 설계 시 지켜야 할 핵심 원칙

원칙	설명
📦 캡슐화	필드는 private, 메서드로만 상태를 변경
🚫 setter 금지	setter 남발은 객체지향이 아님. 메서드로 의미 있는 변경만 허용
🔴 생성자 유효성 검사	생성 시점부터 무결한 객체를 만들어야 함
❄ 불변성 고려	VO는 반드시 불변, 엔티티도 가능한 불변 필드 우선
🧪 테스트 중심 설계	외부 기술 없이 테스트 가능하게 만들 것
🚫 기술 의존 제거	어떤 프레임워크에도 의존하지 않고 단독으로 작동 가능해야 함

✅ 순수 객체 vs 기술 의존 객체 비교

항목	순수 자바 객체 (POJO)	기술 의존 객체 (Spring, JPA 등)
어노테이션	❌ 없음	✅ <code>@Entity</code> , <code>@Service</code> , <code>@Autowired</code>
생성 방식	<code>new</code> 로 직접 생성 가능	DI, 프록시 필요
테스트 용이성	✅ 단위 테스트 쉬움	❌ 환경 세팅 필요
목적	도메인 표현	실행/구현 목적
기술 결합도	낮음	높음

✅ 마무리 요약

개념	설명
POJO	기술에 의존하지 않고, 도메인의 순수한 의미와 동작을 표현하는 자바 객체
왜 필요한가	비즈니스 중심 모델링, 테스트 가능성, 유지보수 용이성

개념	설명
어디에 사용하나	Entity, VO, Domain Service, Command, Policy 등
무엇을 피하나	JPA, Spring, Lombok 등 기술 프레임워크에 결합된 설계