

값 타입

---

# 목차

---

- 기본값 타입
- 임베디드 타입(복합 값 타입)
- 값 타입과 불변 객체
- 값 타입의 비교
- 값 타입 컬렉션
- 실전 예제 - 6. 값 타입 매핑

기본값 타입

# JPA의 데이터 타입 분류

---

- **엔티티 타입**

- @Entity로 정의하는 객체
- 데이터가 변해도 식별자로 지속해서 **추적** 가능
- 예) 회원 엔티티의 키나 나이 값을 변경해도 식별자로 인식 가능

- **값 타입**

- int, Integer, String처럼 단순히 값으로 사용하는 자바 기본 타입이나 객체
- 식별자가 없고 값만 있으므로 변경시 추적 불가
- 예) 숫자 100을 200으로 변경하면 완전히 다른 값으로 대체

# 값 타입 분류

---

- **기본값 타입**
  - 자바 기본 타입(int, double)
  - 래퍼 클래스(Integer, Long)
  - String
- **임베디드 타입**(embedded type, 복합 값 타입)<sub>x, y 좌표 - Position</sub>
- **컬렉션 값 타입**(collection value type)

# 기본값 타입

---

- 예): String name, int age
- 생명주기를 엔티티의 의존
  - 예) 회원을 삭제하면 이름, 나이 필드도 함께 삭제
- 값 타입은 공유하면X
  - 예) 회원 이름 변경시 다른 회원의 이름도 함께 변경되면 안됨

# 참고: 자바의 기본 타입은 절대 공유X

- int, double 같은 기본 타입(primitive type)은 절대 공유X

- 기본 타입은 항상 값을 복사함

```
Int a = 10;
```

```
Int b = a;
```

```
a = 20;
```

```
Sout("a") // 20
```

```
Sout("b") // 10
```

a 를 20으로 바꾸더라도 b 는 여전히 10이다.  
항상 값을 복사하기 때문이다

- Integer같은 래퍼 클래스나 String 같은 특수한 클래스는 공유 가능한 객체이지만 **변경X**

< 판타지 >

```
Integer a = new Integer(10);
```

```
Integer b = a;
```

```
a.setValue(20);
```

```
Sout("a"); // 20
```

```
Sout("b"); // 20
```

값 복사가 아니라 레퍼런스 복사가 되기 때문에  
b도 덩달아서 20으로 변한다

이런식으로 setValue 하게 되면 사이드이펙트가  
발생할 수 있어서 애초에 변경이 안된다.

임베디드 타입(복합 값 타입)



# 임베디드 타입

---

- 새로운 값 타입을 직접 정의할 수 있음
- JPA는 임베디드 타입(embedded type)이라 함
- 주로 기본 값 타입을 모아서 만들어서 복합 값 타입이라고도 함
- int, String과 같은 값 타입

엔티티가 아니라서 추적이 안되고 변경하면 끝나버린다.

## 값 타입 분류

- 기본값 타입
  - 자바 기본 타입(int, double)
  - 래퍼 클래스(Integer, Long)
  - String
- 임베디드 타입(embedded type, 복합 값 타입)
- 컬렉션 값 타입(collection value type)

# 임베디드 타입

---

- 회원 엔티티는 이름, 근무 시작일, 근무 종료일, 주소 도시, 주소 번지, 주소 우편번호를 가진다.

Member	
id	
name	
(	startDate
	)
endDate	
(	city
	street
	)
zipcode	

이렇게 묶어서 자주 쓰이지 않을까?

# 임베디드 타입

---

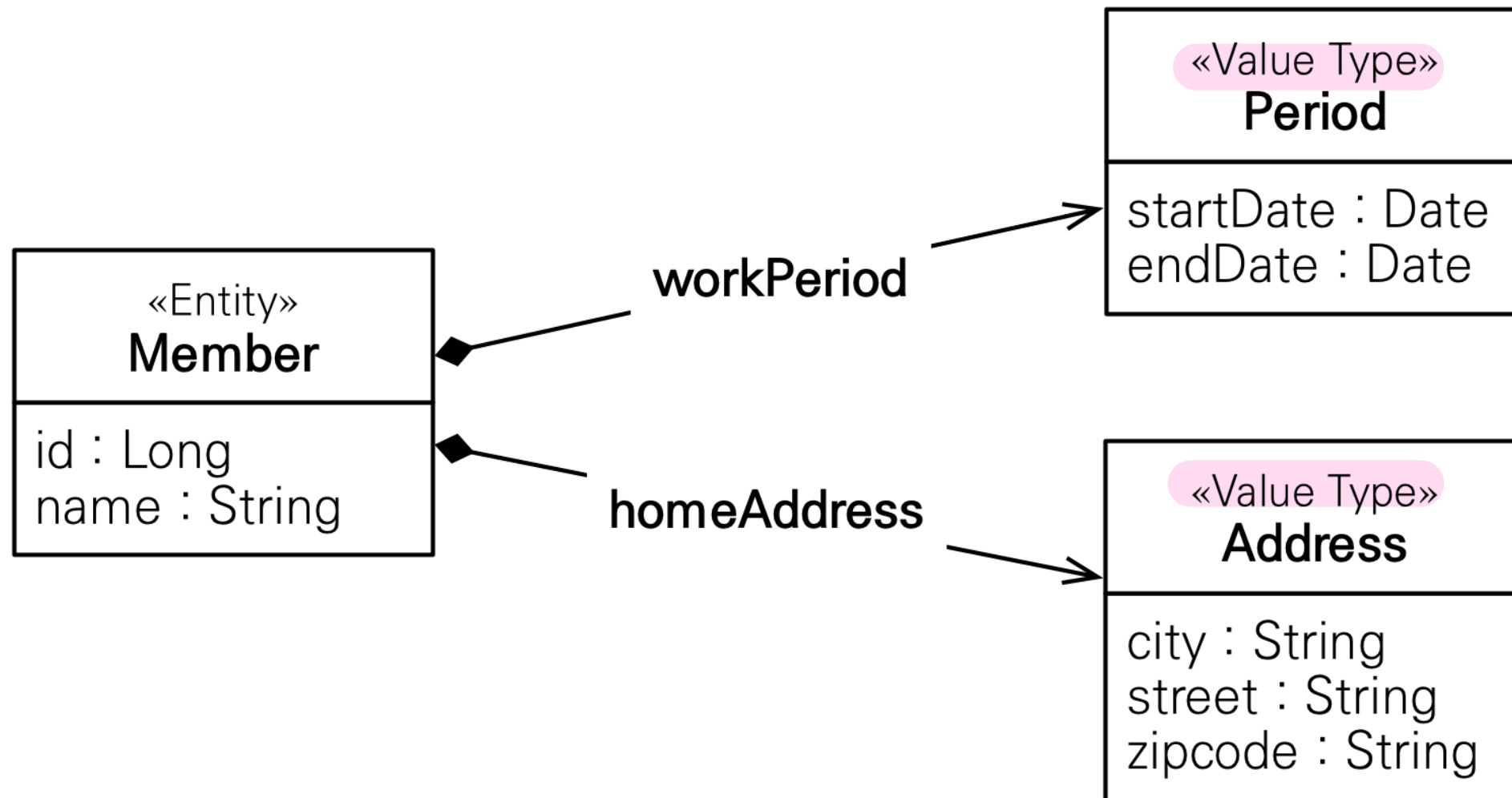
- 회원 엔티티는 이름, 근무 기간, 집 주소를 가진다.

«Entity» Member
id : Long name : String <b>workPeriod</b> : Period <b>homeAddress</b> : Address

Member
id
name
<b>workPeriod</b>
<b>homeAddress</b>

# 임베디드 타입

---



# 임베디드 타입 사용법

---

- @Embeddable: 값 타입을 정의하는 곳에 표시
- @Embedded: 값 타입을 사용하는 곳에 표시
- 기본 생성자 필수

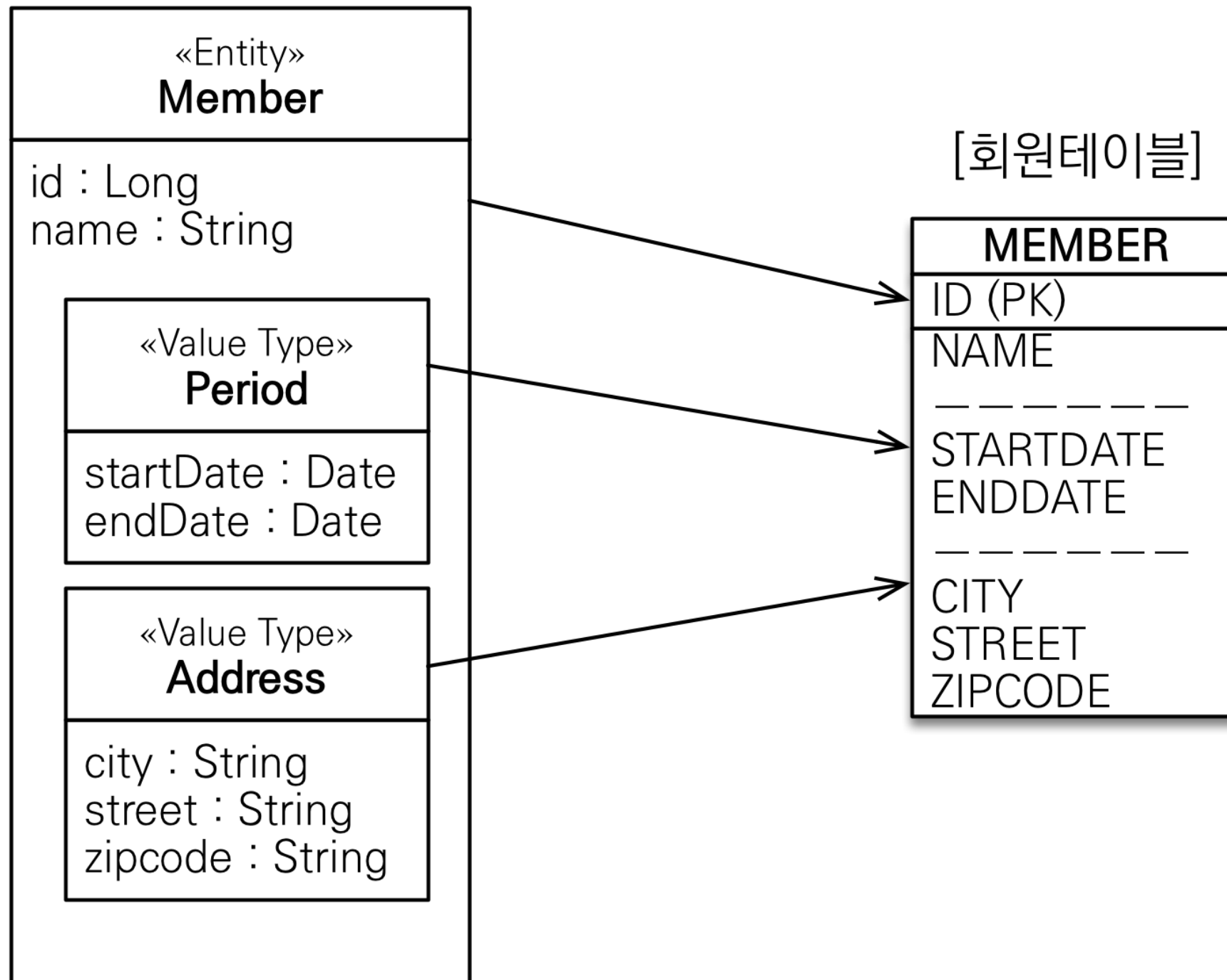
# 임베디드 타입의 장점

---

- 재사용
- 높은 응집도
- `Period.isWork()`처럼 해당 값 타입만 사용하는 의미 있는 메소드를 만들 수 있음      상당히 객체지향적으로 설계가 가능해지는 것.
- 임베디드 타입을 포함한 모든 값 타입은, 값 타입을 소유한 엔티티에 생명주기를 의존함

엔티티가 생겨나면 값 타입이 존재할 수 있고, 엔티티가 삭제되면 값 타입도 같이 삭제됨.

# 임베디드 타입과 테이블 매핑



```

@Entity
public class Member extends BaseEntity {

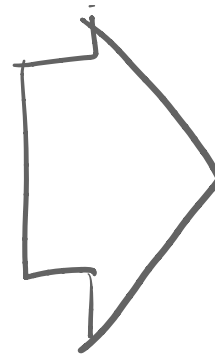
    @Id
    @GeneratedValue
    @Column(name = "MEMBER_ID")
    private Long id;

    @Column(name = "USERNAME")
    private String username;

    private LocalDateTime startDate;
    private LocalDateTime endDate;

    private String city;
    private String street;
    private String zipcode;
}

```



```

@Entity
public class Member {

    @Id
    @GeneratedValue
    @Column(name = "MEMBER_ID")
    private Long id;

    @Column(name = "USERNAME")
    private String username;

    //기간 Period
    @Embedded
    private Period workPeriod;

    //주소
    @Embedded
    private Address homeAddress;
}

```

```

@Embeddable
public class Period {

    private LocalDateTime startDate;
    private LocalDateTime endDate;
}

```

```

@Embeddable
public class Address {

    private String city;
    private String street;
    private String zipcode;
}

```



# 임베디드 타입과 테이블 매핑

---

- 임베디드 타입은 엔티티의 값일 뿐이다.
- 임베디드 타입을 사용하기 전과 후에 **매핑하는 테이블은 같다.**  
DB 가 달라지지 않는다
- 객체와 테이블을 아주 세밀하게(fine-grained) 매핑하는 것이 가능  
Address 나 Period 를 클래스로 만들어 놓으면, 해당 클래스 내에서 메서드를 만들어서 유용하게 쓸 수 있다.
- 잘 설계한 ORM 애플리케이션은 매핑한 테이블의 수보다 클래스의 수가 더 많음

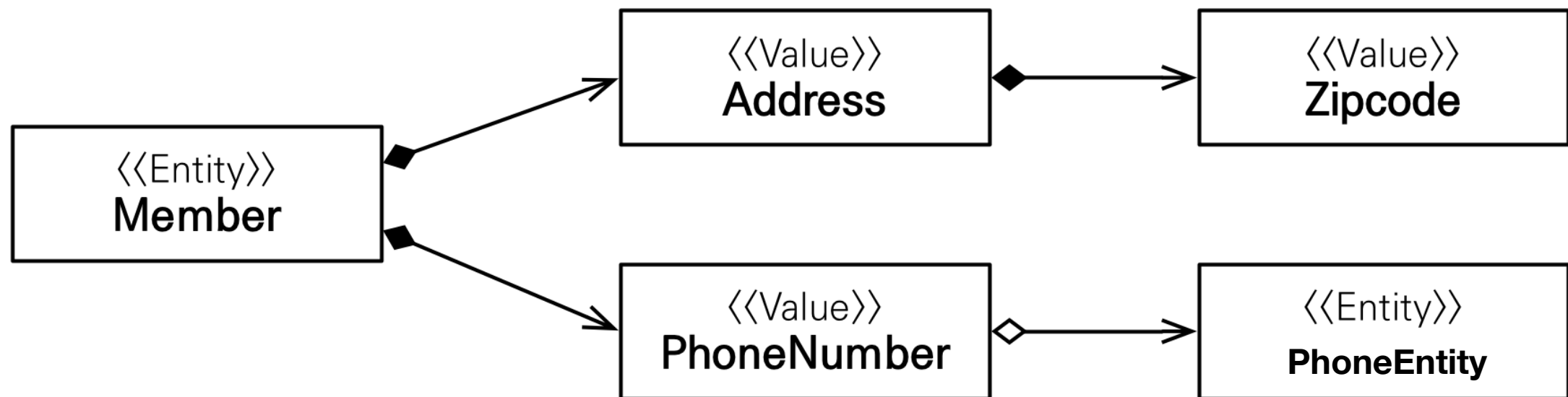
# 임베디드 타입과 연관관계

```
@Embeddable
public class Address {

    private String city;
    private String street;
    private String zipcode;

    private Member member;
```

→ 이게 된다는 것!



## @AttributeOverride: 속성 재정의

---

- 한 엔티티에서 같은 값 타입을 사용하면?
- 컬럼 명이 중복됨
- **@AttributeOverrides, @AttributeOverride**를 사용해서  
컬러 명 속성을 재정의

```
10 @GeneratedValue
11 @Column(name = "MEMBER_ID")
12 private Long id;
13
14 @Column(name = "USERNAME")
15 private String username;
16
17 //기간 Period
18 @Embedded
19 private Period workPeriod;
20
21 //주소
22 @Embedded
23 private Address homeAddress;
24
25 //주소
26 @Embedded
27 private Address workAddress;
28
29 public Long getId() {
30     return id;
31 }
```

Member workAddress

Jun 11, 2019 12:30:02 AM org.hibernate.engine.jdbc.internal.  
INFO: HHH10001001: Connection properties: {user=sa}  
Jun 11, 2019 12:30:02 AM org.hibernate.engine.jdbc.connections.internal.  
INFO: HHH10001003: Autocommit mode: false  
Jun 11, 2019 12:30:02 AM org.hibernate.engine.jdbc.connections.internal.  
INFO: HHH000115: Hibernate connection pool size: 20  
Jun 11, 2019 12:30:02 AM org.hibernate.dialect.Dialect.  
INFO: HHH000400: Using dialect: org.hibernate.dialect.  
Exception in thread "main" javax.persistence.PersistenceException:   
 at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:55)  
 at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:55)  
 at hellojpa.JpaMain.main(JpaMain.java:11)  
Caused by: org.hibernate.MappingException: Repeated column in mapping for entity  
... 4 more

같은 값 타입인  
Address 필드를 중복해 놓으면  
컬럼이 중복된다는 에러가 뜬다

해결책!

```
//주소
@Embedded
private Address homeAddress;

//주소
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="city",
        column=@Column("WORK_CITY")),
    @AttributeOverride(name="street",
        column=@Column("WORK_STREET")),
    @AttributeOverride(name="zipcode",
        column=@Column("WORK_ZIPCODE"))
})
private Address workAddress;
```

# 임베디드 타입과 null

---

- 임베디드 타입의 값이 null이면 매핑한 컬럼 값은 모두 null

Period가 null 이면 이에 매핑되는 startDate, endDate 컬럼 값은 모두 null 이다

값 타입과 불변 객체

값 타입은 복잡한 객체 세상을 조금이라도 단순화하려고 만든 개념이다. 따라서 값 타입은 단순하고 안전하게 다룰 수 있어야 한다.

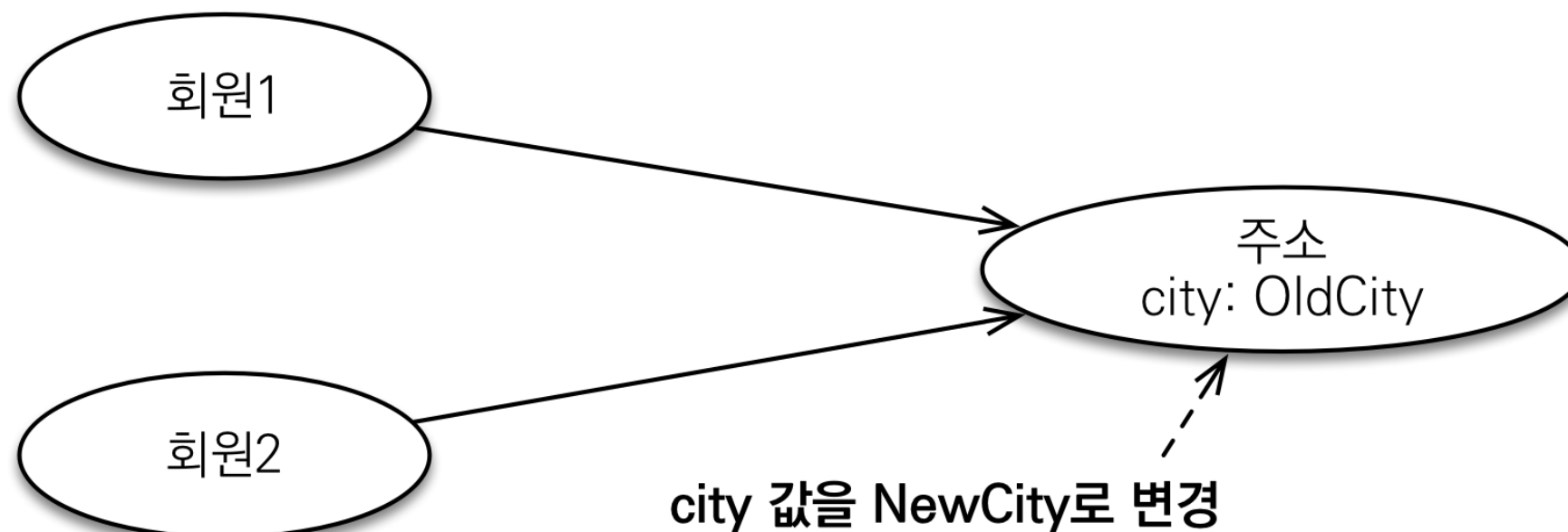
그래서 우리가 엔티티를 다룰때는 좀 여러면을 신경쓰지만, 스트링이나 인트값을 바꿀 때는 걱정할 거 없이 써왔던 것

근데! 값 타입인 임베디드 타입을 여러 엔티티에서 공유하면 문제가 생길 수 있다!

## 값 타입 공유 참조

---

- 임베디드 타입 같은 값 타입을 여러 엔티티에서 공유하면 위험함
- 부작용(side effect) 발생





```

Member member = new Member();
member.setUsername("member1");
member.setHomeAddress(address);
em.persist(member);

Member member2 = new Member();
member2.setUsername("member2");
member2.setHomeAddress(address);
em.persist(member2);

```

select \* from member;

MEMBER_ID	CITY	STREET	ZIPCODE	USERNAME	ENDDATE	STARTDATE	TEAM_ID
1	city	street	10000	member1	null	null	null
2	city	street	10000	member2	null	null	null

(2 행, 3 ms)

```

member.getHomeAddress().setCity("newCity");

```

member1, 2 가 address 를 공유하고 있음

이 상태에서 member1 의 주소를 변경하면 member2 의 주소까지 변경되어 버린다!

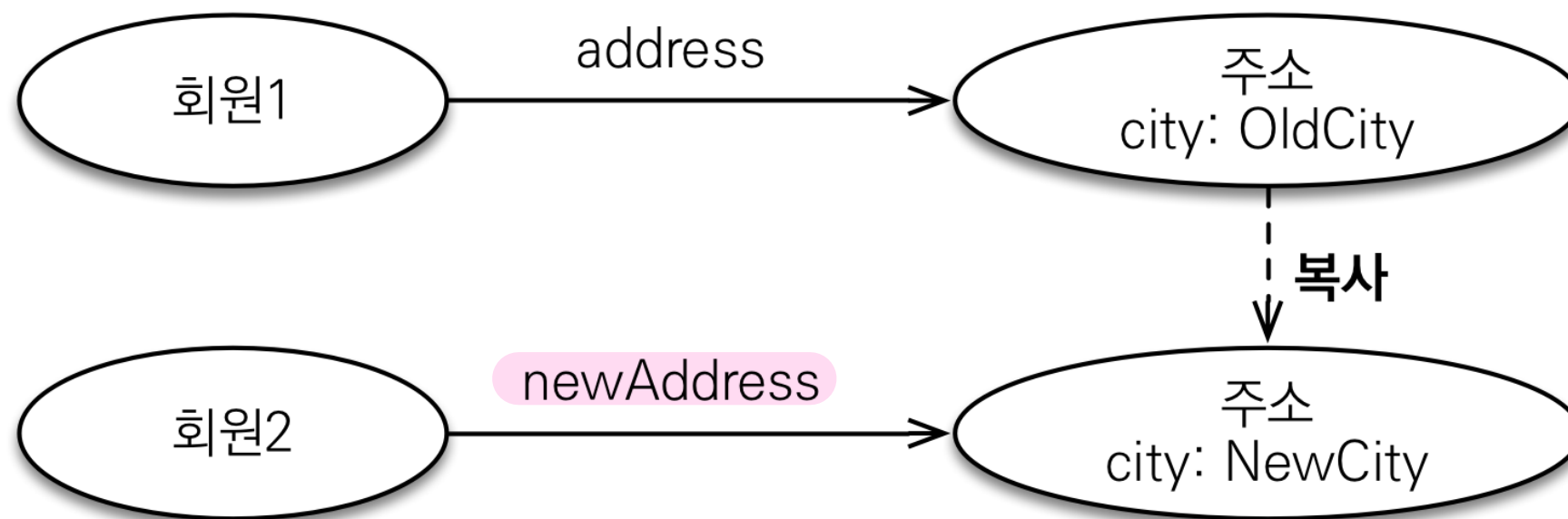
만약 이게 의도적인 거라면 주소가 값 타입이 아니라 엔티티였어야 함.

값 타입은 부작용이 발생하면 안된다!

# 값 타입 복사

---

- 값 타입의 실제 인스턴스인 값을 공유하는 것은 위험
- 대신 값(인스턴스)를 복사해서 사용



# 객체 타입의 한계

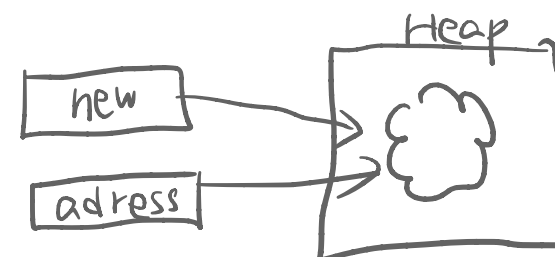
- 항상 값을 복사해서 사용하면 공유 참조로 인해 발생하는 부작용을 피할 수 있다.

primitive type 은 = 로 할당하면 값이 복사되어 넘어가기 때문에 문제가 없으나 객체 타입은 참조 변수이다.

- 문제는 임베디드 타입처럼 직접 정의한 값 타입은 자바의 기본 타입이 아니라 객체 타입이다.
- 자바 기본 타입에 값을 대입하면 값을 복사한다.
- 객체 타입은 참조 값을 직접 대입하는 것을 막을 방법이 없다.
- 객체의 공유 참조는 피할 수 없다.

```
Address adress = new Address ("")
```

```
Address newAdd = adress
```



# 객체 타입의 한계

---

## 기본 타입(primitive type)

```
int a = 10;  
int b = a; // 기본 타입은 값을 복사  
b = 4;
```

## 객체 타입

```
Address a = new Address("Old");  
Address b = a; // 객체 타입은 참조를 전달  
b.setCity("New") // 이러면 a 의 city 도 "New" 가 되버린다.
```

# 불변 객체

---

- 객체 타입을 수정할 수 없게 만들면 **부작용을 원천 차단**
- **값 타입은 불변 객체(immutable object)로 설계해야함**
- **불변 객체: 생성 시점 이후 절대 값을 변경할 수 없는 객체**
- 생성자로만 값을 설정하고 **수정자(Setter)를 만들지 않으면 됨**  
or private setter
- 참고: **Integer, String**은 자바가 제공하는 대표적인 불변 객체

불변이라는 작은 제약으로 부작용이라는 큰 재앙을 막을 수 있다.

만약 멤버1의 주소를 바꾸고 싶으면

```
Address newAddress = new Address("city-name", address.getStreet(), address.getZipcode());
```

```
Member1.setHomeAddress(newAddress);
```

위처럼 통째로 갈아 치워야 한다.

# 값 타입의 비교

# 값 타입의 비교

---

- 값 타입: 인스턴스가 달라도 그 안에 값이 같으면 같은 것으로 봐야 함

```
int a = 10;  
int b = 10;
```

```
Sout(a == b); // true
```

```
Address a = new Address ("서울시")  
Address b = new Address ("서울시")
```

```
Sout(a == b); // false
```



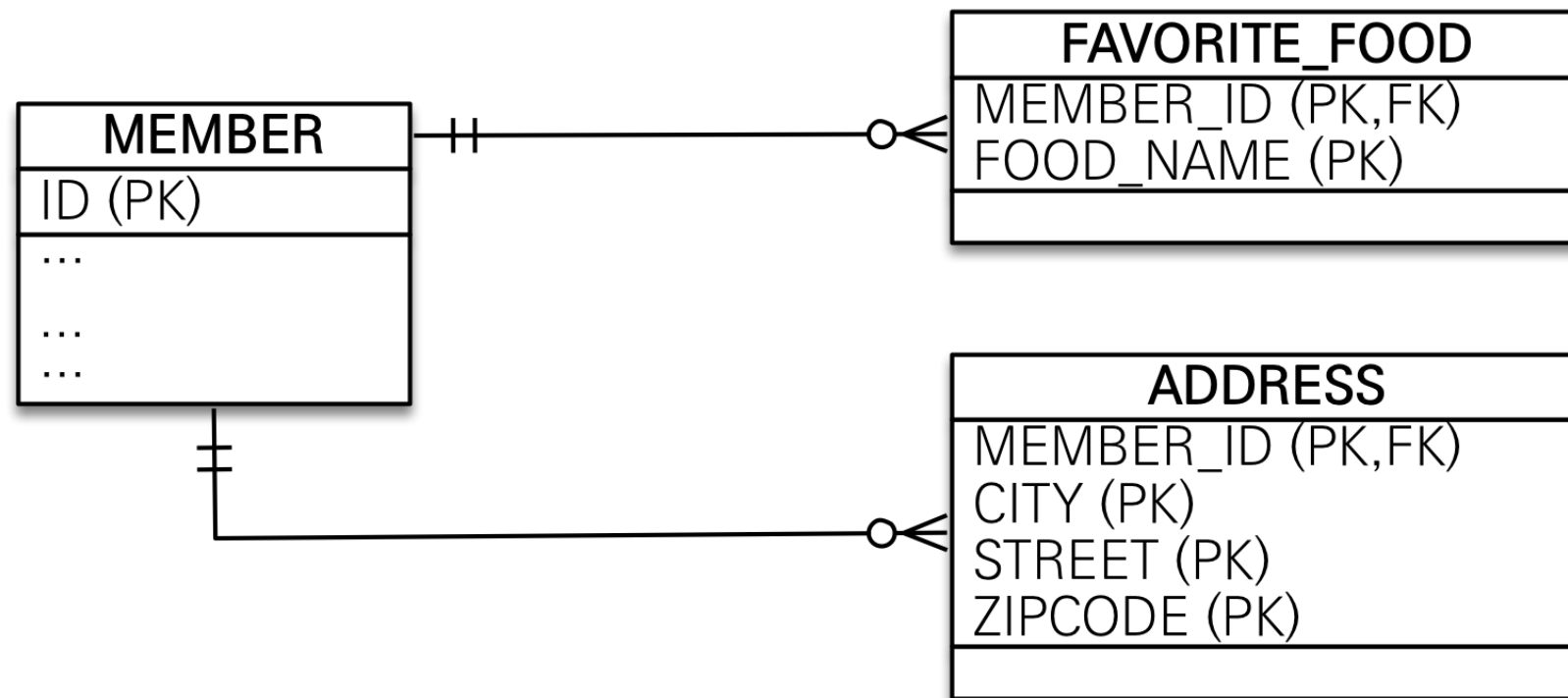
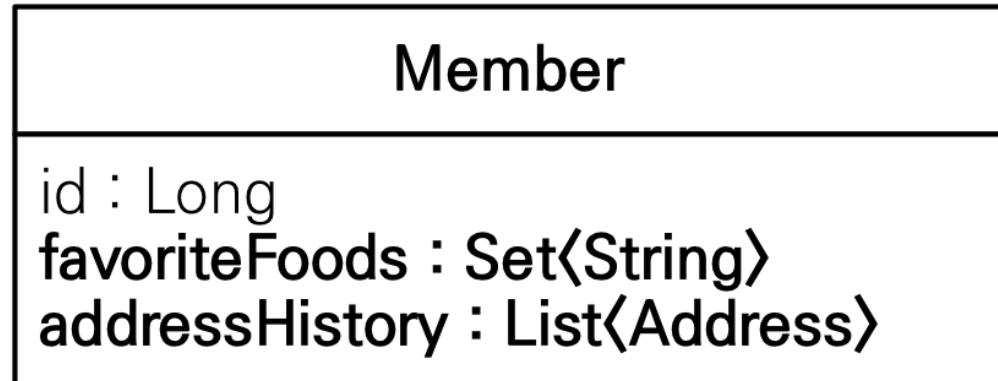
# 값 타입의 비교

---

- **동일성(identity) 비교**: 인스턴스의 참조 값을 비교, == 사용
- **동등성(equivalence) 비교**: 인스턴스의 값을 비교, equals() 사용
- 값 타입은 a.equals(b)를 사용해서 동등성 비교를 해야 함
- 값 타입의 equals() 메소드를 적절하게 재정의(주로 모든 필드 사용)
  - 오버라이딩 안하면 equals() 는 기본적으로 == 로 비교한다
  - 값 타입의 데이터들은 모두 equals(), hashCode() 를 재정의해줘야 한다

값 타입 컬렉션

# 값 타입 컬렉션



# 값 타입 컬렉션

---

- 값 타입을 하나 이상 저장할 때 사용
- @ElementCollection, @CollectionTable 사용
- 데이터베이스는 컬렉션을 같은 테이블에 저장할 수 없다.
- 컬렉션을 저장하기 위한 별도의 테이블이 필요함

# 값 타입 컬렉션 사용

---

- 값 타입 저장 예제
- 값 타입 조회 예제
  - 값 타입 컬렉션도 지연 로딩 전략 사용
- 값 타입 수정 예제
- 참고: 값 타입 컬렉션은 영속성 전에(Cascade) + 고아 객체 제거 기능을 필수로 가진다고 볼 수 있다.

# 값 타입 컬렉션의 제약사항

---

- 값 타입은 엔티티와 다르게 식별자 개념이 없다.
- 값은 변경하면 추적이 어렵다.
- 값 타입 컬렉션에 변경 사항이 발생하면, 주인 엔티티와 연관된 모든 데이터를 삭제하고, 값 타입 컬렉션에 있는 현재 값을 모두 다시 저장한다.
- 값 타입 컬렉션을 매핑하는 테이블은 모든 컬럼을 묶어서 기본 키를 구성해야 함: **null 입력X, 중복 저장X**

# 값 타입 컬렉션 대안

---

- 실무에서는 상황에 따라 **값 타입 컬렉션 대신에 일대다 관계를 고려**
- 일대다 관계를 위한 엔티티를 만들고, 여기에서 값 타입을 사용
- 영속성 전이(Cascade) + 고아 객체 제거를 사용해서 값 타입 컬렉션 처럼 사용
- EX) AddressEntity

# 정리

---

- 엔티티 타입의 특징

- 식별자O
- 생명 주기 관리
- 공유

- 값 타입의 특징

- 식별자X
- 생명 주기를 엔티티에 의존
- 공유하지 않는 것이 안전(복사해서 사용)
- 불변 객체로 만드는 것이 안전



값 타입은 정말 값 타입이라 판단될 때만 사용

엔티티와 값 타입을 혼동해서 엔티티를 값 타입으로 만들면 안됨

식별자가 필요하고, 지속해서 값을 추적, 변경해야 한다면 그것은 값 타입이 아닌 엔티티

## 실전 예제 - 6. 값 타입 매핑

## 실전 예제 - 6. 값 타입 매핑

