<JSTORM>

Enterprise Java Beans 4부

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	Enterprise Java Beans 4부
Document file name:	EJB4_JSTORM.doc
Revision number:	<1.0>
Issued by:	<박지훈> pjh@jstorm.pe.kr
	<남궁윤> yoonng@hanmail.net
	<김종윤> foxkjy@chollian.net
	<전부현> kanaloa@orgio.net
	정리 : <윤준호> junoyoon@orgio.net
Issue Date:	<2000/9/7>
Status:	final

Content Information

Audience	EJB 프로그래밍을 학습하는 중급 JAVA 프로그래머
Abstract	저번장에서는 Entity Bean과 Session 빈에 대해서 간단히 살펴보았다. 이번 장에서는 Bean의 고급 기술들을 살펴보고 또한 몇가지 사례를 살펴봄으로써 실질적인 EJB 적용의 기초를 다진다.
Reference	1) Enterprise Java Bean, Second Edition (Oreill'y)
	2) Mastering EJB (Wiley)
	3) http://www.itplus.co.kr 자료
	4) http://www.bea.com의 자료
	5) http://www.javasoft.com의 자료
	6) http://www.javaworld.com의 자료
	7) Sun사의 EJB 스펙 1.1 PDF파일
	8) J2EE SDK의 example9
	9) J2EE blueprint 및 java pet store example
Benchmark information	

Document Approvals

고문	Signature	date
		
지위	Signature	date

Revision History

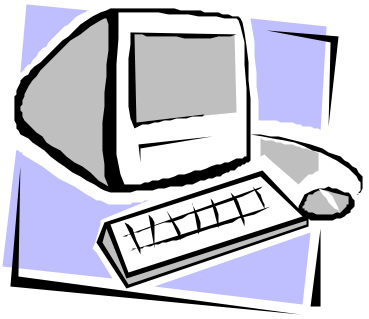
Revision	Date	Author	Description of change

Table of Contents

4 부

Entity Bean, Session Bean 고급개발

저번장에서는 Entity Bean과 Session 빈에 대해서 간단히 살펴보았다. 이번 장에서는 Bean의 고급 기술들을 살펴보고 또한 몇가지 사례를 살펴봄으로써 실질적인 EJB 적용의 기초를 다진다.



Chapter 10

고급 엔티티빈 개발

Persistence

엔티티 빈의 상태(state)는 관계형 데이터베이스와 같은 저장 메가니즘에 저장되기 때문에 엔티티 빈은 영속적(persistent) 이라고 할 수 있다. Persistence는 엔티티 빈이 어플리케이션의 생명주기 또는 J2EE 서버(또는 다른 어플리케이션 서버)의 프로세스와 관계없이 계속 존재하는 것을 의미한다. 만일 데이터베이스와 연동된 일을 하는 경우라면, 이러한 영속적인 데이터(persistent data)를 자주 볼 수 있다. 데이터베이스에 있는 데이터는 영속적이다. 왜냐하면 제공하고 있는 데이터베이스 서버나 응용 프로그램이 종료되더라도 여전히 존재할 것이기 때문이다.

Persistence 작업이란 엔티티 빈 인스턴스에 의해서 표현되는 데이터를 데이터베이스에 반영하는 과정을 말하는 것이다. 즉, 새로운 빈이 생성될 때, 데이터베이스에 새로운 레코드가 추가되어야 하고 빈 인스턴스는 그 데이터와 연관되어야 한다. 또한, 빈이 사용되어지고 그 상태가 변화함에 따라, 이러한 변화는 데이터베이스의 데이터와 동기화되어야 하고, 각 항목은 추가/변경 및 제거 되어야 한다. 이러한 과정을 엔티티 빈의 persistence 작업 또는 persistence 관리라고 한다.

CMP와 BMP

엔티티 빈은 Persistence를 관리하는 방법에 의해 CMP(Container-managed persistence)기반 엔티티 빈과 BMP(Bean-managed persistence)기반 엔티티 빈의 두 가지 종류로 구분된다.

Container-managed persistence(CMP) 기반 엔티티 빈

- CMP는 빈의 persistence가 EJB 컨테이너에 의해 자동으로 관리된다. 컨테이너는 엔티티 빈 인스턴스의 상태 필드가 어떻게 데이터베이스에 대응되어야 하는지를 알고, 데이터베이스 내에 있는 엔티티 빈의 상태와 관련된 데이터를 추가, 변경, 삭제하는 것을 자동으로 관리한다.
- CMP의 장점은 개발이 용이하고 코드의 크기가 줄어든다는 것이다. 그것은 persistence에 대한 관리 자체를 EJB 컨테이너가 해결해 주기 때문에, 개발자는 비즈니스 로직에만 집중하여 개발할 수 있기 때문이다. 따라서, 개발자는 개발한 빈을 deployment 할 때 엔티티 빈의 어떤 필드가 EJB 컨테이너에 의해 관리되고 데이터베이스에 대응되는 방법을 지정해 주기만 하면 된다. 이와 같은 작업만 마치면, 컨테이너는 빈의 모든 persistence 작업에 필요한 코드를 자동으로 생성하게 된다. CMP의 또 다른 장점은 빈이 자신의 상태를 저장하는데 사용하는 데이터베이스와 독립된다는 것이다. 즉, CMP 기반 엔티티 빈은 관계형 또는 객체지향 데이터베이스의 장점을 모두 이용할 수 있다. 또한, 이와 같은 특징 때문에 응용 프로그램 간의 빈에 대한 재사용성과 유연성이 높아질 수 있게 된다.
- CMP의 단점은 빈의 필드를 데이터베이스와 대응시킬 때 BMP에 비해 정교함이 떨어진다는 것이다. 즉, 좀 더 정교하게 CMP기반 엔티티 빈과 데이터베이스를 대응시키기 위해서는 훨씬 강력한 툴이 필요하게 된다. 예를들어, 특정 빈의 상태는 복잡한 관계형 데이터베이스의 join 연산으로 정의될 수 있다. 또한, CICS나 IMS와 같은 기존의 시스템에 대응될 수도 있다. 실제 CMP기반으로 엔티티 빈을 설계할때는 여러가지 제약사항이 뒤따르게 된다.
- CMP Limitations : 엔티티 빈에 대한 CMP기반 구현은 객체들을 관계형 데이터베이스에 맵핑시키는데 다음과 같은 제한이 있다.(2000-08-02 현재 J2EE Server에 대한 SUN's Release Note로 모든 EJB 컨테이너에 해당하는 것은 아님.)

  - 엔티티 빈 클래스는 데이터베이스의 오직 하나의 테이블에만 맵핑될 수 있다.

  - Container-managed field는 테이블의 오직 하나의 컬럼에만 맵핑될 수 있다.

  - 컨테이너가 기반 데이터베이스(underlying database)에서 단지 몇 개의 container-managed field들을 불러올 때에도 컨테이너는 모든 field들을 불러오게 된다. 만일 불러올 데이터의 양이 매우 많다고 하면, 이것은 아주 비효율적이 된다. 왜냐하면, 비즈니스 method에게 사실상 이 모든 container-managed filed들의 데이터가 필요하지는 않기 때문이다.

  - 만일 여러 개의 엔티티 빈들에 정의되어 있는 container-managed field들이 데이터베이스의 동일한 하나의 데이터 item에 맵핑되어 있고, 이 빈 인스턴스들이 동일한 트랜잭션을 호출하게 된다면, 이 데이터 item에 대한 consistency를 보장할 수 없게 된다.

  - Application Deployment Tool은 ejbCreate, ejbRemove, ejbLoad, ejbStore method들에 대한 SQL 구문을 생성한다. 개발자는 이 SQL 구문에서 오직 테이블과 컬럼의 이름만을 수정할 수 있다. 개발자는 입력 전달인자에 대한 위치를 대신 나타내 주는 느낌표의 개수나 순서를 바꿀 수 없을 것이다.

  - 생성된 SQL 구문에서 내장 프로시져(stored procedure)를 호출할 수 없다.

- " Create Table " SQL 구문에서, 테이블 컬럼의 SQL 형은 그것에 대응되는 인스턴스 변수에게 호환되도록 제공되는 형으로만 바꿀 수 있다.

- 모든 SQL 구문에 있는 테이블과 컬럼의 이름은 반드시 일관되어야 한다.

- 생성된 SQL 구문은 다음 종류의 데이터베이스들에서 테스트 되었다. 만일 다른 종류의 데이터베이스를 사용하고 있다면, 그 시스템의 요구사항에 맞게 생성된 SQL 구문을 수정해야 한다.

Cloudscape / Oracle / Microsoft SQL Server

Bean-managed persistence(BMP) 기반 엔티티 빈

- BMP는 엔티티 빈의 persistence에 관련된 모든 작업을 개발자가 명시적으로 해주어야 한다. 즉, 개발자가 작성하는 엔티티 빈 코드는 데이터베이스에 접근하고 제어하는 호출을 포함하고 있어야 한다. 이를 위해 빈 개발자는 데이터베이스를 다루는 코드를 반드시 사용해야 하고, EJB 컨테이너는 빈 인스턴스에게 언제 데이터베이스의 데이터를 추가, 변경, 삭제를 하는 것이 안전한지 만을 알려준다. 따라서 BMP 기반의 엔티티 빈은 모든 persistence 작업에 대한 SQL 호출 구문에 대한 처리를 스스로 해야 하며, 이것은 전적으로 개발자의 몫이다.
- BMP의 경우 빈 클래스 내에 persistence 관리에 관한 코드가 명시적으로 존재해야 하기 때문에 BMP는 CMP에 비해 개발하는 과정이 복잡하다고 할 수 있다.
- 이와 같이 BMP기반 엔티티 빈은 코드내에 어떤 유형의 데이터베이스를 사용할 것인지와 엔티티 빈 클래스의 각 필드들을 데이터베이스와 어떻게 대응시킬 것인지를 명시적으로 작성해야 하므로 개발자는 이에 대한 충분한 지식이 필요하다.
- BMP의 장점은 빈 인스턴스와 데이터베이스 사이의 상태를 관리함에 있어서 보다 유연하다는 것이다. 즉, 복잡한 join 연산이나 서로 다른 데이터베이스들의 연동 또는 기존 시스템과의 연동과 같은 시스템에서는 BMP 기반의 엔티티 빈이 훨씬 유용하게 적용될 수 있다. 또한, 기존의 기업 비즈니스 시스템이 단일한 플렛폼에 적용되어 있지 않기 때문에 CMP 기반의 엔티티 빈은 이에 최적화 되기 어렵다. 하지만, BMP 기반의 엔티티 빈의 경우 다양한 환경에 알맞게 디자인이 가능함으로 비즈니스 시스템에 대한 최적화된 빈의 설계 및 생성이 가능하다는 장점이 있다.
- BMP의 단점은 빈을 설계하기가 CMP에 비해 훨씬 까다롭다는 것이다. 즉, BMP 기반의 엔티티 빈을 설계하는 사람은 데이터베이스의 구조에 대한 이해와 지식이 필요하고 entity에 관련된 데이터를 생성하고 변경 및 제거하는 로직을 개발해야 한다. 또한, 개발자는 빈의 홈 인터페이스에 정의된 finder method들을 모두 구현해야 한다. 또 하나의 단점은 빈을 특정한 데이터베이스 타입이나 구조에 종속시킨다는 점이다. 이는 데이터베이스 또는 데이터 구조 내의 모든 변화가 빈 인스턴스의 정의에 영향을 미치는 것을 의미한다. 따라서 이러한 경우 빈의 flexibility가 나빠지게 된다.

엔티티 빈의 Object Interaction Diagram(OID)

- Object Interaction Diagram은 엔티티 빈 인스턴스(instance)와 그것을 담고 있는

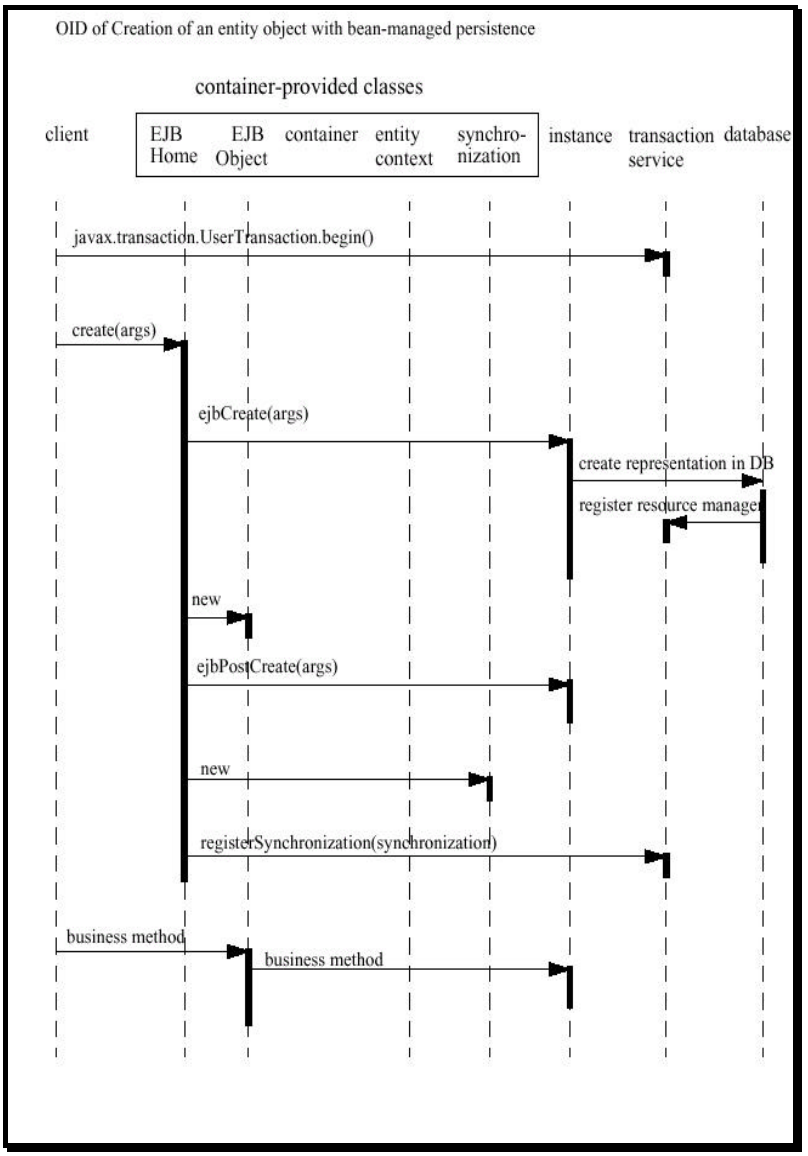
컨테이너와의 상호작용(interactions)을 표현한다.

OID에는 " container-provided classes " 라는 이름의 상자가 있다. 이 클래스들은 컨테이너의 일부분이거나, 컨테이너의 tool들에 의해 생성된다. 이 클래스들은 컨테이너 구현 스펙에 정의된 프로토콜로 서로간에 통신한다. 그러므로, 이러한 클래스들간의 통신은 다이어그램에 나타나지 않는다.

다이어그램에 나타난 클래스들은 규정에 따른 것이라기 보다는 실제 예로써 구현된 것으로 생각하라.

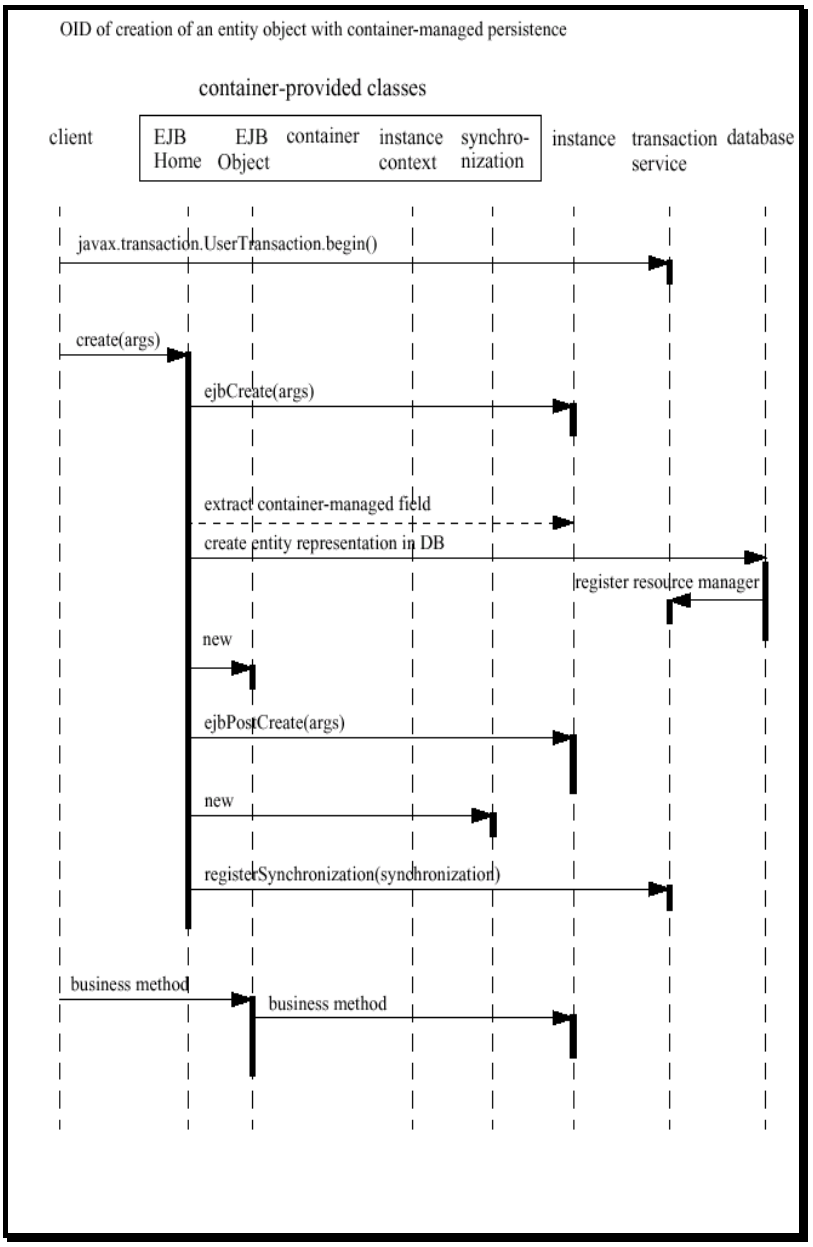
✍✍ Entity 객체(Object)의 생성

BMP 기반 entity 객체의 생성



<그림> BMP기반 엔티티 객체의 생성 OID

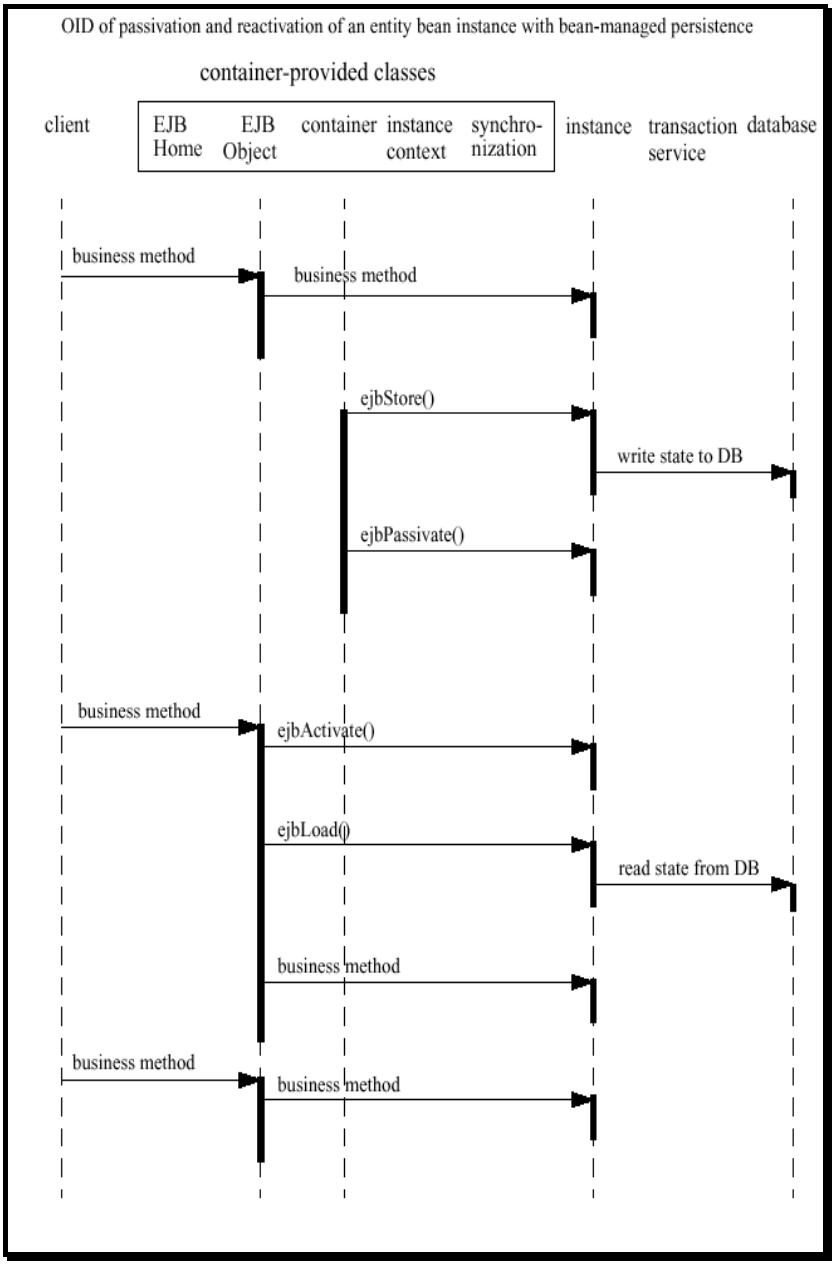
CMP 기반 entity 객체의 생성



<그림> CMP기반 엔티티 객체의 생성 OID

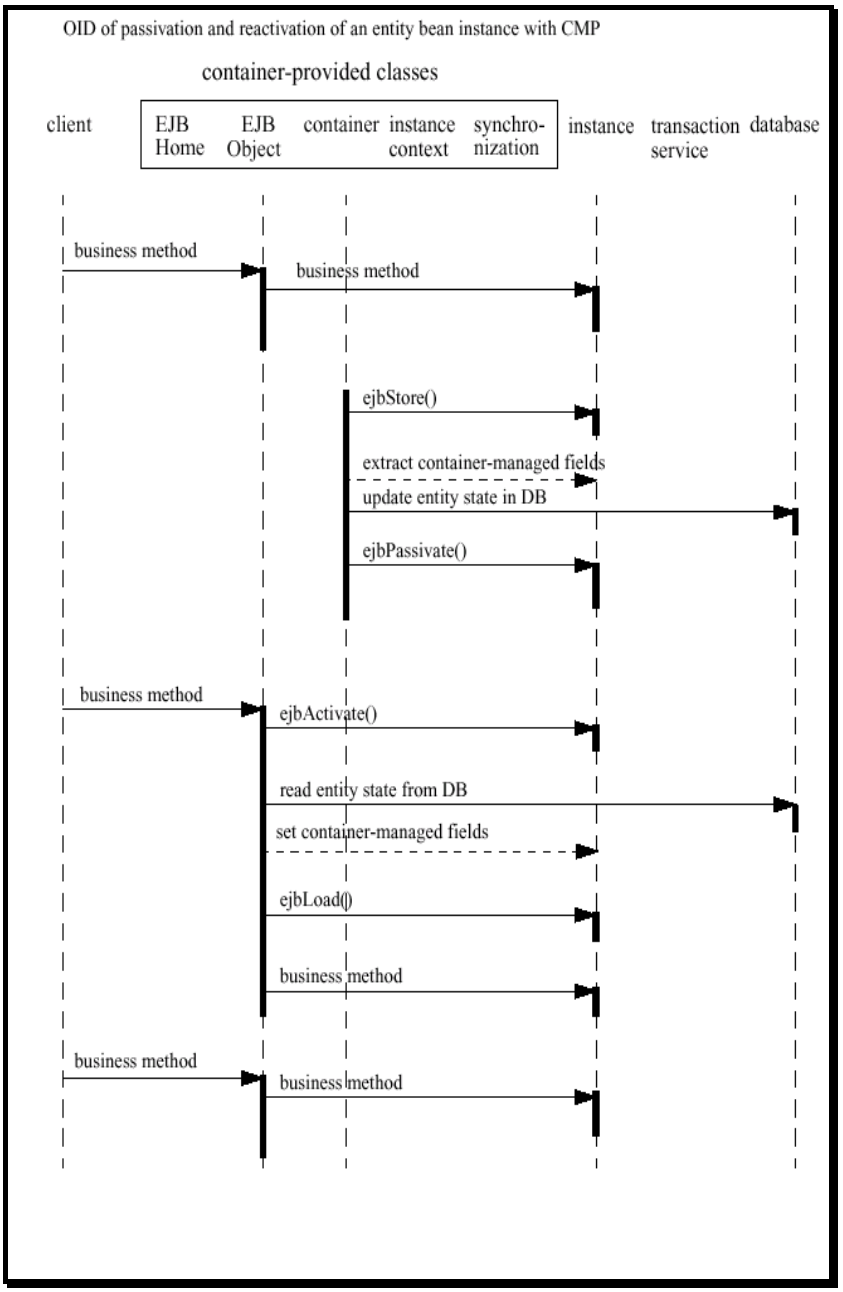
트랜잭션에서 인스턴스의 Passivating and Activation

✍✍ BMP 기반 인스턴스의 Passivation and Reactivation



<그림> BMP기반 엔티티 빈 인스턴스의 활성화 및 비활성화 OID

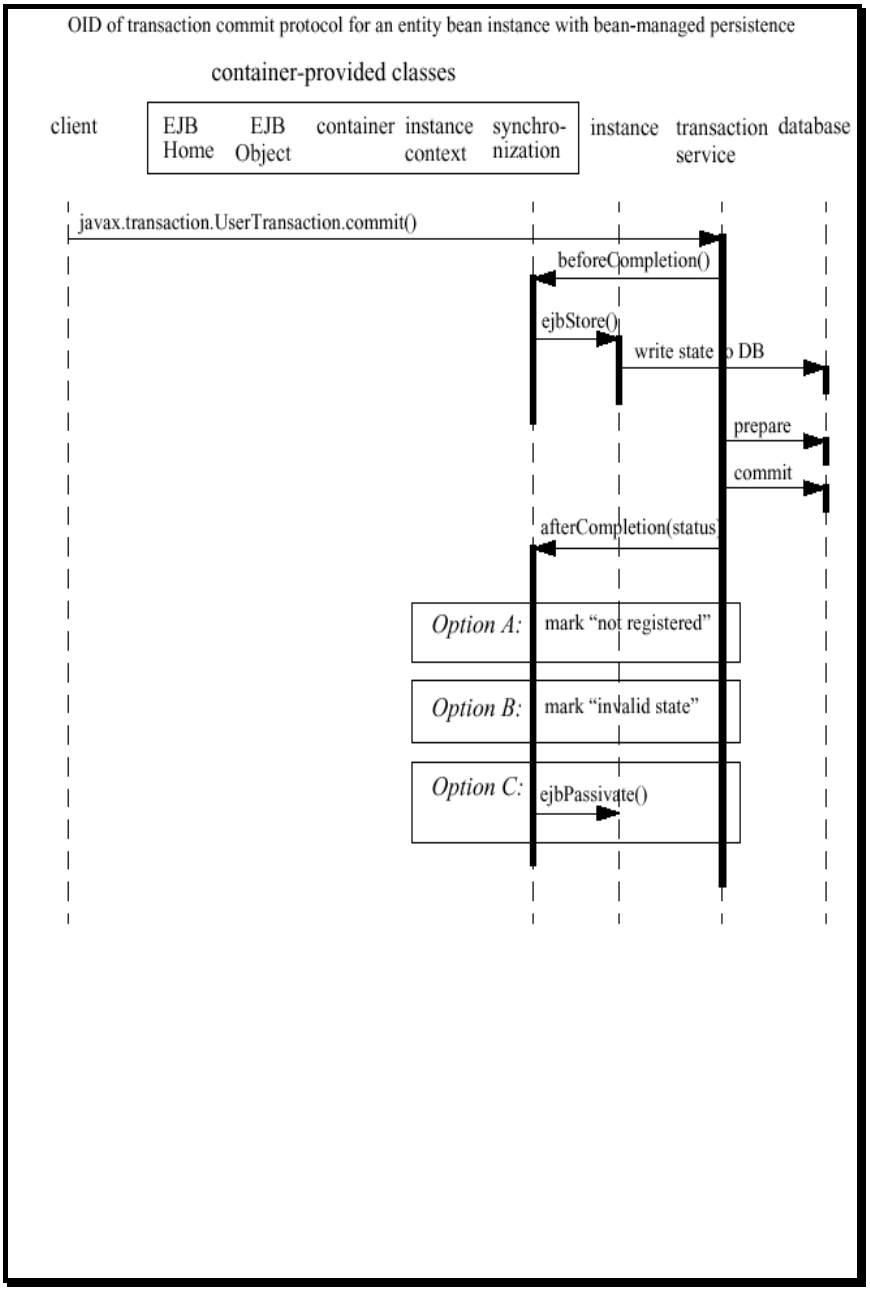
CMP 기반 인스턴스의 Passivation and Reactivation



<그림> CMP기반 엔티티 빈 인스턴스의 활성화 및 비활성화 OID

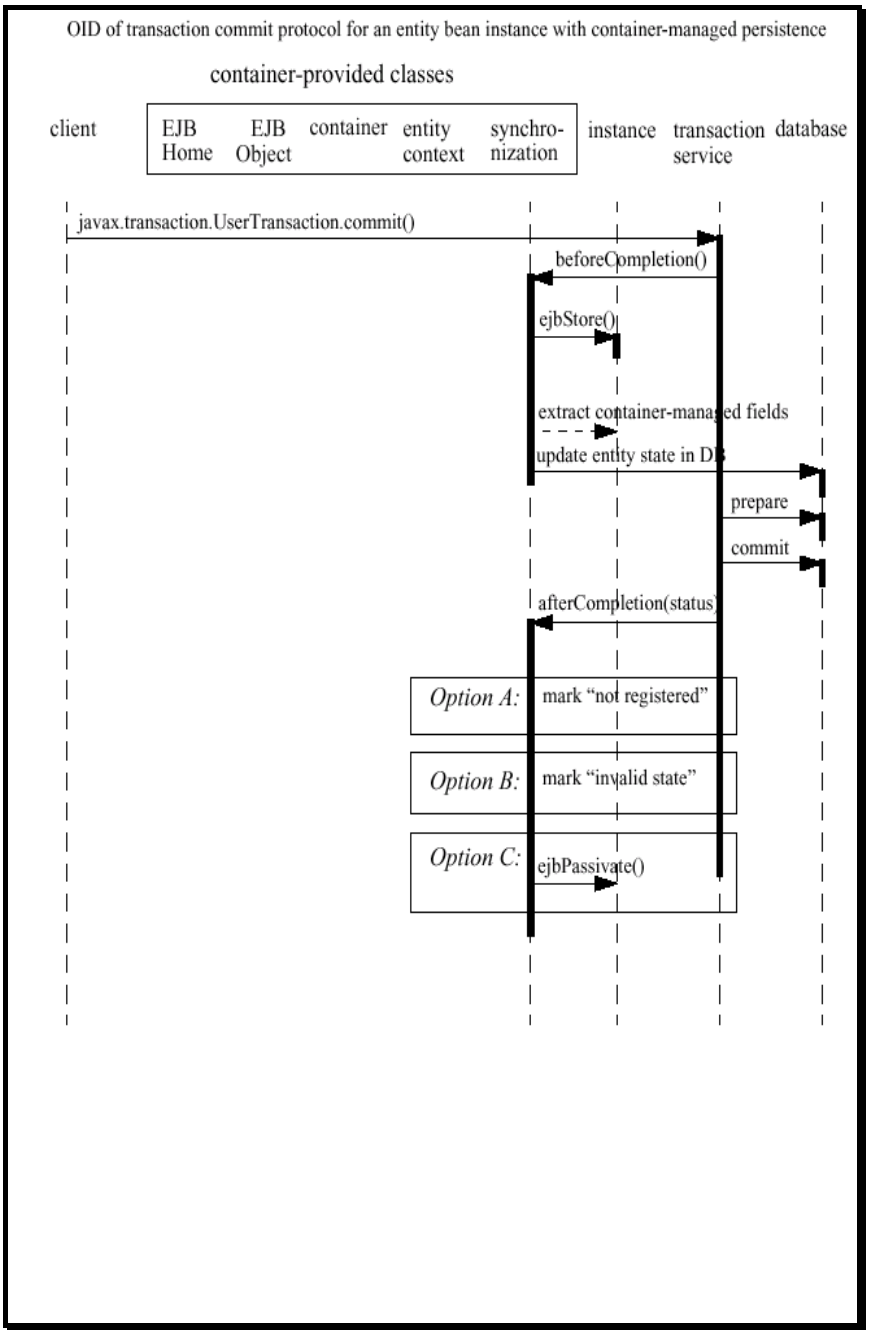
트랜잭션 Committing

✍✍ BMP 기반 엔티티 빈 인스턴스에 대한 트랜잭션 commit protocol



<그림> BMP기반 엔티티 빈 인스턴스에 대한 트랜잭션 commit protocol OID

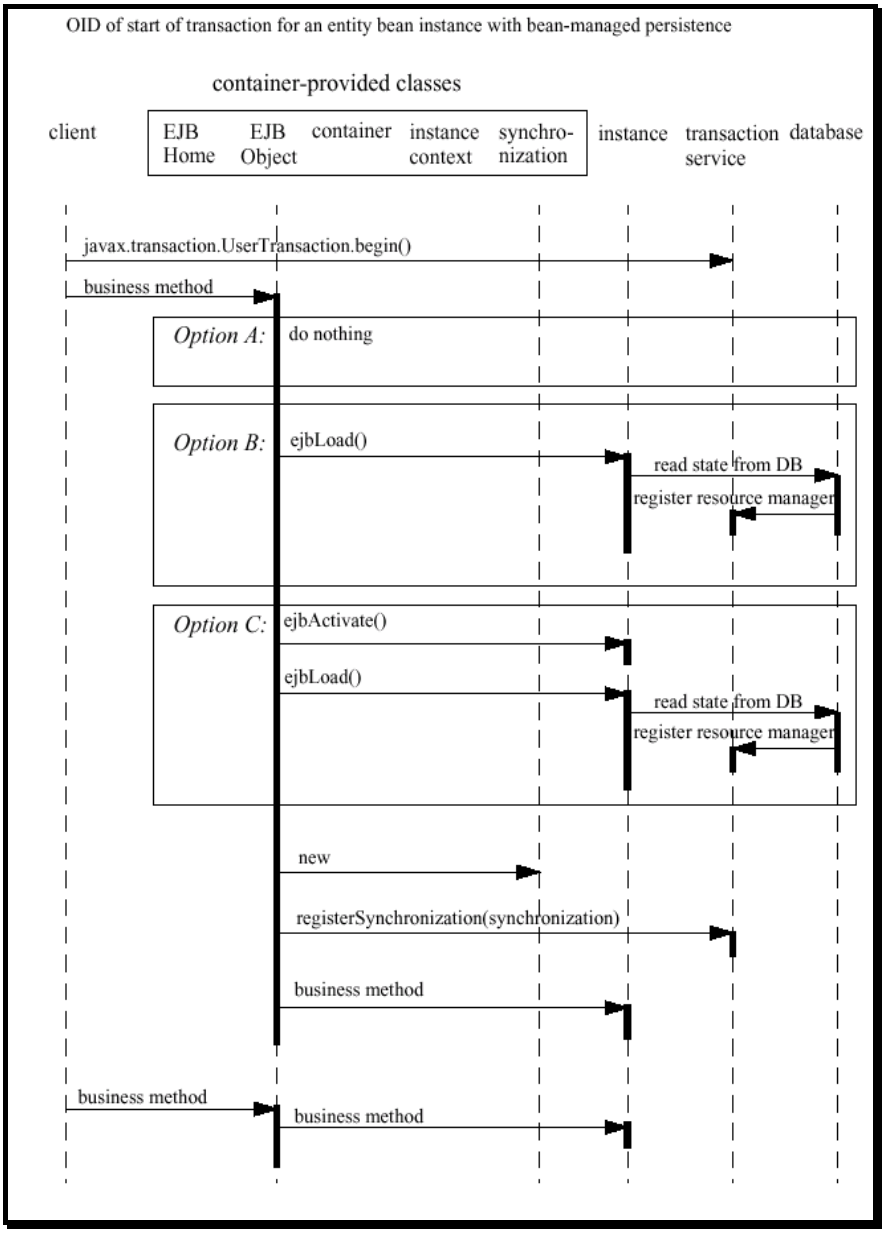
CMP 기반 엔티티 빈 인스턴스에 대한 트랜잭션 commit protocol



<그림> CMP기반 엔티티 빈 인스턴스에 대한 트랜잭션 commit protocol OID

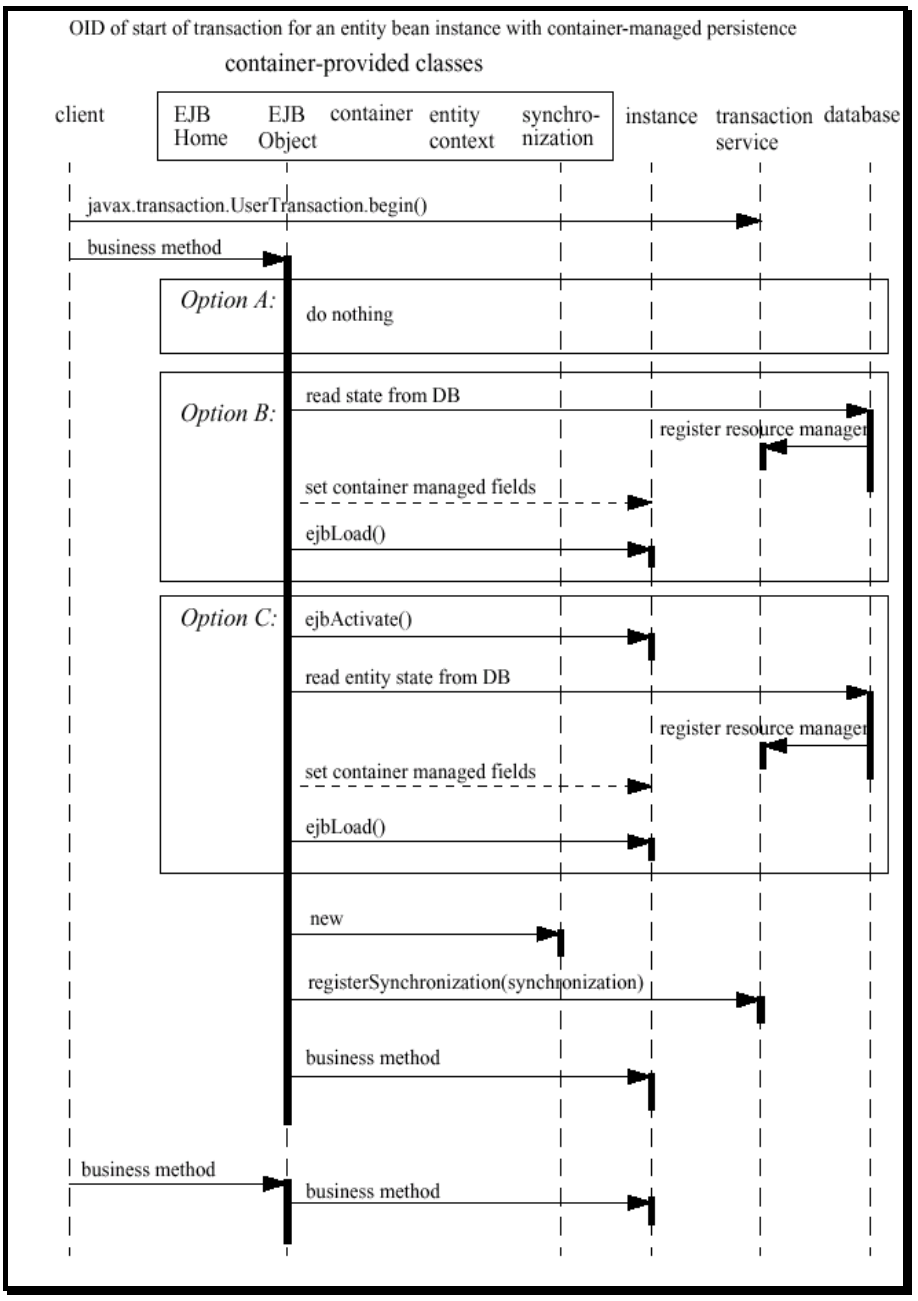
새로운(Next) 트랜잭션의 시작

✍✍ 새로운 트랜잭션의 시작점에서 BMP 기반 엔티티 빈 인스턴스에 대해 실행되는 프로토콜(다이어그램의 3개의 option들은 바로 전의 세개의 commit option들에 대응)



<그림> BMP기반 엔티티 빈 인스턴스에 대한 트랜잭션 시작 OID

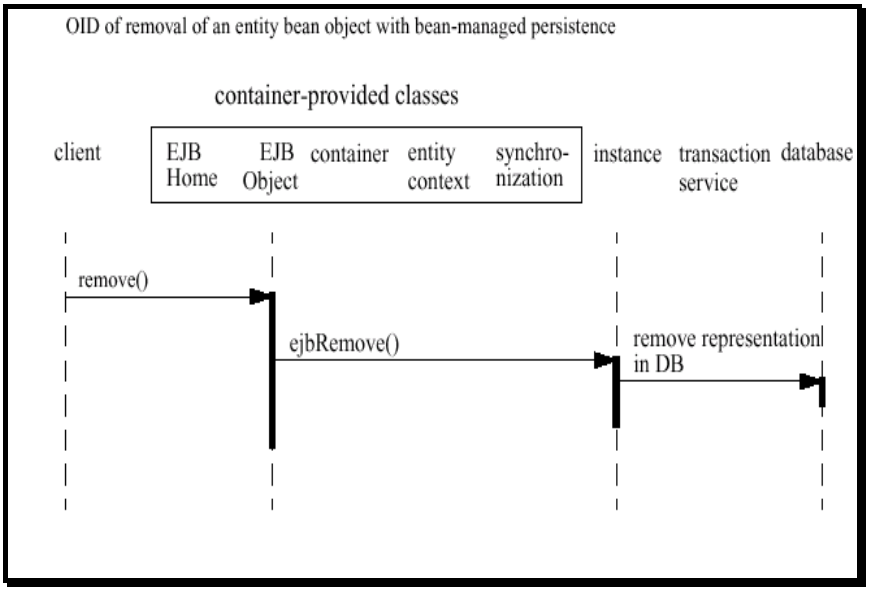
✍✍ 새로운 트랜잭션의 시작점에서 CMP 기반 엔티티 빈 인스턴스에 대해 실행되는 프로토콜(다이어그램의 3개의 option들은 바로 전의 세개의 commit option들에 대응)



<그림> CMP기반 엔티티 빈 인스턴스에 대한 트랜잭션 시작 OID

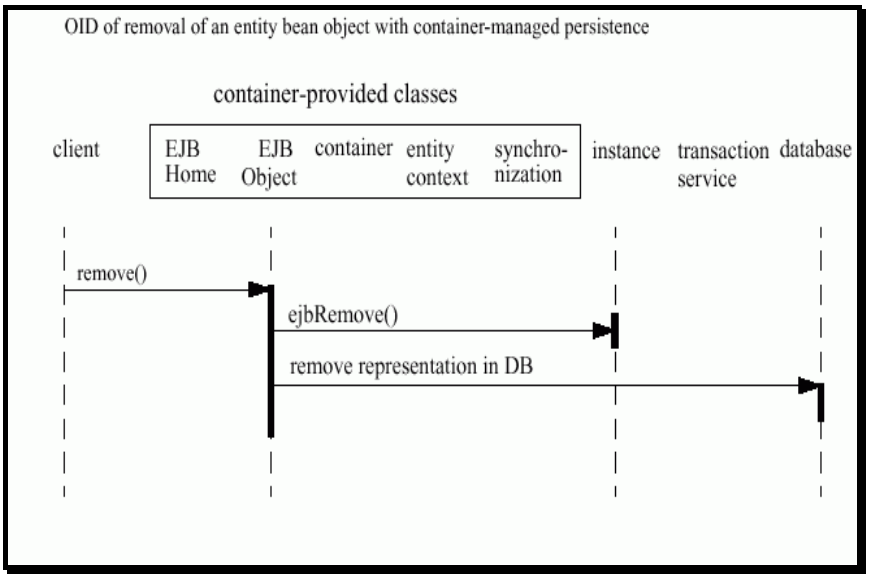
entity 객체(Object)의 삭제

✍ BMP 기반 엔티티 빈 객체의 삭제



<그림> BMP기반 엔티티 빈 객체의 삭제 OID

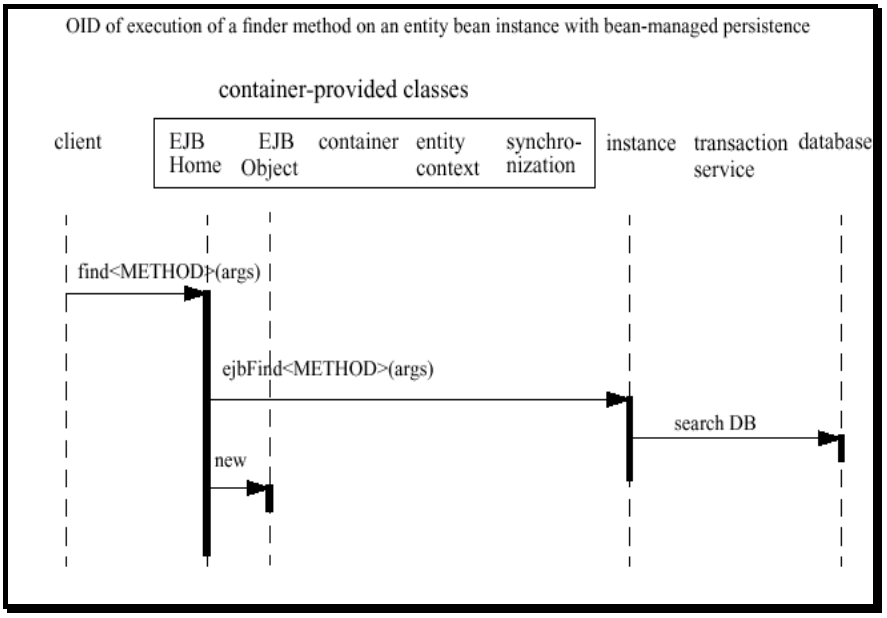
CMP 기반 엔티티 빈 객체의 삭제



<그림> CMP기반 엔티티 빈 객체의 삭제 OID

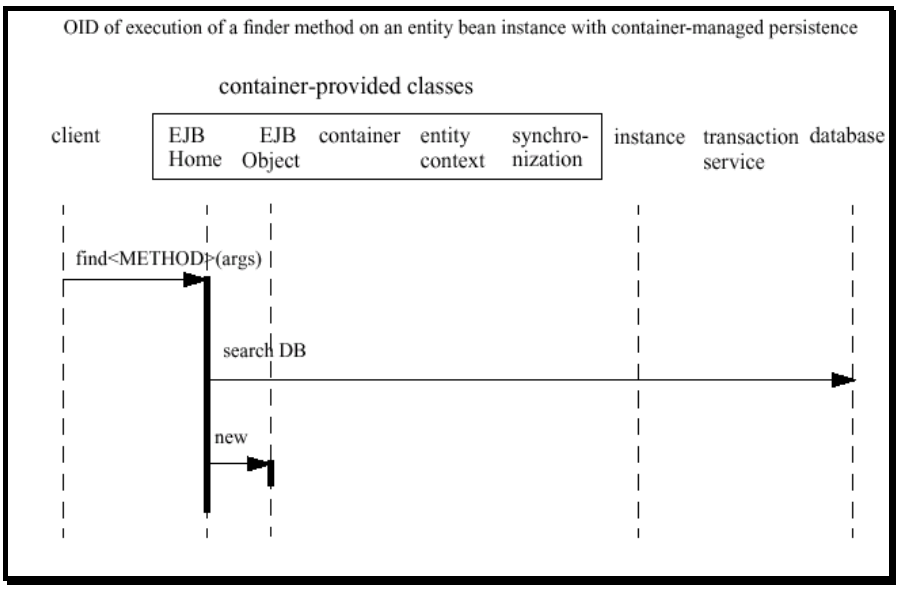
Entity 객체 찾기

✍ BMP 기반 엔티티 빈 인스턴스에 대한 finder method 실행



<그림> BMP기반 엔티티 빈 인스턴스에서 finder method의 실행 OID

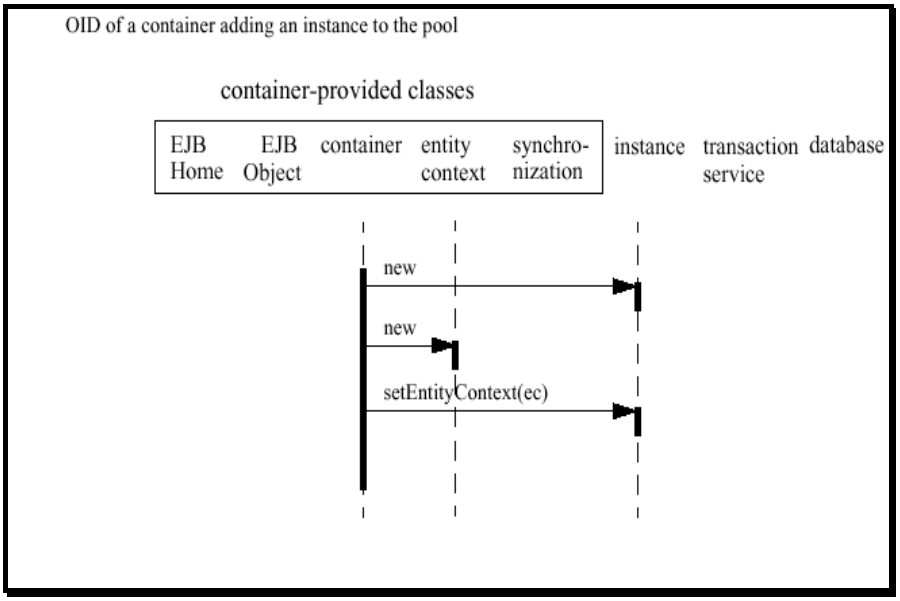
✍ CMP 기반 엔티티 빈 인스턴스에 대한 finder method 실행



<그림> CMP기반 엔티티 빈 인스턴스에서 finder method의 실행 OID

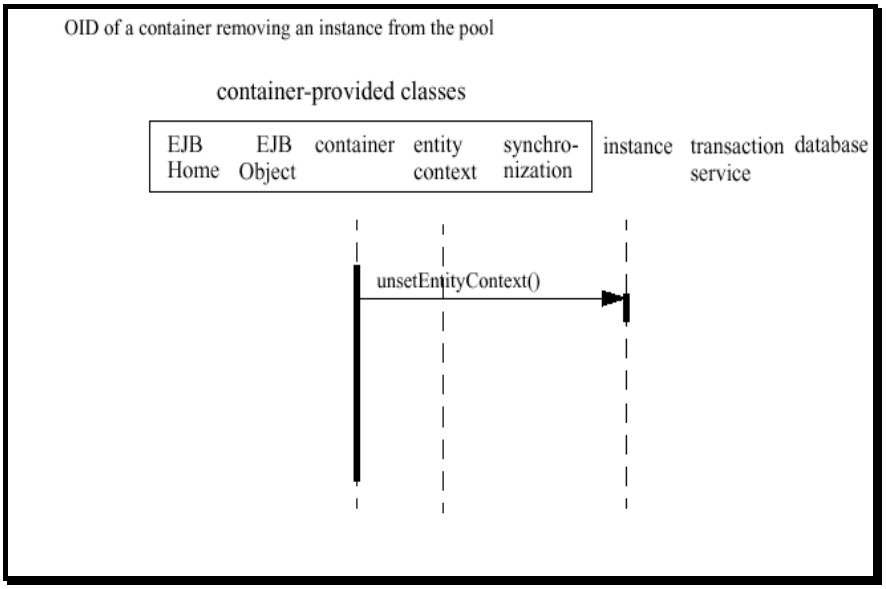
인스턴스를 풀(pool)에 추가 / 삭제

✍✍ 인스턴스를 풀(pool)에 추가



<그림> 컨테이너가 엔티티 빈 인스턴스를 풀에 추가시 OID

✍✍ 인스턴스를 풀(pool)에서 삭제



<그림> 컨테이너가 엔티티 빈 인스턴스를 풀에서 삭제시 OID

✍✍ [3-나]부터 [3-사]까지의 다이어그램은 엔티티 빈의 life cycle중에서 "does not exist"에서 "pooled" 상태까지의 sequence가 나타나 있지 않다.

CMP 기반 엔티티 빈 구현

CMP 기반 엔티티 빈 예제 개요

- 다음과 같은 경우를 생각해 보자.
- 슈퍼마켓에 진열되어 있는 상품들의 경우는 상품 번호, 상품 설명(또는 상품명), 가격으로 크게 세가지의 속성(property)를 가진다.
- 이 상품들은 슈퍼마켓이 영업을 끝내도 존재해야 하고 다음날 다시 팔릴 준비를 해야한다. 즉, 영속적인 성격(Persistence)을 가지고 있다. 또한, 상품은 주인에 의해 새롭게 진영되기도 하고 기존에 있던 상품이 진열장에서 치워지기도 한다.
- 슈퍼마켓 주인은 어떤 상품에 대해 가격을 새로 매기거나 또는 그 상품이 얼마인지 알고 싶다. 그리고, 특정 상품 번호에 대해 그 상품이 어떤것인지 상품 설명를 알고 싶다.
- 슈퍼마켓 주인은 또한, 상품 번호를 통한 상품 검색과 상품 설명을 통한 상품 검색 그리고 몇번부터 몇번까지의 범위를 통해서 여러 개의 상품을 검색하고 싶다. 그러나 주인은 상품이 어디에 위치하는지 정확히 알 필요는 없다. 왜냐하면 슈퍼마켓은 아르바이트 학생을 고용하였기 때문에 상품의 진열이나 찾아오는 것등은 모두 아르바이트생이 전적으로 도맡아서 해결해 주기 때문이다.
- 여기서 슈퍼마켓 주인이 EJB를 사용한다면 과연 어떻게 이러한 슈퍼마켓의 비즈니스를 형상화 하고 구현했을까?
- 슈퍼마켓 주인은 EJB의 엔티티 빈에 대해 알아야 한다. 왜냐하면 상품들은 영속적인 속성을 가지고 있고 이들은 데이터베이스의 테이블에 대응될 수 있으므로 이러한 특징을 가지는 엔티티 빈에 의해 구현되어야 하는 것이다.
- 상품들을 데이터베이스에 저장하고 이 들을 이제 구현할려고 하는 Product 엔티티 빈에 대응시키면 슈퍼마켓의 비즈니스 로직은 쉽게 해결될 수 있다.

  슈퍼마켓의 상품 집합은 데이터베이스의 상품 테이블에 대응될 수 있다.

  각 상품은 데이터베이스의 상품 테이블의 열(row)로 생각할 수 있다.

  상품은 3가지 속성(상품 번호, 상품 설명, 가격)을 가지고 있고, 우리가 사용할 테이블의 열은 3가지 컬럼(productId, Description, Price)을 가지고 있다.

  슈퍼마켓의 주인이 하기 원하는 작업들(상품의 가격정하기, 상품 가격 알기, 상품 설명 알기) 즉, 비즈니스 method들은 리모트 인터페이스에 정의되고 엔티티 빈 클래스에서 실제 구현된다.

  주인이 새로운 상품을 들여오거나 기존의 상품들을 찾고, 진열대에서 없애는 작업들은 홈 인터페이스에 정의된다.

  주인은 직접 상품을 찾으러 가거나 진열할 필요가 없다. 왜냐면 아르바이트생이 이 모든 일들을 알아서 해주기 때문이다. 따라서 이러한 비즈니스 로직은 엔티티 빈에서도 Persistence를 컨테이너가 관리해 주는 CMP 기반 엔티티 빈으로 구현할 수 있다.

✍✍ 여기서 예제로 살펴볼 CMP 기반 엔티티 빈은 상품(product)를 나타내는 것으로 다음 3개의 엔티티 빈 구성파일과 1개의 client program으로 이루어진다.

엔티티 빈 구성 파일

리모트 인터페이스 : Product.java

홈 인터페이스 : ProductHome.java

엔티티 빈 클래스 : ProductEJB.java

Client program

ProductClient.java

✍✍ 엔티티 빈이 CMP 기반이거나 BMP 기반이거나 관계없이 리모트 인터페이스와 홈 인터페이스는 동일하다. 엔티티 빈 클래스에 있는 코드는 당연히 CMP와 BMP가 서로 다르다. CMP의 엔티티 빈 클래스는 어떤 데이터베이스 접근 / 제어 코드도 가지고 있지 않다. Application Deployment Tool이 필요한 SQL 구문들을 자동으로 생성해 준다. 이렇게 하기 위해서는 tool이 데이터베이스에 저장되어야 하는 인스턴스 변수들을 알아야 한다. 이런 인스턴스 변수들을 container-managed field라도 한다.

✍✍ Container-managed field

엔티티 빈 클래스인 ProductEJB.class 파일은 다음과 같은 container-managed field들을 가지고 있다.

```
public String productId;
public String description;
public double price;
```

이 필드들이 ProductEJB 인스턴스들의 상태를 나타내고 데이터베이스에 저장되게 된다. 개발자는 이러한 container-managed field들을 Application Deployment Tool에서 지정해 주어야 한다(J2EE 서버의 경우, Deploy tool의 New Enterprise Bean Wizard나 Entity tab).

Container-managed field는 반드시 public 이고 임시적인 것 이어서는 안된다. 이것은 다음중 하나의 형이어야 한다.

Java serializable 클래스

Java primitive

홈 인터페이스에 대한 레퍼런스

리모트 인터페이스에 대한 레퍼런스

Application Deployment Tool을 사용할 때, 하나 이상의 container-managed filed를 엔티티 빈의 primary key field로 선언해야 한다.

리모트 인터페이스 구현

✍✍ 리모트 인터페이스의 정의는 비즈니스의 목적을 정의하는 것이라고 할 수 있다. 실제 클라이언트는 리모트 인터페이스와 홈 인터페이스를 통해서만 Enterprise bean에 접근한다. 홈 인터페이스의 경우 bean 인스턴스의 생성과 라이프 사이클에 관련된 method를 클라이언트에게 제공한다. 결국 비즈니스 로직에 관련된 method는 리모트 인터페이스에서 정의된다고 할 수 있고, 여기서 정의된 비즈니스 method가 실제 엔티티 빈 클래스에서 구현되는 것이다.

✍✍ 속성의 설정(set) 및 참조(get)

리모트 인터페이스에서는 주로 엔티티 빈 인스턴스 변수의 값을 설정(set)하거나 참조(get)하는 method들로 이루어진다. 이들은 때때로 setter와 getter로 불리고 이들이 조작하는 변수들을 프로퍼티(properties)라고 한다. 이런 프로퍼티들은 이들이 실제 저장될 데이터베이스와 같은 데이터의 저장매체와 독립적으로 정의되어야 한다. 즉, 리모트 인터페이스는 사용할 데이터가 아니라 비즈니스 로직을 만족할 수 있도록 설계되어야 하는 것이다.

기반 데이터(underlying data)가 아닌 비즈니스 로직에 따라 엔티티 빈의 속성을 정의하는 것은 쉽지 않다. 하지만, 다음과 같은 이유로 이와 같이 하기 위해 노력해야 한다.

기반 데이터가 항상 엔티티 빈에 통해 구현하려는 비즈니스의 목적이나 개념을 명확하게 정의하지는 않는다. 리모트 인터페이스는 데이터베이스에 의해 사용되는 것이 아니라 엔티티 빈에 비즈니스 로직이 반영되는 것이 중요한 비즈니스를 아는 개발자에 의해 사용될 것이기 때문이다.

엔티티 빈의 속성이 데이터로부터 독립적이 되면, 엔티티 빈과 데이터의 상호 독립적인 성격을 강하게 한다. 이것은 데이터베이스의 구현이 시간과 환경에 따라 변화할 수 있기 때문에 중요한 문제이다. 또한, 데이터로 부터의 독립은 필요에 따라 엔티티 빈에 새로운 동작을 추가할 수 있게 한다.

Product의 리모트 인터페이스에는 다음과 같은 3개의 method가 정의되어 있다. 엔티티 빈 인스턴스의 price 속성을 설정하고 참조하는 setPrice()와 getPrice(), 엔티티 빈 인스턴스의 Description을 읽어오는 getDescription() method이다.

다음은 리모트 인터페이스의 전체 소스 코드이다.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface Product extends EJBObject
{
    public void setPrice(double price)
            throws RemoteException;
    public double getPrice()
            throws RemoteException;
    public String getDescription()
            throws RemoteException;
}
```

<소스> Product.java

홈 인터페이스 구현

홈 인터페이스는 엔티티 빈 인스턴스의 생명주기에 관련된 method들을 제공한다. 즉, 모든 엔티티 빈의 홈 인터페이스는 EJB 시스템에서 인스턴스를 생성하고, 찾고, 삭제하는데 사용된다.

각각의 엔티티 빈 타입은 그 자체의 홈 인터페이스를 가진다. 홈 인터페이스는 없거나 또는 하나 이상의 create method와 하나 이상의 finer method를 정의한다. Create method는 product의 기본 속성 값을 입력 전달인자로써 가지는 형태이고, finder method는 3개가 정의되어 있다.

EJB의 홈 인터페이스는 반드시 javax.ejb.CreateException을 포함해야 하는데 CMP의 경우 컨테이너는 생성 과정에서 유발되는 통신문제에 대한 일반적인 예외처리가 필요하다.

✍✍ create method

엔티티 빈의 홈 인터페이스는 0 또는 그 이상의 create method(각각은 저마다의 방법으로 entity 객체를 생성한다.)를 정의할 수 있다. 일반적으로 create method들의 전달인자는 생성된 entity 객체들의 상태(state)를 초기화 하는데 사용된다.

create method의 반환값은 엔티티 빈의 리모트 인터페이스이다.

모든 create method의 throws 절은 java.rmi.RemoteException과 javax.ejb.CreateException을 포함하고 있다. 또한, 추가적으로 응용프로그램 차원에서 제공되는 Exception들을 포함할 수 있다.

다음은 ProductHome.java의 create method이다.

```
public Product create(String productId, String description,
        double balance) throws RemoteException, CreateException;
```

클라이언트는 다음과 같이 홈 인터페이스의 create method를 이용하여 새로운 entity 객체를 생성한다.

```
ProductHome home =
    (ProductHome)PortableRemoteObject.narrow(objref,
        ProductHome.class);
Product duke = home.create("123", "Ceramic Dog", 10.00);
System.out.println(duke.getDescription() + ": " + duke.getPrice());
```

다음은 홈 인터페이스의 전체 소스 코드이다.

```
import java.util.Collection;
import java.rmi.RemoteException;
import javax.ejb.*;
```

```
public interface ProductHome extends EJBHome
{
    public Product create(String productId, String description,
            double balance) throws RemoteException, CreateException;

    public Product findByPrimaryKey(String productId)
            throws FinderException, RemoteException;

    public Collection findByDescription(String description)
            throws FinderException, RemoteException;

    public Collection findInRange(double low, double high)
            throws FinderException, RemoteException;
}
```

<소스> ProductHome.java

엔티티 빈 클래스 구현

✍ ProductEJB 클래스는 CMP에 기반하고 있다. 이 클래스는 어떤 데이터베이스 제어 루틴도 가지고 있지않기 때문에 상당히 간결하다.

✍ ejbCreate Method

ejbCreate method는 입력 전달인자로 들어오는 container-managed field들을 초기화 한다. 이 method는 반환값으로 null을 가지는데 그 이유는 CMP 기반의 경우 컨테이너가 반환값을 무시하기 때문이다. ejbCreate method가 실행되고 나면 컨테이너는 container-managed field들을 데이터베이스에 삽입한다.

다음은 ProductEJB.class의 ejbCreate method의 코드이다.

```
public String ejbCreate(String productId, String description,
                        double price) throws CreateException
{
    if (productId == null) {
            throw new CreateException("The productId is required.");
    }

    this.productId = productId;
    this.description = description;
    this.price = price;
    return null;
}
```

✍ ejbRemove method

클라이언트가 remove method를 호출하면, 컨테이너는 ejbRemove method를 호출한다. ejbRemove의 실행이 완료되면, 컨테이너는 데이터베이스에서 해당 열(row)를 삭제한다. 만일 컨테이너가 데이터 열을 삭제하는데 실패하게 되면, 예외(exception)을 발생시킨다.

만일 엔티티 빈이 삭제하기 전에 즉각적인 어떤 처리가 필요하다면 ejbRemove method에서 하면 된다. ProductEJB 클래스는 그런 작업이 필요없기 때문에 ejbRemove는 아무일도 하지 않는 비어있는 method이게 된다.

✍✍ ejbLoad method

컨테이너는 다음 과정을 거쳐 데이터베이스로부터 엔티티 빈의 상태를 재설정한다.

데이터베이스에서 재설정하고자 하는 엔티티 빈에 해당하는 열(row)을 가져(select) 온다.

가져온 열의 각 컬럼 값을 대응하는 각 container-managed field에 할당한다.

ejbLoad method를 호출한다.

일반적으로 ejbLoad method는 아무것도 하지않고 비어있다. 그러나, 엔티티 빈은 이 method를 데이터베이스로부터 읽어온 값들을 전달하는데 사용할 수도 있다. 예를 들면, ejbLoad method는 데이터베이스로부터 읽어온 특정 형태로 압축된 텍스트 데이터를 비즈니스 method들이 사용할 수 있는 형태로 바꾸는 역할(압축 해제)을 할 수도 있는 것이다.

✍✍ ejbStore method

컨테이너는 다음 과정을 거쳐 엔티티 빈의 상태를 데이터베이스에 저장한다.

ejbStore method를 호출한다.

Container-managed field들의 값들을 읽어온다.

해당 엔티티 빈에 대응하는 데이터베이스의 열(row)의 각 컬럼값들을 읽어온 container-managed field값들로 대체한다.

ejbLoad method와 마찬가지로 ejbStore method는 일반적으로 아무것도 하지 않은채로 비어있다. 그러나, 만일 container-managed field들을 컨테이너가 데이터베이스에 저장하기 전에 특정한 형태로 변환하기를 원한다면, 이러한 작업은 ejbStore method에서 실행하면 된다. 예를 들면, 개발자는 ejbStore method에서 텍스트 데이터를 컨테이너가 데이터베이스에 저장하기 전에 특정 형태로 압축할 수 있다.

✍✍ Finder methods

다음은 Product홈 인터페이스에 정의되어 있는 finder method들이다.

```
public Product findByPrimaryKey(String productId)
throws FinderException, RemoteException;

public Collection findByDescription(String description)
throws FinderException, RemoteException;
```

```
public Collection findInRange(double low, double high)
        throws FinderException, RemoteException;
```

ProductEJB 클래스가 CMP 기반이기 때문에, 이 finder method들은 구현할 필요가 없다. Application Deployment Tool가 데이터베이스로부터 엔티티 빈의 상태를 저장하고 있는 열(row)을 읽어오는 SQL 구문을 포함한 findByPrimaryKey method를 구현해 준다. 또한 tool은 findByDescription이나 findInRange와 같은 응용프로그램에 커스터마이징된 method들도 구현해준다. 다만, 이런 경우에 개발자는 이 method들에 해당하는 select SQL 구문의 where절을 설정해 주어야 한다. 이것은 뒷 부분의 Deployment하는 과정에서 살펴보게 된다.

✍✍ 데이터베이스 테이블 만들기

CMP에서는 Application Deployment Tool이 엔티티 빈에 대한 create table SQL 구문을 생성해 준다. 따라서 개발자가 미리 테이블을 생성할 필요가 없다. 예를 들면, ProductEJB 클래스에 대해서는 다음과 같은 SQL 구문을 생성한다.(여기서 주의 해야 할 점은 큰 따옴표도 테이블, 컬럼 및 constraint name들의 일부분이라는 것이다. 따라서 아래 SQL 구문에서, 테이블 이름은 따옴표 없는 ProductEJBTable이 아니라 "ProductEJBTable"이다.)

```
CREATE TABLE "ProductEJBTable" (
   "description" VARCHAR(255) ,
   "price" DOUBLE PRECISION NOT NULL ,
   "productId" VARCHAR(255),
   CONSTRAINT "pk_ProductEJBTable"
PRIMARY KEY ("productId") )
```

EJB 컨테이너는 엔티티 빈을 담고 있는 응용프로그램이 배치(deploy)될때 이 create table SQL 구문을 기본으로 실행한다. 또한, 당연히 응용프로그램이 uninstall되면 EJB 컨테이너는 생성된 테이블을 drop시킨다.

J2EE Server에서는 다음과 같은 과정을 통해 deployment동안 컨테이너가 테이블을 생성하지 않도록 할 수 있다.

Application Deployment Tool에서 entity bean의 Entity 탭을 선택한다.

Entity 탭 영역에서 Deployment Settings을 클릭한다.

Deployment Settings 대화상자에서 "Create Table on Deploy" 항목의 선택을 해제한다.

✍✍ 다음은 엔티티 빈 클래스의 전체 소스 코드이다.

```
import java.util.*;
import javax.ejb.*;

public class ProductEJB implements EntityBean
{
    public String productId;
    public String description;
    public double price;
    private EntityContext context;
    public void setPrice(double price)
    {                   this.price = price; }
    public double getPrice()
    {           return price;
    }

    public String getDescription()
    {           return description;
    }

    public String ejbCreate(String productId, String description,
                double price) throws CreateException
    {
                if (productId == null) {
                        throw new CreateException("The productId is required.");
                }

                this.productId = productId;
                this.description = description;
                this.price = price;
                return null;
    }

    public void setEntityContext(EntityContext context)
    {
                this.context = context;
    }

    public void ejbActivate()
```

```
    {
            productId = (String)context.getPrimaryKey();
    }

    public void ejbPassivate()
    {
            productId = null;
            description = null;
    }

    public void ejbRemove() { }
    public void ejbLoad() { }
    public void ejbStore() { }
    public void unsetEntityContext() { }
    public void ejbPostCreate(String productId, String description, double
                                        balance) { }
} // ProductEJB
```

<소스> ProductEJB.java

Primary Key 클래스 구현

- Primary key는 bean의 형태, 홈 인터페이스, 그리고 그것이 사용되는 컨테이너 컨텍스트(container context)에 따른 엔티티 빈을 유일하게 구분하는 구분자이다. CMP에서 primary key는 bean에 한정되어 정의된 직렬화된 객체이다.
- Product 엔티티 빈에서는 Primary key 클래스를 구현하지 않았다. 왜냐하면, product 클래스에서 primary key는 String형의 productId로써 클래스에 대한 아무런 정의를 하지않고 java.lang.String 클래스를 primary key 클래스로 사용하게 되는 것이다.
- 하지만, 대부분의 어플리케이션들은 매우 복잡한 비즈니스 로직을 담고 있다. 따라서 primary key도 단순하게 위와 같은 경우도 있겠지만, 여러 개의 필드가 모여서 하나의 primary key를 형성할 수도 있다. 이러한 경우에는 반드시 primary key 클래스를 구현해야 한다. 또한, primary key가 primitive type 일지라도 primary key 클래스를 구현해 놓으면 차후 어플리케이션의 확장이나 데이터베이스 및 테이블의 변동에서 오는 영향을 최소화 시키고, bean의 finder method들을 좀 더 유연하고 융통성있게 만든다.
- Primary key 클래스의 반환값의 형태는 반드시 RMI-IIOP에 적법한 값의 형태이어야 한다. 또한, primary key 클래스는 클라이언트 프로그램의 primary key에 대한 관리를 간단히 하기 위해서 적합한 hashCode() 와 equals(Object other) method들을 제공해야 한다.
- 만일 Product 엔티티 빈에 대한 primary key 클래스를 구현한다면 다음과 같이 간단하게 구현할 수 있다. 하지만, 여러 개의 값을 가진 보다 복잡한 primary ke

y의 경우에는 hash 테이블에서의 충돌을 가능한 줄이기 위해 좀더 개선된 has h code 알고리즘을 필요로 하게 된다.

```
import java.io.Serializable;

public class ProductPK implements java.io.Serializable
{
    public String productId;
    public ProductPK() {}
    public ProductPK(String _Id)
    {
            productId = _Id;
    }
    public boolean equals(Object obj)
    {
            if (obj == null || !(obj instanceof ProductPK)) {
                    return false;
            }
            else if ( ((ProductPK)obj).productId.compareTo(productId)
== 0 ) {
                    return true;
            }
            else {
                    return false;
            }
    }
    public String hashCode()
    {
            return productId;
    }
};
```

<소스> ProductPK.java

Deployment Descriptor 작성 및 어플리케이션 배치

✍✍ 우리는 6장에서 Sample BMP 기반 엔티티 빈의 deployment descriptor의 작성과 실제 J2EE 어플리케이션 서버에 Enterprise bean을 deploy하는 과정을 살펴보았다. 현재 구현하고 있는 엔티티 빈이 CMP 기반이기 때문에 데이터베이스 관련한 부분만 약간 다를뿐 전체적인 과정은 매우 비슷하다. 여기에서는 J2EE 서버에서 CMP 기반 엔티티 빈의 deployment descriptor를 작성할 때 데이터베이

스 부분의 처리를 어떻게 하는지를 중점적으로 살펴보면서 설명하도록 하겠다.

✍✍ 각 소스 코드 컴파일

Enterprise bean을 작성하는데 필요한 리모트 인터페이스, 홈 인터페이스, 엔티티 빈 클래스를 컴파일한다. 컴파일 하는 과정은 6장의 sample 프로그램과 동일하므로 이를 참고한다.

```
Javac –classpath %CPATH% Product.java
Javac –classpath %CPATH% ProductHome.java
Javac –classpath %CPATH% ProductEJB.java
```

✍✍ 데이터베이스 시동

J2EE Server에는 Cloudscape DBMS가 번들로 내장되어 있다. 우리의 예제는 이 DB를 사용하여 구동되게 될 것이다.

```
cloudscape - start
```

CMP기반의 엔티티 빈 Deployment descriptor 작성에서 중요한 점은 미리 데이터베이스 테이블을 생성하지 않는다는 것이다. 즉, CMP기반에서는 컨테이너가 자동으로 필요한 테이블을 생성해준다.

✍✍ Application Deployment Tool

J2EE 서버를 실행 시키고 Application Deployment Tool을 실행시킨다.

```
j2ee –verbose
deploytool          (새로운 command shell에서 실행)
```

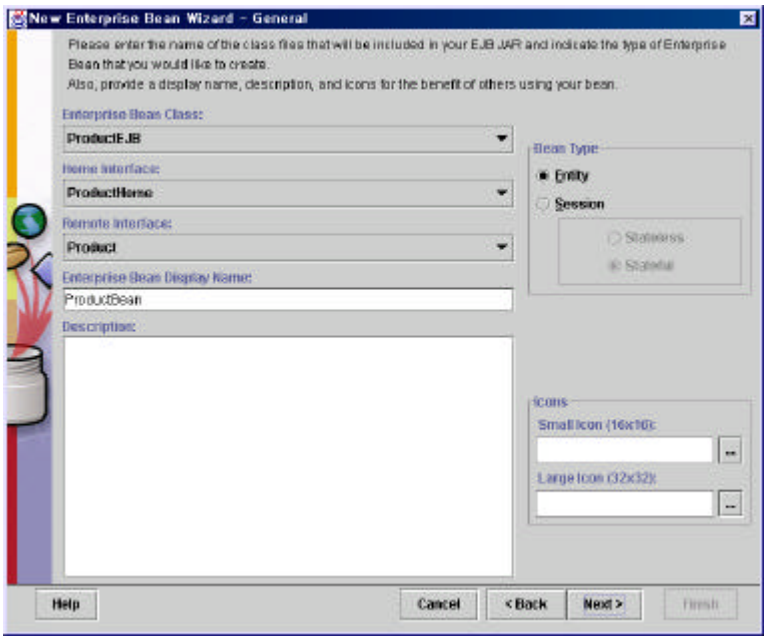
Tool에서 [New Enterprise Bean Wizard]를 실행시킨다.

앞의 과정은 6장과 동일하므로 ProductEJB와 같은 CMP기반의 엔티티 빈에 대해서만 적용되는 특별한 설정만 나타내겠다.

✍✍ [General Dialog Box]

[Entity] 라디오 버튼을 선택한다.

[Display Name] 필드에 ProductBean이라고 입력한다.



<그림. General Settings
대화상자>

✍ [Entity Settings Dialog Box]

[Container-managed Persistence] 라디오 버튼을 선택한다.

엔티티 빈 클래스의 container-managed field 체크 박스를 선택한다.

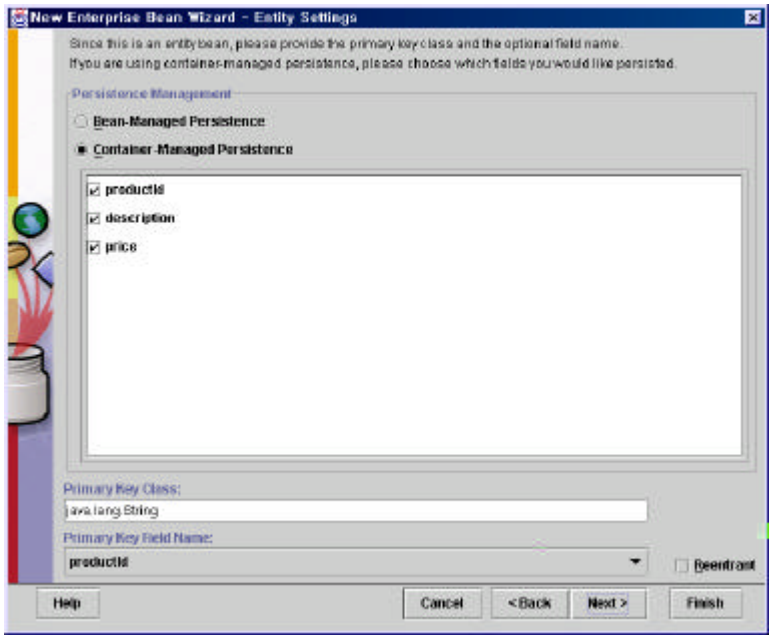
productId

description

price

[Primary Key Class] 필드에 java.lang.String을 입력한다. 즉, 여기서 새롭게 primary key 클래스를 정의하지 않고 String 클래스를 사용하는 것이다.

[Primary Key Field Name] 필드에서 productId를 선택한다.



<그림. Entity Settings 대화상자>

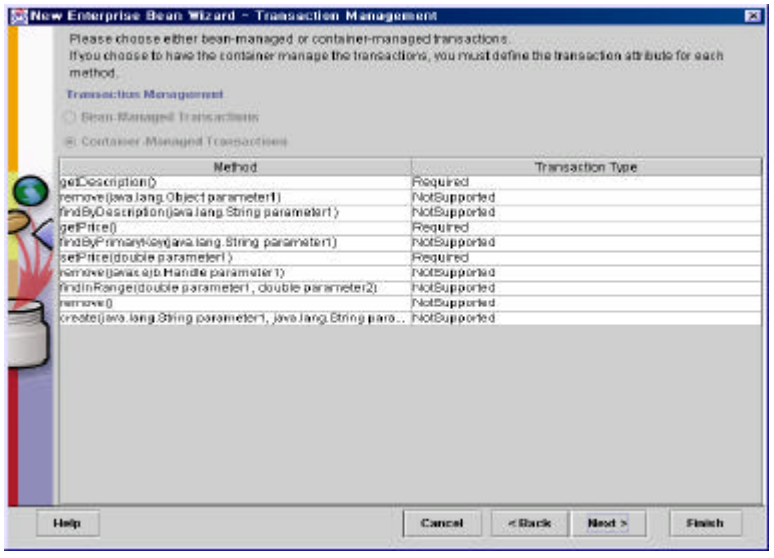
✍ [Transaction Managedment Dialog Box]

비즈니스 method들에 대해 [Transaction Type]란에서 [Required]를 선택한다. ProductBean의 비즈니스 method들은 다음과 같다.

getDescription

setPrice

getPrice

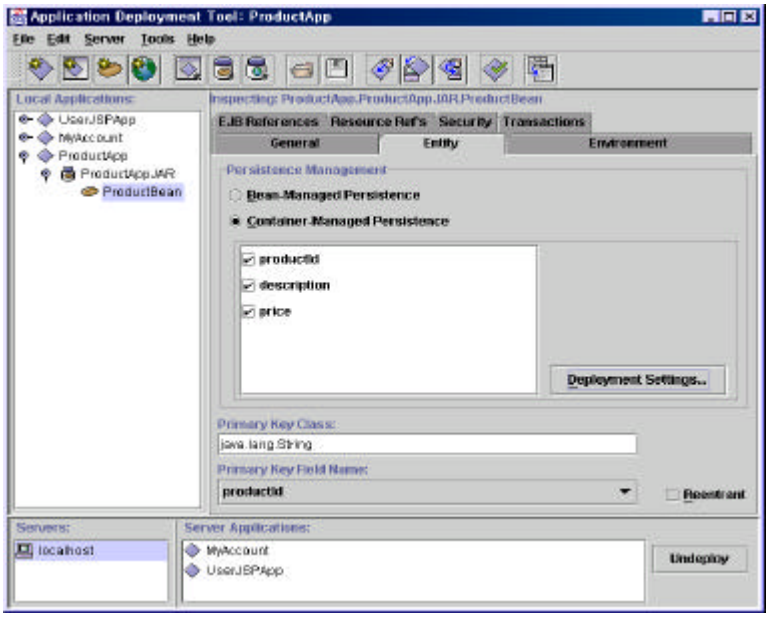


<그림. Transaction Management 대화상자>

✍✍ Deployment Setting 설정하기

Application Deployment Tool에서 ProductBean에 대한 Entity 탭 영역을 선택한다.

Entity 탭 영역에서 [Deployment Settings]를 클릭한다.

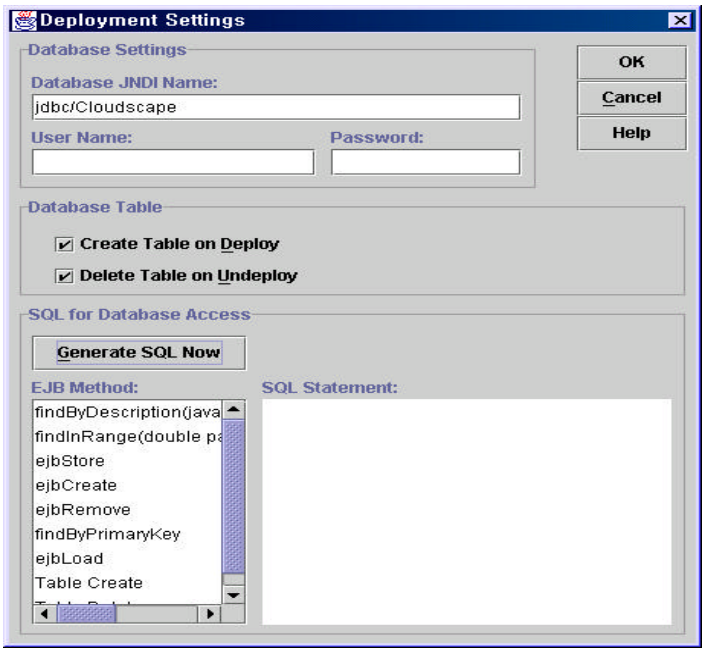


<그림. Application Deployment Tool – Entity Tab 화면>

[Database JNDI Name] 필드에 jdbc/Cloudscape를 입력한다.

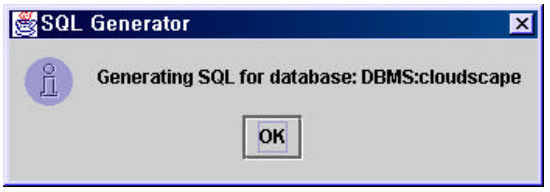
“Create Table on Deploy ” 체크박스가 선택되어 있는지 확인한다.

[Generate SQL Now]를 클릭한다.



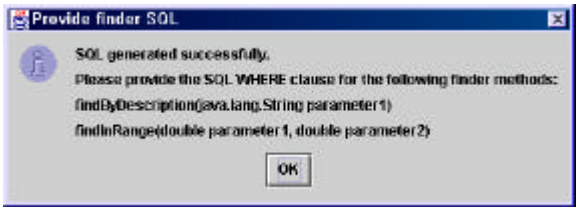
<그림. Deployment Settings 대화상자>

Cloudscape 데이터베이스에 대한 SQL 구문이 생성되었음을 알리는 팝업 윈도우가 뜰 것이다. [OK]를 클릭한다.



<그림. SQL Generator 대화상자>

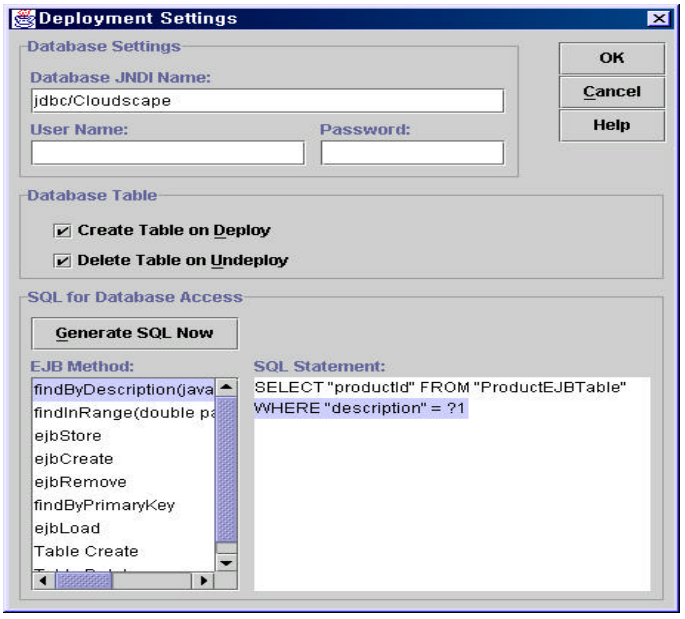
findByDescription과 findInRange method들에 대한 SQL where 절을 입력해야 한다고 알리는 팝업 윈도우가 뜰것이다. [OK]를 클릭한다.



<그림. Provide finder SQL 대화상자>

EJB method 리스트에서 findByDescription method를 선택하면, SQL select 구문이 일부분만 나타난다.

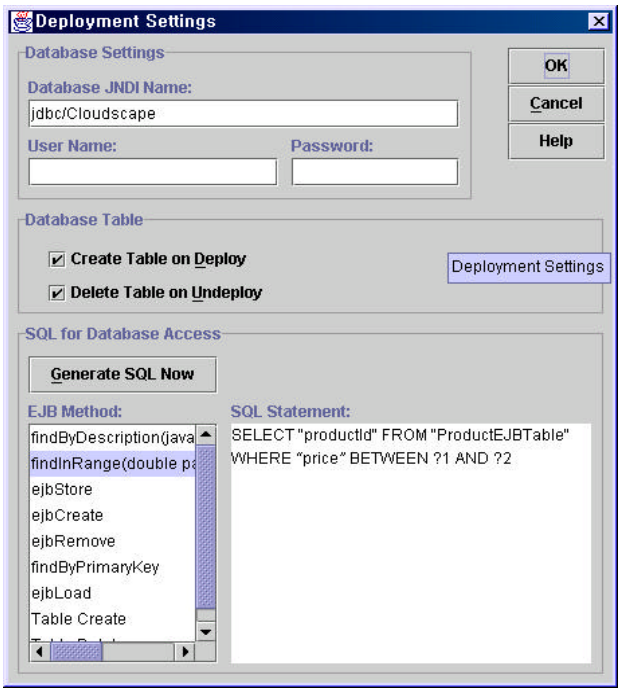
이 select 구문에 "WHERE "description" = ?1"의 where 절을 추가한다.(반드시 따옴표를 포함시켜야 한다. ?1은 finder method의 첫번째 전달인자 임을 나타낸다.)



<그림. Deployment Settings 대화상자>

EJB method 리스트에서 findInRange method를 선택하면, 또 다시 SQL select 구문의 일부분이 나타난다.

이 select 구문에 "WHERE "price" BETWEEN ?1 AND ?2 "의 WHERE 절을 추가한다.(여기서도 반드시 따옴표를 포함해야 하고 ?1과 ?2는 각각 첫번째와 두 번째 전달인자를 나타낸다.)

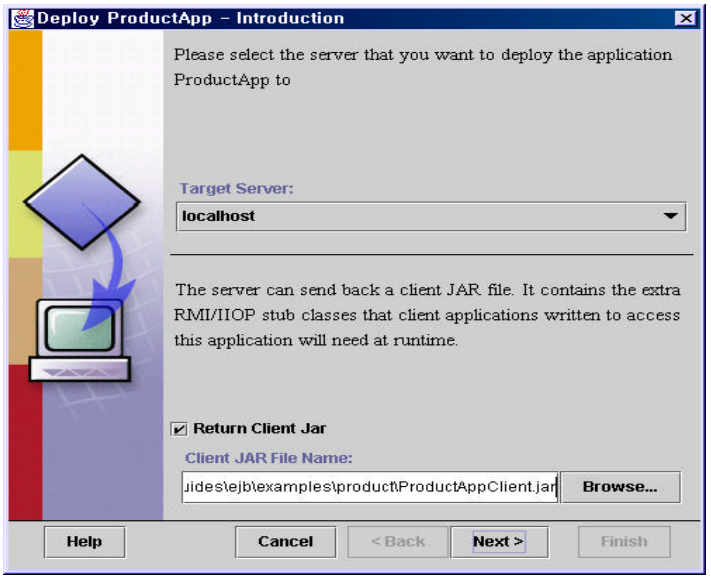


<그림. Deployment Settings 대화상자>

[OK]를 클릭한다.

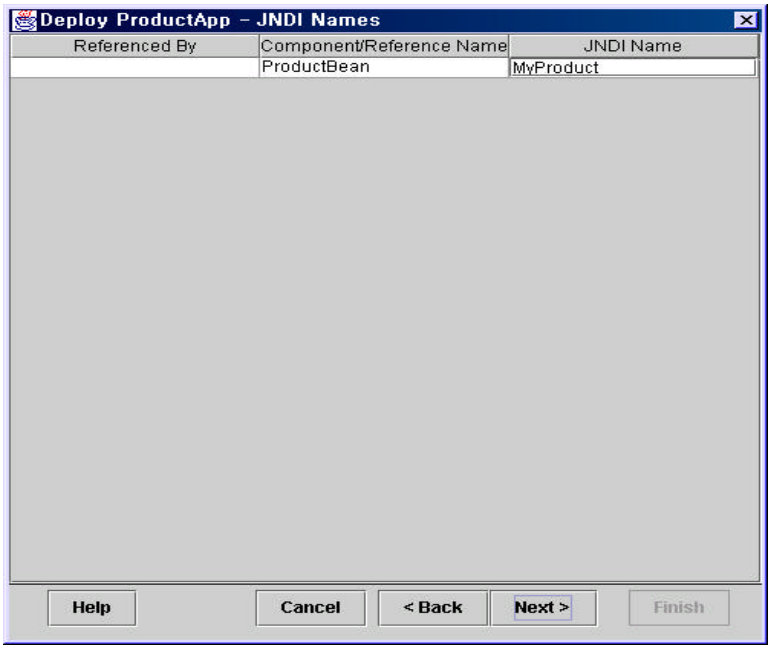
✍✍ J2EE 어플리케이션 deployment

[Return Client Jar]을 선택한다.

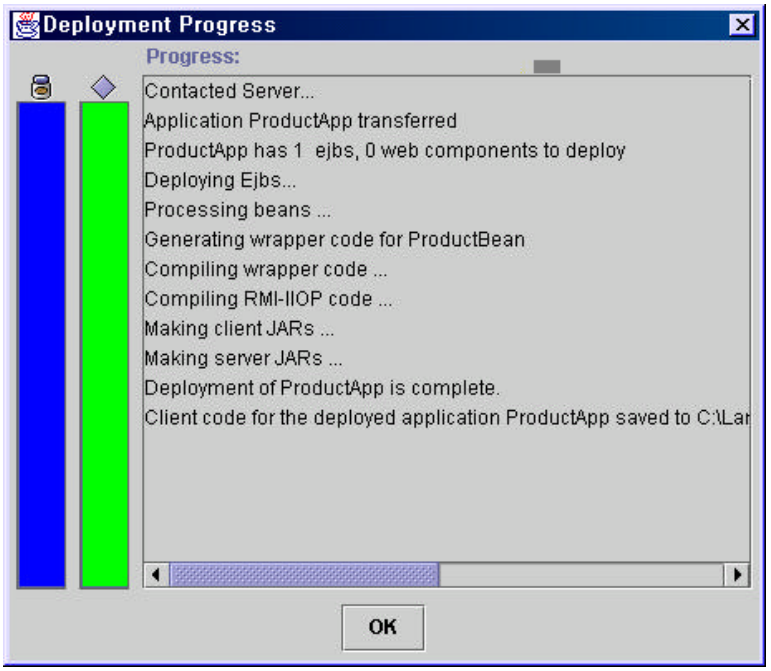


<그림. Introduction 대화상자

두번째 대화 상자에서 ProductBean의 [JNDI Name] 필드에 MyProduct라고 입력한다.



<그림. JNDI Names 대화상자>



<그림. Deployment Progress 대화상자>

Client Program 구현

클라이언트 프로그램은 다음과 같은 사항을 구현하고 있다.

✍ JNDI Context 생성

```
Context initial = new InitialContext();
```

✍ MyProduct의 Enterprise bean 레퍼런스 획득

```
Object objref = initial.lookup("MyProduct");
```

✍ 홈 인터페이스 가져오기

```
ProductHome home = (ProductHome)PortableRemoteObject.narrow(objref,
                                                    ProductHome.class);
```

✍ 엔티티 빈 인스턴스 생성

```
Product duke = home.create("123", "Ceramic Dog", 10.00);
…
duke = home.create("456", "Wooden Duck", 13.00);
duke = home.create("999", "Ivory Cat", 19.00);
duke = home.create("789", "Ivory Cat", 33.00);
duke = home.create("876", "Chrome Fish", 22.00);
```

✍ 비즈니스 method 실행

```
System.out.println(duke.getDescription() + ": " + duke.getPrice());
duke.setPrice(14.00);
System.out.println(duke.getDescription() + ": " + duke.getPrice());
```

✍ finder method 실행

```
Product earl = home.findByPrimaryKey("876");
System.out.println(earl.getDescription() + ": " + earl.getPrice());

Collection c = home.findByDescription("Ivory Cat");
…
c = home.findInRange(10.00, 20.00);
```

다음은 클라이언트 프로그램의 소스 코드이다.

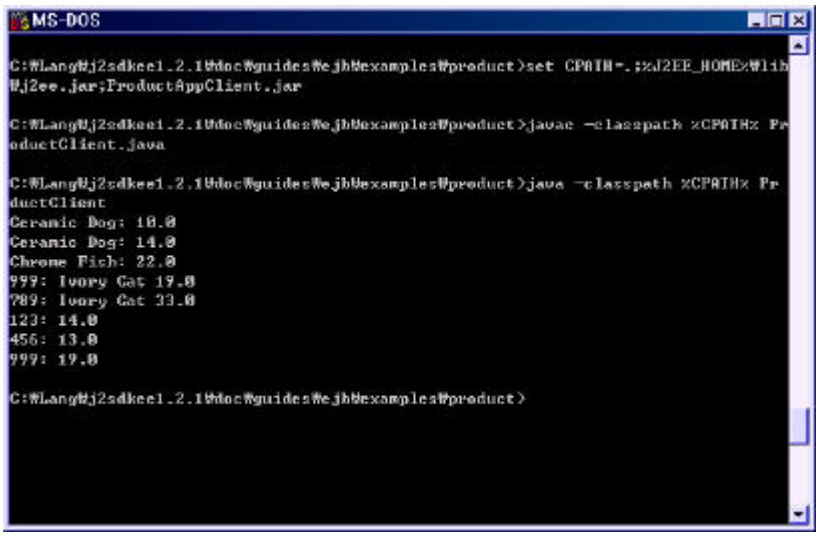
```
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import javax.rmi.PortableRemoteObject;

public class ProductClient
{
    public static void main(String[] args)
    {
       try {
                Context initial = new InitialContext();
                Object objref = initial.lookup("MyProduct");
                ProductHome home
=(ProductHome)PortableRemoteObject.narrow(
                                        objref, ProductHome.class);
                Product duke = home.create("123", "Ceramic Dog", 10.00);
                System.out.println(duke.getDescription() + ": "
+ duke.getPrice());
                duke.setPrice(14.00);
                System.out.println(duke.getDescription() + ": "
+ duke.getPrice());
                duke = home.create("456", "Wooden Duck", 13.00);
                duke = home.create("999", "Ivory Cat", 19.00);
                duke = home.create("789", "Ivory Cat", 33.00);
                duke = home.create("876", "Chrome Fish", 22.00);
                Product earl = home.findByPrimaryKey("876");
                System.out.println(earl.getDescription() + ": "
+ earl.getPrice());

                Collection c = home.findByDescription("Ivory Cat");
                Iterator i = c.iterator();

                while (i.hasNext()) {
                        Product product = (Product)i.next();
                        String productId = (String)product.getPrimaryKey();
                        String description = product.getDescription();
                        double price = product.getPrice();
                        System.out.println(productId + ": " + description
                                        + " " + price);
                }
                c = home.findInRange(10.00, 20.00);
                i = c.iterator();
```

```
            while (i.hasNext()) {
                Product product = (Product)i.next();
                String productId = (String)product.getPrimaryKey();
                double price = product.getPrice();
                System.out.println(productId + ": " + price);
            }
        } catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
    }
}
```

<소스> ProductClient.java

다음은 클라이언트 프로그램인 ProductClient의 실행 결과이다.



<그림. 클라이언트 프로그램 실행 콘솔 화면>

BMP 기반 엔티티 빈 구현

BMP 기반 엔티티 빈 예제 개요

✍✍ 9.2절에서 우리는 슈퍼마켓을 경영하고 있는 주인의 예를 통해 CMP 기반의 엔티티 빈을 통한 상품을 처리하는 비즈니스 로직에 대해 알아보았다. 그런데 슈퍼마켓에 다음과 같은 일이 발생하였다.

슈퍼마켓에서 일하던 아르바이트생이 일을 그만 두었다.

슈퍼마켓 주인은 새로운 아르바이트생을 고용하려고 하였지만, 상품의 종류가 많고 복잡하여 이전과 같이 아르바이트생이 상품 정보 및 관리의 모든일을 모두 해줄 수 없게되었다.

이제 주인은 상품의 정보와 위치 및 가격을 관리하는 일을 직접 수행하려고 한다.

기존에 슈퍼마켓의 비즈니스 로직을 담담했떤 CMP 기반의 엔티티 빈으로는 이러한 일을 해결할 수 없다.

✍✍ 슈퍼마켓 주인은 이제 새로운 개념의 엔티티 빈을 사용하려고 한다. 주인이 사용하려는 것은 BMP 기반의 엔티티 빈이다. 이것은 상품의 정보가 저장되어 있는 데이터베이스와 상품 사이의 모든 작업들을 주인이 직접 작성한 방법에 의해 수행하게 된다. CMP 기반에서는 주인이 지시한 일을 아르바이트생이 모두 알아서 처리하였다. 이는 매우 편리한 일이었지만, 주인이 원하는 아주 특별하고 복잡한 업무를 아르바이트생에게 시키기가 매우 어려웠다. BMP의 경우 이제 주인은 그와 관련된 모든 일들을 직접 알아서 하기로 했기 때문에 복잡하고 어려운 일들도 적절하게 수행할 수 있게된다.

✍✍ 그러면 슈퍼마켓의 비즈니스는 어떻게 변화하게 될까? CMP기반의 엔티티 빈에서 BMP기반으로 변화했다고 해서 전부 변하지는 않는다. 기본적인 수퍼마켓의 업무에서 변화하는 것은 이제는 상품의 배치라든지 가격 결정 사항에 대한 수정 및 상품 진열과 정리등을 모두 직접 주인이 알아서 해야한다는 것이다. 즉, 이전에는 주인이 상품의 가격을 변화시키면 아르바이트생이 가서 상품에 적힌 가격표를 바꿨지만, 이제는 주인이 직접가서 바꿔주어야 한다. 또한, 상품의 배치와 이동등도 모두 주인이 직접해 주어야 하는 것이다.

✍✍ CMP기반 엔티티 빈을 BMP기반으로 바꾸기

리모트 인터페이스와 홈 인터페이스

리모트 인터페이스와 홈 인터페이스는 CMP와 BMP에 관계없이 동일하다. 즉, 엔티티 빈의 persistence 관리기법에 영향을 받지 않는다.

이와 같이 독립적으로 구현함으로써 클라이언트가 바뀌지 않고 변화된 서버 어플리케이션을 사용할 수 있다는 이점이 있다. 즉, 클라이언트 API와 클라이언트 프로그램의 변경없이 서버의 어플리케이션을 CMP에서 BMP로 바꿀 수 있는 것이다.

엔티티 빈 클래스

CMP에서 BMP로 바뀌면서 실제적으로 변화하는 부분이 바로 엔티티 빈 클래스이다. 다음은 새로 작성될 엔티티 빈 클래스의 특징이다.

CMP기반의 엔티티 빈 클래스에서는 리모트 인터페이스에서 선언된 모든 method들과 홈 인터페이스에서 선언된 일부 method들(life cycle에 관련된 method들)만을 구현했지만, BMP기반에서는 리모트 인터페이스와 홈 인터페이스에 선언된 finder method들을 포함한 모든 method들을 구현해야 한다.

Home inteface에 정의된 모든 method들을 구현하기 위해서는 개발자가 직접 엔티티 빈 클래스내에 데이터베이스를 제어하기 위한 각종 method들과 connection 및 SQL 구문을 실행하기 위한 작업을 해 주어야 한다.

클라이언트 프로그램

클라이언트 프로그램은 어플리케이션 서버에 배치된 Enterprise bean을 이용하기 위해 bean의 리모트 인터페이스와 홈 인터페이스에 대한 레퍼런스를 이용한다. 즉, 직접적으로 엔티티 빈 클래스에 접근하지는 않는다.

따라서, 이 예제와 같이 엔티티 빈을 CMP기반에서 BMP기반으로 바꾸더라도 실제 리모트 인터페이스와 홈 인터페이스가 변화하지 않았으므로 클라이언트 프로그램은 새롭게 작성되거나 수정되지 않고 계속 사용될 수 있다.

클라이언트 프로그램은 ProductClient을 그대로 사용한다.

새로운 BMP기반의 엔티티 빈은 다음 3개의 파일로 구성된다.

Product.java ? 리모트 인터페이스(변화 없음)

ProductHome.java ? 홈 인터페이스(변화 없음)

ProductBmpEJB.java ? 엔티티 빈 클래스(수정 필요)

엔티티 빈 클래스 구현

✍ import 문

BMP기반 엔티티 빈 클래스는 SQL명령을 위해 몇가지 import문장이 새롭게 추가된다.

```
/*
 * 기존 productEJB.java에서 import된 API
 */
import java.util.*;
import javax.ejb.*;

/*
 * productBmpEJB.java에 새롭게 추가된 API
 */
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
```

✍✍ **데이터베이스 연결**

CMP에서는 DB 연결을 컨테이너가 자동으로 해주었지만, BMP에서는 개발자가 직접 연결 코드를 작성해 주어야 한다.

```
/*
 * 데이터베이스에 접근하기 위해 새롭게 추가된 코드
 */
private Connection con;
private String dbName = "java:comp/env/jdbc/AccountDB";
```

✍✍ **리모트 인터페이스** method **구현**

리모트 인터페이스에 선언된 method들은 CMP에서 구현된 method들과 동일하다.

✍✍ **홈 인터페이스** method **들의 구현**

다음은 홈 인터페이스에서 선언된 method들이다.

```
public Product create(String productId, String description,
    double balance) throws RemoteException, CreateException;

public Product findByPrimaryKey(String productId)
    throws FinderException, RemoteException;

public Collection findByDescription(String description)
    throws FinderException, RemoteException;

public Collection findInRange(double low, double high)
    throws FinderException, RemoteException;
```

ejbCreate() method의 경우 bean 인스턴스를 생성하고 DB 테이블에 새롭게 생성된 인스턴스에 대한 열(row)을 삽입하는 코드가 추가되었다. 또한, CMP기반 ejbCreate()경우 반환값이 null이었지만, BMP인 경우 반드시 primary key 값이어야 한다.

```
…
try {
    insertRow(productId, description, price);
```

```
} catch (Exception ex) {
    throw new EJBException("ejbCreate: " + ex.getMessage());
}
...
return productId;
```

finder method들은 모두 개발자에 의해서 구현이 제공되어야 한다. SQL 구문에 대한 실행문을 함수안에 직접 구현해 넣지 않고 다른 method로 구현하여 적용하였다.

```
public String ejbFindByPrimaryKey(String primaryKey)
throws FinderException
{
boolean result;

    try {
            result = selectByPrimaryKey(primaryKey);
    } catch (Exception ex) {
            throw new EJBException("ejbFindByPrimaryKey: "
+ ex.getMessage());
    }

    if (result) {
            return primaryKey;
    }
    else {
            throw new ObjectNotFoundException
("Row for productId " + primaryKey
+ " not found.");
    }
}

public Collection ejbFindByDescription(String description)
throws FinderException
{
    ...
    ...
}

public Collection ejbFindInRange(double low, double high)
```

```
throws FinderException
{
    …
…
}
```

CMP에서는 bean 인스턴스가 삭제되면 컨테이너가 자동으로 데이터베이스에서 해당 인스턴스에 대한 상태(row)를 삭제해 주었으나, BMP에서는 ejbRemove() method 를 구현하여 이를 직접 구현해 주어야 한다.

```
public void ejbRemove()
{
    try {
            deleteRow(productId);
    } catch (Exception ex) {
            throw new EJBException("ejbRemove: " + ex.getMessage());
    }
}
```

✍ JNDI 설정 및 데이터베이스 연결

setEntityContext() method : bean의 JNDI Context와 데이터베이스에 대한 연결을 설정한다.

```
public void setEntityContext(EntityContext context)
{
    this.context = context;

    /*
     * DB Connection을 위해 새롭게 추가된 코드
     */
    try {
            makeConnection();
    } catch (Exception ex) {
            throw new EJBException
("Unable to connect to database. "
+ ex.getMessage());
    }
}
```

unsetEntityContext() : 데이터베이스에 대한 연결 설정을 해제한다.

```
public void unsetEntityContext()
{
try {
                        con.close();
    } catch (SQLException ex) {
            throw new EJBException("unsetEntityContext: "
+ ex.getMessage());
    }
}
```

✍✍ ejbLoad() **와** ejbStore() method

이 method들은 bean 인스턴스의 상태를 데이터베이스와 동기화 시켜주는 것들로 인스턴스의 변수들(description, price)등을 데이터베이스에서 읽어오는 작업과 데이터베이스에 저장하는 작업을 수행한다.

- ejbLoad() method

```
public void ejbLoad()
{
    try {
            loadRow();
    } catch (Exception ex) {
            throw new EJBException("ejbLoad : "
+ ex.getMessage());
    }
}
```

- ejbStore() method

```
public void ejbStore()
{
    try {
            storeRow();
    } catch (Exception ex) {
            throw new EJBException("ejbStore: "
+ ex.getMessage());
    }
}
```

✍✍ **데이터베이스 및 SQL 구문 관련 개발자 정의 methods**

- **데이터베이스 연결 method**

```
private void makeConnection()
throws NamingException, SQLException
{
    InitialContext ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup(dbName);
    con =  ds.getConnection();
}
```

- **bean 인스턴스 정보 삽입 및 삭제 method**

  insertRow() method : **새로운 인스턴스 상태 정보 삽입**

```
private void insertRow (String productId, String description,
double price) throws SQLException
{
    String insertStatement = "insert into productTbl values ( ? , ? , ?)";
    PreparedStatement prepStmt = con.prepareStatement(insertStatement);

    prepStmt.setString(1, productId);
    prepStmt.setString(2, description);
    prepStmt.setDouble(3, price);

    prepStmt.executeUpdate();
    prepStmt.close();
}
```

deleteRow() method : 인스턴스 상태정보 삭제

```
private void deleteRow(String productId) throws SQLException
{
    String deleteStatement =
            "delete from productTbl where productId = ? ";
    PreparedStatement prepStmt =
            con.prepareStatement(deleteStatement);

    prepStmt.setString(1, productId);
    prepStmt.executeUpdate();
    prepStmt.close();
}
```

- finder method에 관련 일을 하는 method들

selectByPrimaryKey() method : findByPrimaryKey()

```
private boolean selectByPrimaryKey(String primaryKey)
throws SQLException
{
    String selectStatement =
            "select productId " +
```

```
                "from productTbl where productId = ? ";
    PreparedStatement prepStmt =
                con.prepareStatement(selectStatement);

    prepStmt.setString(1, primaryKey);

    ResultSet rs = prepStmt.executeQuery();
    boolean result = rs.next();
    prepStmt.close();
    return result;
}
```

selectByDescription() method : findByDescription()

```
private Collection selectByDescription(String description)
throws SQLException
{
    String selectStatement = "select productId " +
                                "from productTbl where description = ? ";
    PreparedStatement prepStmt = con.prepareStatement(selectStatement);

    prepStmt.setString(1, description);
    ResultSet rs = prepStmt.executeQuery();
    ArrayList a = new ArrayList();

    while (rs.next()) {
                String productId = rs.getString(1);
                a.add(productId);
    }

    prepStmt.close();
    return a;
}
```

selectInRange() method : findInRange()

```
private Collection selectInRange(double low, double high)
throws SQLException
```

```
{
    String selectStatement =
            "select productId from productTbl " +          "where price
between  ? and ?";
    PreparedStatement prepStmt = con.prepareStatement(selectStatement);

    prepStmt.setDouble(1, low);
    prepStmt.setDouble(2, high);
    ResultSet rs = prepStmt.executeQuery();
    ArrayList a = new ArrayList();

    while (rs.next()) {
            String productId = rs.getString(1);
            a.add(productId);
    }

    prepStmt.close();
    return a;
}
```

bean 인스턴스의 상태와 데이터베이스에 저장된 정보와의 동기화에 관련된 methods

loadRow() method : ejbLoad()

```
private void loadRow() throws SQLException
{
    String selectStatement =
            "select description, price " +          "from productTbl where
productId = ? ";
    PreparedStatement prepStmt = con.prepareStatement(selectStatement);
    prepStmt.setString(1, this.productId);
    ResultSet rs = prepStmt.executeQuery();

    if (rs.next()) {
            this.description = rs.getString(1);
            this.price = rs.getDouble(2);
            prepStmt.close();
    }
    else {
```

```
            prepStmt.close();
            throw new NoSuchEntityException(
            "Row for productId " + productId + " not found in database.");
        }
    }
```

storeRow() method : ejbStore()

```
private void storeRow() throws SQLException
{
    String updateStatement = "update productTbl set description =  ? , price
= ? "
                                + "where productId = ?";
    PreparedStatement prepStmt = con.prepareStatement(updateStatement);
    prepStmt.setString(1, description);
    prepStmt.setDouble(2, price);
    prepStmt.setString(3, productId);
    int rowCount = prepStmt.executeUpdate();
    prepStmt.close();

    if (rowCount == 0) {
            throw new EJBException(
                    "Storing row for productId " + productId + " failed.");
    }
}
```

데이터베이스 테이블 만들기

✍✍ BMP기반 엔티티 빈에서는 CMP기반과는 달리 필요한 데이터베이스 테이블을 컨테이너가 자동으로 만들어 주지 않는다. 따라서 개발자가 직접 데이터베이스에 접근하여 필요한 데이터베이스 테이블을 생성해 주어야 한다.

✍✍ 이 예제에서 사용되는 데이터베이스 테이블은 CMP기반의 엔티티 빈예제에서 사용되었던 것과 완벽하게 동일하다.

✍✍ 다음은 데이터베이스 테이블에 대한 create SQL 구문이다.

```
CREATE TABLE productTbl  (
    productId VARCHAR(255)
CONSTRAINT pk_ProductEJBTable  PRIMARY KEY
```

```
    description VARCHAR(255) , price DECIMAL(10,2) NOT NULL ,
);
```

✍✍ 본 예제에서는 SUN사의 J2EE Application Server를 사용하고 데이터베이스로는 J2EE SDK에 번들로 포함되어 있는 Cloudscape를 사용하고 있다. 이 데이터베이스에서 테이블을 생성하는 과정은 다음과 같다.

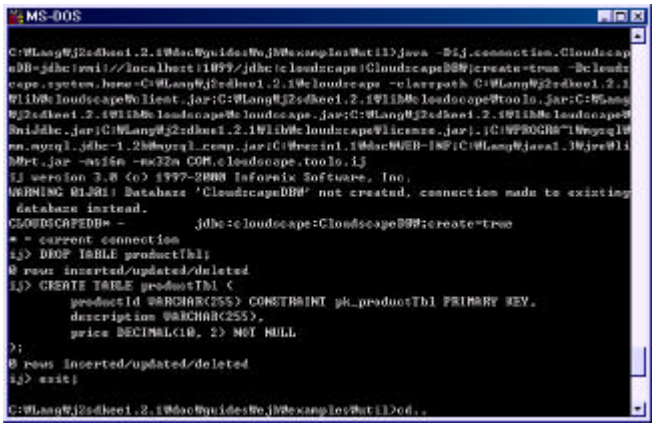
✍✍ 데이터베이스 접속하여 직접 입력하기 : 다음 문장을 실행하여 데이터베이스에 접속할 수 있다.

데이터베이스를 실행한다. 실행하는 명령어는 cloudscape ? start로 과정은 CMP기반의 엔티티 빈 예제를 참고한다.

J2EE SDK를 설치한 디렉토리의 부디렉토리를 살펴보면 %J2EE_HOME% \ doc \ guides \ ejb \ examples \ util에 cloudlj.bat 파일이 있음을 알 수 있다. 이 파일은 Oracle의 SQL Plus와 동일한 환경을 제공해 주는 배치파일이다.

참고로 %J2EE_HOME%은 J2EE SDK가 설치된 HOME 디렉토리이다. 즉, J2EE SDK가 설치된 디렉토리가 C: \ j2sdkee1.2.1이라면 cloudlj.bat가 위치한 디렉토리는 C: \ j2sdkee1.2.1 \ \ doc \ guides \ ejb \ examples \ util가 된다.

다음은 위 파일을 실행한 화면이다. 화면과 같이 직접 SQL 명령어를 입력하면 테이블이 생성된다.



<그림. 테이블 생성 콘솔 화면>

다음은 BMP기반 엔티티 빈 클래스의 전체 소스 코드이다.

```
/*
 * 기존 productEJB.java에서 import된 API
 */
import java.util.*;
import javax.ejb.*;

/*
 * productBmpEJB.java에 새롭게 추가된 API
```

```
 */
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class ProductBmpEJB implements EntityBean
{
    public String productId;
    public String description;
    public double price;

    private EntityContext context;

    /*
     * 데이터베이스에 접근하기 위해 새롭게 추가된 코드
     */
private Connection con;
private String dbName = "java:comp/env/jdbc/AccountDB";

    /*
     * 리모트 인터페이스에 선언된 method들의 implementation
     */
    public void setPrice(double price)
    {
            this.price = price;
    }

    public double getPrice()
    {
            return price;
    }

    public String getDescription()
    {
            return description;
    }

    /*
     * 홈 인터페이스에 선언된 엔티티 빈의
```

```
    *life cycle과 관련된 method들이다.
    *
    * 1. ejbCreate() method            : 기존의 method 수정
    * 2. ejbFindByPrimaryKey()         : 새로 구현
    * 3. ejbFindByDescription()        : 새로 구현
    * 4. ejbFindInRange() method       : 새로 구현
    */

   /* 1. 홈 인터페이스의 create() method */
   public String ejbCreate(String productId, String description,
double price) throws CreateException
   {
            if (productId == null) {
                     throw new CreateException("The productId is required.");
            }

            /*
             * DB에 열을 삽입하는 SQL method인 insertRow()를 이용하여
             * 새로운 엔티티 빈 인스턴스를 DB에 저장한다.
             */
            try      {
                     insertRow(productId, description, price);
            } catch (Exception ex) {
                     throw new EJBException("ejbCreate: " + ex.getMessage());
            }

            this.productId = productId;
            this.description = description;
            this.price = price;

            return productId;
   }

   /* 2. 홈 인터페이스의 findByPrimaryKey() method */
   public String ejbFindByPrimaryKey(String primaryKey) throws
FinderException
   {
            boolean result;
```

```
                try {
                        result = selectByPrimaryKey(primaryKey);
                } catch (Exception ex) {
                        throw new EJBException("ejbFindByPrimaryKey: "
+ ex.getMessage());
                }

                if (result) {
                        return primaryKey;
                }
                else {
                        throw new ObjectNotFoundException
("Row for productId " + primaryKey + " not found.");
                }
        }

        /* 3. 홈 인터페이스의 findByDescription() method */
        public Collection ejbFindByDescription(String description)
throws FinderException
        {
                Collection result;

                try {
                        result = selectByDescription(description);
                } catch (Exception ex) {
                        throw new EJBException("ejbFindByDescription: "
+ ex.getMessage());
                }

                if (result.isEmpty()) {
                        throw new ObjectNotFoundException("No rows found.");
                }
                else {
                        return result;
                }
        }

        /* 4. 홈 인터페이스의 findInRange() method */
        public Collection ejbFindInRange(double low, double high)
```

```
throws FinderException
    {
            Collection result;

            try {
                    result = selectInRange(low, high);
            } catch (Exception ex) {
                    throw new EJBException("ejbFindInRange: "
+ ex.getMessage());
            }

            if (result.isEmpty()) {
                    throw new ObjectNotFoundException("No rows found.");
            }
            else {
                    return result;
            }
    }

    /*
     * CMP 기반 엔티티 빈 클래스에서는
     * 아무일도 하지 않은 빈 method들이었지만,
     * BMP 기반 엔티티 빈 클래스에서는
     * bean instance의 life cycle과 이에 따른
     * 데이터베이스에서의 생성, 수정, 삭제들을
     * 모두 개발자가 처리해야 한다.
     */

    /*
     * ejbRemove() method
     * -------------------------------------------------------
     * 데이터베이스에서 bean 인스턴스에 대한 정보를 삭제한다.
     */
    public void ejbRemove()
    {
            try {
                    deleteRow(productId);
            } catch (Exception ex) {
                    throw new EJBException("ejbRemove: "
```

```
+ ex.getMessage());
            }
    }

    /* setEntityContext() method
     * -------------------------------------------------------
     * 엔티티 빈의 JNDI Context를 설정하고
     * 데이터베이스 연결을 설정한다.
     */
    public void setEntityContext(EntityContext context)
    {
            this.context = context;

            /*
             * DB Connection을 위해 새롭게 추가된 코드
             */
            try {
                    makeConnection();
            } catch (Exception ex) {
                    throw new EJBException("Unable to connect
to database. " + ex.getMessage());
            }
    }

    /* unsetEntityContext() method
     * -----------------------------------
     * 데이터베이스 연결을 해제한다.
     */
    public void unsetEntityContext()
    {
            try {
                    con.close();
            } catch (SQLException ex) {
                    throw new EJBException("unsetEntityContext: "
+ ex.getMessage());
            }
    }
```

```
/* ejbActivate()
 * -------------------------------------
 * bean 인스턴스를 활성화시키는 method로써
 * CMP와 BMP 모두 동일하다.
 */
public void ejbActivate()
{
        productId = (String)context.getPrimaryKey();
}

/* ejbPassivate() method
 * -------------------------------------
 * bean 인스턴스를 비활성화 시키는 method로써
 * CMP와 BMP 모두 동일하다.
 */
public void ejbPassivate()
{
        productId = null;
        description = null;
}

/* ejbLoad() method
 * -------------------------------------
 * 데이터베이스에 저장되어 있는
 * bean 인스턴스의 정보를 읽어온다.
 */
public void ejbLoad()
{
        try {
                loadRow();
        } catch (Exception ex) {
                throw new EJBException("ejbLoad : " + ex.getMessage());
        }
}

/* ejbStore() method
 * -------------------------------------
 * bean 인스턴스의 정보를
 * 데이터베이스에 저장한다.
```

```
     */
    public void ejbStore()
    {
            try {
                    storeRow();
            } catch (Exception ex) {
                    throw new EJBException("ejbStore: " + ex.getMessage());
            }
    }

    /* ejbPostCreate() method
     * ----------------------------------
     * ejbCreate() method 실행 후 컨테이너에 의해
     * 바로 그 다음에 실행되는 method로서 이 예제
     * 에서는 아무일도 하지 않는다.
     */
    public void ejbPostCreate(String productId, String description,
double price) { }


    /*********************************************************************/
    /********************** Database Routines ************************/
    /*********************************************************************/

    private void makeConnection()
throws NamingException, SQLException
    {
            InitialContext ic = new InitialContext();
            DataSource ds = (DataSource) ic.lookup(dbName);
            con =  ds.getConnection();
    }

    private void insertRow (String productId, String description,
double price) throws SQLException
    {
            String insertStatement =
                    "insert into productTbl values ( ? , ? , ?)";
            PreparedStatement prepStmt =
                    con.prepareStatement(insertStatement);
```

```
                prepStmt.setString(1, productId);
                prepStmt.setString(2, description);
                prepStmt.setDouble(3, price);

                prepStmt.executeUpdate();
                prepStmt.close();
        }

        private void deleteRow(String productId) throws SQLException
        {
                String deleteStatement =
                        "delete from productTbl where productId = ? ";
                PreparedStatement prepStmt =
                        con.prepareStatement(deleteStatement);

                prepStmt.setString(1, productId);
                prepStmt.executeUpdate();
                prepStmt.close();
        }

        private boolean selectByPrimaryKey(String primaryKey)
    throws SQLException
        {
                String selectStatement =
                        "select productId " +
                        "from productTbl where productId = ? ";
                PreparedStatement prepStmt =
                        con.prepareStatement(selectStatement);

                prepStmt.setString(1, primaryKey);

                ResultSet rs = prepStmt.executeQuery();
                boolean result = rs.next();
                prepStmt.close();
                return result;
        }

        private Collection selectByDescription(String description)
```

```
throws SQLException
    {
        String selectStatement =
            "select productId " +
            "from productTbl where description = ? ";
        PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);

        prepStmt.setString(1, description);
        ResultSet rs = prepStmt.executeQuery();
        ArrayList a = new ArrayList();

        while (rs.next()) {
            String productId = rs.getString(1);
            a.add(productId);
        }

        prepStmt.close();
        return a;
    }

    private Collection selectInRange(double low, double high)
throws SQLException
    {
        String selectStatement =
            "select productId from productTbl " +
            "where price between  ? and ?";
        PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);

        prepStmt.setDouble(1, low);
        prepStmt.setDouble(2, high);
        ResultSet rs = prepStmt.executeQuery();
        ArrayList a = new ArrayList();

        while (rs.next()) {
            String productId = rs.getString(1);
            a.add(productId);
        }
```

```
            prepStmt.close();
            return a;
    }

    private void loadRow() throws SQLException
    {
            String selectStatement =
                    "select description, price " +
                    "from productTbl where productId = ? ";
            PreparedStatement prepStmt =
                    con.prepareStatement(selectStatement);

            prepStmt.setString(1, this.productId);

            ResultSet rs = prepStmt.executeQuery();

            if (rs.next()) {
                    this.description = rs.getString(1);
                    this.price = rs.getDouble(2);
                    prepStmt.close();
            }
            else {
                    prepStmt.close();
                    throw new NoSuchEntityException(
                            "Row for productId " + productId + "
not found in database.");
            }
    }

    private void storeRow() throws SQLException
    {
            String updateStatement =
                    "update productTbl set description =  ? , price = ? "
+ "where productId = ?";
            PreparedStatement prepStmt =
                    con.prepareStatement(updateStatement);

            prepStmt.setString(1, description);
```

```
                prepStmt.setDouble(2, price);
                prepStmt.setString(3, productId);
                int rowCount = prepStmt.executeUpdate();
                prepStmt.close();

                if (rowCount == 0) {
                        throw new EJBException(
                                "Storing row for productId " + productId
+ " failed.");
                }
        }
}
```

<소스> ProductBmpEJB.java

Primary Key 클래스 구현

✍ 본 예제의 primary key인 productId는 java.lang.String 클래스 인스턴스로써 단순하기 때문에 따로 primary key 클래스를 구현하지 않고 String 클래스를 primary key 클래스로 사용한다.

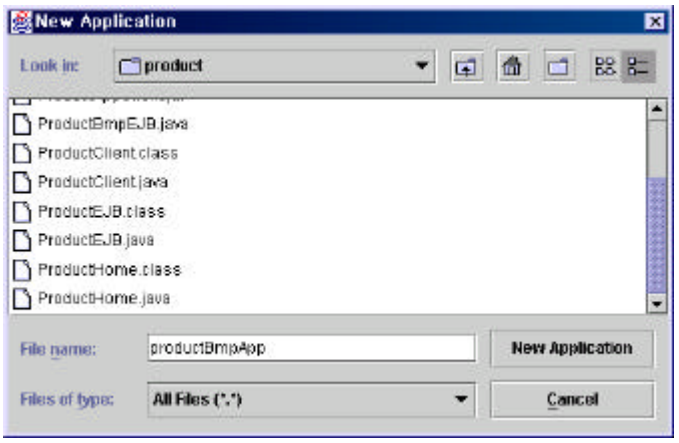
Deployment Descriptor 작성 및 어플리케이션 배치

EMP 기반의 엔티티 빈의 deployment descriptor 작성과 Enterprise bean packaging은 CMP 기반의 것과 몇몇 부분에서 다를 뿐 대부분의 과정을 동일하다. 또한 이미 6장에서 EMP 기반의 sample 엔티티 빈에 대한 작업을 했었기 때문에 중요한 부분만을 살펴 보도록 하자.

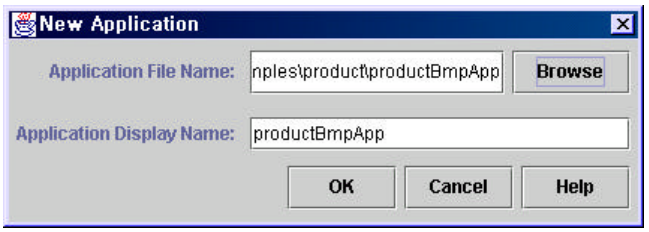
✍ Application Deployment Tool

[New Application]

배치 tool의 [File] 메뉴에서 [New Application]을 선택하고, Application File Name에 productBmpApp를 입력한다.



<그림. New Application 대화상자>



<그림. New Application 대화상자>

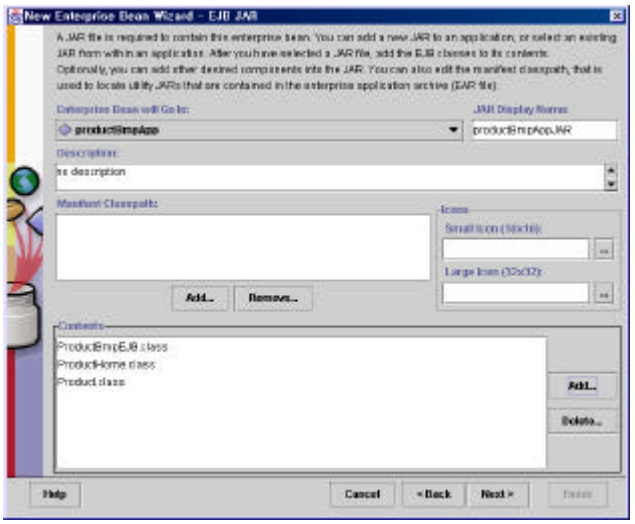
여기서는 새로운 어플리케이션 이름(productBmpApp.ear)을 지정하였다. 그러나, 꼭 이렇게 하지 않고 기존에 deploy된 어플리케이션을 undeploy하고 동일한 이름(ProductApp.ear)으로 다시 작성해도 관계 없다. 그러나 뒤에 나오는 JNDI 설정에서 어플리케이션에 대한 JNDI 이름은 클라이언트 프로그램을 수정하지 않고 싶다면 반드시 이전과 동일한 이름(MyProduct)를 사용해야 한다.

New Enterprise Bean Wizard

✍✍ [EJB JAR 대화상자]

[Enterprise Bean will Go In:] 콤보상자에서 productBmpApp를 선택하고 [JAR Display Name:] 칸에 productBmpAppJAR을 입력한다.

[Contexts] 텍스트 영역 오른쪽에 있는 [Add…] 버튼을 클릭하여 Enterprise bean을 구성할 리모트 인터페이스, 홈 인터페이스, 엔티티 빈 클래스를 추가한다.



<그림. EJB JAR 대화상자>

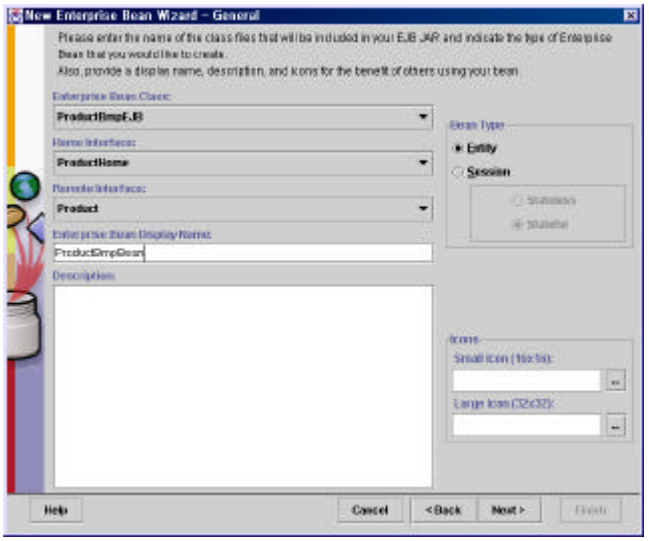
✍✍ [General 대화상자]

Enterprise Bean Class: ProductBmpEJB

Home Interface : ProductHome

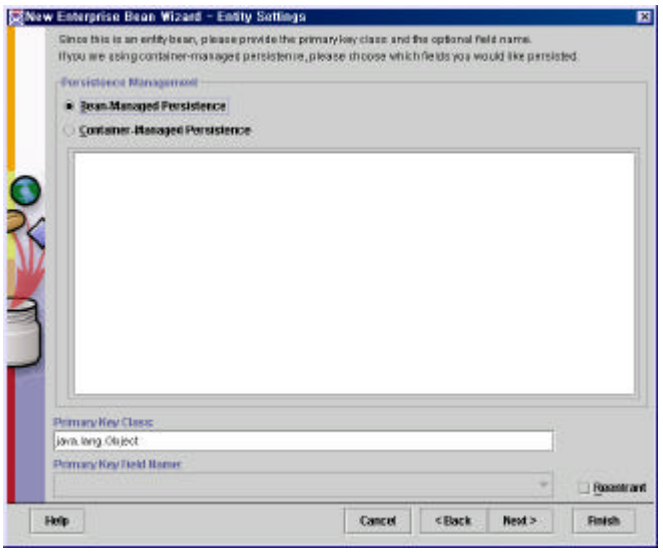
Remote Interface : Product

[Bean Type]은 Entity를 선택하고, [Enterprise Bean Display Name] 칸에는 ProductBmpBean을 입력한다.



<그림. General 대화상자>

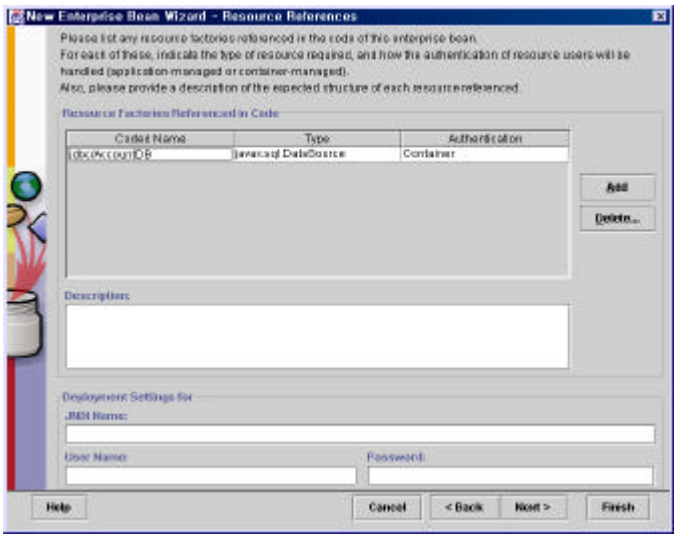
✍ [Entity Settings] **대화상자**



<그림. Entity Settings 대화상자>

✍ [Resource References] **대화상자**

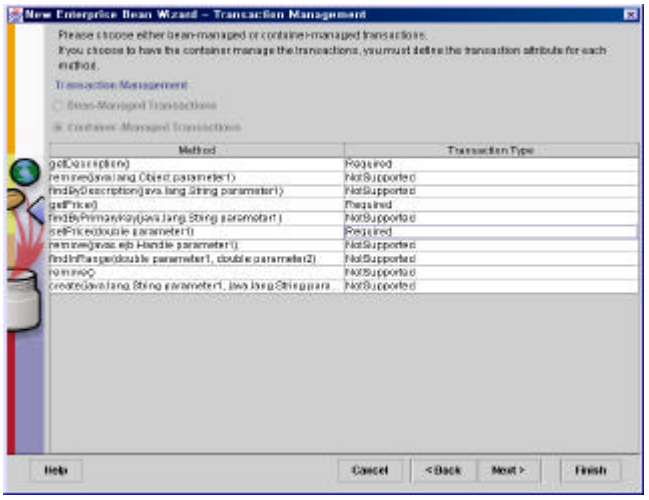
[Add] 버튼을 누르고 아래와 같이 입력한다.



<그림. Resource Reference 대화상자>

✍✍ [Transaction Management] 대화상자

비즈니스 method들을 [Required]로 설정한다.



<그림. Transaction Management 대화상자>

✍✍ 다음은 상기 과정을 통해 생성된 Deployment descriptor XML 문서이다.

```
<?xml version="1.0" encoding="MS949"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
 <description>no description</description>
 <display-name>productBmpAppJAR</display-name>
 <enterprise-beans>
  <entity>
   <description>no description</description>
   <display-name>ProductBmpBean</display-name>
   <ejb-name>ProductBmpBean</ejb-name>
   <home>ProductHome</home>
   <remote>Product</remote>
   <ejb-class>ProductBmpEJB</ejb-class>
   <persistence-type>Bean</persistence-type>
   <prim-key-class>java.lang.Object</prim-key-class>
   <reentrant>False</reentrant>
   <resource-ref>
  <res-ref-name>jdbc/AccountDB</res-ref-name>
```

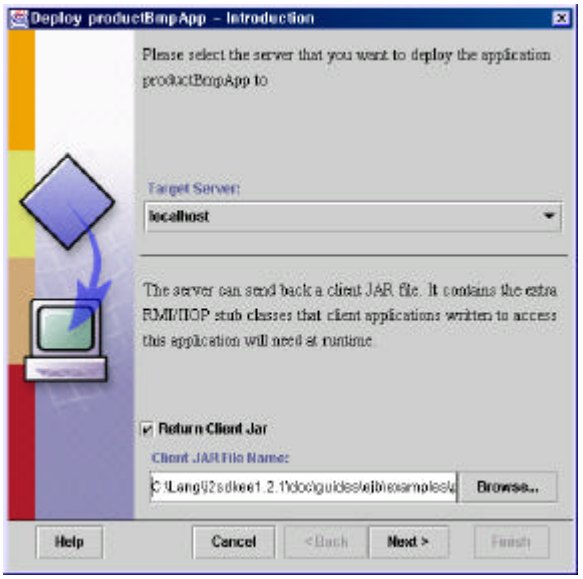
```
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
     </resource-ref>
   </entity>
  </enterprise-beans>
  <assembly-descriptor>
   <container-transaction>
    <method>
    <ejb-name>ProductBmpBean</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getDescription</method-name>
    <method-params />
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>
   <container-transaction>
     <method>
    <ejb-name>ProductBmpBean</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>setPrice</method-name>
    <method-params>
      <method-param>double</method-param>
    </method-params>
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>
   <container-transaction>
     <method>
    <ejb-name>ProductBmpBean</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getPrice</method-name>
    <method-params />
     </method>
     <trans-attribute>Required</trans-attribute>
   </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Application Deployment

✍ [Introduction] 대화상자

[Return Client Jar] 체크박스를 선택한다. 이 박스를 선택하면 클라이언트용 API인 .jar파일이 생성되게 된다. 클라이언트가 일반 java 어플리케이션일 경우 클라이언트는 서버에 접속하기 위한 rmi API 클래스가 필요하게 된다. 이 예제에서는 productBmpAppClient.jar 파일이 생성되게 되는데 이 파일안에는 rmi Stub 클래스와 서버 어플리케이션에 대한 .class 파일과 META 파일등이 들어있다.

웹 어플리케이션의 경우 [Return Client Jar] 체크박스를 선택할 필요가 없다. 즉, 웹 어플리케이션은 클라이언트측의 어플리케이션이 없고 사용자가 Html문서나 JSP(Java Server Page) 또는 Servlet등을 이용하여 서버 어플리케이션을 이용하기 때문에 클라이언트를 위한 API(.jar)이 필요하지 않게 되는 것이다.



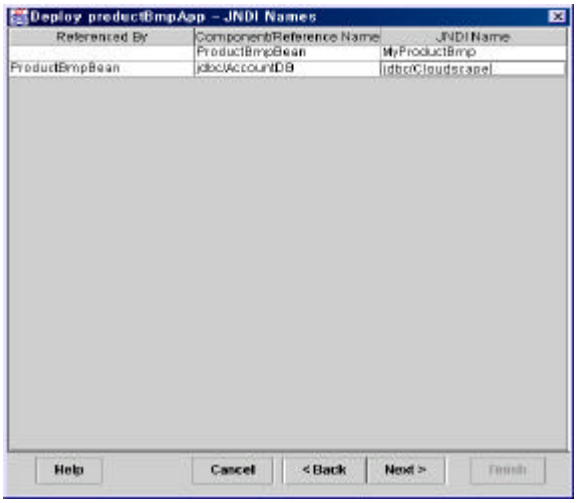
<그림. Introduction 대화상자>

✍ [JNDI Names] 대화상자

여기에서는 ProductBmpBean에 대한 JNDI Name을 MyProductBMP로 하고 있다. 이것은 CMP기반 product 예제에서의 MyProduct와 다르다. 이렇게 한 이유는 시스템에 CMP기반의 product 엔티티 빈 어플리케이션이 이미 설치되어 있는 상태이기 때문이다.

이와 같이 JNDI 이름이 다르게 설정된 경우 클라이언트 프로그램의 수정이 필요하게 된다. 즉, 클라이언트 프로그램이 JNDI 이름을 통해서 서버의 배치된 어플리케이션에 접근하므로 소스 코드에서 JNDI 이름에 관련된 부분을 MyProduct에서 MyProductBMP로 수정해 주어야 하는것이다.

일반적으로 서버측 어플리케이션이 수정되었을 경우 클라이언트가 따라서 수정되는 것은 바람직하지 않다. 따라서 이와 같이 수정이 가해졌을 경우에는 기존에 수정하기 전의 어플리케이션을 undeploy하고 수정된 새로운 어플리케이션을 기존과 동일한 JNDI 이름을 적용하여 deploy 하게 되면 클라이언트 프로그램은 전혀 수정될 필요성이 없게 된다.



<그림. JNDI Names 대화상자>

설정을 완료하고 [finish]를 누르면 deploy가 시작된다.

Client Program 구현

✍ 클라이언트 프로그램의 수정

원칙적으로 EJB에서는 서버측 어플리케이션의 비즈니스 로직에 대한 인터페이스가 바뀌지 않는 한 클라이언트 프로그램는 전혀 수정할 필요가 없다.

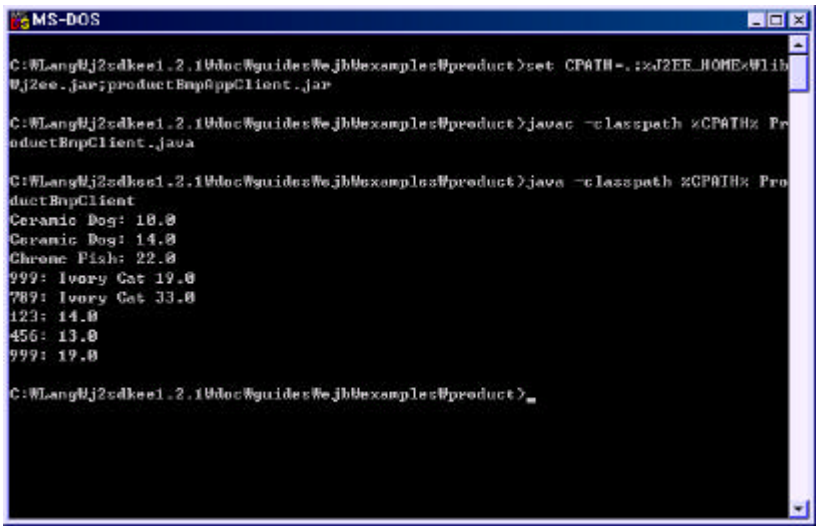
즉, 비즈니스 로직에 대해서 서버 어플리케이션이 어떻게 구현을 하던 그것은 클라이언트와 관계가 없는 것이다. 따라서 클라이언트는 서버 어플리케이션이 제공하는 인터페이스를 통해 서버에 배치된 Enterprise bean을 이용하기만 하면되는 것이다.

본 예제도 이와 동일하다 엔티티 빈을 구현하고 있는 클래스가 어떻게 변화해도 이에 접근하는 인터페이스만 전과 동일하면 클라이언트를 수정할 필요가 없다. 그러나, 위에서 살펴보았듯이 BMP기반의 product 예제는 서버에 deployment 할 때 JNDI 이름이 바뀌었다. 이것은 클라이언트가 서버에 배치된 Enterprise bean을 찾는 이름이 바뀐것이다. 따라서 클라이언트 프로그램의 JNDI look up 부분의 코드를 다음과 같이 수정해 주어야 한다.

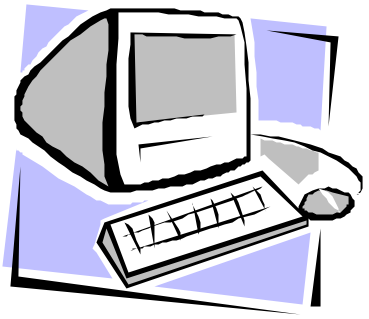
```
public class ProductBmpClient
{
    public static void main(String[] args)
    {
        try {
            Context initial = new InitialContext();
/* 이 부분이 수정된 부분이다. CMP기반 예제에서는
 JNDI 이름이 MyProduct였는데 현재는 MyProductBmp로 수정되었다. */
            Object objref = initial.lookup("MyProductBmp");
            ...
```

```
        }
    }
```

✍✍ 다음은 새로운 클라이언트 프로그램의 컴파일 및 실행 화면이다.



<그림. 클라이언트 프로그램 실행 화면>



Chapter 11

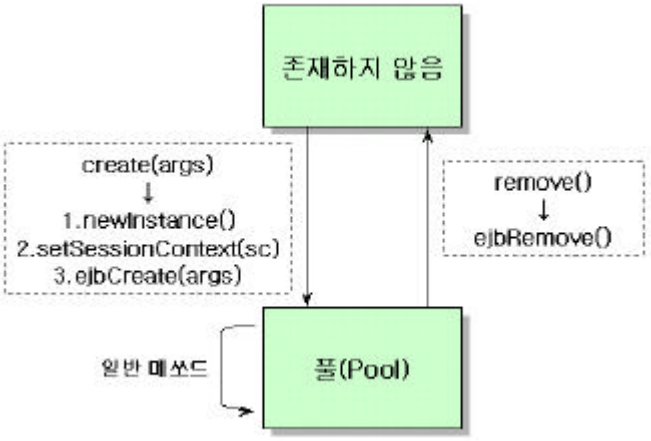
고급 세션빈 개발

세션 빈

개요

- 세션 빈에는 무상태 세션 빈과 상태정보유지 세션 빈이 있다. 세션 빈에서의 상태(state)의 의미는 빈 내부에서 transient라는 변경자가 붙어 있지 않는 필드 변수들이다. 세션 빈에서 사용하는 변수값으로 데이터베이스에는 저장되지 않고 2차저장소에 저장되어 지속적으로 참조하는 값을 의미한다. 상태유지 세션빈은 이런 필드 변수를 갖는 세션 빈을 의미하고, 무상태 세션 빈은 이런 필드 변수가 없는 경우가 된다.
- 컨테이너는 클라이언트가 요청하면 세션 빈을 메모리에 올리는 활성화(Activate) 작업을 하고, 세션 빈 인스턴스가 필요 없어지는 상황이 오면 인스턴스를 메모리에서 풀로 옮기는 비활성화(Passivation) 시킨다.
- 무상태 세션 빈은 상태가 유지될 필요가 없기 때문에 활성화와 비활성화가 메쏘드 호출 단위로 벌어질 수 있다. 즉, 메쏘드 호출이 벌어졌을 때는 세션 빈 풀에서 해당 세션 빈 인스턴스를 가져와서 사용하고 메소드 호출이 끝났을 때 상태 보존없이 다시 세션 빈 풀에 넣으면 된다.
- 상태유지 세션 빈은 활성화와 비활성화시에 상태를 유지해야 되는 부담이 있기 때문에 메쏘드 단위로 비활성화가 벌어지는 것이 아니라, 컨테이너에서 정한 규칙에 따라 비활성화가 일어난다.
- 상태유지 세션 빈은 오직 하나의 클라이언트에 할당되며 미리 정해진 타임아웃 기간을 갖는데, 이 기간은 배치 디스크립터에서 초 단위로 정해져 클라이언트가 하나의 비즈니스 메쏘드를 호출하고, 다음 메쏘드를 호출하는 사이의 시간에 대하여 적용된다.
- 클라이언트가 타임아웃 기간내에 상태유지 빈을 사용하지 않으면, 빈의 인스턴스가 없어지고 리모트 레퍼런스는 무효화된다. 클라이언트가 종료되거나 빈을 더 이상 사용하지 않게 되었는데도 빈이 쓸데 없이 유지되는 것을 막아준다.

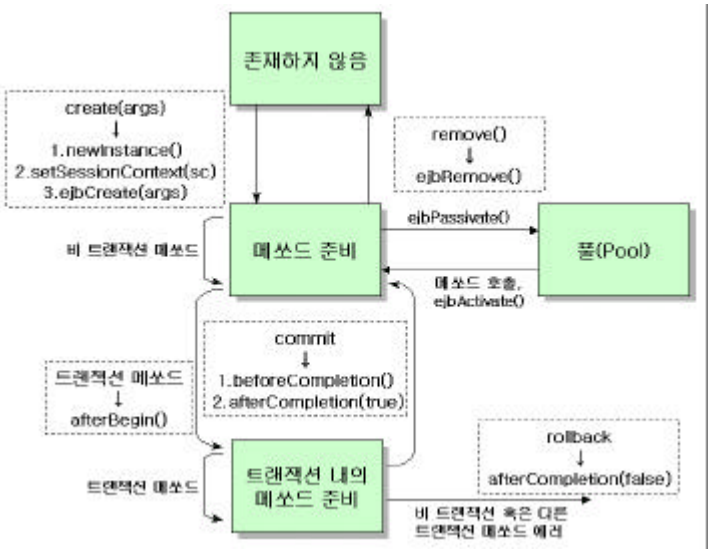
무상태 세션 빈이 가질수 있는 상태



- 무상태 세션 빈은 상태정보유지 세션 빈에 비해 단순한 데, 그 이유는 상태가 유지될 필요가 없으므로 모든 메쏘드 호출이 끝나면 즉시 비활성화되어 버리기 때문이다.
- 무상태 세션 빈은 시스템의 사용 가능한 메모리를 늘려주는 역할을 하기 때문에 가급적 많이 발생하면 좋고, 이런 디자인이 좋은 디자인이 될 것이다.
- 무상태 세션 빈은 클라이언트의 상태 정보를 가질 필요가 없기 때문에 각각의 클라이언트별로 인스턴스를 생성할 필요가 없고 풀에 존재하는 인스턴스는 어떠한 사용자 요청이 들어오는 경우에도 바로 처리할 수 있다. 클라이언트에서 메쏘드 호출이 끝나면 빈은 바로 풀로 이동하게 된다.
- 클라이언트가 create() 메쏘드를 호출하는 경우 실제로 인스턴스가 생성되고 ejbCreate()가 호출되는 것이 아니고, EJBObject만을 생성하며 클라이언트가 리모트 인터페이스에 있는 메쏘드를 호출할 때 비로소 새로운 인스턴스를 생성하며 setSessionContext(), ejbCreate()가 호출된다. 이미 인스턴스가 pool에 존재하는 경우에는 ejbCreate()가 호출되지 않으며 EJBObject에 의해서 바로 이 pool에 있는 인스턴스를 사용하게 된다. 따라서, setSessionContext(), ejbCreate() 메쏘드는 pool에 인스턴스가 없어서 새로운 인스턴스를생성해야 하는 경우에만 호출된다.

상태정보유지 세션 빈이 가질 수 있는 상태

메쏘드 호출관계로 본 상태도



✍✍ '존재하지 않음' 은 세션빈이 생성되지 않은 상태이다

✍✍ "메쏘드 준비" 상태는 클라이언트에서 홈 인터페이스에 정의된 create() 메쏘드를 호출하여 새로운 EJBObject 인스턴스가 생성되는 경우에 호출된다. 이 상태는 리모트 인터페이스에 정의된 메쏘드를 클라이언트가 호출시에 바로 서비스할 수 있는 상태이다. 클라이언트의 메쏘드 요청을 수행하기 위해서는 세션 빈의 인스턴스의 상태가 "메쏘드 준비" 상태에 있어야 한다. 클라이언트가 create()메소드를 호출하면 컨테이너는 해당 빈에 대해서 newInstace()메쏘드를 호출하고 setSessionContext()를 호출한 다음 ejbCreate() 메쏘드를 호출한다. 무상태 세션 빈은 클라이언트가 create()메쏘드를 호출하더라도 현재 풀에 존재하는 인스턴스가 있는경우에는 새롭게 인스턴스를 생성하지 않는다. SetSessioncontext(), ejbCreate()가 호출되지 않는다. 메쏘드는 트랜잭션 하에서 수행될 수도 있고, 그렇지 않을 수도 있다. 이 상태는 메쏘드가 트랜잭션 하에서 수행될 필요가 없는 메쏘드인 경우이다.

✍✍ "트랜잭션 내의 메소드 준비" 상태는 메쏘드가 트랜잭션 하에서 수행되어야 하는 경우이다.

✍✍ "풀(Pool)" 상태는 운영체계들이 이용하는 메모리 운영방식과 같이 자주 사용되는 인스턴스들은 메모리상에 존재하게 되고 오랫동안 사용되지 않는 인스턴스들은 하드디스크등의 2차 저장공간으로 이동한다. 언제 Passivate 될지에 대한 time-out은 deploy시에 설정할 수 있다. 풀에 인스턴스가 모두 차 있는 경우에는 Passivate될 시간이 되지 않은 인스턴스라 할지라도 클라이언트에 의해 호출되고 있지 않다면 풀로 이동한다. 현재 인스턴스가 메쏘드 호출중에 있다면 이 인스턴스는 풀로 이동하지 않는다. 트랜잭션 상태에 있는 동안에도 인스턴스는 Passivate되지 않는다. 상태정보유지 세션 빈은 무상태 세션 빈과 다르게 두가지 풀을 가지는데, 인스턴스가 메모리상에 존재하는 ready pool(cache pool)과 디스크에 저장되는 풀(free pool)이 존재한다.

기타 사항

- 엔터프라이즈 빈을 생성할 때 컨테이너는 setSessionContext()를 통해 SessionContext를 엔터프라이즈 빈에 전달해주고 ejbCreate(args) 메쏘드를 호출한다. 개발자는 ejbCreate(args)를 이용해서 엔터프라이즈 빈의 초기 상태를 설정한다.
- '메쏘드 준비' 상태에서는 '트랜잭션 내의 메쏘드 준비' 상태로 들어가게 된다. 트랜잭션 메쏘드가 호출될 때는 afterBegin() 메쏘드가 호출된다. 이후에 Commit이 발생하기 전에 beforeCompletion()이 호출되고, Commit이 된 이후에는 afterCompletion(true)이 불려진다. Rollback의 경우에는 afterCompletion(false)만 호출된다.
- 비활성화는 엔터프라이즈 빈이 트랜잭션 상태에 있지 않을때만 일어난다. 비활성화가 일어날 때는 ejbPassivate()가 호출되며, 이후에 메쏘드 호출을 통해서 활성화가 되면 ejbActivate()가 호출된다. 비활성화되는 정확한 주기는 컨테이너에 따라 달라질 수 있다.
- 트랜잭션 메쏘드 호출이 끝나고 트랜잭션이 종료될 때 발생하거나 LRU(least recently used) 알고리즘에 의해서 발생할 수도 있다. 실제 동작은 컨테이너가 담당하고 전후에 이 메쏘드들을 호출해 주게 된다. 개발자들은 컨테이너가 호출하는 메쏘드를 이용해서 다양한 작업을 처리할 수 있게 된다.

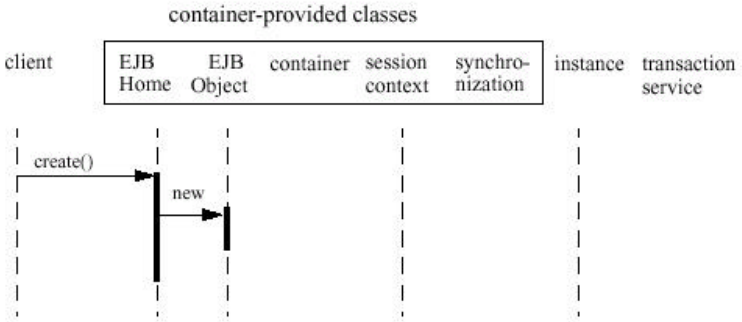
무상태 세션 빈와 상태정보유지 세션 빈의 비교

무상태 세션빈	상태정보유지 세션빈
클라이언트의 상태정보를 유지할 수 없다	클라이언트의 상태정보를 계속 유지할 수 있다
하나의 인스턴스는 여러 명의 클라이언트에 의해 사용될 수 있다	하나의 인스턴스는 한 클라이언트에 의해서만 사용된다
Passivate 상태로 이동하지 않는다	Passivate/Activate 상태로 이동한다

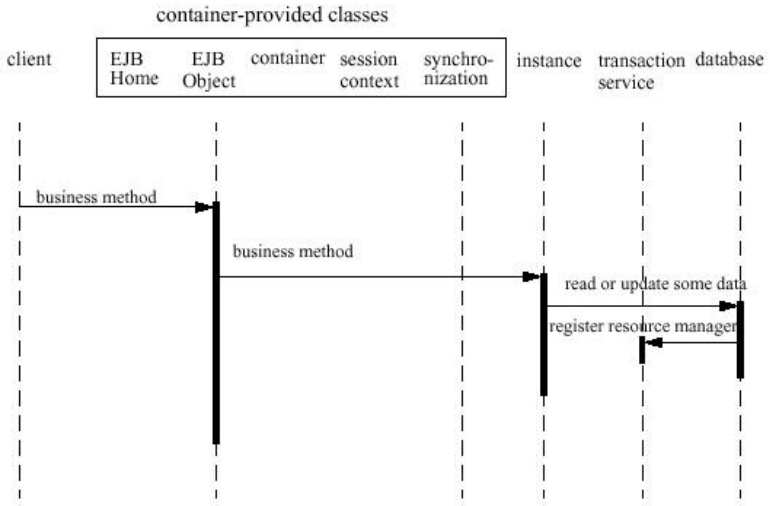
각 세션 빈의 Object Interaction Diagram

✍✍ 무상태 세션 빈

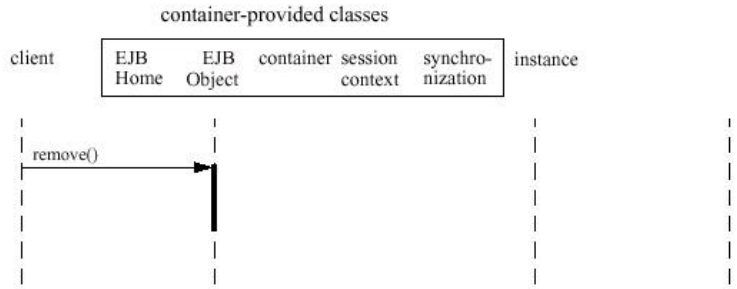
객체 Creation



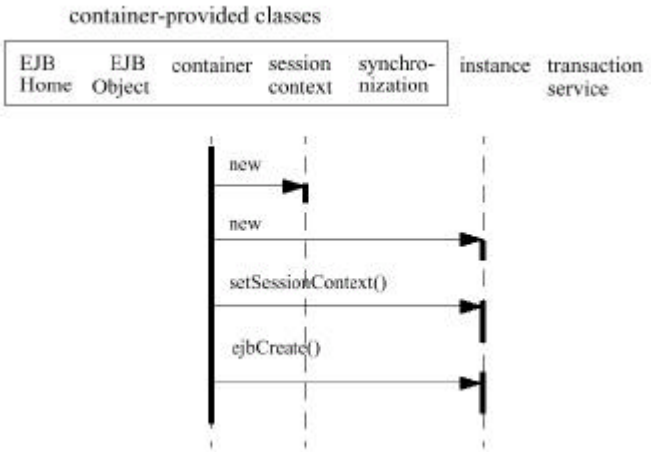
Business 메쏘드 invoke



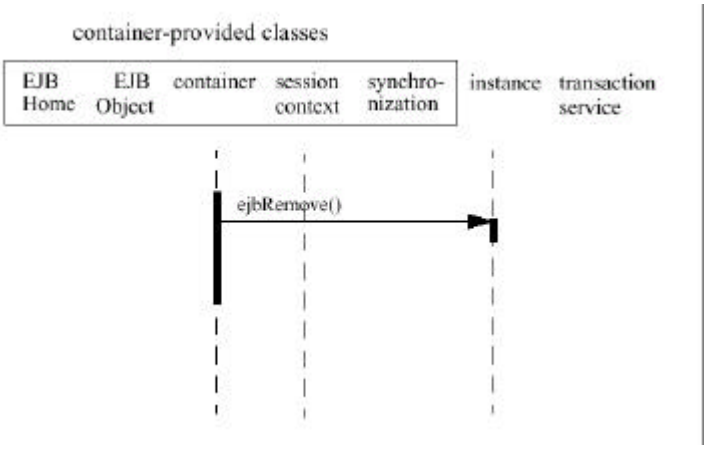
객체 Remove



Pool에 Instance Add

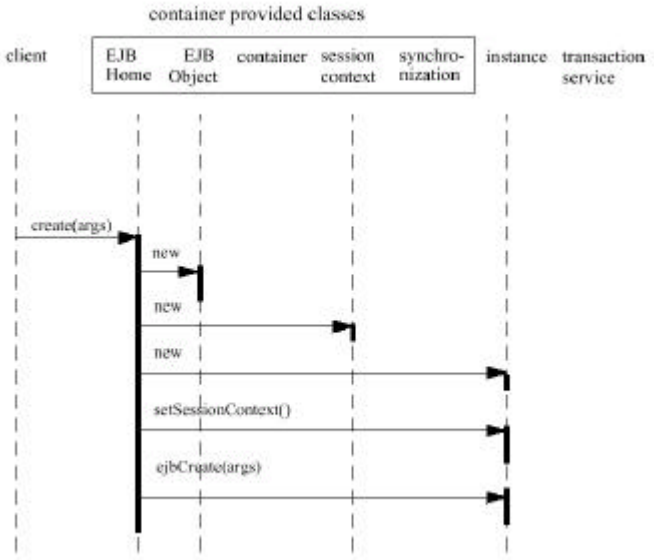


Pool에서 Instance Remove

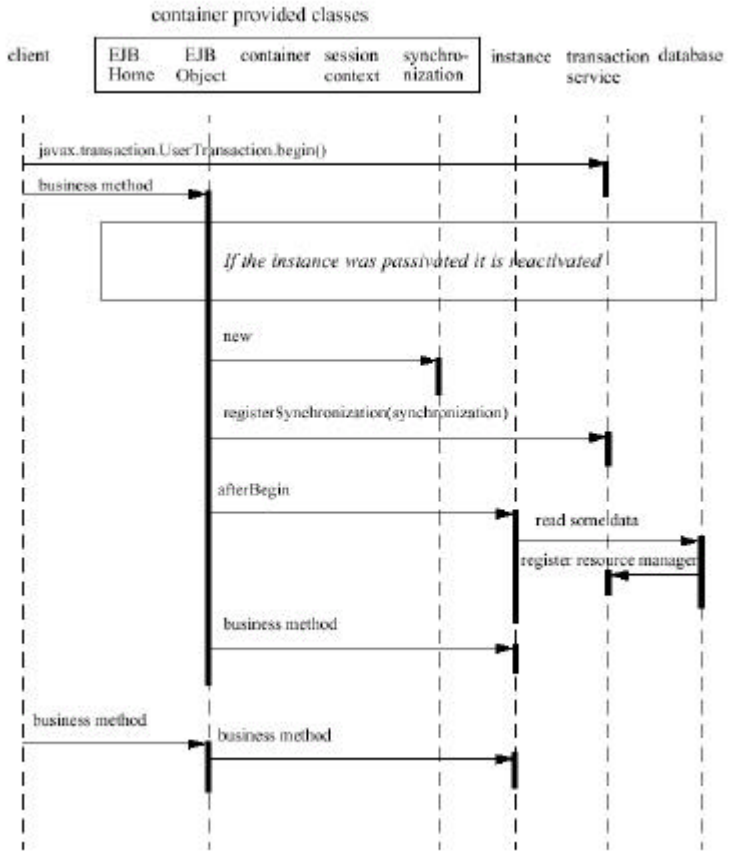


✍✍ **상태정보유지 세션 빈**

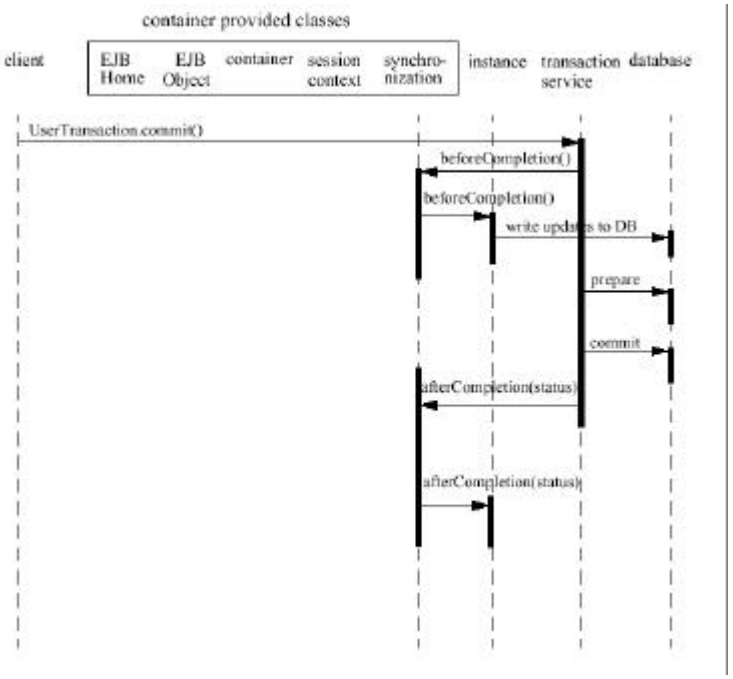
객체 Creation



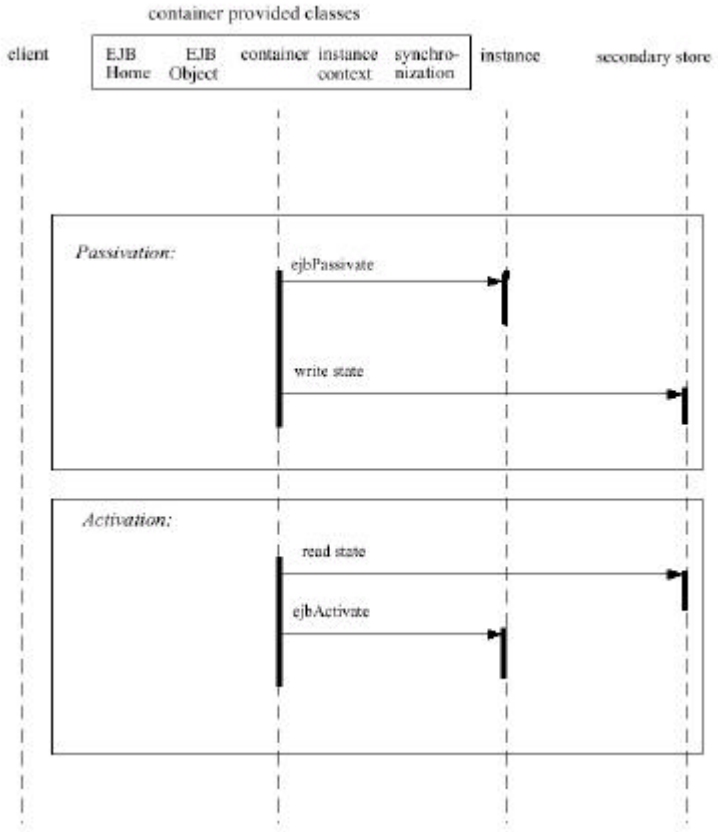
✍✍ **트랜잭션** Start



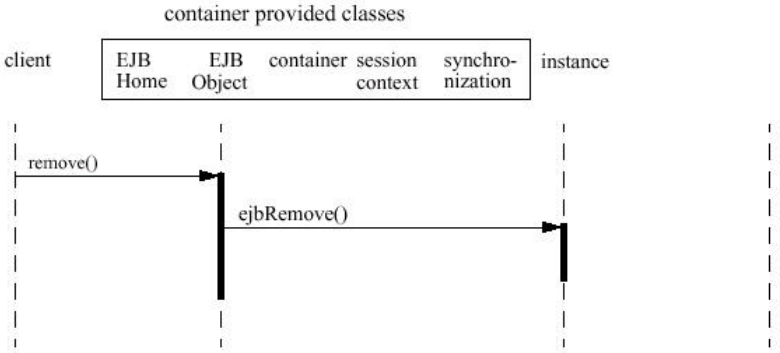
트랜잭션 Commit



객체의 Activation/Passivation



객체 Remove


container provided classes
client
EJB Home
EJB Object
container
session context
synchro-nization
instance
remove()
ejbRemove()

무상태 세션 빈 구현

무상태 세션 빈인 간단한 컨테이너의 동작을 확인할 수 있는 프로그램으로 한사람의 이름과 나이를 가지는 빈으로 Person이라는 빈은 한 사람의 이름과 나이를 가지고 있다. 클라이언트는 이 Person에 정의되어 있는 메쏘드를 통해서 이름과 나이를 설정, 얻을 수 있다. 또 한 메소드 안에서 sleep을 두어 메소드 아에서 멤버변수를 사용하는 도중에 다른 사용자에의해서 값이 변경되었을 때의 데이터 무결성을 확인하는 프로그램이다.

홈 인터페이스

클라이언트가 EJB를 얻기위해 EJB 홈에 접근해야 한다.

<소스> PersonHome.java

```
package sample.pse.stateless;

import javax.ejb.*;
import java.rmi.*;
public interface PersonHome extends EJBHome    {
   public Person create() throws CreateException, RemoteException;
}
```

리모트 인터페이스

한 사람의 이름과 나이를 설정하거나 얻는 메쏘드이다. SetNameSleep(), setAgeSleep() 메쏘드는 setName(), setAge()와 같은 기능을 하는 메쏘드지만, 빈 안에서 2초 동안 sleep 상태로 빠지게 한다.

<소스> Person.java

```
package sample.pse.stateless;
import javax.ejb.*;
import java.rmi.*;

public interface Person extends EJBObject    {
   public void setName(String name) throws RemoteException;
   public void setAge(int age) throws RemoteException;
   public String getName() throws RemoteException;
   public int getAge() throws RemoteException;
```

```
    public void setNameSleep(String name) throws RemoteException;
    public void setAgeSleep(int age) throws RemoteException;
}
```

빈 클래스

리모트 인터페이스의 구현 클래스이다. 각각의 메쏘드에 대해서 해당 빈의 어떤 메쏘드가 호출이 되는지를 확인하기 위해 시스템 아웃으로 출력하는 문장만을 가지고 있다.

setNameSleep(), setAgeSleep()**는 앞에서 설명했듯이 내부적으로** 2**초 동안** sleep **상태로 빠져 있는 동안에 다른 클라이언트가 해당 빈의** name, age**값을 변경하였을 때** 2**초 동안** sleep**상태에 있었던 이 메쏘드는** name, age **등의 멤버변수가 어떻게 바뀌는지를 확인하여 동시사용자에 대한 데이터 무결성을 보장하는지를 확인 한다**.

<소스> PersonBean.java

```
package sample.pse.stateless;

import javax.ejb.*;
import java.rmi.*;

public class PersonBean implements SessionBean    {
    String name;
    int    age;

    public PersonBean()      {
        System.out.println("constructor called");
    }
    public void ejbCreate()    {
        System.out.println("ejbCreate called");
    }
    public void ejbRemove()    {
        System.out.println("ejbRemove called");
    }
    public void ejbActivate()  {
        System.out.println("ejbActivate called");
    }
    public void ejbPassivate() {
        System.out.println("ejbPassivate called");
    }
    public void setSessionContext(SessionContext ctx)  {
```

```
        System.out.println("SessionContext called");
    }
    public void setName(String name)    {
        this.name = name;
    }
    public void setAge(int age)  {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public void setNameSleep(String name)   {
        System.out.println("setNameSleep called");
        try  Thread.sleep(2000); catch(Exception e)
        System.out.println("name,age : " + this.name + "," + this.age);
        this.name = name;
    }
    public void setAgeSleep(int age)    {
        System.out.println("setAgeSleep called");
        try  Thread.sleep(2000); catch(Exception e)
        System.out.println("name, age : " + this.name + "," + this.age);
        this.age = age;
    }
}
```

Client 프로그램

✍ 무상태 세션빈을 사용하는 클라이언트는 상태정보를 유지하지 않는다. 한 사용자와의 상태유지는 한 메쏘드 단위로 이루어지게 된다. 만약 무상태 세션빈에 멤버변수가 정의되어 있고, 이 멤버변수를 한 클라이언트가 변경하면 다른 클라이언트에도 변경된 값이 적용된다.

✍ 서블릿이나 RMI 객체처럼 이 인스턴스는 모든 사용자에 의해 공유되는 인스턴스라고 볼 수 있다. 한 사용자가 서블릿이나 RMI에 정의된 멤버변수를 수정하면 실시간에 다른 클라이언트도 동시에 변경된다. 하지만 서블릿이나 RMI의 경우는 동시사용자에 대한 데이터 무결성에 대한 보장이 없기 때문에 데이터 무결성을 보장하지 못한다. 하지만 무상태 세션빈은 서블릿과 RMI처럼 다른 사용자가 수정한 멤버변수가 모든 사용자에 적용되지만 동시사용자 접근

시의 데이터 무결성을 보장한다.

✍✍ 이 프로그램은 하나의 홈 인터페이스를 통해서 두 개의 빈을 얻은 후 name, age 등의 멤버변수 값을 변경하는 두 개의 메쏘드(setName(), setAge())를 호출한 후 홈 인터페이스를 통해 얻은 두 개의 빈이 같은 값을 가지는지를 확인하는 과정이다.

✍✍ 그리고 두 개의 빈을 쓰레드로 호출하여 동시 사용자에 대해 어떻게 처리가 되는지를 확인한다. 쓰레드 생성시에 4번째 파라미터가 true인 경우는 setNameSleep(), setAgeSleep()를 호출하며, false인 경우는 setName(), setAge() 메쏘드를 호출한다. 한 쓰레드가 setNameSleep(), setAgeSleep()를 호출하여 2초 동안 빈이 sleep 상태에 있는 동안 다른 쓰레드가 이 빈에 대해서 setName(), setAge()를 호출할 경우 sleep에서 빠져 나온 클라이언트의 name, age값이 어떻게 바뀌는지를 확인한다.

<소스> StatelessClient.java

```
import java.rmi.*;
import javax.naming.*;
import java.util.*;
import sample.pse.stateless.*;

public class StatelessClient     {
   public static void main(String[] args)  {
      try    {
         Context ctx = getInitialContext();
         PersonHome home =
(PersonHome)ctx.lookup("sample.pse.stateless.PersonHome");
         Person obj1 = home.create();
         Person obj2 = home.create();
         obj1.setName("홍길동");
         obj1.setAge(20);
         System.out.println("----obj1의 내용----");
         System.out.println("name<obj1> : " + obj1.getName());
         System.out.println("age<obj1> : " + obj1.getAge());
         System.out.println("----obj2의 내용----");
         System.out.println("name<obj2> : " + obj2.getName());
         System.out.println("age<obj2> : " + obj2.getAge());
         obj2.setName("임꺽정");
         obj2.setAge(30);
         System.out.println("----obj1의 내용----");
         System.out.println("name<obj1> : " + obj1.getName());
         System.out.println("age<obj1> : " + obj1.getAge());
```

```
            System.out.println("----obj2 의 내용----");
            System.out.println("name<obj2> : " + obj2.getName());
            System.out.println("age<obj2> : " + obj2.getAge());
            obj1.remove();
            obj2.remove();

            System.out.println("thread start...");

            SLThread t1 = new SLThread(home, "thread1", 40, true);
            SLThread t2 = new SLThread(home, "thread2", 50, false);
            t1.start();
            t2.start();
            }
        catch(Exception e)  {
            e.printStackTrace();
        }
    }
    public static Context getInitialContext() throws NamingException {
        Hashtable h = new Hashtable();
        h.put(Context.PROVIDER_URL, "t3://localhost:7001");
        h.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        return new InitialContext(h);
    }
}

class SLThread extends Thread  {
    PersonHome home = null;
    String name = null;
    int    age;
    boolean sleep;

    public SLThread(PersonHome home, String name, int age, boolean sleep)  {
        this.home = home;
        this.name = name;
        this.age = age;
        this.sleep = sleep;
    }
    public void run()   {
```

```
        if(sleep) System.out.println(name + " 2초동안 sleep");
        try    {
            Person obj = home.create();
            if(sleep)
            {
                obj.setNameSleep(name); System.out.println("setNameSleep
called");
            }
            else obj.setName(name);
            if(sleep)
            {
                obj.setAgeSleep(age); System.out.println("setAgeSleep called");
            }
            else obj.setAge(age);

            System.out.println("<"+name+"> name : " + obj.getName());
            System.out.println("<"+name+"> age: " + obj.getAge());
            obj.remove();
            System.out.println("finished");
    }       catch(Exception e)    {
            e.printStackTrace();
        }
    }
}
```

Deployment Descriptor

Deployment Descriptor 파일을 작성하는 형식은 EJB 서버마다 다를 수 있다. 하지만 deployment Descriptor 파일이 가져야 할 정보는 표준이므로 방법이 다르더라도 의미는 같다.

Deployment descriptor를 작성하는 이유는 개발자가 부담해야 할 구현부분을 컨테이너가 알아서 처리할 수 있도록 하기 위한 것이다. 해당 EJB가 컨테이너 안에서 어떻게 수행되어야 하는지와 환경에 따라 커스터마이징되어야 할 부분에 대해 정의함으로써 환경이 바뀌더라도 소스를 수정하는 경우를 없애기 위해서이다.

```
(SessionDescriptor
 beanHomeName                statelessSession.TraderHome
 enterpriseBeanClassName
```

```
examples.ejb.basic.statelessSession.TraderBean
  homeInterfaceClassName
examples.ejb.basic.statelessSession.TraderHome
  remoteInterfaceClassName      examples.ejb.basic.statelessSession.Trader
  isReentrant                   false
  ; session EJBean-specific properties
  stateManagementType           STATELESS_SESSION
  sessionTimeout                5; seconds
  ; end session EJBean-specific properties

  (accessControlEntries
;   DEFAULT                     [admin managers]
  ); end accessControlEntries

  (controlDescriptors
    (DEFAULT
      isolationLevel            TRANSACTION_SERIALIZABLE
      transactionAttribute      TX_REQUIRED
      runAsMode                 CLIENT_IDENTITY
;     runAsIdentity             traderAdmin
    ); end DEFAULT
  ); end controlDescriptors

  (environmentProperties
    ; Application-specific environment properties
    (StockPrices
      ; Stock prices is a nested hash table
      WEBL   10.0
      INTL   15.0
    ); end Stock Prices

    tradeLimit                  500

    ; end Application-specific environment properties

;   homeClassName
;   ejbObjectClassName
    maxBeansInFreePool          100
    maxBeansInCache             100
```

```
    idleTimeoutSeconds          5
  ); end environmentProperties
); end SessionDescriptor
```

수행 결과 분석

클라이언트쪽 출력 결과	서버쪽 출력 결과
----obj1의 내용---- name<obj1> : 홍길동 age<obj1> : 20 ----obj2의 내용---- name<obj2> : 홍길동 age<obj2> : 20 ----obj1의 내용---- name<obj1> : 임꺽정 age<obj1> : 30 ----obj2의 내용---- name<obj2> : 임꺽정 age<obj2> : 30 thread start... thread1 2초동안 sleep <thread2> name : thread2 <thread2> age: 50 finished setNameSleep called setAgeSleep called <thread1> name : thread1 <thread1> age: 40 finished	constructor called SessionContext called ejbCreate called setNameSleep called constructor called SessionContext called ejbCreate called name,age : 임꺽정,30 setAgeSleep called name, age : thread1,30

✍✍ 상태 정보 유지

무상태 세션빈은 각각의 클라이언트의 상태정보를 가지지 않는다. 한 클라이언트에 의해 값이 바뀌면 다른 클라이언트도 바뀐 값이 적용된다.

클라이언트 프로그램이 처음으로 하는 작업은 JNDI를 통해 하나의 홈 인터페이스를 얻고 이 인터페이스를 통해서 두 개의 EJB 객체를 얻는다.

obj1에 대해서 setName("홍길동"); setAge(20); 메쏘드를 호출하고 obj1과 obj2의 값을 출력한다. 결과에서 보면 이 두 개의 빈은 모두 같은 값을 가진다. 그 다음 obj2에 대해서 마찬가지로 setName("임꺽정"); setAge(30);를 호출하고 난 결과를 보면 obj1과 obj2가 같은 값을 갖는다.

이처럼 무상태 세션빈인 경우는 해당 빈을 사용하는 모든 클라이언트가 같은 값을 갖고, 한 메쏘드 단위로 세션이 유지된다고 볼 수 있다. 따라서 대화형으로 빈을 사용할 수 없으며 클라이언트의 상태정보를 계속 유지하고자 하는 경우에는 이 형태의 빈은 사용할 수 없다.

✍ 동시사용자 처리

멀티쓰레드로 수행하면서 한 쓰레드는 2초 동안 sleep 상태에 빠지는 메쏘드인 setNameSleep(), setAgeSleep()를 호출하며 다른 쓰레드는 setName(), setAge()를 호출하여 다른 쓰레드에 의해서 값이 변경이 되었을 때 sleep 상태에서 빠져 나온 setNameSleep()와 setAgeSleep() 메쏘드가 어떠한 값을 가지고 있는지를 확인하도록 한다. SetNameSleep()와 setAgeSleep() 메쏘드는 2초 동안 sleep 상태에서 빠져 나온 후에 name, age 멤버변수 값을 출력한다.

서버쪽의 출력결과에서 알 수 있듯이 다른 쓰레드(Thread2)에 의해서 name과 age 변수값이 thread2, 50으로 바뀌었지만 setNameSleep()를 호출한 후 sleep 상태에서 빠져 나온 obj1의 name, age값은 내부적으로 "임꺽정", 30의 값을 가진다.

하나의 메쏘드가 끝나기 전에 다른 사용자가 멤버변수 값을 변경하더라도 변경된 값으로 바뀌지 않는다. 그리고 setAgeSleep()메쏘드가 sleep에서 빠져 나온 후의 값은 thread1, 30이라는 값을 가진다. SetNameSleep()에 의해 name이라는 멤버변수가 thread1으로 변경되었기 때문이다. 따라서 동시사용자에 의해 호출이 되더라도 메쏘드 안에서 값이 변경되지 않기 때문에 데이터 무결성을 보장한다.

✍ 생명주기

서버쪽 메시지를 보면 최초에 생성자가 호출되며 뒤이어 setSessionContext()가 호출되고, ejbCreate()가 호출된다. 클라이언트가 홈 인터페이스에 정의되어 있는 create() 메쏘드를 호출하면 새로운 EJB 인스턴스가 생성되며, 해당 EJBHome에 의해 setSessionContext()와 ejbCreate()가 호출된다.

새로운 빈 인스턴스가 생성되고 setSessionContext(), ejbCreate()가 호출되는 시점은 클라이언트가 create() 메쏘드를 호출하는 시점이 아니다. 클라이언트에서 create()메쏘드를 호출하면 실제로는 EJBObject만 생성할 뿐이다. 사용자가 리모트 인터페이스에 정의되어 있는 메쏘드를 호출하는 시점에 새로운 빈을 생성하고 호출된다.

클라이언트 프로그램이 이 세션빈을 사용하는 동안 실제로 서버상에는 두 개의 인스턴스가 생성된다. 클라이언트에서 홈 인터페이스의 create()메쏘드는 4번을 호출했지만 실제로 서버상에는 2개의 인스턴스만 생성된다. 이 인스턴스가 생성된 시점을 잘 살펴보면, 첫 번째 인스턴스는 당연히 메쏘드 호출이 발생하기 전에 생겼으며, 두 번째 인스턴스는 thread1이 setNameSleep()를 호출한 후에 생성되었다. SetNameSleep() 메쏘드가 호출되기 전에 클라이언트에서는 두 번의 create()메쏘드 호출을 했지만, 서버상에는 한 개의 인스턴스만 생성된 것이다.

무상태 세션빈은 method-ready pool만이 존재한다. 클라이언트가 create()메쏘드를 호출하였을 때 서버상의 method -ready pool에 인스턴스가 존재하는 경우에는 새로운 인스턴스를 생성하는 것이 아니라 이 풀에 있는 인스턴스를 사용하

게 된다. 클라이언트에서 create()메쏘드를 호출하면 풀에 있는 하나의 인스턴스를 클라이언트에 할당한다.

무상태 세션빈에서는 메쏘드 단위로 세션이 이루어지기 때문에 메쏘드 호출이 끝난 후에는 해당 클라이언트에 의해 사용되었던 빈은 이 풀로 이동하게 된다.

클라이언트 프로그램에서 쓰레드로 호출하기 전에 두 번의 create()호출을 했지만, obj1의 메쏘드 호출이 끝나면 인스턴스가 풀로 이동한다. 따라서 그 다음의 호출은 새로운 인스턴스를 생성하는 것이 아니고 풀에 있는 인스턴스를 사용하게 된다.

클라이언트에 의해 메쏘드 호출중에 있는 동안 다른 클라이언트가 create()호출한 후 메쏘드를 호출하였을 때, 풀에 인스턴스가 없으면 새로운 인스턴스를 생성하게 된다. 서버쪽 메시지에서 한 쓰레드가 SetNameSleep()를 호출하여 2초 동안 인스턴스를 할당받은 동안 다른 쓰레드에서 호출하였기 때문에 setNameSleep()호출 후에 인스턴스가 생성된 것이다.

사례 연구-1

쇼핑몰에서 A 고객은 5% 할인, B 상품 10개 구입시 15% 할인등의 가격정책을 적용하여 가격을 재계산하는 무상태 세션 빈 프로그램이다.

리모트 인터페이스

<소스> Pricer.java

```
package com.wiley.compBooks.ecommerce;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.rmi.Remote;

/**
 * These are the business logic methods exposed publicly
 * by PricerBean.
 *
 * This interface is what clients operate on when they
 * interact with beans. The container vendor will
 * implement this interface; the implementation object is
 * called the EJB Object, which delegates invocations to
 * the actual bean.
 */
public interface Pricer extends EJBObject {

    /**
     * Computes the price of a set of goods
     */
    public void price(Quote quote) throws RemoteException, PricerException;
}
```

홈 인터페이스

<소스> PricerHome.java

```
package com.wiley.compBooks.ecommerce;

import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * This is the home interface for Pricer.  The
 * container implements this interface; the
 * implementation object is called the Home Object,
 * and serves as a factory for EJB Objects.
 */
public interface PricerHome extends EJBHome {

   /*
    * This method creates the EJB Object.
    *
    * @return The newly created EJB Object.
    */
  Pricer create() throws RemoteException, CreateException;
}
```

Bean Class

<소스> PricerBean.java

```
package com.wiley.compBooks.ecommerce;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.*;

/**
 * Stateless Session Bean which computes prices based
 * upon a set of pricing rules.  The pricing rules are
 * deployed with the bean as environment properties.
 */
```

```
public class PricerBean implements SessionBean {

    // Constants used for reading properties
    public static final String ENV_DISCOUNT = "DISCOUNT_";
    public static final String ENV_TAX_RATE = "TAX_RATE";

    /*
     * Although this is a stateless session bean, we
     * do have state - the session context.  Remember
     * that stateless session beans can store state, they
     * just can't store state on behalf of particular
     * clients.
     */
    private SessionContext ctx;

    // Begin business methods

    /**
     * Computes the total price of a quote.  This includes
     * subtotal and taxes.  Sets the appropriate quote
     * fields.  We expose this coarse-grained method to
     * avoid network roundtrips for pricing individual
     * parts of a quote.
     */
    public void price(Quote quote) throws PricerException, RemoteException {
            priceSubtotal(quote);
            priceTaxes(quote);
    }

    /**
     * Computes the subtotal price for a set of products the
     * customer is interested in.  The subtotal takes into
     * account the price of each product the customer wants,
     * the quantity of each product, and any discounts the
     * customer gets.  However, the subtotal ignores taxes.
     * @param quote All the data needed to compute the
     * subtotal is in this parameter.
     */
    private void priceSubtotal(Quote quote) throws PricerException,
```

```
RemoteException {
        System.out.println("PricerBean.priceSubtotal() called");

        /*
         * Customers with certain names get discounts.
         * The discount rules are stored in the
         * environment properties that the bean is
         * deployed with.
         *
         * Get the name of this customer.
         */
        Customer customer = quote.getCustomer();
        String customerName = customer.getName();

        double percentDiscount = 0;
        try {
         for (int i=0; ; i++) {
                /*
                 * Get the next discount equation from
                 * the environment properties.  A
                 * discount equation has the form
                 * "<name>=<% discount>".  For example,
                 * "Ed Roman=50" means that the
                 * Customer whose name is "Ed Roman"
                 * gets a 50% discount.
                 */
                String discount = (String)
ctx.getEnvironment().get(ENV_DISCOUNT + i);

                /*
                 * Break the equation up into
                 * customer name, discount
                 */
                StringTokenizer tokens = new StringTokenizer(discount,
"=", false);
                String discountCustomerName = tokens.nextToken();
                percentDiscount =
Double.valueOf(tokens.nextToken()).doubleValue();
```

```
                        /*
                         * If this discount applies to this
                         * customer, then stop.  Otherwise,
                         * move on to the next discount
                         * equation.
                         */
                        if (!discountCustomerName.equals(customerName)) {
                                continue;
                        }

                        System.out.println("Pricer.priceSubtotal(): " +
customerName + " gets a " + percentDiscount + "% discount.");
                }
                }
                catch (Exception e) {
                        System.out.println("Pricer.priceSubtotal(): " +
customerName + " doesn't get a discount.");
                }
                /*
                 * Now we know how much discount to apply.  The
                 * next step is to apply the discount to each
                 * product the customer is interested in.
                 *
                 * We need to get the price and quant ity of each
                 * product the customer* wants.  The quote object
                 * stores this information in individual quote
                 * line-items.  Each line-item has price and
                 * quantity information for a single product.
                 */

                /*
                 * First, get the Line Items contained within the
                 * Quote.
                 */
                Enumeration lineItems = quote.getLineItems().elements();

                /*
                 * Compute the subtotal
                 */
```

```
                double subTotal = 0;
                try {
                        /*
                         * For each line item...
                         */
                        while (lineItems.hasMoreElements()) {
                        QuoteLineItem li = (QuoteLineItem)
                                        lineItems.nextElement();

                                /*
                                 * Get the price of the line item...
                                 */
                                double basePrice = li.getBasePrice();

                                /*
                                 * Set the discount for this line item...
                                 */
                                li.setDiscount(basePrice * percentDiscount);

                                /*
                                 * Apply the discount...
                                 */
                                double aggregatePrice =
                                basePrice - (basePrice * percentDiscount);

                                /*
                                 * And add the price to the subtotal.
                                 */
                                subTotal += aggregatePrice;
                        }

                        quote.setSubtotal(subTotal);
                }
                catch (Exception e) {
                        throw new PricerException(e.toString());
                }
        }

        /**
```

```
     * Computes the taxes on a quote.
     * Since the taxes are based on the subtotal, we assume
     * that the subtotal has already been calculated.
     */
    private void priceTaxes(Quote quote) throws PricerException,
RemoteException {
            System.out.println("PricerBean.priceTaxes() called");

            Properties props = ctx.getEnvironment();
            /*
             * Compute the taxes
             */
            String taxRateStr = (String)
                    ctx.getEnvironment().get(ENV_TAX_RATE);
            double taxRate = Double.valueOf(taxRateStr).doubleValue();
            double subTotal = quote.getSubtotal();
            quote.setTaxes(taxRate * subTotal);
    }

    // End business methods
    // Begin EJB-required methods.  The methods below are
    // called by the Container, and never called by client
    // code.

    public void ejbCreate() throws RemoteException {
            System.out.println("ejbCreate() called.");
    }

    public void ejbRemove() {
            System.out.println("ejbRemove() called.");
    }

    public void ejbActivate() {
            System.out.println("ejbActivate() called.");
    }

    public void ejbPassivate() {
            System.out.println("ejbPassivate() called.");
    }
```

```
    public void setSessionContext(SessionContext ctx) {
            System.out.println("setSessionContext() called");
            this.ctx = ctx;
    }
}
```

사례 연구-2

쇼핑몰에서 가장 중요한 결제등과 관련된 로직을 처리하는 무상태 세션 빈 프로그램이다.

리모트 인터페이스

<소스> Teller.java

```
package com.wiley.compBooks.ecommerce;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.rmi.Remote;

/**
 * These are the business logic methods exposed publicly
 * by TellerBean.
 *
 * This interface is what clients operate on when they
 * interact with beans. The container vendor will implement
 * this interface; the implementation object is called the
 * EJB Object, which delegates invocations to the actual bean.
 */
public interface Teller extends EJBObject {

  /**
   * Looks up an account number (if you forgot it)
   * @param name Your full name
   */
  public String lookupAccountNumber(String name) throws RemoteException,
TellerException;

  /**
   * Creates a new account with a starting deposit.
   *
   * @param name Your full name
   * @param initialDeposit The initial starting balance you
```

```
     *                  want to deposit.
     *
     * @return Your new account number
     */
    public String createNewAccount(String yourName, double initialDeposit)
throws RemoteException, TellerException;

    /**
     * Closes your account.
     *
     * @return The funds left in your account are returned to
     *         you.
     */
    public double closeAccount(String accountNum) throws RemoteException,
TellerException;

    /**
     * Gets your current balance for an account.
     *
     * @param accountNum the account number string.
     */
    public double getBalance(String accountNum) throws RemoteException,
TellerException;

    /**
     * Switches names on an account.
     */
    public void changeNames(String accountNum, String newName) throws
RemoteException, TellerException;

    /**
     * Transfers funds from accountNum1 to accountNum2
     */
    public void transfer(String accountNum1, String accountNum2, double funds)
throws RemoteException, TellerException;

    /**
     * Withdraws from your account.
     *
```

```
   * @exception TellerException thrown if you have
   *            insufficient funds.
   * @return Your withdrawn funds.
   */
  public double withdraw(String accountNum, double funds) throws
RemoteException, TellerException;

  /**
   * Deposits into your account.
   */
  public void deposit(String accountNum, double funds) throws
RemoteException, TellerException;
}
```

홈 인터페이스

<소스> TellerHomejava

```
package com.wiley.compBooks.ecommerce;

import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * This is the home interface for TellerBean.  This
 * interface is implemented by the EJB Server's glue-
 * code tools - the implemented object is called the
 * Home Object, and serves as a factory for EJB Objects.
 *
 * One create() method is in this Home Interface, which
 * corresponds to the ejbCreate() method in the
 * TellerBean file.
 */
public interface TellerHome extends EJBHome {

   /*
    * This method creates the EJB Object.
    *
```

```
     * @return The newly created EJB Object.
     */
    Teller create() throws RemoteException, CreateException;
}
```

Bean Class

<소스> TellerBean.java

```
package com.wiley.compBooks.ecommerce;

import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import javax.naming.*;

/**
 * Stateless Session Bean which acts as a bank teller,
 * operating a bank.  Behind the scenes, this Teller
 * Bean interacts with the Account entity bean.
 */
public class TellerBean implements SessionBean {

    /*
     * Although this is a stateless session bean,
     * we do have state - the session context, and
     * a home object used to find/create accounts.
     * Remember that stateless session beans can
     * store state, they just can't store state on
     * behalf of particular clients.
     */
    private AccountHome home;
    private SessionContext ctx;

    // Begin internal methods
    /**
     * Finds the Home Object for Bank Accounts.
     */
```

```
    private void findHome() throws Exception {
        try {
                /*
                 * Get properties from the
                 * Session Context.
                 */
                Properties props = ctx.getEnvironment();

                /*
                 * Get a reference to the AccountHome
                 * Object via JNDI.  We need the
                 * Account's Home Object to create
                 * Accounts.
                 *
                 * We rely on app-specific properties
                 * to define the Initial Context params
                 * which JNDI needs.
                 */
                Context initCtx = new InitialContext(props);
                home = (AccountHome)
initCtx.lookup("Ecommerce.AccountHome");
            }
            catch (Exception e) {
                e.printStackTrace();
                throw e;
            }
    }

    // End internal methods
    // Begin business methods
    /**
     * Private helper method.  Finds an Account Entity
     * Bean based upon its ID.
     */
    private Account findAccountByNumber(String accountID) throws
TellerException, RemoteException {
            try {
                /*
                 * Construct a Primary Key from the
```

```
                        * passed accountID, and call the
                        * Entity Bean Home's default finder
                        * method.
                        */
                        return home.findByPrimaryKey(new
AccountPK(accountID));
                }
                catch (FinderException e) {
                        e.printStackTrace();
                        throw new TellerException(e.toString());
                }
        }

        /**
         * Private helper method.  Finds an Account Entity
         * Bean based upon it's owner's name.
         */
        private Account findAccountByOwnerName(String name) throws
TellerException, RemoteException {
                try {
                        /*
                         * Call our Entity's custom finder
                         * method that finds by name.
                         */
                        return home.findByOwnerName(name);
                }
                catch (FinderException e) {
                        e.printStackTrace();
                        throw new TellerException(e.toString());
                }
        }

        /**
         * Looks up an account number based upon your name
         * (perhaps you forgot it).
         *
         * @exception TellerException thrown if error occurs,
         *            such as a name not found.
         * @param name Your full name
```

```
     */
    public String lookupAccountNumber(String name) throws RemoteException,
TellerException {

            /*
             * Get the Account Entity Bean
             */
            Account account = findAccountByOwnerName(name);

            /*
             * Return the value of the getAccountID()
             * accessor method on our Entity Bean.
             */
            return ((AccountPK)account.getPrimaryKey()).accountID;
    }

    /**
     * Creates a new account with a starting deposit.
     *
     * @param name Your full name
     * @param initialDeposit The initial starting balance
     *        you want to deposit.
     *
     * @return Your new account number
     */
    public String createNewAccount(String name, double initialDeposit) throws
RemoteException, TellerException {
            /*
             * Generate a unique Bank Account Number.  This
             * is a hard problem to solve, in general, with
             * EJB.  We'll see the issues at hand in Chapter
             * 13.  For now, we'll punt and use the System
             * Clock to create a unique identifier.  However,
             * note that this will cause our bank account to
             * fail if two accounts are created in the same
             * millisecond.
             */
            String accountNum = Long.toString(System.currentTimeMillis());
```

```
                try {
                        /*
                         * Create a new Account Entity Bean
                         */
                        Account account = home.create(accountNum, name);

                        /*
                         * Start the account off with an initial deposit
                         */
                        account.deposit(initialDeposit);

                        /*
                         * Return the new account number
                         */
                        return accountNum;
                }
                catch (CreateException e) {
                        e.printStackTrace();
                        throw new TellerException("Could not create account: " +
e.toString());
                }
        }

        /**
         * Closes your account.
         *
         * @return The funds left in your account are
         *         returned to you.
         */
        public double closeAccount(String accountNum) throws RemoteException,
TellerException {

                /*
                 * Find the correct Account Entity Bean
                 */
                Account account = findAccountByNumber(accountNum);

                /*
                 * Withdraw all the funds from the Entity
```

```
                 */
                double balance = account.getBalance();

                /*
                 * Close the account by removing the Entity Bean
                 */
                try {
                        account.remove();
                }
                catch (RemoveException e) {
                        e.printStackTrace();
                        throw new TellerException("Could not close account: " +
e.toString());
                }

                /*
                 * Return the remaining funds
                 */
                return balance;
        }

        /**
         * Gets your current balance for the account
         * numbered accountNum.
         */
        public double getBalance(String accountNum) throws RemoteException,
TellerException {

                /*
                 * Find the correct Account Entity Bean
                 */
                Account account = findAccountByNumber(accountNum);

                /*
                 * Get the Entity Bean's balance, and return it
                 */
                return account.getBalance();
        }
```

```
    /**
     * Switches names on an account.
     */
    public void changeNames(String accountNum, String newName) throws
RemoteException, TellerException {

            /*
             * Find the correct Account Entity Bean
             */
            Account account = findAccountByNumber(accountNum);

            /*
             * Change the Entity Bean's owner name field
             */
            account.setOwnerName(newName);
    }

    /**
     * Transfers funds from accountNum1 to accountNum2
     */
    public void transfer(String accountNum1, String accountNum2, double
funds) throws RemoteException, TellerException {

            /*
             * Withdraw from account #1 into account #2.
             */
            withdraw(accountNum1, funds);
            deposit(accountNum2, funds);
    }

    /**
     * Withdraws from your account.
     *
     * @exception TellerException thrown if you have
     *            insufficient funds.
     * @return Your withdrawn funds.
     */
    public double withdraw(String accountNum, double amt) throws
RemoteException, TellerException {
```

```
                /*
                 * Find the correct Account Entity Bean
                 */
                Account account = findAccountByNumber(accountNum);

                /*
                 * Withdraw from it
                 */
                try {
                        return account.withdraw(amt);
                }
                catch (AccountException e) {
                        e.printStackTrace();
                        throw new TellerException(e);
                }
        }

        /**
         * Deposits into your account.
         */
        public void deposit(String accountNum, double funds) throws
RemoteException, TellerException {

                /*
                 * Find the correct Account Entity Bean
                 */
                Account account = findAccountByNumber(accountNum);

                /*
                 * Deposit into it
                 */
                account.deposit(funds);
        }

        // End business methods
        // Begin EJB-required methods.  The methods below
        // are called by the Container, and never called
        // by client code.
```

```
    public void ejbCreate() throws RemoteException {
        System.out.println("ejbCreate() called.");

        /*
         * Retrieve Home Object for Bank Accounts.
         * If failure, throw a system-level error.
         */
        try {
            findHome();
        }
        catch (Exception e) {
            throw new RemoteException(e.toString());
        }
    }

    public void ejbRemove() {
        System.out.println("ejbRemove() called.");
    }

    public void ejbActivate() {
        System.out.println("ejbActivate() called.");
    }

    public void ejbPassivate() {
        System.out.println("ejbPassivate() called.");
    }

    public void setSessionContext(SessionContext ctx) {
        System.out.println("setSessionContext called");
        this.ctx = ctx;
    }
}
```

사례 연구-3

유람선에 대한 Ticket 예약등을 수행하는 상태정보유지 세션 빈이다.

리모트 인터페이스

✍✍ 고객이 예약하고자 하는 유람선과 객실의 ID를 수집하는 메쏘드와 예약 처리를 하는 bookPassage() 메쏘드등의 인터페이스이다.

<소스> TravelAgent.java

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.FinderException;
import com.titan.cruise.Cruise;
import com.titan.customer.Customer;
import com.titan.processpayment.CreditCard;

public interface TravelAgent extends javax.ejb.EJBObject {

   public void setCruiseID(int cruise)
      throws RemoteException, FinderException;
   public int getCruiseID() throws RemoteException,
                           IncompleteConversationalState;

   public void setCabinID(int cabin)
      throws RemoteException, FinderException;
   public int getCabinID() throws RemoteException,
                          IncompleteConversationalState;
   public int getCustomerID( ) throws RemoteException,
                             IncompleteConversationalState;

   public Ticket bookPassage(CreditCard card, double price)
      throws RemoteException,IncompleteConversationalState;

   public String [] listAvailableCabins(int bedCount)
      throws RemoteException, IncompleteConversationalState;
```

```
}
```

홈 인터페이스

<소스> TravelAgentHome.java

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import com.titan.customer.Customer;

public interface TravelAgentHome extends javax.ejb.EJBHome {

   public TravelAgent create(Customer cust)
      throws RemoteException, CreateException;
}
```

빈 클래스

✍ JNDI 커넥션을 얻고, 여러 정보를 가지고 새로운 Ticket을 발급을 수행하고 고객의 예약을 처리하는 데 트랜잭션 무결성을 제공한다.

<소스> TravelAgentBean.java

```
package com.titan.travelagent;

import com.titan.cabin.*;
import com.titan.cruise.*;
import com.titan.customer.*;
import com.titan.processpayment.*;
import com.titan.reservation.*;

import java.sql.*;
import javax.sql.DataSource;
import java.util.Vector;
import java.rmi.RemoteException;
import javax.naming.NamingException;
import javax.ejb.EJBException;


public class TravelAgentBean implements javax.ejb.SessionBean {
```

```
public Customer customer;
public Cruise cruise;
public Cabin cabin;

public javax.ejb.SessionContext ejbContext;
public javax.naming.Context jndiContext;

public void ejbCreate(Customer cust){
    customer = cust;
}
public int getCustomerID( )
throws IncompleteConversationalState{
    try{
        if(customer == null)
            throw new IncompleteConversationalState();
        return ((CustomerPK)customer.getPrimaryKey()).id;
    }catch(RemoteException re){
        throw new EJBException(re);
    }
}
public int getCruiseID( )
throws IncompleteConversationalState{
    try{
        if(cruise == null)
            throw new IncompleteConversationalState();
        return ((CruisePK)cruise.getPrimaryKey()).id;
    }catch(RemoteException re){
        throw new EJBException(re);
    }

}

public int getCabinID( )
throws IncompleteConversationalState{
    try{
        if(cabin==null)
            throw new IncompleteConversationalState();
        return ((CabinPK)cabin.getPrimaryKey()).id;
```

```
        }catch(RemoteException re){
            throw new EJBException(re);
        }
    }
    public void setCabinID(int cabinID)
    throws javax.ejb.FinderException{
        try{
            CabinHome home =
(CabinHome)getHome("CabinHome",CabinHome.class);
            CabinPK pk = new CabinPK();
            pk.id=cabinID;
            cabin = home.findByPrimaryKey(pk);
        }catch(RemoteException re){
            throw new EJBException(re);
        }
    }
    public void setCruiseID(int cruiseID)
    throws javax.ejb.FinderException{
        try{
            CruiseHome home = (CruiseHome)getHome("CruiseHome",
CruiseHome.class);
            cruise = home.findByPrimaryKey(new CruisePK(cruiseID));
        }catch(RemoteException re){
            throw new EJBException(re);
        }

    }
    public Ticket bookPassage(CreditCard card, double price)
    throws IncompleteConversationalState{


        if(customer == null || cruise == null || cabin == null){
            throw new IncompleteConversationalState();
        }
        try{
            ReservationHome resHome =
                (ReservationHome)
getHome("ReservationHome",ReservationHome.class);
            Reservation reservation =
```

```
            resHome.create(customer, cruise, cabin,price);
            ProcessPaymentHome ppHome =
               (ProcessPaymentHome)
getHome("ProcessPaymentHome",ProcessPaymentHome.class);
            ProcessPayment process = ppHome.create();
            process.byCredit(customer, card, price);

            Ticket ticket = new Ticket(customer,cruise,cabin,price);
            return ticket;
         }catch(Exception e){
            throw new EJBException(e);
         }
      }
      public void ejbRemove(){}
      public void ejbActivate(){}
      public void ejbPassivate(){}

      public void setSessionContext(javax.ejb.SessionContext cntx){
         ejbContext = cntx;
         try{
           jndiContext = new javax.naming.InitialContext();
         }catch(NamingException ne){
            throw new EJBException(ne);
         }
      }
      protected Object getHome(String name,Class type){
         try{
            Object ref = jndiContext.lookup("java:comp/env/ejb/"+name);
            return javax.rmi.PortableRemoteObject.narrow(ref, type);
         }catch(NamingException ne){
            throw new EJBException(ne);
         }
      }
      private Connection getConnection() throws SQLException{
         try{
           DataSource ds =
(DataSource)jndiContext.lookup("java:comp/env/jdbc/titanDB");
            return ds.getConnection( );
         }catch(NamingException ne){throw new EJBException(ne);}
```

```
}
public String [] listAvailableCabins(int bedCount)
throws IncompleteConversationalState{

   if(cruise == null) throw new IncompleteConversationalState();



   Connection con = null;
   PreparedStatement ps = null;;
   ResultSet result = null;
   try {
      int cruiseID = ((CruisePK)cruise.getPrimaryKey()).id;
      int shipID = cruise.getShipID();
      con = getConnection();
      ps = con.prepareStatement(
         "select ID, NAME, DECK_LEVEL  from CABIN "+
         "where SHIP_ID = ? and ID NOT IN "+
         "(SELECT CABIN_ID FROM RESERVATION WHERE CRUISE_ID = ?)");

      ps.setInt(1,shipID);
      ps.setInt(2,cruiseID);
      result = ps.executeQuery();
      Vector vect = new Vector();
      while(result.next()){
         StringBuffer buf = new StringBuffer();
         buf.append(result.getString(1));
         buf.append(',');
         buf.append(result.getString(2));
         buf.append(',');
         buf.append(result.getString(3));
         vect.addElement(buf.toString());
      }
      String [] returnArray = new String[vect.size()];
      vect.copyInto(returnArray);
      return returnArray;
   }
   catch (Exception e) {
      throw new EJBException(e);
   }
```

```
            finally {
                try {
                    if (result != null) result.close();
                    if (ps != null) ps.close();
                    if (con!= null) con.close();
                }catch(SQLException se){se.printStackTrace();}
            }
        }
    }
```

사례 연구-4

쇼핑몰의 구현에서 각 상품에 대한 정보 즉, 상품 번호, 수량, 가격, 할인정보 등의 정보를 보여주는 상태정보유지 세션 빈이다.

리모트 인터페이스

<소스> QuoteLineItem.java

```
package com.wiley.compBooks.ecommerce;

import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * These are the business logic methods exposed publicly
 * by QuoteLineItemBean.
 *
 * This interface is what clients operate on when they
 * interact with beans. The container vendor will implement
 * this interface; the implementation object is called the
 * EJB Object, which delegates invocations to the actual
 * bean.
 */

public interface QuoteLineItem extends EJBObject {

    // Getter/setter methods for Entity Bean fields

    public Product getProduct() throws RemoteException;

    public double getBasePrice() throws RemoteException;

    public double getDiscount() throws RemoteException;
    public void setDiscount(double discount) throws RemoteException;

    public int getQuantity() throws RemoteException;
    public void setQuantity(int quantity) throws RemoteException;
}
```

홈 인터페이스

<소스> QuoteLineItemHome.java

```
package com.wiley.compBooks.ecommerce;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.Enumeration;

/**
 * This is the home interface for QuoteLineItem.  This interface
 * is implemented by the EJB Server's glue-code tools - the
 * implemented object is called the Home Object, and serves as a
 * factory for EJB Objects.
 *
 * One create() method is in this Home Interface, which
 * corresponds to the ejbCreate() method in the QuoteLineItem
 * file.
 */
public interface QuoteLineItemHome extends EJBHome {

    /*
     * This method creates the EJB Object.
     *
     * Notice that the Home Interface returns an EJB Object,
     * whereas  the Bean returns void.  This is because the
     * EJB Container is responsible for generating the
     * EJB Object, whereas the Bean is responsible for
     * initialization.
     *
     * @param product Product Entity Bean for this line item
     *
     * @return The newly created EJB Object.
     */
    public QuoteLineItem create(Product product) throws CreateException,
RemoteException;
}
```

빈 클래스

<소스> QuoteLineItemBean.java

```
package com.wiley.compBooks.ecommerce;

import java.sql.*;
import javax.naming.*;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;
import javax.naming.*;

/**
 *
 * This stateful session bean represents an individual
 * line-item, which is part of a quote that a customer is
 * working on.  A quote is a set of products that the
 * customer is interested in, but has not committed to buying
 * yet.  You can think of this bean as a shopping cart - as the
 * customer surfs our E-Commerce web site, the quote stores all
 * the products and quantities he is interested in.
 *
 * A quote consists of a series of line-items.  Each line item
 * represents one particular product the customer wants, as
 * well as a quantity for that product.  This bean models a
 * line-item.
 *
 * The distinction between a quote and an order is that a quote
 * is only temporary and in-memory (hence a stateful session
 * bean), whereas an order is a persistent record (hence an
 * entity bean).  This quote bean is smart enough to know how
 * to transform itself into an order (via the purchase() method).
 */
public class QuoteLineItemBean implements SessionBean {

    protected SessionContext ctx;
```

```
    // Begin Stateful Session Bean Conversational State fields
    // (Conversational state is non-transient, and is
    // perserved during passivation and activation)
    /*
     * The product that this line-item represents.
     */
    private Product product;

    /*
     * The quantity of the product desired
     */
    private int quantity;

    /*
     * Any discount the customer gets
     */
    private double discount;

    // End Stateful Session Bean Conversational State fields

    public QuoteLineItemBean() {
            System.out.println("New QuoteLineItem Session Bean created by
EJB Container.");
    }

    // Begin public business methods
    // Simple getter/setter methods of Session Bean fields.

    public Product getProduct() throws RemoteException {
            System.out.println("QuoteLineItem.getProduct() called.");
            return product;
    }

    /**
     * Returns the base price.  The base price is the
     * product's price times the quantity ordered.  This
     * figure does not take discounts into consideration.
     */
      public double getBasePrice() throws RemoteException {
```

```
            System.out.println("QuoteLineItem.getBasePrice() called.");
            return quantity * product.getBasePrice();
    }

    /**
     * Returns the discount that the customer gets on this
     * order.
     *
     * Note: The discount is a whole number, not a percentage
     *       discount.
     */
    public double getDiscount() throws RemoteException {
            System.out.println("QuoteLineItem.getDiscount() called.");
            return discount;
    }

    /**
     * Sets the discount that the customer gets on this order.
     *
     * Note: The discount is a whole number, not a percentage
     *       discount.
     */
      public void setDiscount(double discount) throws RemoteException {
            System.out.println("QuoteLineItem.setDiscount() called.");
            this.discount = discount;
    }

    public int getQuantity() throws RemoteException {
            System.out.println("QuoteLineItem.getQuantity() called.");
            return quantity;
    }

    public void setQuantity(int quantity) throws RemoteException {
            System.out.println("QuoteLineItem.setQuantity() called.");
            this.quantity = quantity;
    }

    // End public business methods
    // Begin EJB-required methods.  The methods below are
```

```
        // called by the Container, and never called by client
        // code.

        /**
         * Associates this Bean instance with a particular context.
         */
        public void setSessionContext(SessionContext ctx) throws RemoteException
    {
                System.out.println("QuoteLineItem.setSessionContext called");
                this.ctx = ctx;
        }

        /**
         * The container calls this method directly after
         * activating this instance.  You should acquire any
         * needed resources.
         */
        public void ejbActivate() throws RemoteException {
                System.out.println("QuoteLineItem.ejbActivate() called.");
        }

        /**
         * The container calls this method directly before
         * passivating this instance.  You should release
         * resources acquired during ejbActivate().
         */
        public void ejbPassivate() throws RemoteException {
                System.out.println("QuoteLineItem.ejbPassivate () called.");
        }

        /**
         * This is the initialization method that corresponds
         * to the create() method in the Home Interface.
         *
         * When the client calls the Home Object's create()
         * method, the Home Object then calls this ejbCreate()
         * method.
         */
        public void ejbCreate(Product product) throws CreateException,
```

```
RemoteException {
            System.out.println("QuoteLineItem.ejbCreate(" +
product.getName() + ") called");

            this.product = product;
            this.quantity = 1;
            this.discount = 0;
    }

    /**
     * Removes this quote line-item, and hence all
     * client-specific state.
     */
    public void ejbRemove() throws RemoteException {
            System.out.println("QuoteLineItem.ejbRemove() called.");
    }
}
```