<JSTORM>

Enterprise Java Beans 3부

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

Document Information

Document title:	Enterprise Java Beans 3부
Document file name:	EJB1_JSTORM.doc
Revision number:	<0.9>
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	이번 장에서는 Entity Bean과 Session 빈에 대해서 살펴본다. 또한 각 빈즈 프로그래밍을 해봄으로써 EJB 프로그램의 이해를 높여보자.
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

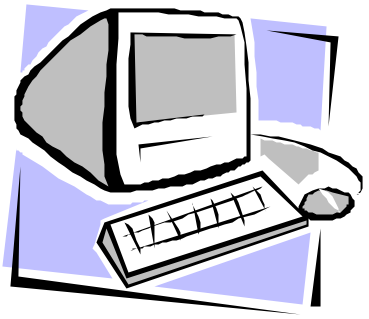
Revision	Date	Author	Description of change

Table of Contents

3 부

Entity Bean, Session Bean과 Client API

이번 장에서는 Entity Bean과 Session 빈에 대해서 살펴본다. 또한 각 빈즈 프로그래밍을 해봄으로써 EJB 프로그램의 이해를 높여보자.



Chapter 7

엔티티 빈 개발

엔티티 빈에 대한 여러가지 정의

- 엔티티 빈은 저장장치(보통 데이터베이스)에 저장되어 있는 하나의 entity를 나타낸다.
- 엔티티 빈은 명사로 표현될 수 있는 비즈니스적인 개념을 형상화한다. 즉, 현실 세계의 상태와 동작을 표현하고 일반적으로 데이터베이스상의 하나의 데이터 항목과 대응된다.
- 세션 빈이 EJB를 통해 구현하고자 하는 비즈니스 로직을 담고 있다면, 엔티티 빈은 이 비즈니스 로직의 구현에 필요한 각종 데이터들과 대응하는 데이터베이스상의 지속적으로 유지되어야 하는 정보(persistent information) 들과의 연동을 담담하고 있다.
- 데이터베이스를 직접 접근하지 않고 엔티티 빈을 통해 접근함으로써 개발자는 데이터베이스에 대한 좀 더 추상화되고 단순한 접근을 할 수 있게된다. 또한 이를 통해 이와 유사한 다른 데이터에 대한 일관성을 보장할 수 있게 된다.

엔티티 빈과 세션 빈과의 차이점

Persistence(영속성)

- 엔티티 빈의 상태가 저장장치(보통 데이터베이스)에 저장되기 때문에 엔티티 빈은 영속적이다. 여기서 영속적이라는 것은 엔티티 빈이 어플리케이션의 생명주기나 J2EE 서버 프로세스와 관계없이 계속 존재하는 것을 의미한다.
- 엔티티 빈은 Bean-managed와 Container-managed의 2가지 Persistence 형태가 존재하고 개발자가 이들 형태를 Application Deployment Tool에서 선언하고, 그 정보를 배치 디스크립터에 저장하는데 이는 뒤에서 살펴본다.

Shared Access(공유 접근)

- 엔티티 빈은 여러 개의 클라이언트로부터 동시에 접근될 수 있다. 이때 클라이

언트들은 동일한 데이터에 대한 수정작업을 시도할 수 있으므로 엔티티 빈이 트랜잭션 안에서 수행되는 것은 중요하다. 일반적으로 EJB 컨테이너는 트랜잭션 Management를 제공한다.

✍ 개발자는 빈의 배치 디스크립터에서 트랜잭션 속성을 지정하고 트랜잭션 한계(Transaction boundary)는 지정할 필요가 없다. 왜냐하면 컨테이너가 개발자를 위해 이를 표시하기 때문이다.

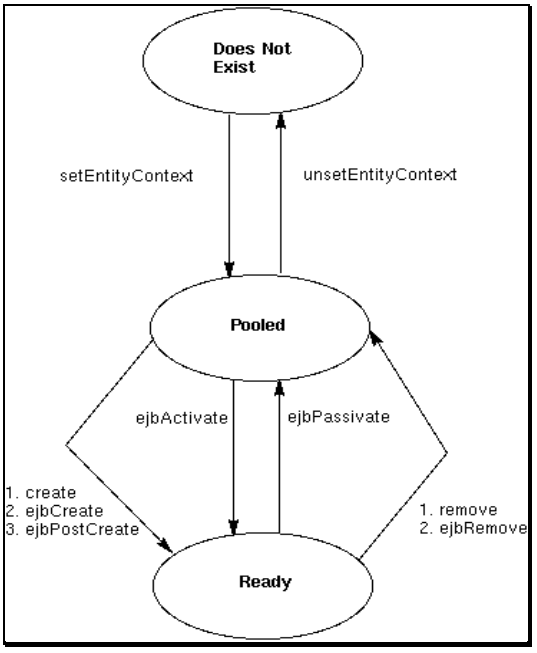
Primary Key

✍✍각각의 엔티티 빈은 유일한 객체 구분자(Object Identifier)를 가지고 있다. 이것은 데이터베이스의 테이블에서 각 필드를 구별해주는 주키와 동일한 것으로 클라이언트가 특정한 엔티티 빈을 이용하는 것을 가능하게 해준다.

엔티티 빈의 Life cycle

✍ 엔티티 빈의 life cycle은 응용프로그램에 의해 결정되는 것이 아니라 EJB 컨테이너에 의해 결정된다. 그러나, 엔티티 빈의 life cycle을 알아두는 것은 중요하다. 그럼으로써 엔티티 빈의 어떤 method가 데이터베이스와 연결하게 될 것인지를 결정할 때 유용하게 적용할 수 있게 되기 때문이다.

✍ 다음 그림은 life cycle동안 엔티티 빈의 상태가 변화하는 것을 보여준다. EJB 컨테이너는 엔티티 빈의 인스턴스(EJB 빈 인스턴스)를 생성하고 나서 엔티티 빈 클래스에 있는 setEntityContext method를 호출한다. SetEntityContext method는 entity context 를 빈에 전달한다.



<그림> 엔티티 빈의 Life Cycle

✍ 인스턴스화된 이후, 엔티티 빈은 가용 인스턴스 풀(a pool of available instances)에 들어가게 된다. 인스턴스는 풀된 상태동안에는 어떤 EJB 객체와도 연관되어 있지 않는다. 풀에 있는 모든 인스턴스들은 다 동일하다. EJB 컨테이너가 인

스턴스를 ready state로 이동시키면 비로서 인스턴스는 EJB 객체와 연동되게 되는 것이다.

✍✍ 인스턴스가 Ready state에서 Pooled state로 변하는 경우는 두가지가 존재하는데, 첫 번째는 클라이언트가 remove method를 호출하는 경우로 이때 EJB 컨테이너는 ejbRemove method를 호출하게 된다. 두 번째는 EJB 컨테이너가 ejbPassivate method를 호출하는 경우이다.

✍✍ Life cycle의 마지막은 EJB 컨테이너가 풀에서 인스턴스를 제거하는 것으로 unsetEntityContext method를 호출하게 된다.

✍✍ Pooled state에서 인스턴스는 어떤 EJB 객체와도 연동되지 않는다. BMP(Bean-Managed Persistence) 기반에서 EJB 컨테이너가 인스턴스를 pooled state에서 ready state로 이동시킬 때, 인스턴스는 primary key를 자동으로 설정하지 않는다. 따라서, ejbCreate와 ejbActivate method들은 반드시 primary key를 설정해 주어야 한다. 만일 primary key가 틀릴 경우에는 ejbLoad와 ejbStore method가 인스턴스 변수들을 데이터베이스와 동기화 시킬 수 없게된다. 뒤에서 살펴볼 AccountEJB 예제에서 보면, ejbCreate method가 하나의 입력 전달인자로부터 primary key를 할당하고 있는 것을 볼 수 있다. ejbActivate method는 다음과 같이 primary key를 설정한다.

```
Id = (String)context.getPrimaryKey();
```

✍✍ Pooled state에서 인스턴스 변수들의 값들은 불필요하다. 따라서 이런 인스턴스 변수들을 ejbPassivate method에서 null로 설정함으로써 garbage collection에 적절하도록 만들 수 있다.

엔티티 빈 사이의 비교

✍✍ 클라이언트는 isIdentical method를 통해서 두개의 엔티티 빈에 대한 객체 레퍼런스(Object reference)가 동일한지 비교할 수 있다.

```
Account accta, acctb;
…
if (accta.isIdentical(acctb))
System.out.println(“identical”);
```

✍✍ 두 엔티티 빈의 primary key를 비교하여 구분할 수도 있다.

```
String key1 = (String)accta.getPrimaryKey();
String key2 = (String)acctb.getPrimaryKey();
If (key1.compareTo(key2) == 0)
System.out.println(“equal”);
```

엔티티 빈의 객체 레퍼런스(Object reference) 전달하기

✍ 하나의 엔티티 빈을 다른 빈에 전달할려면 어떻게 해야 할까? 자바에서 처럼 this 참조를 넘기는 것은 허용되지 않는다. 대신, 엔티티 빈의 객체 레퍼런스(Object reference)를 넘겨야 한다.

✍ 다음 코드는 엔티티 빈의 EntityContext에 있는 getEJBObject method를 호출하여 객체 레퍼런스(Object reference)를 얻어오는 과정을 보여주고 있다.

```
public void setEntityContext(EntityContext ec)
{
   // save the entity context in an instance variable
   this.context = ec;
   }
   . . .
   public void passItOn(Inventory tally)
   {
   // pass the object reference
   tally.copyItems(context.getEJBObject());
}
```

Sample 엔티티 빈 개발

여기서는 간단한 은행 계좌(account)를 나타내는 예제를 살펴본다.

엔티티 빈은 Bean-managed Persistence이다.

엔티티 빈의 상태는 관계형 데이터베이스의 ACCOUNT 테이블에 저장된다.

개발 과정

✍ ACCOUNT 테이블 생성
✍ 리모트 인터페이스(Account.java) 코드 작성
✍ 홈 인터페이스(AccountHome.java) 코드 작성
✍ 엔티티 빈 클래스(AccountEJB.java) 코드 작성
✍ InsufficientBalanceException : Helper 클래스 작성
✍ Deployment Descriptor 작성
✍ Application Deployment(어플리케이션 배치)
✍ AccountClient.java : Client program

ACCOUNT 테이블 생성

✍✍ 다음의 SQL 구문에 의해 생성된다.

```
CREATE TABLE ACCOUNT
    (id      VARCHAR(3) CONSTRAINT pk_account PRIMARY KEY,
    firstname       VARCHAR(24),
    lastname        VARCHAR(24),
    balance         DECIMAL(10, 2));
```

리모트 인터페이스 구현

✍✍ 리모트 인터페이스는 javax.ejb.EJBObject를 상속하고 클라이언트가 호출할 비즈니스 method를 정의한다.

✍✍ Account 리모트 인터페이스는 다음과 같은 비즈니스 method들을 정의하고 있다.

- 인출

```
public void debit(double amount)
        throws InsufficientBalanceException, RemoteException;
```

- 예금

```
public void credit(double amount) throws RemoteException;
```

- FirstName 반환

```
public String getFirstName() throws RemoteException;
```

- LastName 반환

```
public String getLastName() throws RemoteException;
```

- 잔고 확인

```
public double getBalance() throws RemoteException;
```

✍✍ 다음은 Account 리모트 인터페이스이다.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface Account extends EJBObject {
    public void debit(double amount)
            throws InsufficientBalanceException, RemoteException;
    public void credit(double amount) throws RemoteException;
    public String getFirstName() throws RemoteException;
    public String getLastName() throws RemoteException;
    public double getBalance() throws RemoteException;
}
```

<소스> Account.java

✍✍ 리모트 인터페이스의 method에 대한 요구사항은 세션 빈이나 엔티티 빈의 요구사항과 동일하다.

- 리모트 인터페이스의 각각의 method는 엔티티 빈 클래스의 method와 대응해야 한다.

- 리모트 인터페이스의 method의 시그너쳐(signatures)는 엔티티 빈 클래스의 대응되는 method의 시그너쳐와 동일해야 한다.

- 전달인자와 반환값은 반드시 유효한 Java RMI 형이어야 한다.

- throws 절은 반드시 java.rmi.RemoteException을 포함해야 한다.

홈 인터페이스 구현

✍✍ 홈 인터페이스는 클라이언트가 엔티티 빈 객체를 생성하고 그것을 찾도록 해주는 method들을 정의한다. 즉, 홈 인터페이스가 엔티티 빈 객체의 life cycle을 담당하고 있다고 할 수 있다.

✍✍ 다음은 AccountHome interface이다.

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

public interface AccountHome extends EJBHome
{
public Account create(String id, String firstName,String lastName)
        throws RemoteException, CreateException;
public Account findByPrimaryKey(String id)
        throws FinderException, RemoteException;
   public Collection findByLastName(String lastName)
        throws FinderException, RemoteException;
   public Collection findInRange(double low, double high)
        throws FinderException, RemoteException;
}
```

<소스> AccountHome.java

✍✍ 홈 인터페이스의 create method는 반드시 다음 요구사항을 만족해야 한다.

- 대응되는 엔티티 빈 클래스에 위치한 ejbCreate method와 동일한 수와 형태의 전달인자를 가져야 한다.

- 엔티티 빈 객체의 리모트 인터페이스를 반환해야 한다.

- throws절은 대응하는 ejbCreate와 ejbPostCreate method의 throws절에 설정된 exception들을 포함해야 한다.

✍✍ 홈 인터페이스에 있는 모든 finder method는 엔티티 빈 클래스에 있는 finder method와 대응된다.

✍✍ 홈 인터페이스에 정의된 finder method의 이름은 find로 시작되는 반면에 엔티티 빈 클래스에 있는 것들의 이름은 ejbFind로 시작된다. 예를 들면, AccountHome 클래스는 findByLastName method를 선언하고 있고, AccountEJB 클래스는 ejbFindByLastName method를 구현하고 있다.

✍✍ 홈 인터페이스의 finder method의 시그너쳐들(signatures)을 정의하는 규칙들은 다음과 같다.

- 전달인자의 개수와 형은 엔티티 빈 클래스에 있는 대응되는 method의 것들과 동일해야 한다.

- 반환값은 엔티티 빈의 리모트 인터페이스 형이거나 그런 형들의 집합(Enumeration type)이다.

- throws 절의 exception들은 엔티티 빈 클래스의 대응되는 method에 나타나 있는 것을 포함해야 한다.

- throws 절은 javax.ejb.FinderException과 javax.ejb.Remote Exception을 담고 있다.

엔티티 빈 클래스 구현

✍✍ AccountEJB라 불리는 간단한 엔티티 빈 클래스이다. 제공되는 코드는 모든 엔티티 빈의 요구사항과 일치한다.

- EntityBean Interface를 구현한다.
- 클래스는 public으로 선언되어 있다.
- 클래스는 abstract나 final로 선언될 수 없다.
- 0개 이상의 ejbCreate와 ejbPostCreate method를 구현한다.
- finder method를 구현한다.(bean-managed persistence일 경우에만)
- 비즈니스 method를 구현한다.
- empty constructor를 가지고 있다.
- finalize method를 구현하지 않는다.

✍✍ EntityBean Interface

- EntityBean interface는 Serializable interface를 상속(extends)한 EnterpriseBean interface를 상속한다.

- EntityBean interface는 ejbActivate나 ejbLoad와 같은 몇 개의 method를 선언하는데, 이것은 엔티티 빈 클래스에서 반드시 구현되어야 한다.

✍✍ ejbCreate method

클라이언트가 create method를 호출할 때, EJB 컨테이너는 대응되는 ejbCreate method를 호출한다. 전형적으로, 엔티티 빈 클래스에 있는 ejbCreate method는 다음과 같은 일을 수행한다.

- 데이터베이스에 엔티티 상태를 삽입한다.
- 인스턴스 변수를 초기화 한다.

- Primary Key 를 반환한다.

AccountEJB의 ejbCreate method는 SQL insert 구문을 수행하는 private insertRow method를 호출함으로써 엔티티 상태를 데이터베이스에 저장한다. 다음은 AccountEJB 클래스의 ejbCreate method의 소스코드이다.

```
public String ejbCreate(String id, String firstName, String
lastName,double balance) throws CreateException {
   if (balance < 0.00) {
   throw new CreateException ("A negative initial
   balance is not allowed.");
    }
   try {
        insertRow(id, firstName, lastName, balance);
    }
   catch (Exception ex) {
        throw new EJBException("ejbCreate: "
                                    + ex.getMessage());
    }

   this.id = id;
   this.firstName = firstName;
   this.lastName = lastName;
   this.balance = balance;
   return id;
}
```

AccountEJB 클래스는 단 하나의 ejbCreate method를 가지고 있지만, 일반적으로 EJB 빈은 여러 개의 ejbCreate method를 가지고 있을 수 있다.

엔티티 빈에 대한 ejbCreate method를 작성할 때는 다음의 규칙들을 적용하라.

- access control modifier는 반드시 public이어야 한다.
- 반환형은 반드시 Primary Key 이어야 한다.(bean-managed persistence일 경우에만)
- 전달인자는 반드시 Java RMI에 적합한 형이어야 한다.
- Method modifier는 final이나 static이 될 수 없다.
- throws절은 javax.ejb.CreateException과 여러분의 어플리케이션에 적합한 다른 Exception들을 포함할 수 있다.
- ejbCreate method는 보통 입력 인자가 유효하지 않을 경우 CreateException을 발생시키고, 만일 이미 존재하는 동일한 Primary Key를 가지는 다른 엔티티로 인해 ejbCreate method가 엔티티를 생성할 수 없으면 그것은 javax.ejb.DuplicateKeyException(CreateException의 서브클래스)를 발생시킨다. 만일 클라이언트가 CreateException이나 DuplicateKeyException을 수신하면 엔티티가 생성되지 않은 것으로 간주한다.

✍✍ 엔티티 빈의 상태는 non-J2EE 어플리케이션에 의해 직접적으로 데이터베이스에 삽입된다. 예를 들면, SQL 스크립트가 ACCOUNT 테이블에 열을 삽입할 수 있다. 비록 이 열에 대한 엔티티 빈이 ejbCreate method에 의해 생성되지 않았다고 해도, 이 빈에 클라이언트가 위치할 수 있다.

✍✍ ejbPostCreate method

- 각각의 ejbCreate method에 대해, 엔티티 빈 클래스에 ejbPostCreate method를 작성해 주어야 한다.

- EJB 컨테이너는 ejbCreate method를 호출한 바로 직후에 ejbPostCreate method를 호출한다. ejbCreate method와는 다르게, ejbPostCreate method는 EntityContext interface의 getPrimaryKey와 getEJBObject method를 호출할 수 있다.

- 일반적으로 ejbPostCreate method는 empty method이다.

- ejbPostCreate method의 signature는 다음의 요구사항을 만족해야 한다.

  ✍✍ 전달인자의 개수와 형태는 대응되는 ejbCreate method와 동일해야 한다.
  ✍✍ access control modifier는 반드시 public이어야 한다.
  ✍✍ method modifier는 final이나 static이 되어서는 안된다.
  ✍✍ 반환형은 반드시 void형이어야 한다.

- throws절은 javax.ejb.CreateException과 어플리케이션에 적합한 다른 Exception들을 포함할 수 있다.

✍✍ ejbRemove method

- 클라이언트는 remove method를 호출하여 엔티티 빈을 삭제한다.

- 이 호출은 EJB 클라이언트로 하여금 ejbRemove method를 호출하게 하고, ejbRemove method는 데이터베이스로부터 엔티티 상태를 제거한다.

- AccountEJB 클래스의 ejbRemove method에 대한 코드는 다음과 같다.

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

- 만일 ejbRemove method가 시스템 문제에 부딪히면, 그것은 javax.ejb.EJB-Exception을 발생시킨다. 만일 그것이 어플리케이션 에러에 부딪히면, javax.ejb.RemoveException을 발생시킨다.

- 또한 엔티티 빈은 데이터베이스 삭제에 의해 직접적으로 삭제될 수 있다. 예를 들면, 만일 SQL 스크립트가 엔티티 빈 state를 저장하고 있는 열(row)을 삭제하면, 그 엔티티 빈은 삭제되는 것이다.

✍✍ ejbLoad method와 ejbStore method

- 만일 EJB 컨테이너가 엔티티 빈의 인스턴스 변수와 데이터베이스에 저장되어 있는 대응되는 값을 동기화(synchronize)할 필요가 있다면, EJB 컨테이너는 ejbLoad method와 ejbStore method를 호출한다.

- ejbLoad method는 인스턴스 변수를 데이터베이스에서 재설정(refresh)하고, ejbStore method는 인스턴스 변수의 값을 데이터베이스에 쓴다.

- 클라이언트는 ejbLoad나 ejbStore method를 호출하지 않는다.

- 만일 비즈니스 method가 트랜잭션과 연계되어 있다면, 컨테이너는 ejbLoad method를 비즈니스 method가 실행되기 전에 호출한다. 바로 다음에 비즈니스 method가 실행되면, 컨테이너는 ejbStore method를 호출한다.

- 컨테이너가 ejbLoad와 ejbStore를 호출하기 때문에 개발자는 비즈니스 method에서 인스턴스 변수를 재설정(refresh)하거나 저장할 필요가 없다.(컨테이너가 개발자를 위해 이러한 일들을 수행한다.)

- AccountEJB 클래스는 인스턴스 변수를 데이터베이스와 동기화 시키는 것을 컨테이너에 의존한다. 그러므로 AccountEJB의 비즈니스 method는 트랜잭션과 연계되어 있다.

- 만일 ejbLoad와 ejbStore method가 entity 를 기초(underlying) 데이터베이스에 위치시키지 못하게 되면, javax.ejb.NoSuchEntityException 예외를 발생시킨다.

- 이 예외는 EJBException의 서브클래스이다. EJBException이 Runtime-Exception의 서브클래스이기 때문에 이것을 throws 절에 포함시킬 필요가 없다. NoSuchEntityException 예외가 발생할 때, EJB 컨테이너는 그것을 클라이언트에 반환하기 전에 RemoteException안에 포함시킨다.

- AccountEJB 클래스에서 ejbLoad는 loadRow method를 호출하는데, 이것은 SQL select 구문을 수행하고 추출된 데이터를 인스턴스 변수에 할당한다. ejbStore method는 storeRow method를 호출하는데, 이것은 인스턴스 변수를 SQL update 구문을 통해 데이터베이스에 저장한다. 다음은 ejbLoad와 ejbStore method에 대한 코드이다.

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

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

✍✍ finder method

finder method는 클라이언트들이 엔티티 빈에 위치할 수 있게 해준다. AccountClient 프로그램은 3개의 finder method들과 함께 엔티티 빈에 위치한다.

```
Account jones = home.findByPrimaryKey("836");
…
Collection c = home.findByLastName("Smith");
…
Collection c = home.findInRange(20.00, 99.00);
```

클라이언트에게 사용가능한 모든 finder method들에 대해 엔티티 빈 class는 반드시 접두사가 ejbFind로 시작되는 대응되는 method를 구현해야 한다. 예로, AccountEJB 엔티티 빈 class는 다음과 같은 ejbFindByLastName method를 구현하고 있다.

```
public Collection ejbFindByLastName(String lastName)
throws FinderException
{
   Collection result;
   try {
    result = selectByLastName(lastName);
   } catch (Exception ex) {
    throw new EJBException("ejbFindByLastName "
   + ex.getMessage());
   }
   if (result.isEmpty()) {
    throw new ObjectNotFoundException ("No rows found.");
   }else {return result;}
}
```

finder method는 ejbFindByLastName과 ejbFindInRange 처럼 일반적으로 특정 어플리케이션에 국한되어 있기 때문에 선택적이라고 할 수 있다. 그러나 ejbFindByPrimaryKey method는 필수사항이다. 이름이 의미하는 것처럼, ejbFindByPrimaryKey method는 전달인자로서 Primary Key를 받아들이고, 이것은 엔티티 빈에 위치하기 위해 사용된다. AccountEJB 클래스에서 Primary Key는 id 변수이다. 다음은 ejbFindByPrimaryKey method이다.

```
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 id " + primaryKey + " not
   found.");
   }
}
```

- ejbFindByPrimaryKey method가 primaryKey를 method의 전달인자와 동시에 반환값으로 사용하기 때문에 약간 생소하게 보일것이다. 그러나, 클라이언트는 ejbFindByPrimaryKey를 직접적으로 호출하지 않으며, ejbFindByPrimaryKey method는 EJB 컨테이너에 의해 호출된다는 점을 알아야 한다.

- 클라이언트는 홈 인터페이스에 정의된 findByPrimaryKey method를 호출한다.

- 다음은 bean-managed persistence인 엔티티 빈에서 구현하고 있는 finder method를 위한 규칙의 요약이다.

  - ejbFindByPrimaryKey method는 반드시 구현되어야 한다.
  - finder method 이름은 반드시 ejbFind 접두사로 시작되어야 한다.
  - access control modifier는 반드시 public 이어야 한다.
  - method modifier는 final이나 static 이어서는 안된다.
  - 인자와 반환값은 반드시 Java RMI에 적합한 형이어야 한다.
  - 반환값은 반드시 Primary Key 나 Primary Key들의 집합이어야 한다.

- Business method

비즈니스 method는 엔티티 빈에 캡슐화하기를 원하는 비즈니스 로직을 담고 있

다. 보통, 비즈니스 method는 데이터베이스를 제어하지 않고, 개발자로 하여금 비즈니스 로직을 데이터베이스 제어 코드로부터 분리되도록 한다. AccountEJB 엔티티 빈은 이런 비즈니스 method를 담고 있다.

```
public void debit(double amount)  throws InsufficientBalanceException
{
    if (balance - amount < 0) {
            throw new InsufficientBalanceException();
    }
    balance -= amount;
}

public void credit(double amount)
{   balance += amount;}

public String getFirstName()
{   return firstName;}

public String getLastName()
{   return lastName;}

public double getBalance()
{   return balance; }
```

- AccountClient 프로그램은 다음과 같이 비즈니스 method를 호출한다.

```
Account duke = home.create("123", "Duke", "Earl");
duke.credit(88.50);
duke.debit(20.25);
double balance = duke.getBalance();
```

- 비즈니스 method의 시그너쳐(signature)에 대한 요구사항은 세션 빈과 엔티티 빈의 것과 동일하다.

  - method 이름은 EJB 아키텍쳐에 선언된 이름과 대치되어서는 안된다. 예를 들면, ejbCreate나 ejbActivate와 같은 비즈니스 method를 호출할 수 없다.
  - access control modifier는 반드시 public 이어야 한다.
  - method modifier는 final이나 static이 될 수 없다.
  - 전달인자와 반환형은 반드시 자바 RMI에 적합한 형이어야 한다.

- throws 절은 현재 응용프로그램을 위해 새로 정의된 exception을 포함하고 있다. 예를들면, debit method는 InsufficientBalanceException을 사용하는 것을 알 수 있다. 시스템 차원의 예외 핸들링을 위해서 비즈니스 method는 javax.ejb.EJBException을 포함해야 한다.

✍✍ Database Calls

다음 테이블은 AccountEJB 클래스의 데이터베이스 제어 함수들을 요약한 것이다.

Method	결과 SQL 구문
Ate	Insert
ejbFindByPrimaryKey	select
ejbFindByLastName	select
ejbFindInRange	select
EjbLoad	select
EjbStore	update
EjbRemove	delete

<표> AccountEJB의 SQL 구문

AccountEJB의 비즈니스 method들은 위 테이블에 나타나 있지 않다. 이유는 이 method들이 데이터베이스에 접근하지 않기 때문이다. 대신, 이런 method들은 EJB 컨테이너가 ejbStore method를 호출할 때 데이터베이스에 쓰여지게 될 인스턴스 변수들을 바꾼다.

어떤 개발자는 비즈니스 method에서 데이터베이스를 제어하도록 프로그램을 작성할 수도 있다. 이것은 그 응용프로그램의 필요에 따라 변화하는 디자인상의 결정이다.

데이터베이스에 접근하기 위해서는 당연하겠지만, 데이터베이스에 연결되어 있어야 한다. 연결방법은 자료를 참고하라.

다음은 엔티티 빈 클래스의 소스 리스트이다.

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

public class AccountEJB implements EntityBean
{
    private String id;
    private String firstName;
    private String lastName;
    private double balance;
```

```
    private EntityContext context;
    private Connection con;
    private String dbName = "java:comp/env/jdbc/AccountDB";

    public void debit(double amount)
      throws InsufficientBalanceException
{
      if (balance - amount < 0) {
         throw new InsufficientBalanceException();
      }
      balance -= amount;
}

public void credit(double amount)
{
      balance += amount;
}

public String getFirstName()
{
         return firstName;
}

public String getLastName()
{
         return lastName;
}

public double getBalance()
{
return balance;
}

public String ejbCreate(String id, String firstName, String lastName,
double balance) throws CreateException
{
      if (balance < 0.00) {
        throw new CreateException
          ("A negative initial balance is not allowed.");
```

```
        }
        try {
          insertRow(id, firstName, lastName, balance);
        } catch (Exception ex) {
           throw new EJBException("ejbCreate: "
+ ex.getMessage());
        }
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.balance = balance;

        return id;
}

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 id " + primaryKey + " not found.");
     }
}

public Collection ejbFindByLastName(String lastName)
throws FinderException
{
     Collection result;
```

```
        try {
          result = selectByLastName(lastName);
        } catch (Exception ex) {
            throw new EJBException("ejbFindByLastName "
+ ex.getMessage());
        }

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

    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;
        }
    }

    public void ejbRemove()
    {
        try {
          deleteRow(id);
```

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

public void setEntityContext(EntityContext context)
{
    this.context = context;
    try {
      makeConnection();
    } catch (Exception ex) {
        throw new EJBException("Unable to connect to database. "
+ ex.getMessage());
    }
}

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

public void ejbActivate()
{
    id = (String)context.getPrimaryKey();
}

public void ejbPassivate()
{
    id = null;
}

public void ejbLoad()
{
```

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

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


public void ejbPostCreate(String id, String firstName,
      String lastName, double balance) { }

/********************** 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 id, String firstName, String lastName,
double balance) throws SQLException
{
      String insertStatement =
         "insert into account values ( ? , ? , ? , ? )";
      PreparedStatement prepStmt =
         con.prepareStatement(insertStatement);
```

```
        prepStmt.setString(1, id);
        prepStmt.setString(2, firstName);
        prepStmt.setString(3, lastName);
        prepStmt.setDouble(4, balance);

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

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

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

    private boolean selectByPrimaryKey(String primaryKey)
    throws SQLException
    {
        String selectStatement =
            "select id " +
            "from account where id = ? ";
        PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);
        prepStmt.setString(1, primaryKey);

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

    private Collection selectByLastName(String lastName)
```

```
        throws SQLException
    {
        String selectStatement =
            "select id " +
            "from account where lastname = ? ";
        PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);

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

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

        prepStmt.close();
        return a;
    }

    private Collection selectInRange(double low, double high)
        throws SQLException
    {
        String selectStatement =
            "select id from account " +
            "where balance 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 id = rs.getString(1);
          a.add(id);
        }
```

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

    private void loadRow() throws SQLException
    {
        String selectStatement =
            "select firstname, lastname, balance " +
            "from account where id = ? ";
        PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);

        prepStmt.setString(1, this.id);

        ResultSet rs = prepStmt.executeQuery();

        if (rs.next()) {
          this.firstName = rs.getString(1);
          this.lastName = rs.getString(2);
          this.balance = rs.getDouble(3);
          prepStmt.close();
        }
        else {
          prepStmt.close();
          throw new NoSuchEntityException("Row for id " + id +
            " not found in database.");
        }
    }


    private void storeRow() throws SQLException
    {
        String updateStatement =
            "update account set firstname =  ? ," +
            "lastname = ? , balance = ? " +
            "where id = ?";
        PreparedStatement prepStmt =
            con.prepareStatement(updateStatement);
```

```
        prepStmt.setString(1, firstName);
        prepStmt.setString(2, lastName);
        prepStmt.setDouble(3, balance);
        prepStmt.setString(4, id);
        int rowCount = prepStmt.executeUpdate();
        prepStmt.close();

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

<소스> AccountEJB.java

Primary Key 클래스 구현

본 장에서 다루고 있는 예제에서는 특별히 primary key 클래스를 정의하여 사용하고 있지 않다. 왜냐하면 본 장의 예제는 primary key로써 String 클래스의 인스턴스인 id를 사용하고 있기때문이다. 즉, java.lang.String 클래스가 primary key 클래스 역할을 하고 있다고 할 수 있다.

하지만, 일반적으로 많은 엔티티 빈에서는 고유한 primary key 클래스를 따로 정의하여 사용하고 있다. 예를들어, 만일 primary key가 여러 개의 field들로 구성되어 있다고 한다면, 프로그래머는 반드시 primary key 클래스를 정의하여 사용해야 하는 것이다.

다음 예제에서 productId와 vendorId 필드가 조합되어 엔티티 빈의 primary key 역할을 하고 있다.(이 예제는 본 장의 예제와 관련이 없다.)

```
public class ItemKey implements java.io.Serializable
{
    public String productId;
    public String vendorId;

    public ItemKey() { };

    public ItemKey(String productId, String vendorId)
    {
            this.productId = productId;
```

```
            this.vendorId = vendorId;
    }

    public String getProductId()
    {
            return productId;
    }

    public String getVendorId()
    {
            return vendorId;
    }

    public boolean equals(Object other)
    {
            if (other instanceof ItemKey) {
                    return (productId.equals(((ItemKey)other).productId)
                    && vendorId.equals(((ItemKey)other).vendorId));
            }
            return false;
    }

    public int hashCode()
    {
            return productId.hashCode();
    }
}
```

<소스> ItemKey.java

✍✍ primary key 클래스 요구사항

- 클래스의 access control modifier는 public이어야 한다.
- 모든 필드들은 public으로 선언되어야 한다.
- CMP일 경우, primary key 클래스의 필드 이름은 대응되는 엔티티 빈 클래스에 있는 container-managed 필드 이름과 반드시 동일해야 한다.
- Primary key 클래스는 public default 생성자가 있어야 한다.
- Primary key 클래스는 직열화가 가능해야 한다.

✍✍ BMP와 Primary Key 클래스

BMP 기반에서는 ejbCreate method가 primary key 클래스를 반환한다.

```
public ItemKey ejbCreate(String productId, String vendorId,
    String description) throws CreateException
{
    if (productId == null || vendorId == null) {
            throw new CreateException("The productId and
vendorId are required.");
    }

    this.productId = productId;
    this.vendorId = vendorId;
    this.description = description;

    return new ItemKey(productId, vendorId);
}
```

ejbFindByPrimaryKey method는 주어진 primary key에 대해 해당하는 데이터베이스의 필드가 존재하는지 판별한다.

```
public ItemKey ejbFindByPrimaryKey(ItemKey primaryKey)
                throws FinderException
{
    try {
            if (selectByPrimaryKey(primaryKey))
                return primaryKey;
    . . .
}

private boolean selectByPrimaryKey(ItemKey primaryKey)
            throws SQLException
{
    String selectStatement =
                "select productid " +
            "from item where productid = ? and vendorid = ?";
    PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);
    prepStmt.setString(1, primaryKey.getProductId());
    prepStmt.setString(2, primaryKey.getVendorId());
```

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

✍ CMP와 Primary Key 클래스

CMP 기반의 엔티티 빈에서 ejbCreate method는 null값을 반환한다.(BMP 기반에서는 primary key 클래스의 인스턴스를 반환했었다.)

CMP 기반의 엔티티 빈에서는 개발자가 ejbFindByPrimaryKey method를 코딩할 필요가 없다. 왜냐하면, Application Deployment tool이 해당 method들에 대한 해당하는 데이터베이스상의 필드를 추출하는 SQL select 구문을 포함하여 자동으로 그것들을 구현해 주기 때문이다.

또한 아래와 같은 홈 인터페이스에서 ejbFindByPrimaryKey method를 제외한 사용자가 정의한 각종 finder method(findByDescription, findInRange)들에 대해서도 tool이 해당 method들에 대한 SQL 구문을 포함하여 자동으로 그것들을 구현해 준다.

```
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;
…
```

그러나, 이 사용자 정의 method에 대한 SQL 구분에서 개발자는 select 문에 대한 where 절에 대해 Deployment tool에서 지정해 주어야 한다.

✍ Primary Key 사용하기

클라이언트에서 엔티티 빈의 primary key를 얻어오려면 엔티티 빈 class에 선언되어 있는 getPrimaryKey method를 호출하면 된다.

```
Account account;
. . .
String id = (String)account.getPrimaryKey();
```

엔티티 빈이 자신의 primary key를 얻고 싶은때는 EntityContext class에 정의되어 있는 getPrimaryKey method를 호출한다.

```
EntityContext context;
. . .
String id = (String) context.getPrimaryKey();
```

InsufficientBalanceException 클래스 구현

InsufficientBalanceException은 이 예제에서만 적용되도록 만들어진 예외처리용 클래스이다. 다음은 소스 리스트이다.

```
public class InsufficientBalanceException extends Exception
{
   public InsufficientBalanceException() { }
   public InsufficientBalanceException(String msg) {
   s                    uper(msg);
    }
}
```

Deployment Descriptor 작성

- 일반적으로 Deployment Descriptor는 컨테이너에서 제공하는 GUI 환경의 Deployment tool을 이용하여 작성된다.
- GUI 환경이 지원되는 않는 일부 UNIX 환경에서는 직접 수작업으로 Deployment Descriptor를 작성해야 하며, 수작업으로 작성할때는 GUI Deployment tool이 제공하지 못하는 세밀한 설정이 가능하다.
- 여기서는 SUN사에서 제공하는 J2EE SDK 1.2.1 버전의 Application Deployment Tool을 이용한 Deployment Descriptor의 작성과 .JAR 파일 및 .EAR 파일의 작성 과정을 살펴본다.

J2EE SDK 설치

- SUN의 Java 공식 홈페이지에서 J2EE SDK 1.2.1 버전을 다운받아 압축을 풀고 인스톨한다.
- 인스톨이 완료되고 나면 환경변수에 다음과 같이 Home path와 CPATH를 설정해 준다.

```
set J2EE_HOME=<installation-location>
set CPATH=.;%J2EE_HOME% \lib\j2ee.jar
```

- 위 설정에서 <installation-location>에는 실제 J2EE SDK가 설치된 디렉토리의 full path를 기록한다. 예를 들면 다음과 같다.

```
set J2EE_HOME= C:\Lang\j2sdkee1.2.1
```

✍✍ **구현 EJB 파일들을 컴파일하기**

지금까지 구현한 다음 파일들을 컴파일 한다.

Account.java ? 리모트 인터페이스

AccountHome.java ? 홈 인터페이스

AccountEJB.java ? 엔티티 빈 클래스

InsufficientBalanceException.java ? helper 클래스

Command line prompt 창을 열고 다음과 같이 입력하여 각 파일을 컴파일 한다. 이때 따로따로 하지 않고 한꺼번에 해도 상관없다.

```
javac –classpath %CPATH% Account.java
javac –classpath %CPATH% AccountHome.java
javac –classpath %CPATH% AccountEJB.java
javac –classpath %CPATH% InsufficientBalanceException.java
```

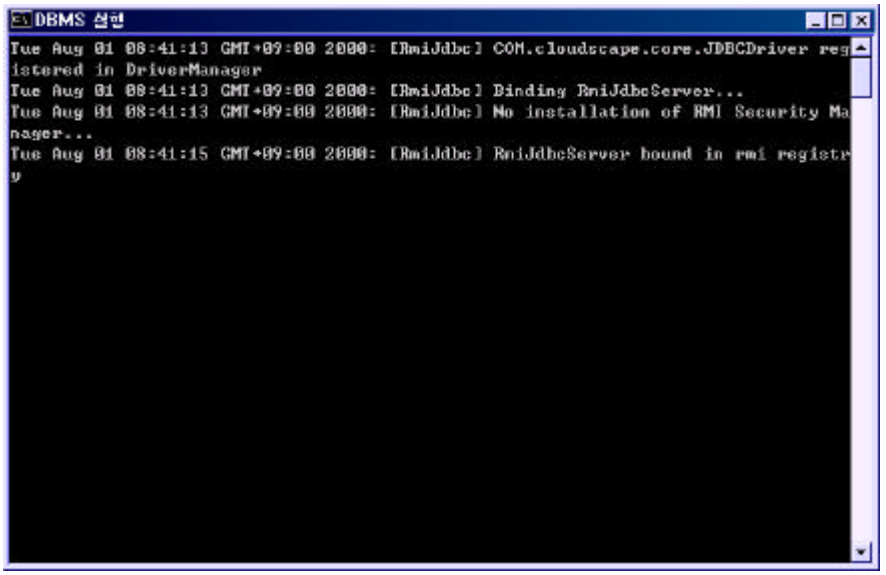
Database 실행하기

✍✍ **이 장에서 사용하는 예제는** Database**와 연동이 되기 때문에** Database **서버를 실행시켜야 한다.**

✍✍ **여기서는** J2EE SDK**에 번들로 따라오는** Cloudscape DB**를 사용하여 예제를 실행시켜본다.**

✍✍ Cloudscape DB**는 다음과 같은** command line **명령을 통해 실행된다.**

```
c:\Lang\j2sdkee1.2.1\bin>cloudscape -start
```



<그림> Cloudscape 데이터베이스 서버 실행시 콘솔화면

✍✍DB 서버가 실행되었으면 이제 응용 프로그램에서 필요로 하는 테이블을 생성해야 한다. 테이블을 생성하는 SQL 구문은 다음과 같다.

```
create table account
   (id varchar(3) constraint pk_account primary key,
   firstname varchar(24),
   lastname varchar(24),
   balance decimal(10,2));
```

✍✍ 위 SQL 구문을 실행하기 위해서는 2가지 방법이 있다.

첫 번째 방법 : cloudIJ에서 직접 입력하는 것으로 cloudIJ는 오라클의 SQL Plus와 비슷한 command line 입력기 이다.

- 다음 디렉토리 위치로 이동한다.

```
cd $J2EE_HOME\doc\guides\ejb \examples\util
```

cloudIJ.bat를 실행한다. 이때 cloudIJ.bat 내에 J2EE_HOME을 지정하는 명령줄이 있는데 이를 삭제하거나 아니면 올바른 Home path를 반드시 설정해 주어야 한다.

```
cloudIJ.bat
```

- 테이블을 생성하는 SQL 구문을 실행한다.

두 번째 방법 : SQL 구문을 배치 파일로 만들어 놓은 cloudTable.bat 파일을 직접 실행하는 것으로 이것 역시 cloudTable.bat 파일 내에 J2EE_HOME을 지정하

는 명령줄이 있는데 이를 삭제하거나 아니면 올바른 Home path를 반드시 설정해 주어야 한다.

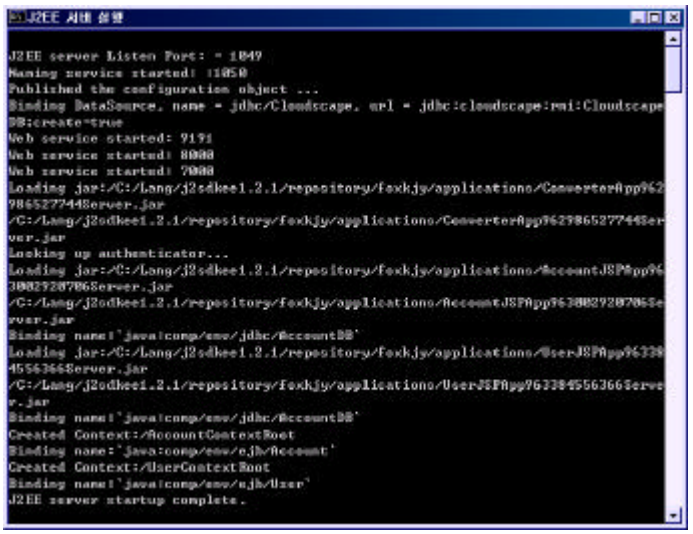
```
cd %J2EE_HOME% \doc\guides\ejb\examples\account
..\util\cloudTable
```

J2EE 서버 실행하기

✍✍설치가 끝나면 J2EE 서버(EJB 컨테이너)를 실행시킨다. J2EE 서버는 command line prompt에서 다음과 같은 명령으로 기동시킬 수 있다.

```
c:\Lang\j2sdkee1.2.1\bin>j2ee –verbose
```

✍✍ 다음은 서버를 기동시킨 command line prompt 화면이다.



<그림> J2EE 서버 실행시 콘솔화면

✍✍서버를 중지 시키려면 새로운 command line prompt에서 다음과 같은 명령어를 사용하면 된다.

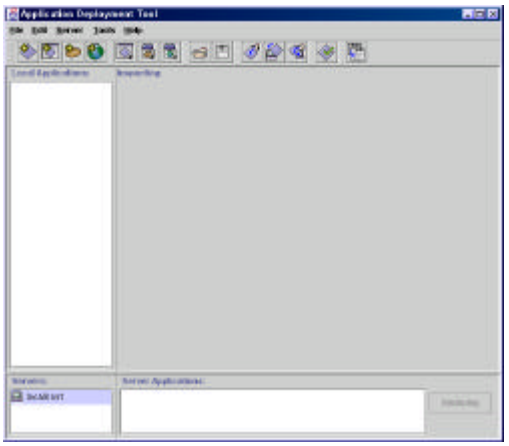
```
c:\Lang\j2sdkee1.2.1\bin>j2ee –stop
```

Application Deployment Tool 실행하기

✍✍J2EE 서버가 실행되고 나면 새로운 command line prompt 에서 J2EE SDK에서 제공되는 Application Deployment tool을 다음과 같이 실행시킨다.

```
deploytool
```

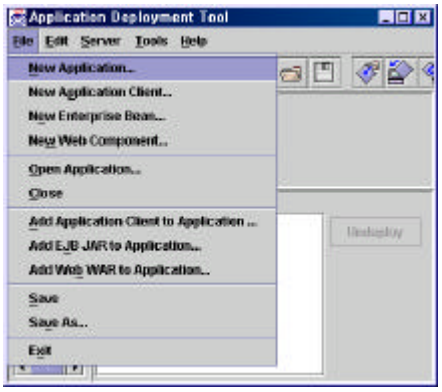
✍✍ 다음은 초기 실행화면이다.



<그림> Application Deployment Tool – 초기 실행화면

새로운 Enterprise bean 만들기

✍✍[File] -> [New Application…]을 선택한다.

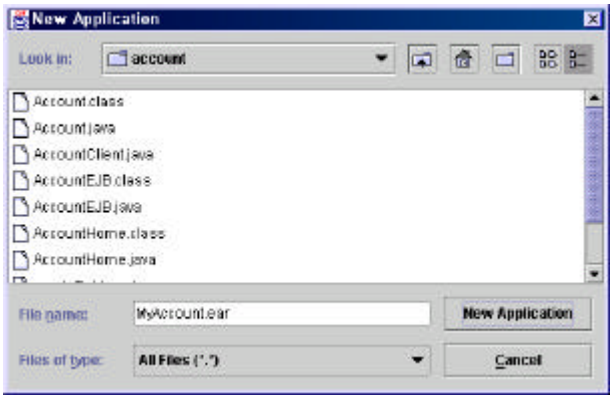


<그림> Application Deployment Tool – File 메뉴 선택화면

✍✍ 그리고 나서 새로 뜬 창에서 [Browse] 버튼을 클릭한다.

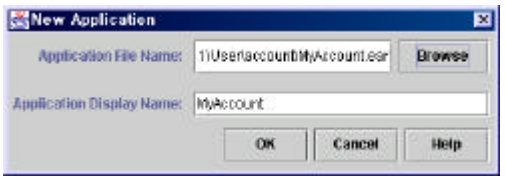
<그림> New Application 대화상자

✍✍File 대화상자에서 J2EE 응용프로그램을 담고 있을 파일인 .ear이 위치할 디렉토리를 선택하고, 파일 이름 필드에 MyAccount.ear(다른 이름을 써도 상관없음)을 입력하고 [New Application] 버튼을 클릭한다.



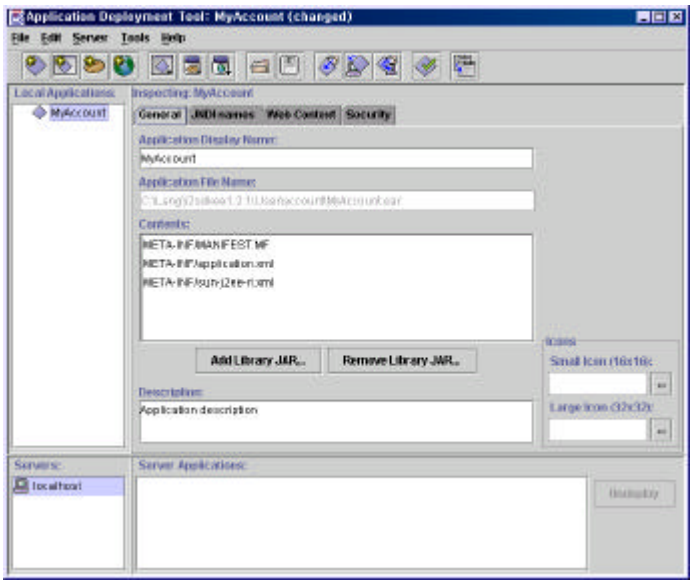
<그림> New Application 대화상자

✍✍ 그러면 다음과 같은 창이 뜨면서 입력한 내용을 확인하는데, 여기서 [OK]를 클릭한다.



<그림> New Application 대화상자

✍✍ 생성이 완료된 화면



<그림> Application Deploymnet Tool 실행화면

Enterprise bean 패키징 하기

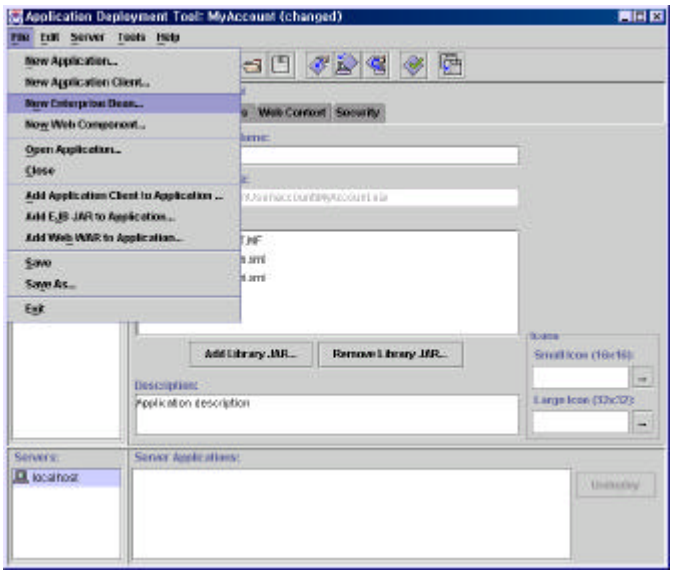
✍✍ 여기서는 위에서 작성하고 컴파일한 파일들을 패키징하는 일이다. 이 부분이 핵심 Deployment descriptor 작성부분으로 다음과 같은 일을 하게 된다.

- Bean의 Deployment descriptor를 작성한다.

- Deployment descriptor와 bean의 각종 class파일들을 EJB 파일인 .jar 파일에 패키징한다.

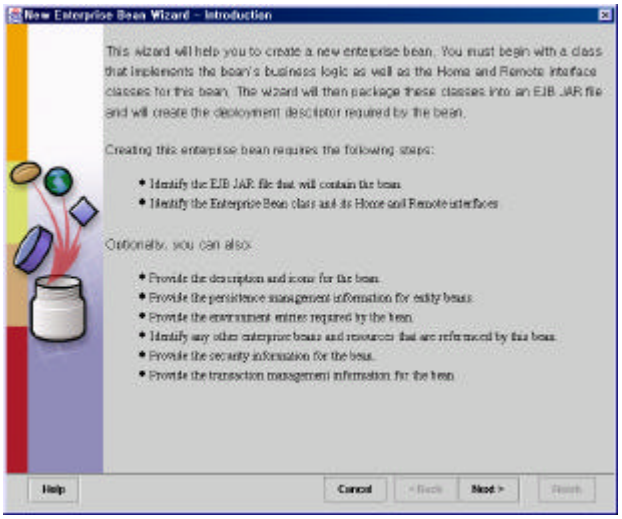
- EJB 파일인 .jar 파일을 어플리케이션 파일인 MyAccount.ear 파일에 집어넣는다.

✍ [File] -> [New Enterprise Bean…]을 선택한다.



<그림> Application Deployment Tool 실행화면

✍✍[New Enterprise Bean Wizard – Introduction] : 한번 읽어보고 [Next]를 선택한다.

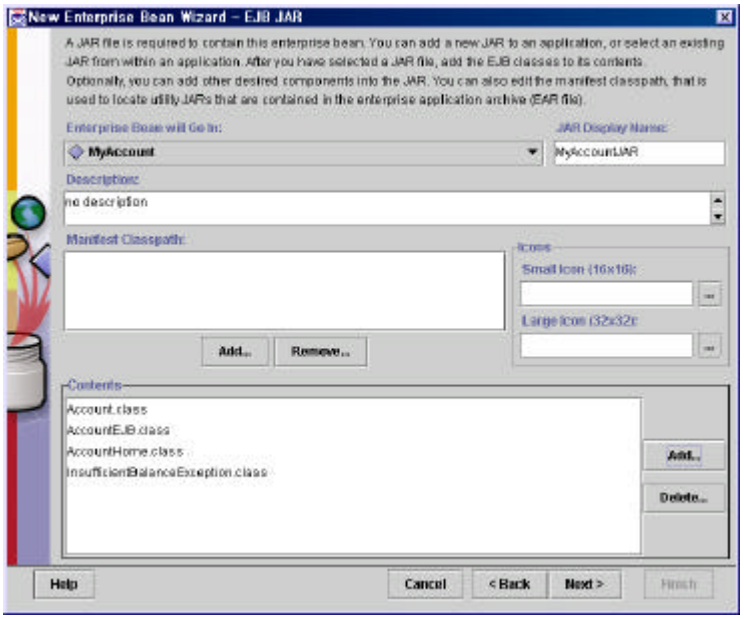


<그림> Introduction 대화상자

✍ [New Enterprise Bean Wizard – EJB JAR]

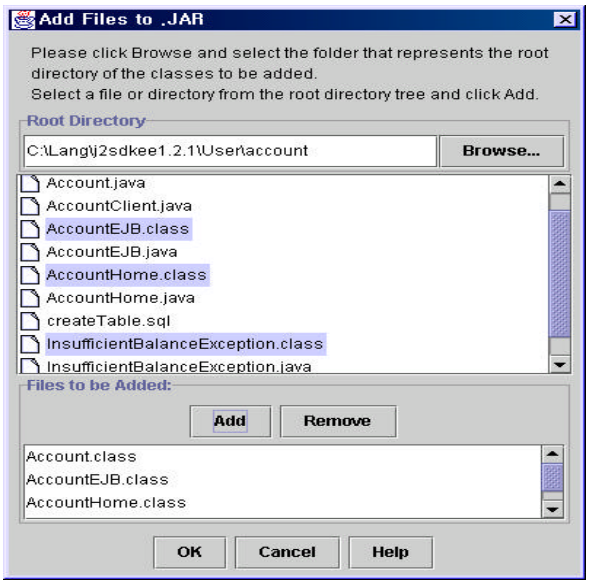
Enterprise Bean will Go In : MyAccount를 선택한다.

JAR Display Name : MyAccountJAR을 입력한다. 이 이름은 bean을 담고 있는 EJB .jar 파일을 나타내는 것으로 트리 view에서 나타나게 된다.



<그림> EJB JAR 대화상자

✍✍ Contents 박스의 [Add] 버튼을 클릭한다.



<그림> Add Files to .JAR 대화상자

[Add Files to .JAR] 대화상자에서 소스코드를 컴파일 하여 생성된 *.class 파일들이 위치하는 디렉토를 선택하고 디렉토리 안에 들어있는 class 파일들을 모두 선택하여 [Add] 한후 [OK] 버튼을 클릭한다.

✍✍ 포함되어야 할 class 파일은 다음 4개 이다.

Account.class

AccountHome.class

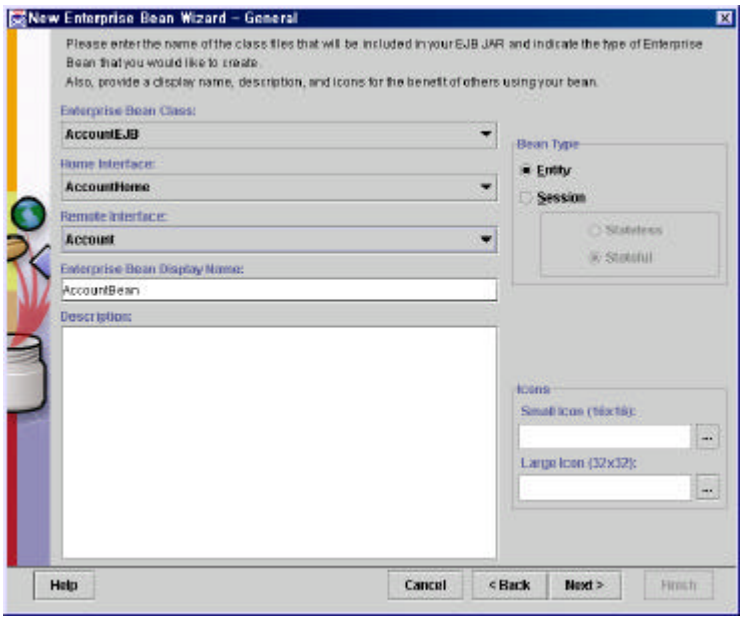
AccountEJB.class

InsufficientBalanceException.class

✍✍ 작업이 완료되었으면 [Next]를 클릭한다.

✍✍ [New Enterprise Bean Wizard – General]

Bean Type : Entity를 선택한다.



<그림> General 대화상자

Enterprise Bean Class : AccountEJB 파일 선택

Home Interface : AccountHome 파일 선택

Remote Interface : Account 파일 선택

Enterprise Bean Display Name : AccountBean 입력한다. 이 이름은 트리 뷰에 나타나게 된다.

[Next]를 클릭한다.

✍✍ [New Enterprise Bean Wizard – Entity Settings]

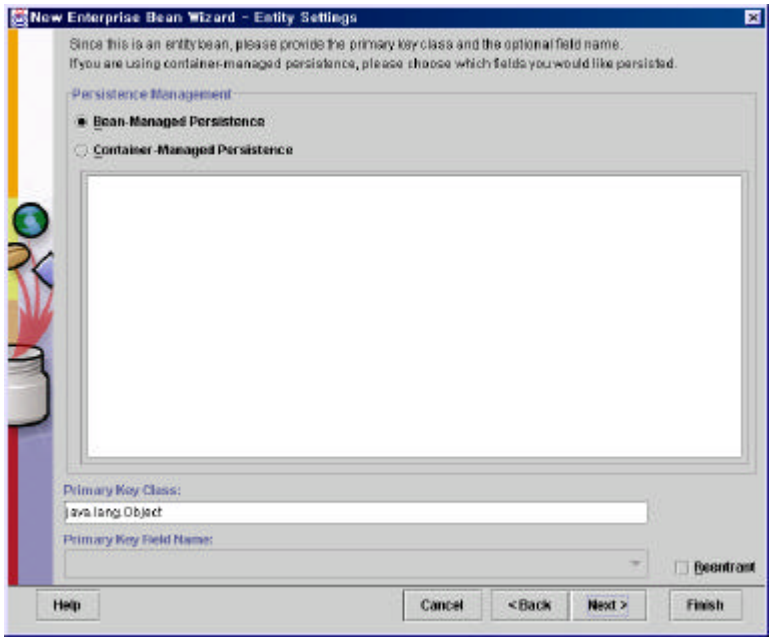
Bean-Managed Persistence를 선택하고 [Next]를 클릭한다.

✍✍ [New Enterprise Bean Wizard – Environment Entries]

[Next]를 클릭한다.(아무 작업도 하지 않는다.)

✍✍ [New Enterprise Bean Wizard – Enterprise Bean References]

[Next]를 클릭한다.(아무 작업도 하지 않는다.)



<그림> Entity Settings 대화상자

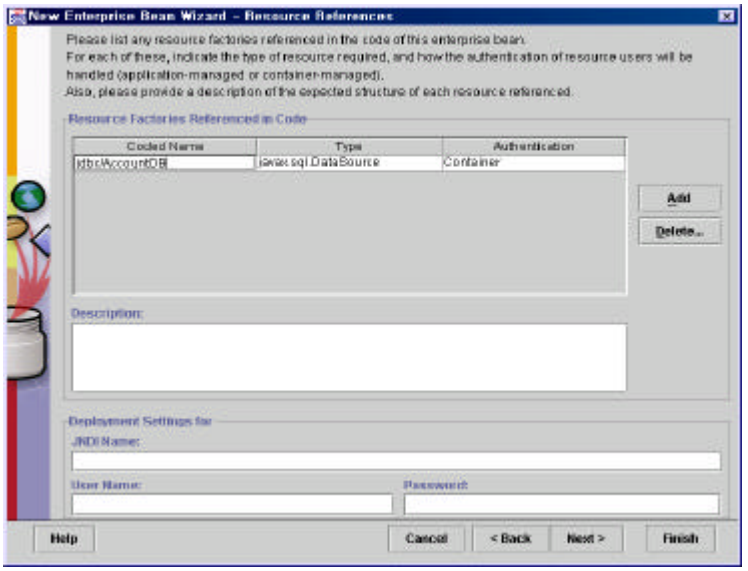
✍ [New Enterprise Bean Wizard – Resource References]

[Add] 버튼을 클릭한다.

[Coded Name] 필드에 jdbc/AccountDB를 입력한다.

[Type] 칸에서 javax.sql.Datasource를 선택한다.

[Authentication] 칸에서 Container를 선택한다.



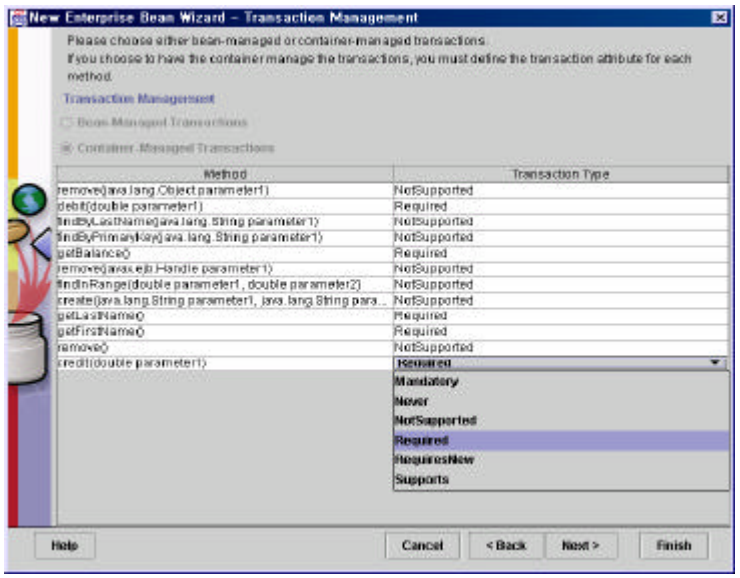
<그림> Resource Reference 대화상자

[Next]를 클릭한다.

✍ [New Enterprise Bean Wizard – Security]

[Next]를 클릭한다.(아무일도 하지 않는다.)

✍✍ [New Enterprise Bean Wizard – Transaction Management]



<그림> Transaction Management 대화상자

비즈니스 method에 대해 트랙잭션을 Required로 설정한다. 다음은 설정해야 할 비즈니스 method들이다.

debit

create

getFirstName

getLastName

getBalance

[Next]를 누르면 지금까지 설정한 내용으 나타나게 되고, [Finish]를 누르면 설정이 종료된다.

지금까지의 과정을 통해 Deployment descriptor를 작성하고 이를 class 파일들과 같이 패키징하여 MyAccount.ear 파일을 생성하였다.

MyAccount.ear 파일안에는 다음과 같은 파일이 들어있다.

Ejb-jar-ic.jar

MANIFEST.MF

application.xml

sun-j2ee-ri.xml

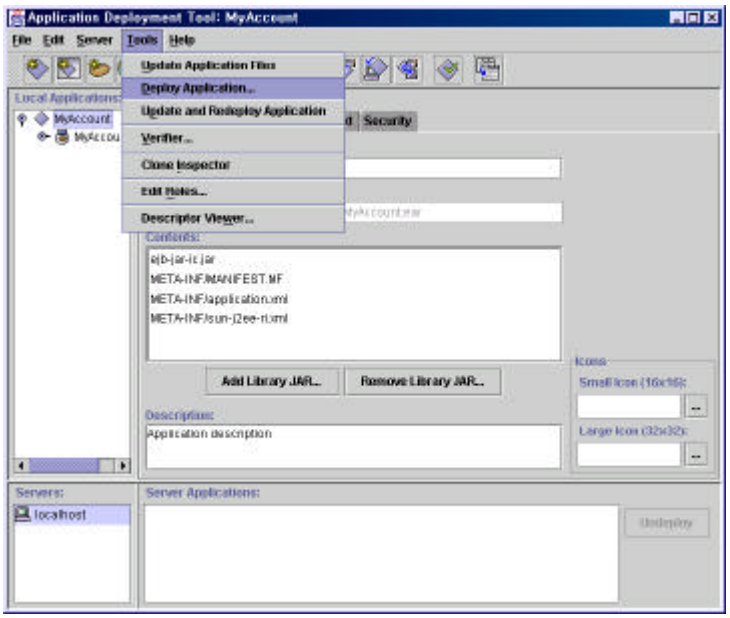
파일들은 Deployment 과정에서 모두 다시 설치되게 된다.

Application Deployment(어플리케이션 배치)

✍ 앞 절에서 Deployment descriptor의 작성과 함께 EJB .ear 파일을 패키징하는데 성공하였다면, 이제 패키징한 EJB .ear 파일은 J2EE 서버에 Deployment해야 한다.

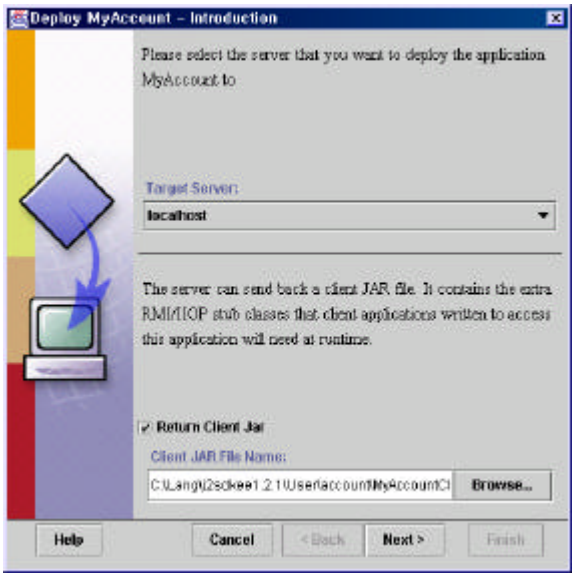
✍ Application Deployment Tool

[Tools] -> [Deploy Application…] 메뉴를 선택한다.



<그림> Application Deployment Tool 실행 화면

✍✍Return Client jar 체크박스를 선택한후 [Next] 버튼을 클릭한다.

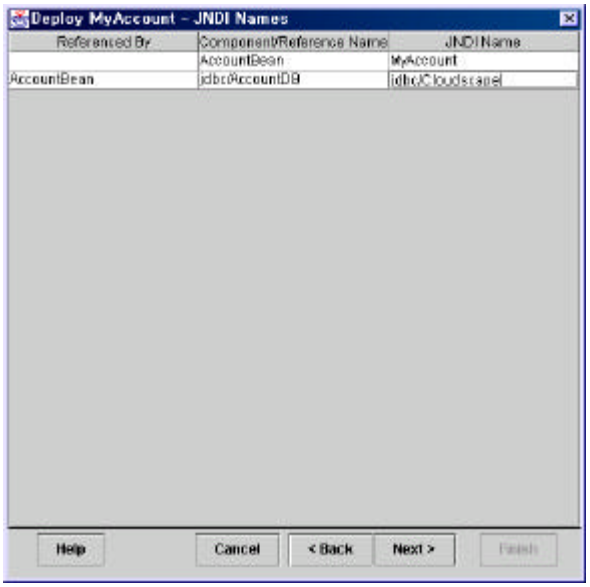


<그림> Introduction 대화상자

✍ JNDI 설정

AccountBean 열의 JNDI Name 필드에 MyAccount를 입력한다.

jdbc/AccountDB 열의 JNDI Name 필드에 jdbc/Cloudscape를 입력한다.



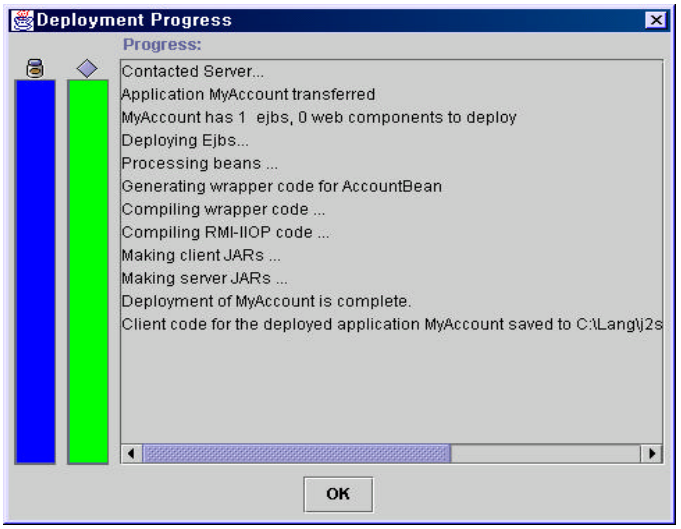
<그림> JNDI Names 대화상자

[Next]를 클릭하면, 지금까지 설정한 내용이 나타나고 [Finish]를 클릭하면, Deployment가 시작된다.

✍ Deployment Progress

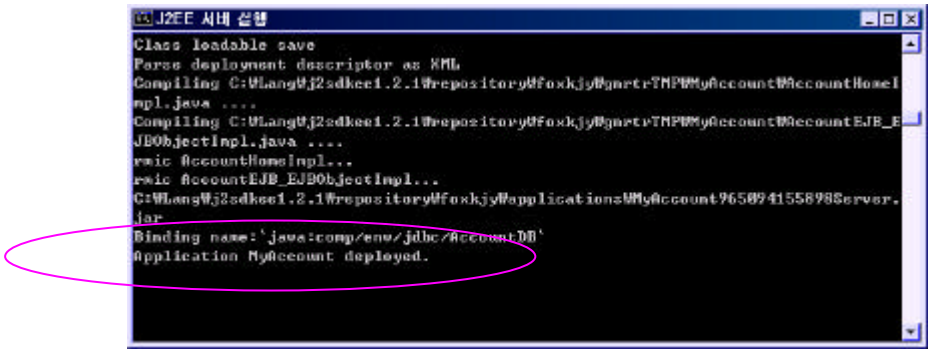
Deployment 과정이 나타나고 완료되면 [OK]버튼으로 바뀐다.

[OK] 버튼을 누르면 Deploy 과정이 비로소 완료된 것이다.



<그림> Deployment Progress

다음음 Deployment 실행 후의 J2EE 서버의 console 화면으로 MyAccount의 Deployment가 실행되었음을 알 수 있다.



<그림> Deployment 실행 후 J2EE 서버의 콘솔 화면

Client Program : AccountClient.java

✍ 클라이언트 프로그램은 다음과 같은 역할을 한다.

계좌 생성

계좌 예금

계좌 인출

계좌 잔고 확인

계좌번호로 검색

이름으로 검색

계좌 삭제

✍ 위와 같은 기능들은 이미 리모트 인터페이스와 엔티티 빈에서 전부 구현된 기능들로 클라이언트 프로그램은 다음과 같은 과정을 거쳐 이들 기능을 사용하게 된다.

EJB 서버에 접속하기 위한 JDNI Naming Context를 생성한다.

JND Naming Contextl를 통해 EJB 서버에 배치되어 있는 EJB 빈중에서 현재 사용하려고 하는 Account bean을 검색한다.

Account 엔티티 빈을 접근하기 위한 Account 홈 인터페이스에 대한 참조 객체를 얻어온다.

참조 객체를 통해 엔티티 빈을 생성한다.

생성된 엔티티 빈은 리모트 인터페이스에 정의된 method들을 통해 원하는 비즈니스 로직을 실행하게 된다.

마지막으로 엔티티 빈은 remove method를 통해 컨테이너로부터 삭제되고 이것은 DB에서의 삭제와 동일하다.

✍✍ 다음은 각각의 과정에 대한 설명이다.

EJB 서버 접속을 위한 JNDI Context 설정 : 클라이언트는 EJB 빈을 획득하기 위해서 JNDI를 통해 EJB 서버에 접근하게 된다. 이를 위해서는 JNDI Naming Context를 생성해 주어야 한다.

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

EJB 빈 레퍼런스 객체 찾기 : 설정된 JNDI Context를 통해 접근하고자 하는 EJB 빈(여기서는 MyAccount)의 레퍼런스 객체를 획득하게 된다.

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

홈 인터페이스 객체 얻어오기 : 실질적으로 엔티티 빈의 라이프 사이클을 제어하는 것이 바로 홈 인터페이스이다. 클라이언트는 홈 인터페이스를 통해 사용할 빈을 생성할 수 있게 되는 것이다.

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

엔티티 빈 인스턴스 생성 : 홈 인터페이스의 정의에 포함된 create method를 사용하여 실제 사용될 엔티티 빈 인스턴스를 생성하게 된다.

```
Account duke = home.create("123", "Duke", "Earl", 0.00);
…
Account joe = home.create("836", "Joe", "Jones", 0.00);
…
Account john = home.create("730", "John", "Smith", 0.00);
```

비즈니스 method의 실행 : 일단 엔티티 빈의 인스턴스가 생성되고 나면 이 인스턴스는 리모트 인터페이스에 정의된 method를 통해 자신에게 주어진 비즈니스 method들를 실행하게 된다.

```
duke.credit(88.50);
duke.debit(20.25);
double balance = duke.getBalance();
```

엔티티 빈의 검색 : 엔티티 빈은 생성되기만 하는 것이 아니라 이미 생성된 엔티

티 빈을 검색하여 이를 값으로 가질 수도 있다. 하나의 EJB 빈에 대해 여러 개의 빈 인스턴스들이 생성될 수 있다. 이 들은 이미 구현된 Finder method들을 통하여 PrimaryKey 를 통한 검색이나 다른 필드를 이용한 검색이 가능하고, 반환값도 하나의 엔티티 빈이 될 수도 있고 엔티티 빈의 집합형(Enumeration type)이 될 수도 있다.

```
Account jones = home.findByPrimaryKey("836");
...
Collection c = home.findByLastName("Smith");
Iterator i=c.iterator();

while (i.hasNext()) {
   Account account = (Account)i.next();
   String id = (String)account.getPrimaryKey();
   double amount = account.getBalance();
   System.out.println(id + ": " + String.valueOf(amount));
}
...
c = home.findInRange(20.00, 99.00);
i=c.iterator();

while (i.hasNext()) {
   Account account = (Account)i.next();
   String id = (String)account.getPrimaryKey();
   double amount = account.getBalance();
   System.out.println(id + ": " + String.valueOf(amount));
}
```

엔티티 빈의 삭제 : 엔티티 빈은 remove method를 통해 컨테이너에서 삭제되고, 엔티티 빈은 또한 DB의 한 필드와 대응하므로 이 필드 또한 삭제되게 된다.

```
duke.remove();
```

✍✍ 다음은 클라이언트 프로그램의 전체 소스 리스트이다.

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

```
public class AccountClient
{
public static void main(String[] args)
{
try {
        Context initial = new InitialContext();
        Object objref = initial.lookup("MyAccount");

        AccountHome home =
           (AccountHome)PortableRemoteObject.narrow(objref,
                                AccountHome.class);

        Account duke = home.create("123", "Duke", "Earl", 0.00);
        duke.credit(88.50);
        duke.debit(20.25);
        double balance = duke.getBalance();
        System.out.println("balance = " + String.valueOf(balance));
        duke.remove();

        Account joe = home.create("836", "Joe", "Jones", 0.00);
        joe.credit(34.55);
        Account jones = home.findByPrimaryKey("836");
        jones.debit(2.00);
        balance = jones.getBalance();
        System.out.println("balance = " + String.valueOf(balance));

        Account pat = home.create("456", "Pat", "Smith", 0.00);
        pat.credit(44.77);
        Account john = home.create("730", "John", "Smith", 0.00);
        john.credit(19.54);
        Account mary = home.create("268", "Mary", "Smith", 0.00);
        mary.credit(100.07);

        Collection c = home.findByLastName("Smith");
        Iterator i=c.iterator();

        while (i.hasNext()) {
          Account account = (Account)i.next();
```

```
            String id = (String)account.getPrimaryKey();
            double amount = account.getBalance();
            System.out.println(id + ": " + String.valueOf(amount));
          }

          c = home.findInRange(20.00, 99.00);
          i=c.iterator();

          while (i.hasNext()) {
            Account account = (Account)i.next();
            String id = (String)account.getPrimaryKey();
            double amount = account.getBalance();
            System.out.println(id + ": " + String.valueOf(amount));
          }
        } catch (InsufficientBalanceException ex) {
          System.err.println("Caught an InsufficientBalanceException: "
                             + ex.getMessage());
        } catch (Exception ex) {
          System.err.println("Caught an exception." );
          ex.printStackTrace();
}
}
}
```

<소스> AccountClient.java

✍✍ **클라이언트 코드 컴파일**

다음 명령어를 통해 컴파일 할 수 있다.

```
set CPATH=.;%J2EE_HOME% \lib\j2ee.jar
javac –classpath %CPATH% AccountClient.java
```

✍✍ **클라이언트 실행하기**

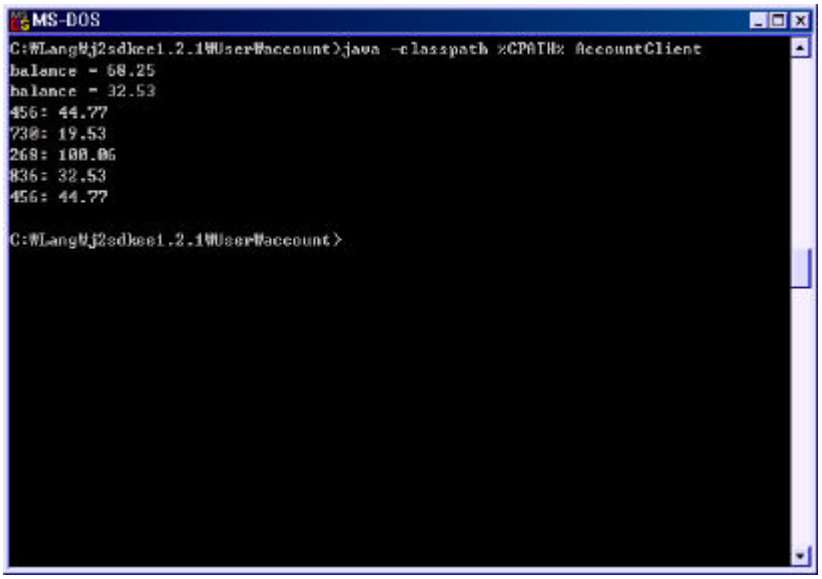
클라이언트 프로그램을 실행하기 위해서는 MyAccountClient.jar 파일이 필요하다.

이 jar 파일은 클라이언트가 EJB 컨테이너에서 실행되고 있는 EJB 빈 인스턴스와 통신을 할 수 있도록 해주는 stub 클래스를 포함하고 있다.

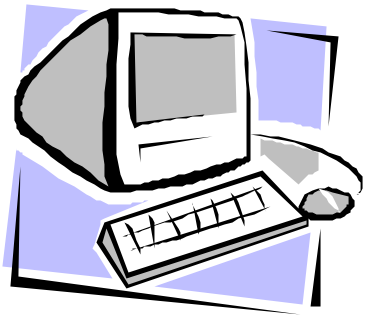
다음 명령을 통해 클라이언트 프로그램을 실행시킬 수 있다.

```
set CPATH=.;%J2EE_HOME% \lib\j2ee.jar;MyAccountClient.jar
java –classpath %CPATH% AccountClient
```

✍ 다음은 실행 결과이다.



<그림> 클라이언트 프로그램 실행 화면



Chapter 8

세션 빈 개발

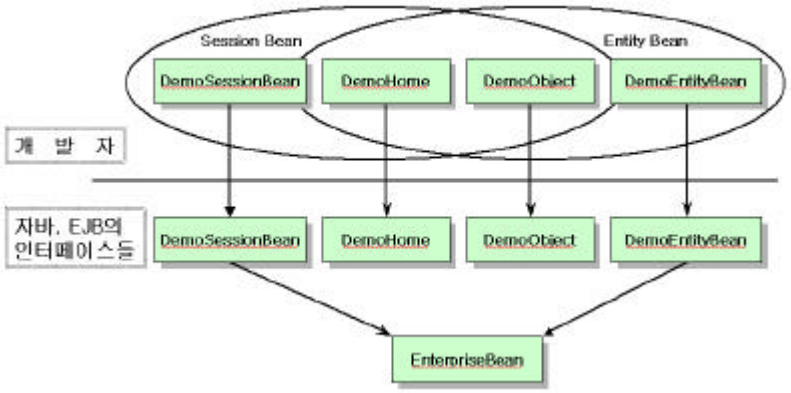
세션 빈이란?

- 세션 빈은 엔티티 빈으로 처리되지 않는 부분들을 채우는 역할로서 서로 다른 빈 간의 상호 작용을 구현하는데 유용하게 쓰인다.
- 공유된 데이터에 액세스하는 역할을 하고, 데이터를 읽고, 변경하고, 추가하는 데 세션 빈을 사용할 수 있다
- 클라이언트에서 홈 인터페이스를 통해 새롭게 생성되거나 삭제될 수 있다. 즉, 세션 빈의 생성과 삭제는 클라이언트에 의존적이다.
- 주로 엔티티 빈은 어떤 개념을 나타내는 공유 데이터들의 집합에 대한 안정적이고 일관성 있는 인터페이스를 제공하기 위해 만들고, 자주 수정될 수 있다. 반면, 세션빈은 어떤 개념을 확장한 데이터를 액세스하고, 공유가 되지 않으며, 대개 읽기 전용인 경우가 많다.
- 세션 빈은 엔티티 빈의 퍼시스턴스를 갖지 않는다. 즉, 데이터 베이스에 저장되지 않는다.

세션 빈의 특징

- 하나의 세션 빈 인스턴스는 한명의 클라이언트에 의해서만 호출된다.
- 트랜잭션 처리가 된다.
- EJB 서버가 다운되는 경우에 사라진다. 따라서 클라이언트는 새로운 세션 빈 객체를 다시 얻어야 한다.
- 한 사용자와의 세션 시작에 의해 생성되고 사라질 수 있다. 일반적으로 생명주기를 한 클라이언트의 세션과 같이한다.
- 데이터베이스에 있는 데이터 그 자체를 표현하지는 않지만, 이 데이터들을 수정하거나, 삭제하거나 추가할 수 있다.
- 비즈니스 로직을 가지고 있다.

엔터프라이즈 빈의 클래스 계층도

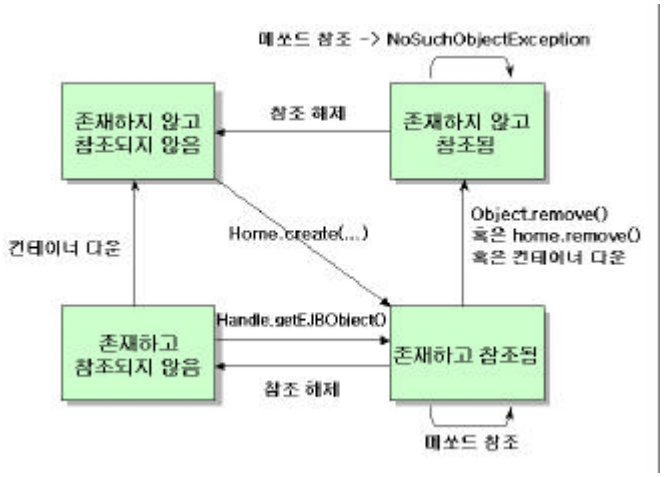


<그림> 클래스 계층도

위의 그림과 같이 EJB 개발시 개발자는 세션 빈, 홈 인터페이스, 리모트 인터페이스(객체), 엔티티 빈등의 클래스를 개발한다. 이외에도 Primary Key Class가 있다.

세션 빈의 라이프사이클

클라이언트측에서의 세션 빈의 라이프 사이클



<그림> 세션 빈의 라이프 사이클

✍ '존재하지 않고 참조되지 않음" 상태는 세션빈이 생성되지 않은 상태를 말한다. 이 상태에서 홈 인터페이스에 정의된 create()메소드를 호출하면 새로운 EJB 객체가 생성되며 클라이언트로 리턴된다. 단, 무상태 세션빈인 경우는 create()호출시에 항상 EJB 객체가 생성되는 것은 아니다. 무상태 세션빈 인스턴스는 여러 사용자에 의해 사용될 수 있으므로 현재 풀에 생성되어 있는 인스턴스가 있고 다른 사용자에 의해 호출되고 있는 상태가 아니라면 기존의 인스턴스를 그대로 사용하게 된다.

✍✍ "존재하고 참조됨" 상태는 위와 같이 홈 인터페이스에 있는 create()메소드를 호출하는 경우나, remove() 메쏘드를 호출하지 않은 상태에서 리모트 인터페이스를 더 이상 참조하지 않고, 나중에 자신이 생성한 인스턴스에 다시 접근하고자 하는 경우이다.

소스 예)

```
a:  DemoRemoteInterface iface = home.create();
b:  Handle handle = iface.getHandle();
      iface = null;
        . . . . . .
      iface – handle.getEJBObject();
```

✍✍ "존재하지 않고 참조됨" 상태는 리모트 인터페이스나 홈 인터페이스로부터 remove() 메쏘드를 호출한다거나 클라이언트가 리모트 인터페이스를 사용하는 중에 서버가 다운된 경우 클라이언트는 리모트 인터페이스를 통해 EJB 객체를 참조하고 있다 하더라도 실제 EJB 객체는 서버상에서 사라져 버린 상태이다. 만약 사용자가 이 리모트 인터페이스를 통해 메쏘드를 호출한다면 java.rmi.NoSuchObjectException 에러가 발생한다.

소스 예)

```
DemoRemoteInterface iface = home.create();
. . . . . .
iface.remove();
. . . . . .
iface.method();   // ✍ 에러 발생
```

✍✍ "존재하고 참조되지 않음" 상태는 클라이언트가 remove() 메쏘드를 호출하지 않고 EJB 객체에 대해 더 이상 참조하지 않을 경우 이 EJB 객체는 서버상에 계속 존재하게 된다. 만약, 무상태 세션빈인 경우는 다른 사용자에 의해 다시 호출되어 사용될 수 있다. 하지만 상태정보유지 세션빈인 경우는 각각의 EJB 객체 인스턴스마다 하나의 클라이언트와 연결되기 때문에 이 EJB 객체 인스턴스는 다른 사용자에 의해 사용되지 않으며 계속해서 서버상에 존재하게 된다.

소스 예)

```
DemoRemoteInterface iface = home.create();
Iface.method();
. . . . . .
iface = null;
```

기타 사항

✍✍ '존재하지 않고 참조되지 않음'상태는 JNDI API를 통해 얻어진 EJB 홈 인터페이스를 통해 EJB 객체 인터페이스를 얻기위해 create()라는 EJB 홈의 메쏘드를 사용한다. EJB 객체 인터페이스를 얻게 되면 상태는 '존재하고 참조됨'으로 바뀐다.

✍✍ '존재하고 참조됨'상태에서 해당 엔터프라이즈 빈의 메쏘드를 EJB 객체 인터페이스를 통해 호출할 수 있고, 이때 메쏘드의 모든 파라미터와 리턴 오브젝트는 직렬화(Serializable)가 가능하도록 해야한다. 생성된 엔터프라이즈 빈을 삭제하려면 EJB 홈이나 EJB 객체에서 remove()를 사용한다.

✍✍ 엔터프라이즈 빈이 삭제되면 상태는 '존재하지 않고 참조됨'으로 바뀌고, 이때 메쏘드 호출이 일어나면 컨테이너는 NoSuchObjectException을 발생시킨다. EJB 객체의 클라이언트 쪽에서의 참조가 가비지 컬렉션에 의해 제거되면 상태는 원점인 '존재하지 않고 참조되지 않음'으로 돌아간다. 그리고 '존재하고 참조됨'상태에서 참조를 해제하면 '존재하고 참조되지 않음'상태로 바뀌게 된다.

✍✍ 세션 빈은 핸들을 통해서 서버가 다운되거나 세션 빈이 삭제되었을 때 그 내용을 유지할 수 있다. '존재하고 참조됨' 상태에서 해당 EJB 객체에서부터 핸들을 얻어낸 후, 핸들을 직렬화 시킨다. 후에 해당 세션 빈이 필요하게 되면 다시 재직렬화(deserializable)하여 핸들을 얻을 수 있고, Handle.getEJBObject()를 통해 세션 빈의 내용이 완벽히 복원된다. 핸들을 얻은 상태가 '존재하고 참조되지 않음'이고, 핸들에서 EJB 객체를 생성하면 '존재하고 참조됨'을 바뀐다.

✍✍

리모트 인터페이스

리모트 인터페이스란?

리모트 인터페이스는 java.rmi.Object를 계승한 것으로 business method를 정의하며 클라이언트는 이 리모트 인터페이스를 통해 컨테이너의 EJB 객체를 호출한다. EJB 객체는 EJB 인스턴스(EJB 구현클래스의 인스턴스)의 business method를 호출한다. 엔티티빈과 마찬가지로 EJBObject를 상속받아야 하며 java.rmi.*, java.ejb.* 등 두 개의 package를 import해야 한다. 모든 메소드들은 RemoteException을 throw하며 자체 Exception을 발생시킬 수도 있다. 세션 빈은 리턴할 프라이머리 키를 갖고 있지 않기 때문에 getPrimaryKey() 메쏘드는 RemoteException을 낼 것이다.

Method	설명
getEJBHome	엔터프라이즈 빈의 홈 인터페이스를 얻음
getHandle	EJB 객체를 위한 핸들을 얻음
getHandle	EJB 객체의 기본 키를 얻음
isIdentical (EJBObject obj)	주어진 EJB 객체가 호출된 EJB 객체와 동일하다면 검사
remove()	EJB 객체를 제거

<표> EJBObject의 메쏘드

예제

아래의 소스는 무상태 세션 빈으로 현금을 지불하는 고개들을 위한 byCash()와 개인 수표로 지불하는 고객을 위한 byCheck()등의 메쏘드를 포함한다. 세가지 메쏘드는 지불 형태오 고나련된 정보를 받아 지불이 제대로 이루어졌는지를 알려주는 불리언(Boolean) 값을 리턴한다. 이 메쏘드들은 RemoteException외에 PaymentException인 지불을 처리하다가 수표번호가 이상하거나 카드의 유효기간이 만효되는 등의 문제가 있을 경우 발생하는 어플리케이션에만 특화된 예외처리도 포함한다. 리모트 인터페이스에 정의된 각각의 메쏘드는 서로에 대하여 완전히 독립적이고, 처리를 위해 필요한 모든 데이터는 메쏘드의 인자를 통해 얻어진다. EJB 프로그램에서는 상속받지 않고 배치할 때 각각의 코드가 홈 인터페이스, 리모트 인터페이스 그리고 빈 자체 중 어느 부분에 속하는지 설정하여 배치를 하면 EJB 컨테이너가 서비스 요청시 리모트 인터페이스에 선언도어 있는 메쏘드와 동일한 이름을 가지는 메쏘드를 빈으로부터 호출하여 서비스를 제공한다.

<소스> ProcessPayment

```
package com.titan.processpayment;

import java.rmi.RemoteException;
import java.util.Date;
import com.titan.customer.Customer;


public interface ProcessPayment extends javax.ejb.EJBObject {

   public boolean byCheck(Customer customer, Check check, double amount)
    throws RemoteException,PaymentException;

   public boolean byCash(Customer customer, double amount)
    throws RemoteException,PaymentException;

  public boolean byCredit(Customer customer, CreditCard card, double
amount)
    throws RemoteException,PaymentException;
}
```

홈 인터페이스

홈 인터페이스란?

✍ 모든 세션 빈의 홈 인터페이스는 인자를 갖지 않는 create() 메쏘드를 하나씩 포함한다. 이는 EJB스펙에서 규정하는 사항이다. 데이터 베이스내의 데이터를 표현하지 않기 때문에 홈 인터페이스는 클라이언트를 위하여 컨테이너가 새로운 세션 빈을 만들 수 있게 한다. 리모트 인터페이스와 마찬가지로 RMI syntax를 사용하여 정의하며 EJB 컨테이너에 의해서 Implement된다. 즉, 클라이언트로부터 서비스요청이 있을 경우 EJB서버 또는 EJB컨테이너는 빈 자체의 코드를 호출하는 것이 아니라 우선 빈의 홈 인터페이스를 이용하여 호출하게 되는데, JNDI를 사용하여 빈을 찾아낼 때 홈 인터페이스를 사용한다.

Method	설명
GetEJBMetaData	엔터프라이즈 빈을 위한 EJBMetaData 인터페이스를 얻음
getHomeHandle	홈 객체를 위한 핸들을 얻음
remove(Handle handle)	핸들에 의해 식별된 EJB 객체를 제거
remove(java.lang.Object primaryKey)	기본 키에 의해 식별된 EJB 객체를 제거

<표> EJBHome의 메쏘드

예제

✍ CreateException은 세션 빈을 생성하다가 문제가 발생했을 때, 빈자체 또는 컨테이너가 발생시킬 수 있는 예외 처리이다. 컨테이너는 클라이언트에 EJB객체를 할당하는 도중에 어떤 시스템 문제가 발생하면 CreateException을 던진다. 예로 이용할 수 있는 PeocessPayment 빈 인스턴스의 수를 넘었을 경우, 컨테이너는 CreateException을 던지고, 다른 시스템적인 문제가 발생할 경우 에는 RemoteException을 던진다.

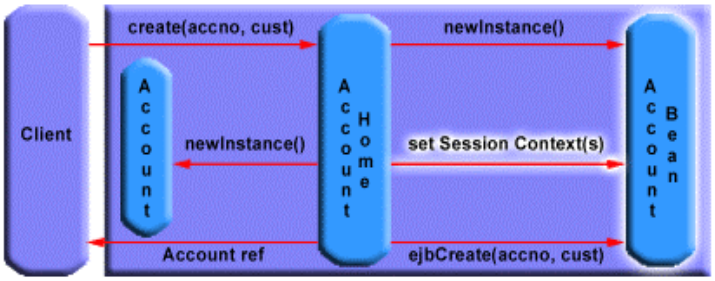
<소스> ProcessPaymentHome

```
package com.titan.processpayment;

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

public interface ProcessPaymentHome extends javax.ejb.EJBHome {
  public ProcessPayment create()
      throws RemoteException, CreateException;
}
```

빈 클래스

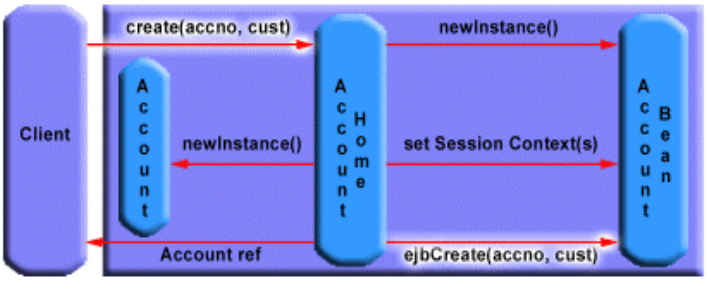
빈 클래스는 리모트 인터페이스에서 정의한 비즈니스 메소드에 대한 실제 구현코드를 작성한다. EJB 구현 클래스는 javax.ejb.*를 import해야 하며 SessionBean이라는 인터페이스를 implement해야 한다. 이 SessionBean에서는 4개의 ejbXXX() 메소드들을 기본적으로 구현해야 한다. 각 메소드는 나름대로의 의미를 가지고 있으며 개발자는 이 부분에 필요에 따라 코드를 삽입할 수 있다.

- ejbActive() : 인스턴스가 pool상태에서 ready 상태로 이동할 때 호출된다.
- ejbPassivate() : 인스턴스가 ready상태에서 pool 상태로 이동할 때 호출된다.
- ejbCreate() : 클라이언트가 홈 인터페이스의 create() 메소드를 호출하여 새로운 인스턴스가 생성될 때 호출된다.
- ejbRemove() : 클라이언트가 remove()메소드를 호출할 때 컨테이너에 의해 호출된다.



<그림> 각 인터페이스와 빈 클래스간의 관계

setSessionContext()메소드는 ejb가 생성된 후 컨테이너에 의해 호출되며 실시간의 session context 정보를 해당 EJB 인스턴스에게 전달한다. 이 session context 객체는 EJB 인스턴스와 생명주기를 같이 한다.



<그림> 각 인터페이스와 빈 클래스간의 관계

컨테이너는 클라이언트가 대응하는 create() 메소드를 호출하였을 때 ejbCreate() 메소드를 호출하며 만일 클라이언트가 요청한 객체가 이미 존재한다면 ejbActivate() 메소드를 대신 호출한다. 양쪽의 경우 모두 인스턴스는 활성화 된 상태가 된다.

예제

✍✍ 리모트 인터페이스에서의 세 개의 지불 메쏘드를 구현한다. byCheck() 메쏘드와 byCredit() 메쏘드는 데이터를 처리하기 전에 그 데이터의 유효성을 체크하는 로직을 갖고 있다. byCredit() 메쏘드는 신용카드의 유효기간이 지났는지를 체크하여 지났을 경우 PaymentException을 던진다.

<소스> ProcessPaymentBean

```
package com.titan.processpayment;

import com.titan.customer.*;

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

import javax.naming.InitialContext;
import javax.sql.DataSource;
import javax.ejb.EJBException;
import javax.naming.NamingException;

public class ProcessPaymentBean implements javax.ejb.SessionBean{

    final public static String CASH = "CASH";
    final public static String CREDIT = "CREDIT";
    final public static String CHECK = "CHECK";

    public SessionContext context;

    public void ejbCreate(){
    }

    public boolean byCash(Customer customer, double amount)
    throws PaymentException{
        return process(getCustomerID(customer),amount,
                    CASH,null,-1,-1,null);
    }

    public boolean byCheck(Customer customer,Check check,double amount)
```

```
    throws  PaymentException{
        int minCheckNumber = getMinCheckNumber();
        if (check.checkNumber > minCheckNumber){
            return process(getCustomerID(customer), amount, CHECK,
                        check.checkBarCode,check.checkNumber,-1,null);
        }
        else {
           throw new PaymentException(
             "Check number is too low. Must be at least "+minCheckNumber);
        }
    }
    public boolean byCredit(Customer customer,CreditCard card, double
amount)
    throws PaymentException {
        if (card.expiration.before(new java.util.Date())){
            throw new PaymentException("Expiration date has passed");
        }
        else {
            return process(getCustomerID(customer), amount, CREDIT, null,
                        -1, card.number,
                        new java.sql.Date(card.expiration.getTime()));
        }
    }
    private boolean process(long customerID, double amount, String type,
                        String checkBarCode, int checkNumber,
                        long creditNumber, java.sql.Date creditExpDate)
    throws PaymentException{

        Connection con = null;

        PreparedStatement ps = null;
        try {
            con = getConnection();
            ps = con.prepareStatement
                ("INSERT INTO payment (customer_id, amount, type,"+
                 "check_bar_code,check_number,credit_number,"+
                 "credit_exp_date) VALUES (?,?,?,?,?,?,?)");
            ps.setLong(1,customerID);
            ps.setDouble(2,amount);
```

```
            ps.setString(3,type);
            ps.setString(4,checkBarCode);
            ps.setInt(5,checkNumber);
            ps.setLong(6,creditNumber);
            ps.setDate(7,creditExpDate);
            int retVal = ps.executeUpdate();
            if (retVal!=1){
                throw new EJBException("Payment insert failed");
            }

            return true;
        } catch(SQLException sql){
             throw new EJBException(sql);
        } finally {
             try {
                 if (ps != null) ps.close();
                 if (con!= null) con.close();
             } catch(SQLException se){se.printStackTrace();}
        }
    }
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void ejbRemove(){}
    public void setSessionContext(SessionContext ctx){
        context = ctx;
    }
    private int getCustomerID(Customer customer){
        try{
            return ((CustomerPK)customer.getPrimaryKey()).id;
        }catch(RemoteException re){
            throw new EJBException(re);
        }
    }

    private Connection getConnection() throws SQLException{
        try{
            InitialContext jndiCntx = new InitialContext( );
            DataSource ds = (DataSource)
            jndiCntx.lookup("java:comp/env/jdbc/titanDB");
```

```
            return ds.getConnection( );
          }catch(NamingException ne){throw new EJBException(ne);}
      }
      private int getMinCheckNumber(){
          try{
            InitialContext jndiCntx = new InitialContext( );
            Integer value =
(Integer)jndiCntx.lookup("java:comp/env/minCheckNumber");
            return value.intValue();
          }catch(NamingException ne){throw new EJBException(ne);}
      }
}
```

Deployment Descriptor

Deployment Descriptor 란?

프로퍼티 정의 파일과 매우 비슷한 직렬화된 클래스로서 배치 디스크립터는 프로그램의 변경없이 런타임시에 프로그램의 동작을 변경하는 것을 가능하게 한다. 개발자가 빈에 대한 모든 프로퍼티를 설정한 후에, 배치 디스크립터는 직렬화되고 파일에 저장된다. 일단 배치 디스크립터가 완성되고 파일에 저장되면 빈은 배치를 위해 JAR파일에 묶일 수 있게 된다. 배치 디스크립터의 프로퍼티들은 배치 툴에게 빈이 어떤 종류의 빈인지, 그 빈이 트랜잭션내에서 어떻게 사용되는지, 등의 빈의 런타임시의 속성들에 대해 알려준다.

대부분의 EJB 서버는 DeploymentDescriptor를 생성하기 위한 위저드를 제공하지만 직접 생성할 수도 있다.

Deployment Descriptor 예제

Deployment Descriptor의 인스턴스를 생성하고, 값을 채우고, 직렬화시키는 프로그램 예이다.

<소스> MakeDD

```
package com.titan.processpayment;

import javax.ejb.deployment.*;
import javax.naming.CompoundName;
import java.util.*;
import java.io.*;

public class MakeDD {

   public static void main(String args [] ){
      try {
      if(args.length <1){
         System.out.println("must specify target directory");
         return;
      }
      SessionDescriptor paymentDD = new SessionDescriptor();
      paymentDD.setRemoteInterfaceClassName(
        "com.titan.processpayment.ProcessPayment");
      paymentDD.setHomeInterfaceClassName(
        "com.titan.processpayment.ProcessPaymentHome");
      paymentDD.setEnterpriseBeanClassName(
```

```
            "com.titan.processpayment.ProcessPaymentBean");
        paymentDD.setSessionTimeout(60);
        paymentDD.setStateManagementType(
            SessionDescriptor.STATELESS_SESSION);

        Properties props = new Properties();
        props.put("jdbcURL","jdbc:<subprotocol>:<subname>");

        props.put("minCheckNumber","1000");
        paymentDD.setEnvironmentProperties(props);

        ControlDescriptor cd = new ControlDescriptor();

    cd.setIsolationLevel(ControlDescriptor.TRANSACTION_READ_COMMITTED);
        cd.setMethod(null);
        cd.setRunAsMode(ControlDescriptor.CLIENT_IDENTITY);
        cd.setTransactionAttribute(ControlDescriptor.TX_REQUIRED);
        ControlDescriptor [] cdArray = {cd};
        paymentDD.setControlDescriptors(cdArray);

        CompoundName jndiName =
            new CompoundName("ProcessPaymentHome", new Properties());
        paymentDD.setBeanHomeName(jndiName);

        String fileSeparator =
            System.getProperties().getProperty("file.separator");
        if (! args[0].endsWith(fileSeparator))
            args[0] += fileSeparator;

        FileOutputStream fis =
            new FileOutputStream(args[0]+"ProcessPaymentDD.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fis);
        oos.writeObject(paymentDD);
        oos.flush();
        oos.close();
        fis.close();
        } catch(Throwable t){t.printStackTrace();}
      }
    }
```

사례 연구

개요

✍✍ 본 프로그램은 쇼핑몰에서 쇼핑카트의 역할을 하는 프로그램으로 여러 상품을 선택할 수 있고, 등록된 고객은 할인율 적용등을 수행하는 프로그램이다.

리모트 인터페이스

<소스> Quote.java

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

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

public interface Quote extends EJBObject {

    // 상품의 가격을 리턴
    public int getNumberOfItems() throws RemoteException;

    // 한 상품의 수량을 라인별로 리턴
    public Vector getLineItems() throws RemoteException;

    //상품의 가격을 합산
    public void addProduct(Product prod) throws QuoteException,
RemoteException;

    // 상품의 수량을 파악하여 0이면 상품을 삭제
// 상품이 존재하지 않으면 QuoteException 발생
    public void setProductQuantity(Product prod, int quantity) throws
QuoteException, RemoteException;

    // 가격을 지움
    public void clear() throws RemoteException;

    // 고객의 구입가격을 리턴
    public Customer getCustomer() throws RemoteException;

// setSubtotal()에 의해 계산된 가격을 리턴
    public double getSubtotal() throws RemoteException;
```

```
    // 고객정보에 기초하여 소계를 재 계산
    public void setSubtotal(double subTotal) throws RemoteException;

    // 세금을 리턴
    public double getTaxes() throws RemoteException;

    // 세금을 정함
    public void setTaxes(double taxes) throws RemoteException;

    // 상품가격과 세금을 합해 총계를 산출
    public double getTotalPrice() throws RemoteException;

    // 구입을 처리
    // 상품이 비었으면 QuoteException 발생
    public Order purchase() throws RemoteException, QuoteException;
}
```

홈 인터페이스

<소스> QuoteHome.java

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

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

/**
 * This is the home interface for Quote.  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 Quote file.
```

```
 */
public interface QuoteHome 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 customer The Customer for this Quote
     *
     * @return The newly created EJB Object.
     */
    public Quote create(Customer customer) throws CreateException,
RemoteException;
}
```

빈 클래스

<소스> QuoteBean.java

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

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

/**
 *
 * This stateful session bean represents a quote which
 * 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, this bean 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.
 *
 * 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).
 *
 * This Bean could easily be extended to include other
 * things, such as:
 * - Shipping charges.
 * - Comments or special instructions the user gives.
 */
public class QuoteBean implements SessionBean {

    private SessionContext ctx;

    // Begin Stateful Session Bean Conversational State
    // fields
    // (Conversational state is non-transient, and
    // is perserved during passivation and activation)
    /*
     * The customer this quote belongs to.
     */
    private Customer customer;

    /*
     * The line items which constitute this quote
     */
    private Vector lineItems;
```

```
    /*
     * The subtotal (price before taxes) of the goods
     */
    private double subTotal;

    /*
     * The taxes on the goods
     */
    private double taxes;

    // End Stateful Session Bean Conversational State
    // fields
    /*
     * These are home objects that we need to find/
     * create EJB Objects.
     *
     * The home objects are not client specific, and
     * are not part of the conversational state on
     * behalf of a single client.  Hence they are
     * marked as transient.
     */
    private transient QuoteLineItemHome qliHome;
    private transient OrderLineItemHome oliHome;
    private transient OrderHome orderHome;

    public QuoteBean() {
            System.out.println("New Quote Stateful Session Bean created by
EJB Container.");
    }

    // Begin internal methods
    /**
     * For internal use only.  Finds a Home Object
     * using JNDI.
     *
     * Usage example:
     *  findHome("Ecommerce.QuoteLineItemHome")
     */
```

```
private EJBHome findHome(String homeName) throws Exception {

        /*
         * Get a reference to the Home Object
         * via JNDI.
         *
         * We rely on app-specific properties
         * to define the Initial Context
         * params which JNDI needs.
         */
        Context initCtx = new InitialContext(ctx.getEnvironment());
        EJBHome home = (EJBHome) initCtx.lookup(homeName);
        return home;
}

/**
 * For internal use only.  Generates a unique ID.
 *
 * When we convert this temporary quote into a
 * permanent order, we need to generate unique
 * IDs for the primary keys of the data we create.
 *
 * This is a poor-man's implementation that uses
 * the current system time.  A real deployment
 * should use a more robust method.
 */
private long uniqueID = System.currentTimeMillis();
private String makeUniqueID() {
        return Long.toString(uniqueID++);
}

/**
 * For internal use only.  Finds a line-item
 * based on a product.
 *
 * @return The matching line item, or null if
 *  not found
 */
private QuoteLineItem findLineItem(Product prod) throws RemoteException
```

```
{

        /*
         * Loop through all Line Items
         */
        Enumeration e = lineItems.elements();
        while (e.hasMoreElements()) {
                QuoteLineItem li = (QuoteLineItem) e.nextElement();

                /*
                 * Get the Product EJB Object
                 * from the line item.
                 */
                Product oldProd = li.getProduct();

                /*
                 * If the Products match (via
                 * EJB Object equivalency) then
                 * we've found our Line Item.
                 */
                if (oldProd.isIdentical(prod)) {
                        return li;
                }
        }

        return null;
}

// End internal methods
// Begin public business methods
/**
 * Returns the # of items in the quote
 */
public int getNumberOfItems() throws RemoteException {
        return lineItems.size();
}

/**
 * Returns the line-items.  Each line-item
```

```
     * represents a quantity of a single product.
     */
    public Vector getLineItems() throws RemoteException {
            return lineItems;
    }

    /**
     * Adds a product to the quote.
     */
    public void addProduct(Product prod) throws RemoteException,
QuoteException {

            /*
             * See if a line-item exists for this product already..
             */
            QuoteLineItem li = findLineItem(prod);

            /*
             * If it exists, simply increase the quantity.
             */
            if (li != null) {
                    li.setQuantity(li.getQuantity() + 1);
                    return;
            }

            /*
             * Otherwise, we need to make a new
             * Quote Line Item.  Use the Quote
             * Line Item Home Object to create
             * a new Quote Line Item EJB Object.
             */
            try {
                    li = qliHome.create(prod);
                    lineItems.addElement(li);
            }
            catch (CreateException e) {
                    throw new QuoteException("Could not create line item: " +
e.toString());
            }
```

```
    }

    /**
     * Call to adjust the quantity of a product.
     * If the quantity is 0, the product is removed.
     *
     * @exception QuoteException thrown if Product
     *  doesn't exist.
     */
    public void setProductQuantity(Product prod, int quantity) throws
QuoteException, RemoteException {
            QuoteLineItem li = findLineItem(prod);
            if (li != null) {
                    if (quantity > 0) {
                            li.setQuantity(quantity);
                    }
                    else {
                            lineItems.removeElement(findLineItem(prod));
                    }

                    return;
            }

            throw new QuoteException("No such Product in Quote!");
    }

    /**
     * Empties the quote
     */
    public void clear() throws RemoteException {
            /*
             * Remove each Quote Line Item
             * EJB Object
             */
            for (int i=0; i < lineItems.size(); i++) {
                    QuoteLineItem li = (QuoteLineItem)
lineItems.elementAt(i);
                    try {
                            li.remove();
```

```
                }
                catch (Exception e) {
                        e.printStackTrace();
                }
        }

        /*
         * Reset quote fields
         */
        lineItems = new Vector();
        taxes = 0;
        subTotal = 0;
}

/**
 * Returns the customer we're quoting.
 */
public Customer getCustomer() throws RemoteException {
        return customer;
}

/**
 * Returns the subtotal price which has been
 * previously set by setSubtotal().
 */
public double getSubtotal() throws RemoteException {
        return subTotal;
}

/**
 * Sets the subtotal price.
 *
 * Our external pricer bean is responsible for
 * calculating the subtotal.  It calculates it
 * based upon customer discounts (and can be
 * extended to include other rules as well).
 */
public void setSubtotal(double subTotal) throws RemoteException {
        this.subTotal = subTotal;
```

```
}

/**
 * Returns the taxes computed for this Quote.
 */
public double getTaxes() throws RemoteException {
        return taxes;
}

/**
 * Sets the taxes for this Quote.
 */
public void setTaxes(double taxes) throws RemoteException {
        this.taxes = taxes;
}

/**
 * Returns the total price.  Total Price
 * is computed from:
 *  1) Subtotal price
 *  2) Tax
 */
public double getTotalPrice() throws RemoteException {
        return subTotal + taxes;
}

/**
 * Converts this temporary quote into a
 * permanent, persistent order.  When the
 * user clicks the "Purchase" button on the
 * web site, a servlet will call this method.
 *
 * @throw QuoteException thrown if no items in quote
 */
public Order purchase() throws RemoteException, QuoteException {

        /*
         * Find home objects via JNDI, so we
         * can create an order and its
```

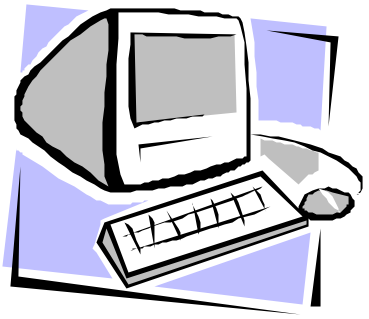
```
                * constituent order line-items.
                */
                try {
                        if (oliHome == null)
                                oliHome = (OrderLineItemHome)
findHome("Ecommerce.OrderLineItemHome");

                        if (orderHome == null)
                                orderHome = (OrderHome)
findHome("Ecommerce.OrderHome");
                }
                catch (Exception e) {
                        throw new RemoteException(e.toString());
                }

                /*
                * Make a unique ID for the order
                */
                String orderID = makeUniqueID();

                /*
                * Make a new persistent Order.
                */
                Order order = null;
                try {
                        order = orderHome.create(orderID, customer);
                        order.setSubtotal(subTotal);
                        order.setTaxes(taxes);
                }
                catch (Exception e) {
                        e.printStackTrace();
                        throw new QuoteException("Error generating an order: " +
e.toString());
                }

                /*
                * Convert each of our quote's line-
                * items into permanent, persistent
                * order line-items.
```

```
                 */
                try {
                        Vector orderLineItems = new Vector();

                        /*
                         * For each quote line item...
                         */
                        for (int i=0; i < lineItems.size(); i++) {

                                /*
                                 * Extract the fields from
                                 * the Quote Line Item...
                                 */
                                QuoteLineItem qli =
                                  (QuoteLineItem) lineItems.elementAt(i);
                                Product prod = qli.getProduct();
                                int quantity = qli.getQuantity();
                                double discount = qli.getDiscount();

                                /*
                                 * And shove the fields into
                                 * a new Order Line Item.
                                 */
                                String id = makeUniqueID();
                                OrderLineItem oli = oliHome.create(id, order,
prod, quantity, discount);
                        }
                }
                catch (Exception e) {
                        e.printStackTrace();
                        throw new QuoteException("Error generating an order line
items: " + e.toString());
                }

                return order;
        }

        // 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("Quote.setSessionContext called");
            this.ctx = ctx;

            /*
             * Get the QuoteLineItemHome Home Object
             * so we can create Line Items.
             */
            try {
                    this.qliHome = (QuoteLineItemHome)
findHome("Ecommerce.QuoteLineItemHome");
            }
            catch (Exception e) {
                    throw new RemoteException("Could not retrieve Quote
Line Item Home Object: " + e.toString());
            }
    }

    /**
     * The container calls this method directly after
     * activating this instance.  You should acquire
     * any needed resources.
     */
    public void ejbActivate() throws RemoteException {
            System.out.println("Quote.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("Quote.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(Customer customer) throws CreateException,
RemoteException {
            System.out.println("Quote.ejbCreate(" + customer + ") called");

            this.customer = customer;
            this.lineItems = new Vector();
            this.subTotal = 0;
            this.taxes = 0;
    }

    /**
     * Removes this quote, and hence all client-
     * specific state.
     *
     * In our example, there is really no way for
     * the client to 'abort' the quote, so the EJB
     * Container will call this when the Session
     * times out.
     */
    public void ejbRemove() throws RemoteException {
            System.out.println("Quote.ejbRemove() called.");

            /*
             * Remove all the Line Items which are
             * part of the quote.  We want to remove
             * the Line Items because an Quote is
```

```
                * made up of unique Line Items, which
                * other quotes cannot use.
                *
                * (For you UML fans, this is
                *  Composition, rather than
                *  Aggregation).
                */
               Enumeration e = lineItems.elements();
               while (e.hasMoreElements()) {
                        try {
                                QuoteLineItem li = (QuoteLineItem)
e.nextElement();
                                li.remove();
                        }
                        catch (Exception ex) {
                                // Ignore exceptions. EJB Container may
                                // have removed the Line Items already
                                // due to Session timeout.
                        }
               }
    }
}
```



Chapter 9

JNDI와 Client API

JNDI

Java Naming and Directory Interface(JNDI)는 Java 2 Platform, Enterprise Edition(J2EE)의 필수 컴포넌트이다. JNDI는 사용자, 머신(machine), 네트워크, 객체, 그리고 서비스의 locating에 대한 표준 인터페이스를 제공한다. 예를 들어 당신은 회사 인트라넷에서 프린터를 설치하기 위해서 JNDI를 이용할 수 있다. 또한 자바 객체 를 설치하거나 데이터베이스에 연결에 JNDI를 이용할 수 있다. EJB에서 JNDI는 엔터프라이즈 배치시에 객체를 locating할 목적으로 확장되어 사용된다.

Naming and Directory Services

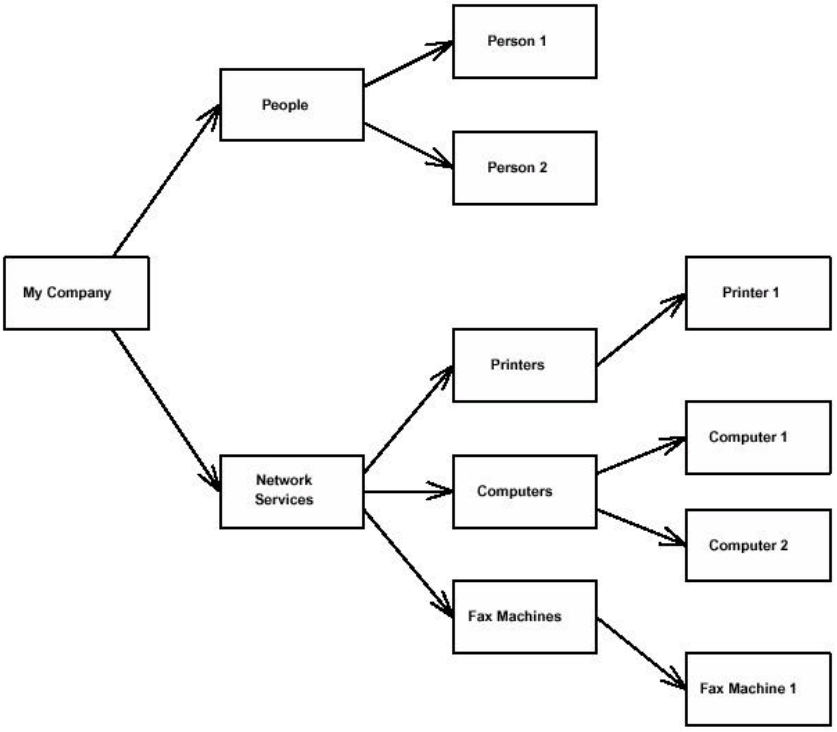
JNDI를 이해하기 위해서는 먼저 네이밍(naming) 서비스와 디렉토리(directory) 서비스를 이해해야 한다.

네이밍 서비스는 전화 교환수의 역할과 유사하다. 당신이 전화로 누군가와 통화를 하고싶은데 전화 번호를 모를 때, 당신은 전화 회사의 정보 서비스 담당자에게 당신이 통하하고 싶은 원하는 사람을 찾아(look up)달라고 전화할 수 있다. 그러면 담당자는 당신이 통화하고 싶어하는 사람의 전화 번호를 찾아 전화를 걸어 당신에게 그 사람을 연결시켜 줄것이다. 네이밍 서비스는 다음의 작업을 수행한다.

- ✍✍ 네이밍 서비스는 name을 객체와 연결한다. 이런 작업을 name과 객체를 바인딩(binding)한다고 표현한다. 이것은 전화 회사가 사람의 이름을 거주지의 전화 번호와 연결하는 것과 매우 유사하다.
- ✍✍ 네이밍 서비스는 name을 기반으로 객체를 찾는 편의를 제공한다. 이런 작업을 객체를 찾는다(look up)라고 표현한다. 이는 전화 교환수가 사람의 이름을 기반으로 해서 사람의 전화 번호를 찾아 두 사람을 연결시켜 주는것과 같다.

디렉토리 서비스는 속성을 다루는 directory object operation을 제공하기 위해 확장되고 향상된 네이밍 서비스이다. 디렉토리는 연결된 디렉토리 객체 의 시스템이다

. 디렉토리 제품의 예로는 Netscape Directory Server와 Microsofts's Active Directory가 있다. 예를 들어 당신의 회사에서는 아마도 컴퓨터의 위치(location), 현재 프린터의 상태, 개인적인 데이터 등의 회사 내부 정보를 저장하기 위해서 이러한 디렉토리 서비스를 사용할 것이다. 디렉토리가 내부적으로 어떤것과 비슷하게 보이지 않는가? 디렉토리의 내용은 보통 계층적인 트리 구조를 가지고 있을 것이다. 그것은 트리 형식이 실세계에서 회사가 조직된 구조를 제시하기 때문이다. 예를 들어 디렉토리 트리의 루트(top node)는 전체 회사를 나타낸다. 루트로 부터의 하나의 가지는 회사의 사람들을 나타내고, 다른 가지는 네트워크 서비스를 나타낸다. 각 가지는 내려갈수록 구성요소가 작아지는 서브트리(subtree)를 가질 수 있으며, 이는 각각의 사용자 객체, 프린터 객체, 머신 객체 등과 같은 것에 도달할 때 까지 계속된다.



<그림> 계층적인 디렉토리 구조

JNDI의 이점

JNDI가 제공하는 이점은 다음과 같다.

- JNDI는 모든 디렉토리 서비스 정보에 접근할 수 있는 통합된 시스템이다.
- JNDI는 서로 다른 디렉토리와 프로토콜에 접근할 수 있는 단일한 API를 제공한다.
- JNDI는 애플리케이션을 프로토콜과 세부적인 구현으로부터 분리한다.
- JNDI는 확장성이 있다. 미래에는 디렉토리 제공자가 클라이언트 코드의 수정 없이 특정 디렉토리 서비스를 JNDI에 추가할 수 있을 것이다.

✍ JNDI를 이용하여 디렉토리로부터 모든 자바 객체들을 읽고 쓸 수 있다.

✍ LDAP와 NDS 디렉토리의 경우처럼 서로 다른 타입의 디렉토리를 연결할 수 있다.

✍ JNDI는 Sun사에 의해 지원되며 표준 자바 확장(extension)이다.

EJB에서의 JNDI

JNDI는 클라이언트 애플리케이션이 EJB 서버를 일반 파일 시스템의 디렉토리와 같은 디렉토리 모음으로 간주할 수 있도록 해준다. 클라이언트 애플리케이션이 JNDI를 이용하여 EJB 홈에 대한 리모트 레퍼런스를 찾고, 이를 얻게 된 후에는 클라이언트가 빈에 대한 리모트 레퍼런스를 얻는 데 그 EJB 홈을 이용할 수 있다. 엔터프라이즈 빈에서 JNDI InitialContext 객체를 얻기 위해서 다음과 같이 getInitialContext() 메소드를 정의할 수 있다.

```
public static Context getInitialContext()
throws javax.naming.NamingException {
  Properties p = new Properties();
  // 공급 업체의 JNDI 프로퍼티를 정의한다.
  return new javax.naming.InitialContext(p);
}
```

initial context는 JNDI lookup의 시작점이라 할 수 있으며, 이는 파일 시스템의 루트 개념과 유사하다. initial context를 생성하는 방법이 독특하기는 하지만 근본적으로 어려운 것은 아니다. initial context를 생성하는 것은 Properties 타입의 프로퍼티 테이블에서 시작하며, 프로퍼티 테이블은 얻어 오게 될 초기 컨텍스트의 종류를 결정하는 다양한 값들을 추가하는 해시 테이블(hash table)이다.

이 코드는 사용하고 있는 EJB 서버가 어떻게 JNDI를 구성했는가에 따라 변화하게 된다. 예를 들어 BEA사의 WebLogic 서버에서는 다음과 같이 initial context를 가져오는 메소드를 정의할 수 있다.

```
public static Context getInitialContext() throws NamingException {
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
  p.put(Context.PROVIDER_URL, "t3://localhost:7001");
  return new InitialContext(p);
}
```

Client API

엔터프라이즈 빈 개발자들은 빈 클래스, 두 개의 인터페이스, 그리고 엔티티 빈의 경우 선택적으로 프라이머리 키 클래스를 작성한다. 이들 중에서 두 인터페이스와 프라이머리 키 클래스는 클라이언트가 볼 수 있으며, 빈 클래스는 볼 수 없다. 따라서 리모트 인터페이스와 홈 인터페이스, 프라이머리 키 클래스는 EJB에서 클라이언트 API의 역할을 한다. 이들 타입에 정의된 메소드들과 이들의 수퍼타입의 메소드들은 클라이언트들이 EJB 비즈니스 시스템과 상호 작용하는 데 사용하는 메커니즘을 제공한다.

Home Interface

홈 인터페이스는 빈에 대한 라이프 사이클과 메타데이터를 제공한다. 빈에 접근하기 위해 JNDI를 사용하면 홈 인터페이스를 구현하고 있는 EJB 홈에 대한 리모트 레퍼런스나 스텁을 얻게 된다. 모든 빈 타입은 하나의 홈 인터페이스를 갖는다. 다음은 홈 인터페이스를 생성하는데 사용되는 javax.ejb.EJBHome 인터페이스의 정의이다.

```
public interface javax.ejb.EJBHome
extends java.rmi.Remote {
public abstract EJBMetaData getEJBMetaData()
throws java.rmi.RemoteException;
public abstract void remove(Handle handle)
throws java.rmi.RemoteException, javax.ejb.RemoveException;
public abstract void remove(Object primaryKey)
throws java.rmi.RemoteException, javax.ejb.RemoveException;
}
```

getEJBMetaData()	엔터프라이즈 빈에 대한 정보에 접근하기 위해서 사용된다. 예를 들면 빈이 세션 빈인지 엔티티 빈인지 에 대한 정보를 들 수 있다. 이 정보는 getEJBMetaData() 메소드가 반환하는 EJBMetaData 객체에 캡슐화되어있다. EJBMetaData는 주로 개발 툴에서 유용하게 쓰인다. 아마도 개발자가 직접 EJBMetaData를 다루는 일은 거의 없을 것이다.
remove()	이 메소드는 EJB 객체를 삭제한다. remove() 메소드는 두 가지 방법으로 호출할 수 있다. 1. javax.ejb.Handle 객체를 파라미터로 넘겨주어 EJB 객체를 삭제한다. 2. 프라이머리 키를 넘겨주어 삭제한다. 이 방법은 엔티티 빈에서만 가능하다.

✍✍ **빈의 제거**

빈의 제거는 remove() 메소드를 통해 이루어진다. remove() 메소드의 파라미터는 javax.ejb.Handle이거나 엔티티 빈의 경우 프라이머리 키가 될 수 있다. 두 remove() 메소드 중 하나가 호출되면 빈에 대한 클라이언트의 리모트 레퍼런스가 무효화된다.

빈의 타입에 따라 remove() 메소드는 다르게 작동한다. 세션 빈에서는 remove() 메소드는 클라이언트에 대한 해당 세션의 서비스를 종료시킨다. remove() 메소드가 호출되면 세션 빈에 대한 리모트 레퍼런스가 무효화 되고, 상태유지 세션 빈의 경우 대화 상태를 잃어버리게 된다.

엔티티 빈의 경우에 remove() 메소드가 호출되면 리모트 레퍼런스가 무효화 될 뿐만 아니라 제거된 엔티티 빈이 나타내던 데이터가 데이터베이스로부터 실제로 지워진다. 따라서 엔티티 빈에서의 remove() 메소드 호출은 주의깊게 이루어져야 한다.

✍✍ **빈의 생성 및 찾기**

홈 인터페이스가 상속받는 EJBHome 인터페이스 내에 정의된 메소드들 외에도 홈 인터페이스는 생성(create) 메소드와 찾기(find) 메소드를 포함한다. 생성 메소드와 찾기 메소드에 대해서는 4장에서 엔티티 빈과 세션 빈의 경우로 나뉘어 자세히 설명하고 있다.

생성 메소드와 찾기 메소드는 빈마다 서로 다른 특징을 갖기 때문에, 홈 인터페이스에서 생성 및 찾기 메소드를 정의하는 것은 개발자의 책임이다. 세션 빈의 경우에는 엔티티 빈과 달리 찾기 메소드가 없다. 엔티티 빈은 데이터베이스 내의 고유한 데이터를 표현하므로 찾기 메소드의 사용이 가능하지만 세션 빈은 그렇지 않으므로 찾기 메소드를 정의하는 것은 의미없는 일이다.

홈 인터페이스의 생성 메소드들은 반드시 빈 클래스의 ejbCreate() 메소드와 대응되어야 한다. 클라이언트가 홈 인터페이스의 생성 메소드 호출 시 빈 인스턴스의 ejbCreate() 메소드에게 임무를 위임할 수 있다. 찾기 메소드 역시 마찬가지로 빈 클래스의 ejbFind() 메소드와 대응되어야 한다는 사실에 유의해야 한다.

✍✍ EJBMetaData

getEJBMetaData() 메소드는 javax.ejb.EJBMetaData 의 인스턴스를 반환한다. EJBMetaData 인스턴스는 빈이 세션 빈인지 아니면 엔티티 빈인지의 여부, 홈 인터페이스, 리모트 인터페이스, 그리고 프라이머리 빈 클래스 등과 같은 엔터프라이즈 빈에 대한 메타데이터를 포함한다. 이러한 메타 데이터는 일반적인 클라이언트 코드에서는 거의 쓰이지 않지만 동적으로 엔터프라이즈 빈의 정보를 관리해야 하는 클라이언트나 EJB 배치 툴과 같은 클라이언트의 경우에 유용하게 쓰일 수 있다.

Remote Interface

✍✍ Remote Interface

엔터프라이즈 빈의 비즈니스 메소드는 빈 개발자가 제공하는 리모트 인터페이스

에 의해 정의된다. 리모트 인터페이스는 클라이언트가 호출할 수 있도록 엔티티 빈의 인터페이스를 제공하며 외부 세계에 제공되는 빈의 비즈니스 메소드를 정의한다. 다음은 리모트 인터페이스를 생성하는데 사용되는 javax.ejb.EJBObject 인터페이스의 정의이다.

```
public interface javax.ejb.EJBObject
        extends java.rmi.Remote {
  public abstract javax.ejb.EJBHome getEJBHome()
        throws java.rmi.RemoteException;
  public abstract java.lang.Object getPrimaryKey()
        throws java.rmi.RemoteException;
  public abstract void remove()
        throws java.rmi.RemoteException, javax.ejb.RemoveException;
  public abstract javax.ejb.Handle getHandle()
        throws java.rmi.RemoteException;
  public abstract boolean isIdentical(javax.ejb.EJBObject)
        throws java.rmi.RemoteException;
}
```

getEJBHome()	해당 EJB 홈에 대한 레퍼런스를 반환한다.
getPrimaryKey()	EJB 객체에 대한 프라이머리 키를 반환한다. 프라이머리 키는 엔티티 빈에서만 사용할 수 있다.
remove()	EJB 객체를 파괴한다. 엔티티 빈에서 remove() 메소드를 호출했을 경우 데이터베이스의 데이터가 삭제된다.
getHandle()	EJB 객체로부터 핸들을 가져온다. EJB 핸들은 EJB 객체에 대한 지속성있는 (persistent) 레퍼런스이다.
isIdentical()	두 EJB 객체가 동일한지를 비교한다.

EJB 홈 얻기

getEJBHome() 메소드는 빈의 EJB 홈에 대한 리모트 레퍼런스를 반환한다. 리모트 레퍼런스는 javax.ejb.EJBHome 객체로 돌아오며, 그것은 특정 빈의 홈 인터페이스로 전달될 수 있다. 이 메소드는 EJB 객체가 자신을 만들었던 EJB 홈의 범위를 벗어날 때 유용하다. 리모트 레퍼런스는 여타의 클라이언트측 자바 객체처럼 레퍼런스 형태로 전달되어 메소드로부터 반환될 수 있기 때문에, 원래 자신의 홈과 완전히 다른 곳에서도 자신을 빠르게 찾아낼 수 있다.

프라이머리 키

getPrimaryKey() 메소드는 빈의 프라이머리 키를 반환한다. 이 메소드는 엔티티 빈을 나타내는 EJB 객체만이 지원한다. 세션 빈의 경우 데이터의 표현이 아닌 특정 임무나 프로세스를 표현하므로 프라이머리 키가 무의미하다. 엔티티 빈은 이 프라이머리 키를 이용하여 다른 빈들을 구별할 수 있는 특정 데이터를 나타낸다.

빈의 비교

isIdentical() 메소드는 두개의 EJB 객체의 리모트 레퍼런스를 비교한다. 왜 Object.equals() 메소드가 EJB 객체들을 비교하는데 불충분한지 생각해보는 것도 중요하다. EJB 객체는 분산 객체 스텁이므로 많은 네트워킹과 기타의 여러 가지 상태를 포함하고 있다. 따라서 두 개의 EJB 객체가 동일한 하나의 빈을 나타내면서도 같지 않을 수 있다. isIdentical() 메소드는 만일 EJB 객체 스텁이 동일한 빈을 레퍼런스하는 서로 다른 객체 인스턴스라 하더라도 두 개의 EJB 객체 레퍼런스가 동일한 빈을 나타내고 있으면 true를 반환한다.

빈의 제거

remove() 메소드는 세션 빈이나 엔티티 빈을 제거하는데 사용된다. 이 메소는 EJBHome.remove() 메소드와 동일한 효과를 가져온다. 세션 빈에 대한 remove() 메소드의 호출은 세션이 끊어지고 EJB 객체 레퍼런스가 무효화 되는 효과를 가져오며, 엔티티 빈에서는 실제 엔티티 데이터가 데이터베이스에서 삭제되고 리모트 레퍼런스가 무효화되는 결과를 가져온다.

Handle

많은 EJB 애플리케이션에서 빈과의 연결이 끊긴 클라이언트가 나중에 빈에 재연결하여 빈을 계속 사용할 수 있는 기능이 요구된다. EJB는 이러한 필요성을 위해 핸들을 제공한다. 만약 당신이 어떤 이유로 EJB 컨테이너/서버로부터 연결을 끊었다면, 당신은 핸들을 이용해 해당 EJB 객체에 재연결함으로써 빈과의 대화 상태(conversational state)를 유지할 수 있다. 예를 들어, 클라이언트가 주문 프로세스를 처리하는 상태유지 세션 빈을 사용하고 있다고 가정해 보자. 클라이언트는 어떤 이유로 프로세스를 멈추어야 할 필요가 있을 수 있다. 이때 해당 세션에서 하던 일을 잃어버리는 대신, 클라이언트 애플릭이션은 EJB 객체로부터 핸들을 얻은 다음 주문 프로세스를 멈출 수 있다. 사용자가 다시 주문을 계속하고자 한다면, 핸들을 사용하여 상태유지 세션 EJB 객체에 대한 레퍼런스를 얻을 수 있다. 이와 같이 핸들은 EJB 객체에 대한 지속성있는(persistent) 레퍼런스이다. 다음은 javax.ejb.Handle 인터페이스의 정의이다.

```
public interface javax.ejb.Handle {
public javax.ejb.EJBObject getEJBObject();
}
```

getEJBObject()	핸들이 생성된 EJB 객체를 반환한다.

Handle 예제

클라이언트는 자바 직렬화를 이용하여 Handle 객체를 저장하고, 같은 EJB 객체에 대한 레퍼런스를 얻기 위해 그것을 다시 비직렬화 시켜 사용할 수 있다. 다음의 코드는 핸들의 사용예이다.

```
// 먼저 EJB 객체로부터 EJB 객체 핸들을 가져온다.
javax.ejb.Handle myHandle = myEJBObject.getHandle();

// 다음으로 myHandle을 serialize해서 영구 저장장치에 저장한다.
// 여기서는 파일에 저장한다.
FileOutputStream fos = new FileOutputStream("myhandle.ser");
ObjectOutputStream outStream = new ObjectOutputStream(fos);
outStream.writeObject(myHandle);
outStream.flush();
fos.close();

// 시간이 흐르고…
// EJB 객체를 다시 사용하고자 할 때
// EJB 객체 핸들을 deserialize한다.
FileInputStream fis = new FileInputStream("myhandle.ser");
ObjectInputStream inStream = new ObjectInputStream(fis);
Handle myHandle = (Handle) inStream.readObject();
fis.close();

// EJB 객체 핸들에서 EJB 객체를 가져온다.
MyRemoteInterface myEJBObject =
(MyRemoteInterface) myHandle.getEJBObject();

// EJB 객체를 사용한다.
myEJBObject.callMethod();
```

예제

AdderServlet 예제

이번 예제에서는 서블릿과 상태유지 세션 빈을 사용해 간단한 덧셈기를 구현한다. 웹페이지에서 사용자가 임의의 숫자를 입력하면 지금까지 입력한 숫자의 합과 새로 입력된 숫자가 더해져서 출력된다. 예제를 통해 서블릿으로 구현된 클라이언트 프로그램이 어떻게 홈 인터페이스와 리모트 인터페이스에 접근하여 빈과 상호작용을 하는지 알아보자.

엔터프라이즈 빈 소스코드

AdderEJB 엔터프라이즈 빈은 상태유지 세션 빈으로써 다음과 같은 구성을 가지고 있다.

✍✍ 빈 클래스

```
import javax.ejb.*;
public class AdderEJB implements SessionBean {
int total;
public void ejbCreate() {
total = 0;
}
public void ejbCreate(int initial) {
total = initial;
}
public void add(int number) {
total += number;
}
public int getTotal() {
return total;
}
public AdderEJB() {}
public void setSessionContext(SessionContext sc) {}
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
}
```

<소스> AdderEJB.java

✍ 홈 인터페이스

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface AdderHome extends EJBHome {
   Adder create() throws RemoteException, CreateException;
   Adder create(int initial)
   throws RemoteException, CreateException;
}
```

<소스> AdderHome.java

✍ 리모트 인터페이스

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Adder extends EJBObject {
   void add(int number) throws RemoteException;
   int getTotal() throws RemoteException;
}
```

<소스> Adder.java

서블릿/HTML 소스 코드

이 예제에서는 서블릿으로 클라이언트를 구현하였다. AdderServlet 클래스의 init() 메소드에서 AdderEJB 엔터프라이즈 빈의 EJB 홈 객체와 EJB 객체를 얻는 코드가 작성되어 있으므로 주의깊게 분석해보기 바란다.

✍ AdderServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.rmi.PortableRemoteObject;
import javax.naming.InitialContext;

import Adder;
```

```
import AdderHome;

public class AdderServlet extends HttpServlet {

  Adder adder;

  public void init() throws ServletException {
    System.out.println("in init of AdderServlet");
    try {
      InitialContext ic = new InitialContext();
      Object objref =
      ic.lookup("java:comp/env/ejb/Adder");
      System.out.println("lookup ok");
      AdderHome home =
      (AdderHome)PortableRemoteObject.narrow(objref,
                                    AdderHome.class);
      System.out.println("narrow ok");
      adder = home.create(0);
      System.out.println("create ok");
    } catch(Exception e) {
          e.printStackTrace();
    }
  }

  public void doGet (HttpServletRequest req, HttpServletResponse res)
          throws ServletException, IOException {
    System.out.println("in doGet");
    String inputString = req.getParameter("inputString");
    Integer inputNumber = new Integer(inputString);
    adder.add(inputNumber.intValue());
    int total = adder.getTotal();
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    generatePage(out,total);
  }

  private void generatePage(PrintWriter out, int total) {
    out.println("<html>");
    out.println("<head>");
```

```
        out.println("<title>Input for AdderServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("The running total is: " +
         String.valueOf(total));
        out.println("<p>");
        out.println("<form method = " +
        "get action=\"AdderAlias\">");
        out.println("Please enter an integer:");
        out.println("<input type=text name= \"inputString\">");
        out.println("<p>");
        out.println("<input type=submit>");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }

    public String getServletInfo() {
     return "This servlet accesses an enterprise bean.";
    }
}
```

<소스> AdderServlet.java

✍✍ adder.html

```
<html>
<head>
<title>Initial Page for AdderServlet</title>
</head>
<body>
<form method = get action="AdderAlias">
Please enter an integer:
<input type=text name="inputString">
<p>
<input type=submit>
</form>
</body>
</html>
```

<소스> adder.html